Design and Cryptanalysis in Multivariate Quantum-Safe Cryptography

Jocelyn Ryckeghem

To cite this version:

Jocelyn Ryckeghem. Design and Cryptanalysis in Multivariate Quantum-Safe Cryptography. Cryptography and Security [cs.CR]. Sorbonne Université, 2021. English. NNT : 2021SORUS577 . tel-04326611

HAL Id: tel-04326611
https://theses.hal.science/tel-04326611

Submitted on 24 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Sorbonne Université

École Doctorale Informatique, Télécommunications et Électronique de Paris

Laboratoire d'Informatique de Sorbonne Université / Équipe PolSys

Cryptographie post-quantique : conception et analyse en cryptographie multivariée

Par Jocelyn Ryckeghem

Thèse de doctorat d'Informatique

Dirigée par Jean-Charles Faugère et Ludovic Perret

Présentée et soutenue publiquement le lundi 8 février 2021

Devant un jury composé de :

M. Jean-Guillaume Dumas	Professeur	Rapporteur
M. Jean-Charles Faugère	Directeur de recherche	Directeur de thèse
Mme Aline Gouget Morin	Directrice en cryptographie	Examinatrice
M. Pierre Loidreau	DGA et chercheur associé	Examinateur
M. Ludovic Perret	Maître de conférences	Directeur de thèse
M. Mohab Safey El Din	Professeur	Président
M. Daniel Smith-Tone	NIST and associate professor	Rapporteur

To cite this PhD thesis

Jocelyn Ryckeghem. Cryptographie post-quantique : conception et analyse en cryptographie multivariée PhD dissertation, Sorbonne Université, February 2021. English, 284 pages.

Alternate title

Design and Cryptanalysis in Multivariate Quantum-Safe Cryptography

Funding

This work is financially supported by the French / Cette thèse est soutenue financièrement par le

Ministère des armées – Direction Générale de l'Armement.

License

Except where otherwise noted, this work is licensed under
Creative Commons License CC-BY-NC-ND 3.0.
https://creativecommons.org/licenses/by-nc-nd/3.0/

Printing this book

Color printing of pages 205, 206 and 207 is recommended. The cover page could also be considered. The corresponding absolute page numbers are 239, 240 and 241 (if the cover page number is 1).

Note of the author

This PhD thesis presents the MQsoft software toolkit, which was entirely implemented by Jocelyn Ryckeghem (2017–2020). The latter announces his decision to cease the development of this toolkit. However, a larger project initiated by the author, the FAHP (High-Performance Finite Field Arithmetic) library, could include equivalent features in the future.

Keywords

multivariate cryptography
e cient software implementation
GeMSS
DualModeMS
HFE
Approximate MQ
MQsoft
root finding
binary field

Mots clés

cryptographie multivariée
implémentation logicielle e cace
GeMSS
DualModeMS
HFE
AMQ
MQsoft
recherche de racines
corps binaire

Science sans conscience n'est que ruine de l'âme.

François Rabelais, *Pantagruel*

À Paola,

Remerciements

> Le fait de ne pas croire au hasard, et que le hasard existe et soit fait de façon à ce qu'il ne se passe que des coïncidences, serait une belle coïncidence.
>
> — Jocelyn Ryckeghem, 2013

Bien que cette thèse soit rédigée en anglais, je profite de ces quelques pages pour écrire dans ma langue maternelle, qui m'est si chère. J'ai rencontré d'innombrables personnes durant ma thèse à Sorbonne Université, ce qui a été très motivant pour mon travail de recherche. Je les remercie donc aujourd'hui.

Je souhaite d'abord remercier Jean-Charles Faugère et Ludovic Perret, mes directeurs de thèse. C'est grâce à eux que jadis, en master, je découvris la cryptographie multivariée et la résolution des systèmes polynomiaux non-linéaires. J'ai ensuite pu contribuer en soumettant `GeMSS` et `DualModeMS` au processus de standardisation de cryptographie post-quantique du NIST. Puis j'ai continué en thèse pour obtenir davantage de résultats et défendre nos candidats, en plus d'acquérir la compétence de chercheur. Ainsi, `GeMSS` est arrivé jusqu'au troisième tour de ce processus, et tout cela n'aurait été possible sans Jean-Charles et Ludovic.

Ensuite, je remercie Jean-Guillaume Dumas et Daniel Smith-Tone, mes rapporteurs de thèse, pour l'avoir rapportée dans des délais très courts ! Je remercie Jean-Guillaume pour ses remarques, toutes très pertinentes, qui ne font qu'améliorer la qualité de ce tapuscrit. Especially to Daniel, it is an honor for me to have the comments from a specialist in multivariate cryptography of the NIST team.

Je remercie Aline Gouget Morin, Pierre Loidreau et Mohab Safey El Din, ainsi que Jean-Guillaume et Daniel, d'avoir participé à mon jury de thèse. Je remercie particulièrement Pierre d'avoir été mon tuteur DGA, et plus largement, je remercie la DGA/AID pour son soutien financier. Après avoir effectué mon stage de fin de master à l'ANSSI, cela est très motivant pour moi d'avoir décroché une demi-bourse DGA. J'espère que cette thèse est à la hauteur de la confiance qui m'a été accordée. J'en profite pour remercier Jean-Pierre Flori et Jérôme Plût qui ont encadré mon stage à l'ANSSI, et qui m'ont très bien formé, tel un prélude à cette thèse. Après six mois de travail acharné sur la composition modulaire de polynômes, je ne m'attendais pas à la revoir ici ! Bien que ce ne soit évidemment pas celle de Kedlaya–Umans. Merci aussi à l'INRIA et à Sorbonne Université pour avoir complété les financements de la DGA.

Je remercie les membres de l'équipe `GeMSS` : Antoine Casanova, Gilles Macario-Rat et Jacques Patarin. En particulier, merci à Jacques pour nos échanges sur `GeMSS`, et sur `HFE` plus généralement. C'est très motivant de discuter avec le créateur de `HFE` en personne.

Maintenant, je souhaite remercier toute l'équipe PolSys pour son accueil chaleureux. Merci à Mohab Safey El Din que je connais depuis ma L2 Parcours Informatique et Mathématiques Appliquées (PIMA), et qui aura accentué mon goût pour l'implantation efficace. Nous aurons eu des

discussions très intéressantes sur le sujet durant ma thèse. Merci à Guénaël Renault de m'avoir donné, dès la L3, le goût de la cryptographie ainsi que la manière de penser « crypto ». Si le master Sécurité, Fiabilité et Performance du Numérique (SFPN) n'avait pas été créé, j'aurais sans doute privilégié un master en mathématiques... un chemin bien différent ! Et merci pour tes visites toutes fort sympathiques, et très instructives pour résoudre certains problèmes. Merci à Jérémy Berthomieu de m'avoir enseigné le calcul formel. Tes enseignements resteront sans aucun doute ma partie préférée du master SFPN, et j'ai bien aimé toutes nos conversations, qui auront été fortement utiles pour ma thèse. Cela implique de te remercier pour ta grande disponibilité ! En somme, les permanents de PolSys auront grandement contribué à tout ce que je sais faire aujourd'hui. Et je n'oublie pas Élias Tsigaridas, qui aura toujours eu un mot gentil en entrant dans mon bureau ! Je remercie notre ancien gestionnaire financier, Irphane Khan, qui a toujours fait du très bon travail, et cela avec une grande humanité. Je remercie les ingénieurs Konstantin Kabassanov et Pierre-Emmanuel Le Roux pour la qualité de leur service. Difficile de lancer des tests de performance avec un serveur hors-service ou avec une fiche de chargeur d'ordinateur portable défectueuse !

Je remercie aussi les permanents des autres équipes du troisième étage du couloir 26-00. Je remercie Stef Graillat pour toutes ces conversations si sympathiques. Je remercie également Fabienne Jézéquel pour nos quelques discussions en salle de convivialité. Je remercie Pierre Fortin pour avoir si bien enseigné le calcul haute performance quand j'étais en master SFPN, ainsi que pour nos discussions en la matière. Je n'aurai, hélas, pas eu l'opportunité de coder en AVX-512 ! Mais ce n'est que partie remise. Je remercie Valérie Ménissier-Morain pour ses conseils, tant sous la casquette de vice-présidente de l'enseignement que sous celle de vice-directrice du master SFPN. Je la remercie aussi pour toutes les anecdotes qu'elle m'aura racontées. Je remercie Jean-Marie Chesneaux qui m'aura beaucoup appris quand j'enseignais LI217 (alias 2I011), et cela pendant deux années consécutives ! Je te remercie également d'avoir participé à mon comité de suivi de thèse, et de m'avoir fait part de tes conseils avisés avec une grande honnêteté. Je n'oublie pas Christoph Lauter, qui en plus d'avoir dirigé LI217, m'aura lui-même donné le goût du calcul scientifique lors de ma L2. Je remercie Julien Tierny pour sa légendaire bonne humeur lors de nos nombreuses rencontres inopinées. Et plus largement, merci à Dominique Béréziat et à Marc Mezzarobba pour nos quelques échanges.

L'enseignement aura été une part importante de ma thèse, et sans doute la plus grande source de joie qu'elle m'ait apportée. Ayant déjà remercié la plupart des enseignants que j'ai rencontrés, je termine le travail ici. Je remercie Damien Vergnaud pour ses cours très intéressants d'introduction à la sécurité. J'ai appris (et renforcé) pas mal de choses grâce à cela (incluant ton fameux livre). Je remercie Stéphane Doncieux qui m'aura fait vivre l'organisation d'une UE aussi énorme que LU2IN018 (C avancé), François Bouchet pour le partage de son expérience de l'UE, et mon chargé de TME Daniel pour sa bonne volonté. Un immense merci à tous mes étudiants, pour avoir égayé mes journées. Vous instruire aura été ce qu'il y a de plus motivant durant ma thèse, et je vous souhaite à tous d'exceller. J'adresse un petit clin d'œil à tous les étudiants en licence PIMA et en master SFPN que j'ai rencontrés, ainsi qu'aux étudiants en spécialité MAIN de Polytech Sorbonne. Pour ces derniers, j'ai grandement apprécié d'enseigner à un public de type ingénieur.

Les formations doctorales auront été un autre aspect de ma thèse, parfois un peu folklorique. Je remercie le formateur du PSC1, Catherine pour ses excellentes formations de communication et lecture rapide, puis Anne de l'association EnAct pour sa formation très originale d'expression avec aisance grâce au théâtre. Histoire ! Raconte ! Je remercie aussi tous les doctorants que j'ai rencontrés dans ce contexte, avec une petite pensée pour Sigrid qui m'aura appris ce que sont les cyanobactéries et leur utilité.

Je me souviens d'une expérience très enrichissante. Ma conférence à Atlanta (CHES 2019), qui fut mon premier voyage à l'étranger seul. Il y a un grand nombre de personnes que j'ai croisées ou rencontrées durant ce voyage, et je souhaite toutes les remercier, même celles qui n'ont aucune chance de lire un jour ce passage. Je suis satisfait d'avoir rencontré la petite communauté française en cryptographie à CHES 2019, au Westin Peachtree Plaza, et j'ai une pensée pour chaque personne rencontrée. En particulier, merci à Luk Bettale qui est également l'un de mes prédécesseurs en HFE ! Après avoir tant lu ta thèse, ce fut finalement à mon tour d'écrire la mienne. Et merci à Pierre-Alain Fouque pour l'organisation de cette belle conférence. Je conclus les remerciements de ce voyage en remerciant l'un de mes sympathiques étudiants que j'ai croisé juste avant de rentrer dans l'avion (quelle coïncidence !), et Matthieu, pour cette super conversation dans l'avion alors que tout le monde préférait dormir. Quelle coïncidence d'avoir pour voisin un étudiant en master Mathématiques, Informatique de la Cryptologie et sécurité (MIC) qui semble partager pas mal de points communs et centres d'intérêts.

J'en viens alors à mes chers camarades. Et oui, j'ai encore des mercis en réserve à distribuer ! D'abord à Matías pour avoir été un bon camarade de bureau, pour avoir élargi ma culture musicale, et pour avoir voulu m'aider quand il pensait (pour ne pas dire savait) que je n'allais pas fort. ¡La vida es dura e injusta! Puis à Paola, qui semble connaître Atlanta mieux que moi ! (Ultime coïncidence de te rencontrer peu après mon voyage.) J'ai été ravi de partager avec toi ma connaissance de HFE (le tableau s'en souvient encore), et j'ai été très satisfait de t'accueillir en 26-00-315. Sans le savoir, ta bonté et ta manière naturelle de penser auront grandement contribué à ma construction personnelle. Et je te remercie de m'avoir rappelé ce qui est le plus important. Je tâcherai de ne jamais l'oublier. Un dernier mot à Rémi, qui est selon moi un digne successeur de 26-00-315. Nous n'aurons pas vraiment eu l'occasion de partager ce bureau, mais cela aura été un grand plaisir de parler par écrit de la multiplication dans les corps premiers. Et puis, la nouvelle disposition du bureau me semble très prometteuse ! Je n'oublie pas mes camarades de 26-00-326, où j'ai passé mes deux premiers mois. Ma chère Éliane. Je me souviens encore de mon premier jour au LIP6[1]. Une époque où l'anglais devait parfois prendre la place du français, et qui est révolue aujourd'hui. En tant d'années, je ne compte plus nos conversations sur HFE. Je suis content d'avoir pu côtoyer une mathématicienne ayant un cœur aussi pur, et je te remercie donc pour ton bon cœur. Je remercie également ton amie Nelly ! Viens ensuite Olive. La première fois où je suis (vraiment) forcé de communiquer en anglais. Cela n'aura pas été facile au début, mais mon anglais s'est amélioré grâce à toi. Te retournant l'ascenseur, je te laisse le soin de traduire ce passage. On aura eu quelques fous rires, je vois encore le ventilateur de table s'exploser contre le sol. Ton sens de l'humour mérite nettement des remerciements. Finalement, il reste Solane, qui techniquement n'a fait que récupérer mon bureau... Je te remercie pour l'attention que tu as pu me porter, ainsi que pour nos nombreuses conversations (sur une palette très large de sujets) !

Pour ce qui est des membres du bureau 26-00-325, je remercie Léo pour nos conversations sophistiquées sur l'optimisation de l'utilisation du processeur. C'est rare de voir quelqu'un aussi motivé par le sujet. Mon cher Guillaume, je suis très satisfait de toutes ces fois où je suis passé discuter à ton bureau. Une petite pensée à Charles qui avait toujours des discours intéressants, peu importe le sujet. Je n'oublie pas Jules qui aura quitté le couloir pour de nouvelles aventures (dans le couloir d'à côté). Je remercie le duo de stagiaires Léa et Joseph, puis le trio de stagiaires Arthur, Céline et Élodie, pour toutes nos conversations. Une pensée spéciale à Arthur qui aura été là alors que la plupart de mes camarades avaient déjà quitté le laboratoire. Je te décerne le titre honorifique de camarade de fin de thèse.

[1] Laboratoire d'Informatique de Sorbonne Université, jadis appelé Laboratoire d'Informatique de Paris 6.

Je termine par les trois bureaux restants. Rachel, we did not spend a lot of time together. But I miss your British accent, as well in French as in English! It was really a pleasure to talk to you. Aniss, chacune de nos discussions aura été emplie de ta grande sympathie. Et Mathieu, merci pour tes explications si sophistiquées, notamment sur les théories justifiant un manque de diversité dans le code génétique humain.

Une pensée va également au quatrième étage du couloir 24-25, à Lucas et Jérôme mes anciens camarades de la spécialité SFPN, et à ce cher Vincent dont les visites auront toujours été les bienvenues.

Je suis content d'avoir croisé les PIMA de ma génération : Anne-Élisabeth, Arthur, Mickael, et même Malik (quelle surprise) ! Sans oublier ces super pauses café avec ce très cher Étienne ! Et ces quelques rencontres inopinées avec Pierre.

Mon cher Baptiste, que ce soit en tant que fidèle binôme de projet, camarade PIMA et SFPN, ou en tant qu'ami, je te remercie pour tous ces moments que nous avons passés ensemble. Ce fut toujours une bonne surprise de te voir passer au bureau.

Je remercie tous mes amis, avec en particulier quelques remerciements anonymisés. Je remercie mon frère spirituel, ainsi que sa famille, pour tous ces super moments passés avec eux. Je remercie mon ami de longue date, ce cher Envy, qui m'aura sauvé plus d'une fois lors des coupures réseaux de ma Picardie natale (au moins, nous nous serons revus avant mon départ à CHES 2019). Je remercie un certain professeur des écoles avec qui le nouvel an (mais pas que) aura toujours été un bon moment. Je remercie ma professeur d'anglais préférée, pour son amitié et ses conseils en anglais. Je remercie quelques amis de lycée pour des discussions épiques dans le train. Que des amis de Picardie au final. Bien évidemment, d'autres périphrases me viennent à l'esprit… et je les remercie tout particulièrement pour leur patience en mon absence.

Je conclus en remerciant tous les doctorants, contractuels, stagiaires et post-doctorants que j'aurai rencontrés durant ma thèse. Comme le bouquet final des feux d'artifice, je fais fuser les prénoms : Iryna, Amine, Reine, Nagarjun, Clothilde, Anastasia, Thi Xuan, Huu Phuoc, Andrew, Jorge, Ramtin, Trung-Hieu, Clara, Marie et Jérémie.

Je vous remercie tous d'avoir ØtØ là. J'ai ØtØ en trŁs bonne
ces annØes.

Jocelyn

Table of Contents

Acknowledgments i

1 Introduction 1
- 1.1 Previous Work . . . 3
- 1.2 Organization of the Document . . . 4
- 1.3 Contributions . . . 5

I Preliminaries 11

2 Multivariate Cryptography 13
- 2.1 Public-Key Cryptography and NP-Completeness . . . 13
- 2.2 Introduction to Multivariate Cryptography . . . 14
- 2.3 MI-Based Public-Key Cryptography . . . 15
  - 2.3.1 Keypair Generation . . . 15
  - 2.3.2 Signing Process . . . 16
  - 2.3.3 Verifying Process . . . 17
- 2.4 Families of Trapdoor . . . 17
  - 2.4.1 Big Field Family . . . 18
  - 2.4.2 Unbalanced Oil and Vinegar (UOV) . . . 20
  - 2.4.3 Rainbow . . . 21

3 Public-Key Compression of Signature Schemes 23
- 3.1 Dual Keypair Generation . . . 23
- 3.2 Dual Signing Process . . . 24
- 3.3 Merkle Tree . . . 26
- 3.4 Dual Verifying Process . . . 27
- 3.5 Security . . . 29
- 3.6 SBP Transformation of MI-Based Signature Schemes . . . 31

4 Cryptanalysis Techniques 35
- 4.1 Metrics and NIST Security Strength Categories . . . 35
- 4.2 Generic Attack and Feistel–Patarin Construction . . . 37
- 4.3 Direct Signature Forgery Attacks . . . 38
  - 4.3.1 Exhaustive Search . . . 39

4.3.2 Approximation Algorithm 39
4.4 Gröbner Bases 40
4.4.1 Practically Fast Algorithms 40
4.4.2 Asymptotically Fast Algorithms 42
4.5 Direct Attack against Approximate PoSSo 43
4.6 Key-Recovery Attacks against HFE 44
4.6.1 Exhaustive Search and Equivalent Keys 44
4.6.2 MinRank 45
4.6.3 Kipnis–Shamir Attack 45
4.7 Side-Channel Attacks 46

5 Arithmetic 49
5.1 Basic Arithmetic in the Polynomial Ring $\mathbb{F}_q[x]$ 51
5.1.1 Addition and Subtraction 51
5.1.2 Multiplication 51
5.1.3 Squaring 53
5.1.4 Euclidean Division 53
5.2 Representation of Finite Field Extensions 54
5.2.1 Polynomial Representation 55
5.2.2 Representation using Normal Bases 55
5.3 Arithmetic in Binary Fields 55
5.3.1 Boolean Arithmetic 56
5.3.2 Irreducibility Conditions of Binary Polynomials 56
5.3.3 Modular Reduction by Sparse Polynomials over $\mathbb{F}_2$ 57
5.3.4 Modular Reduction by Cyclotomic Polynomials over $\mathbb{F}_2$ 58
5.4 Advanced Arithmetic in $\mathbb{F}_q[x]$ 59
5.4.1 Extended Euclidean Algorithm 59
5.4.2 Modular Inversion 61
5.4.3 Univariate Evaluation and Modular Composition 61
5.4.4 Multipoint Evaluation of Univariate Polynomial Systems 63
5.4.5 Frobenius Map 64
5.4.6 Frobenius Trace 66
5.4.7 Split Root Finding in Characteristic Two 68
5.4.8 Root Finding 70

6 Software Implementation 71
6.1 Hardware Considerations 71
6.1.1 Processor 71
6.1.2 Memory Cache 72
6.1.3 Vector Instructions 73
6.1.4 Compiler Flags 75
6.2 Experimental Platform and Tools 76
6.2.1 Library and Software 76
6.2.2 Platform and Benchmarking Methodology 77
6.3 Constant-Time Implementation 77
6.3.1 Variable-Time Instructions 77

6.3.2 Constant-Time Conditional Statements . . . 78
6.3.3 Constant-Time Use of Tables . . . 79

II Main Contributions 81

7 GeMSS – a Great Multivariate Short Signature 83
7.1 General Algorithm Specification (Round 3) . . . 83
7.1.1 Parameter Space . . . 83
7.1.2 Secret-Key and Public-Key . . . 84
7.1.3 Signing Process . . . 86
7.1.4 Verification Process . . . 87
7.2 List of Parameter Sets . . . 87
7.2.1 Parameter Sets for a Security of 2^{128} (Level I) . . . 88
7.2.2 Parameter Sets for a Security of 2^{192} (Level III) . . . 88
7.2.3 Parameter Sets for a Security of 2^{256} (Level V) . . . 89
7.3 Design Rationale . . . 89
7.4 Implementation . . . 90
7.4.1 Data Representation . . . 90
7.4.2 Representation of the Secret-Key . . . 90
7.4.3 Generating Invertible Matrices . . . 91
7.4.4 Constant-Time Gaussian Elimination . . . 92
7.4.5 Generating HFEv Polynomials . . . 93
7.4.6 Public-Key Generation via Quadratic Forms . . . 93
7.4.7 Public-Key Generation by Evaluation-Interpolation . . . 93
7.4.8 Packed Representation of the Public-Key . . . 94
7.4.9 Parallel Arithmetic in $\mathbb{F}_2$, $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ . . . 96
7.4.10 Choice of the Field Polynomial for $\mathbb{F}_{2^{d_{ext}}}$ . . . 98
7.4.11 Constant-Time GCD of Polynomials . . . 98
7.4.12 Constant-Time Root Finding . . . 98
7.4.13 About the Use of Hash Functions . . . 99
7.5 Detailed Performance Analysis . . . 99
7.5.1 Experimental Platform . . . 99
7.5.2 Third-Party Open Source Library . . . 99
7.5.3 Reference Implementation . . . 100
7.5.4 Optimized (Haswell) Implementation . . . 101
7.5.5 Additional (Skylake) Implementation . . . 102
7.5.6 MQsoft . . . 103
7.5.7 Space . . . 105
7.5.8 How Parameters Affect Performance . . . 105
7.6 Expected Strength in General . . . 105
7.6.1 Number of Iterations . . . 106
7.6.2 Existential Unforgeability against Chosen Message Attack . . . 106
7.6.3 Signature Failure . . . 108
7.7 Security . . . 108
7.7.1 Minimum Number of Equations . . . 109

7.7.2 Trade-Off between Number of Equations and Number of Iterations 110
7.7.3 Experimental Results for HFEv- 113
7.7.4 Minimum Number of Vinegar Variables 116
7.7.5 Choice of Degree and Number of Modifiers 116
7.8 Design 118
7.8.1 Set 1 of Parameters: GeMSS 119
7.8.2 Set 2 of Parameters: RedGeMSS 119
7.8.3 Set 3 of Parameters: BlueGeMSS 119
7.8.4 Set 4 of parameters: WhiteGeMSS 120
7.8.5 Set 5 of parameters: MagentaGeMSS 120
7.8.6 Set 6 of parameters: CyanGeMSS 120
7.8.7 A Family of Parameters for Low-End Devices 120
7.8.8 FGeMSS(d_{ext}) 121
7.8.9 SparseGeMSS 122
7.8.10 An Exhaustive Table for the Choice of the Parameters 122
7.9 Advantages and Limitations 126
7.10 MI-Based Cryptography in the NIST PQC Standardization Process 126

8 DualModeMS – a Dual Mode for Multivariate-Based Signatures 131
8.1 General Algorithm Specification 131
8.1.1 Parameter Space 131
8.1.2 Cryptographic Operations 132
8.1.3 Implementation 132
8.2 List of Parameter Sets 133
8.2.1 Parameter Sets for a Security of 2^{128} (Level I) 133
8.2.2 Parameter Sets for a Security of 2^{192} (Level III) 133
8.2.3 Parameter Sets for a Security of 2^{256} (Level V), Version 1 133
8.2.4 Corrected Parameter Sets for a Security of 2^{256} (Level V) 133
8.3 Design Rationale 134
8.4 Detailed Performance Analysis 134
8.4.1 Time 134
8.4.2 Time (Updated) 135
8.4.3 Time (Final Version) 135
8.4.4 Space 135
8.5 Security and Selection of Parameters 136
8.5.1 Existential Unforgeability against Chosen Message Attack 136
8.5.2 Approximate PoSSo and Selection of Parameters 137
8.5.3 Safe Extension of the Dual Mode for Vulnerable Inner Layers 138
8.5.4 Minimizing the Size Public-Key Plus Signature 140
8.5.5 Smaller Signatures 141
8.6 Design 142
8.6.1 RedDualModeMS 142
8.6.2 Dual GeMSS 143
8.6.3 Dual Rainbow 143
8.6.4 Performance with a Smaller Secret-Key 144
8.7 Comparison of DualModeMS to Other Signature Schemes 144

8.8 Advantages and Limitations 145

9 MQsoft – a Fast Multivariate Cryptography Library 147
9.1 Data Structure 151
9.2 Arithmetic in $\mathbb{F}_{2^{d_{ext}}}$ 152
9.2.1 Polynomial Squaring over $\mathbb{F}_2$ 153
9.2.2 Polynomial Multiplication over $\mathbb{F}_2$ 154
9.2.3 Modular Reduction by Trinomials over $\mathbb{F}_2$ and Field Product 156
9.2.4 Modular Reduction by Pentanomials over $\mathbb{F}_2$ 161
9.2.5 Multi-Squaring in $\mathbb{F}_{2^{d_{ext}}}$ 162
9.2.6 Modular Inverse in $\mathbb{F}^{\times}_{2^{d_{ext}}}$ 163
9.2.7 Performance of the Arithmetic in $\mathbb{F}_{2^{d_{ext}}}$ 164
9.3 Efficient Implementation of Root Finding over $\mathbb{F}_{2^{d_{ext}}}$ 166
9.3.1 Polynomial Squaring over $\mathbb{F}_{2^{d_{ext}}}$ 166
9.3.2 Polynomial Multiplication over $\mathbb{F}_{2^{d_{ext}}}$ 167
9.3.3 Polynomial Euclidean Division over $\mathbb{F}_{q^{d_{ext}}}$ and Sparse Divisors 168
9.3.4 Modular Composition of Polynomials over $\mathbb{F}_{2^{d_{ext}}}$ 174
9.3.5 Frobenius Map in $\mathbb{F}_{2^{d_{ext}}}[X]$ 174
9.3.6 Greatest Common Divisor of Polynomials over $\mathbb{F}_{2^{d_{ext}}}$ 176
9.3.7 Performance of the Root Finding Algorithm over $\mathbb{F}_{2^{d_{ext}}}$ 177
9.4 Generation and Evaluation of MQ Systems 178
9.4.1 Generating the Components of a HFEv Polynomial 178
9.4.2 Evaluation of a MQ Polynomial over $\mathbb{F}_2$ 180
9.4.3 Variable-Time Evaluation of MQ Systems over $\mathbb{F}_q$ 182
9.4.4 Implementing an Efficient Evaluation of MQ Systems over $\mathbb{F}_2$ 183
9.4.5 Multipoint Evaluation of a MQ Polynomial 186
9.5 Multipoint Evaluation of Univariate Polynomial Systems 189
9.5.1 Multiplication in $\mathbb{F}_q$ and Accumulators 189
9.5.2 Structured Evaluation Point Set 190
9.5.3 Random Evaluation Point Set 190
9.6 Performance of MQsoft (Final Version) 192
9.6.1 Detailed Performance of HFE-Based Keypair Generation 192
9.6.2 Performance of HFE-Based Schemes 193
9.6.3 Performance of the Dual Mode 193

10 Approximate PoSSo 197
10.1 Reduction from APoSSo to Generalized MinRank 197
10.2 Double Reduction between APoSSo and PoSSo 202
10.3 Experimental Attacks on APoSSo 205

Conclusion 209

Bibliography 213

Appendices 229

A Size of MI-Based NIST Candidates 231

B More Algorithms in $\mathbb{F}_q[x]$ 235
B.1 Karatsuba-Like Formulae 235
B.2 Euclidean Division without Computing the Remainder 236
B.3 Newton Iteration 236
B.4 FFT Variant of the Polynomial Evaluation 237
B.5 Frobenius Map, Right-to-Left Version 238
B.6 Structured Exponentiation and Frobenius Norm 238
B.7 Degree-Two Split Root Finding in Characteristic Two 240
B.8 Constant-Time GCD for Berlekamp's Algorithm 244
B.9 List of Irreducible Polynomials over $\mathbb{F}_2$ 246
B.10 Proof of Lemma 9 246

C Addition Chains for the ITMIA 247

Abstract 250

List of Acronyms

AOP	All One Polynomial
AVX	Advanced Vector Extensions
CHES	Cryptographic Hardware and Embedded Systems
CPI	Cycle Per Instruction
CPU	Central Processing Unit
DGA	Direction Générale de l'Armement
ECC	Elliptic Curve Cryptography
ESP	Equally Spaced Polynomial
EUF-CMA	Existential UnForgeability against Chosen Message Attack
FFT	Fast Fourier Transform
GB	gigabytes
GCC	GNU Compiler Collection
GCD	Greatest Common Divisor
GNU	GNU's Not Unix!
HP$\mathbb{F}$A	High-Performance Finite Field Arithmetic
kB	kilobytes
KiB	kibibytes
LTS	Long Term Support
MB	megabytes
MiB	mebibytes
MQ	Multivariate Quadratic
NESSIE	New European Schemes for Signature, Integrity and Encryption
NIST	National Institute of Standards and Technology
NP	Non-deterministic Polynomial time
OS	Operating System
PhD	Philosophiae Doctor
PIT	Polynomial Identity Testing
PQC	Post-Quantum Cryptography
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
SBP	Szepieniec–Beullens–Preneel
SIMD	Single Instruction on Multiple Data
SSE	Streaming SIMD Extensions
SSSE	Supplemental Streaming SIMD Extensions
SWAR	SIMD Within A Register

AES	Advanced Encryption Standard
AMQ	Approximate MQ
APoSSo	ApproximatePoSSo
EIP	Extended Isomorphism of Polynomials
FGeMSS	Family GeMSS
FLINT	Fast Library for Number Theory
GeMSS	Great Multivariate Short Signature
GMP	GNU Multiple Precision Arithmetic Library
GMR	Generalized MinRank
HFE	Hidden Field Equations
HTTPS	HyperText Transfer Protocol Secure
HW	Hamming weight
ITMIA	Itoh–Tsujii Multiplicative Inversion Algorithm
LUOV	Lifted Unbalanced Oil and Vinegar
MAC	Message Authentication Code
MI	Matsumoto–Imai
MQDSS	MQ Digital Signature Scheme
NTL	Number Theory Library
PKP-DSS	Permuted Kernel Problem – Digital Signature Scheme
PoSSo	Polynomial System Solving
RSA	Rivest–Shamir–Adleman
SHA	Secure Hash Algorithm
SHAKE	Secure Hash Algorithm Keccak
SRA	Successive Resultant Algorithm
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UOV	Unbalanced Oil and Vinegar
XKCP	eXtended Keccak Code Package
XL	eXtended Linearization

List of Figures

1.1 Dependencies between the chapters. . . . 4
1.2 (Figure 9.1) Dependencies between the different operations performed in MQsoft. . . . 7

3.1 Merkle tree. . . . 26
3.2 Security model of the dual mode. . . . 30

5.1 Dependencies between the different arithmetic operations in characteristic two. . . . 50

7.1 Example of hybrid representation of a MQ system of 10 equations in 3 variables. . . . 95

9.1 Dependencies between the different operations performed in MQsoft. . . . 149
9.2 Example of (truncated) matrix $\mathbf{F} \in \mathcal{M}_5(\mathbb{F}_{2^{d_{\mathrm{ext}}}})$ for $D = 18$ and $s = 3$. . . . 173

10.1 Degree of regularity of $\mathtt{DPoSSo}(2, r+1, m, m, 2, r)$ in function of $m - r$. . . . 205
10.2 Degree of regularity of $\mathtt{DPoSSo}(256, r+1, 2m, m, 2, r)$ in function of $m - r$. . . . 206
10.3 Degree of regularity of $\mathtt{DPoSSo}(256, r+1, 17, 16, 2, r)$ in function of r. . . . 207

List of Tables

1.1 Performance of GeMSS128 at the first round of the NIST PQC standardization process, followed by the speed-up between the other rounds and the first round. 8
1.2 Performance of RedGeMSS128 at the second round of the NIST PQC standardization process, followed by the speed-up between the other rounds and the second round. . 8
1.3 Performance of BlueGeMSS256 at the second round of the NIST PQC standardization process, followed by the speed-up between the other rounds and the second round. . 8
1.4 (Table 9.27) Number of megacycles for each cryptographic operation with our library for a Skylake processor (LaptopS), followed by the speed-up between the best implementation provided for the NIST submissions (Table 7.37) versus our implementation. 9
1.5 (Table 8.3) Number of megacycles for each cryptographic operation of DualModeMS with our implementation, for a Skylake processor (LaptopS), followed by the speed-up between the Skylake implementation from [84] versus our implementation. 9

2.1 Theoretical probability of finding s distinct roots for a random HFE polynomial in an extension field ($\leqslant D$), followed by experimental tests. 20

3.1 Comparison of the inner and outer layers of HFE-based schemes, as proposed in [157, 25]. 33

4.1 NIST security strength categories. 36
4.2 Estimation of NIST security strength categories. 36
4.3 Degree of regularity of m semi-regular quadratic boolean equations in n variables. . 41
4.4 Degree of regularity in the case of HFE algebraic systems. 41

5.1 Complexity of main operations on degree-d elements of $\mathbb{F}_{q^{d_{ext}}}[x]$. 50
5.2 Main operations in $\mathbb{F}_2$. 56

6.1 Cost of vector instructions on Intel processors. 75
6.2 Processors. 77
6.3 OS and memory. 77

7.1 Performance of GeMSS for a 128-bit security level, with MQsoft. 88
7.2 Performance of GeMSS for a 192-bit security level, with MQsoft. 88
7.3 Performance of GeMSS for a 256-bit security level, with MQsoft. 89
7.4 Irreducible trinomials $x^{d_{ext}} + x^{k_1} + 1$ defining $\mathbb{F}_{2^{d_{ext}}}$ for GeMSS. 98
7.5 Performance of the reference implementation. We use a Skylake processor (LaptopS). 100
7.6 Performance of the optimized implementation. We use a Haswell processor (ServerH). 101

7.7 Performance of the additional implementation. We use a Skylake processor (LaptopS). 102
7.8 Performance of MQsoft. We use a Skylake processor (LaptopS). 103
7.9 Performance of MQsoft. We use a Haswell processor (ServerH). 104
7.10 Memory cost of GeMSS. 105
7.11 Complexity of solving a multivariate quadratic system of m equations in m variables in $\mathbb{F}_2$, with the exhaustive search. 109
7.12 Lower bound on the complexity of solving a multivariate quadratic system of m equations in m variables in $\mathbb{F}_2$, with asymptotically fast algorithms. 109
7.13 Lower bound on the complexity of finding a collision with a generic attack (Lemma 4). 111
7.14 Minimum ratio of the cost of evaluating a boolean system of m equations in m variables by that of SHA-3. 112
7.15 Lower bound on the complexity of finding a collision with the generic attack (Equation (7.4)). 113
7.16 HFE- with $D = 4$; 32 and 41 equations. 114
7.17 HFE- with $D = 17$; 32 and 41 equations. 114
7.18 HFEv with $D = 6$ and 32 equations. 115
7.19 HFEv with $D = 9$ and 25 equations. 115
7.20 HFEv with $D = 16$; 25 and 32 equations. 115
7.21 Values of and v which achieve the security levels against MinRank-based attacks. . 116
7.22 Values of v which achieve the security levels against MinRank-based attacks, for $= 0$. 116
7.23 Lower bound on the complexity of the Support Minors technique against parameters of round 2 proposals of GeMSS. 117
7.24 Smallest degree of regularity required ($= 2$). 117
7.25 Number of modifiers required in GeMSS. 118
7.26 Summary of the parameters of GeMSS. 119
7.27 Summary of the parameters of RedGeMSS. 119
7.28 Summary of the parameters of BlueGeMSS. 119
7.29 Summary of the parameters of WhiteGeMSS. 120
7.30 Summary of the parameters of MagentaGeMSS. 120
7.31 Summary of the parameters of CyanGeMSS. 120
7.32 Slight modification of GeMSS for low-end devices. 121
7.33 Parameters of FGeMSS. 121
7.34 Performance of an exhaustive set of security parameters achieving the level I, with MQsoft. 123
7.35 Performance of an exhaustive set of security parameters achieving the level III, with MQsoft. 124
7.36 Performance of an exhaustive set of security parameters achieving the level V, with MQsoft. 125
7.37 Performance in megacycles of GeMSS and Gui best implementations submitted to the first round of the NIST PQC standardization process. 127
7.38 Size of the keys and signature of the HFEv--based schemes submitted to the NIST PQC standardization process, as well as QUARTZ from the NESSIE project [145]. . . 128
7.39 Size of the keys and signature of Rainbow submitted to the NIST PQC standardization process. 129

8.1 Performance of DualModeMS128 at the first round of the NIST PQC standardization process. . . . 134
8.2 Number of megacycles for each cryptographic operation of DualModeMS with the version of MQsoft used in [84]. . . . 135
8.3 Number of megacycles for each cryptographic operation of DualModeMS with our implementation, for a Skylake processor (LaptopS), followed by the speed-up between the Skylake implementation from [84] versus our implementation. . . . 135
8.4 Memory cost of DualModeMS. . . . 136
8.5 Size of the keys and signature of the inner and dual modes. . . . 142
8.6 Size of the keys and signature of the dual mode of BlueGeMSS and RedGeMSS. . . . 143
8.7 Size of the keys and signature of the dual mode of Rainbow. . . . 143
8.8 Performance of the dual mode in function of 144
8.9 Comparison of the dual mode to the second round candidates (except PKP-DSS [23]), for a 128-bit security level. . . . 145

9.1 (Table 9.27) Number of megacycles for each cryptographic operation with our library for a Skylake processor (LaptopS), followed by the speed-up between the best implementation provided for the NIST submissions (Table 7.37) versus our implementation. 149
9.2 Performance of the PCLMULQDQ instruction in function of the architecture, as presented in the Intel Intrinsics Guide. . . . 153
9.3 Number of cycles for computing the square of an element of $\mathbb{F}_2[x]$ of degree $d_{\text{ext}} - 1$, with MQsoft. . . . 154
9.4 Number of cycles to multiply two elements of $\mathbb{F}_2[x]$ of degree $d_{\text{ext}} - 1$. . . . 156
9.5 Number of cycles to compute the modular reduction of an element of $\mathbb{F}_2[x]$ of degree $2d_{\text{ext}} - 2$ by $x^{d_{\text{ext}}} + x^{k_1} + 1$, with MQsoft. . . . 158
9.6 Number of cycles to compute the multiplication in $\mathbb{F}_{2^{d_{\text{ext}}}}$ in function of the modular reduction, with MQsoft. . . . 158
9.7 Number of cycles to compute the squaring in $\mathbb{F}_{2^{d_{\text{ext}}}}$ in function of the enabled instructions, with MQsoft. We use a Skylake processor (LaptopS). . . . 159
9.8 Number of cycles to compute the squaring in $\mathbb{F}_{2^{d_{\text{ext}}}}$ in function of the enabled instructions, with MQsoft. We use a Haswell processor (ServerH). . . . 159
9.9 Number of cycles to compute the modular reduction of a square, the squaring in $\mathbb{F}_{2^{d_{\text{ext}}}}$ and the inverse in $\mathbb{F}_{2^{d_{\text{ext}}}}^{\times}$, in function of d_{ext} and k_1, with MQsoft. . . . 160
9.10 Number of cycles to compute the modular reduction of an element of $\mathbb{F}_2[x]$ of degree $2d_{\text{ext}} - 2$ by $x^{d_{\text{ext}}} + x^{k_3} + x^{k_2} + x^{k_1} + 1$ or $x^{d_{\text{ext}}} + x^{k_1} + 1$, with MQsoft and the SSSE3 instruction set. . . . 162
9.11 Number of cycles by operation in $\mathbb{F}_{2^{d_{\text{ext}}}}$. We use a Skylake processor (LaptopS). . . . 165
9.12 Number of cycles by operation in $\mathbb{F}_{2^{d_{\text{ext}}}}$, for [30] (Haswell Core i7-4770 CPU at 3.4 GHz) and MQsoft (LaptopS). . . . 165
9.13 Number of megacycles to compute the multiplication of two coefficient polynomials over $\mathbb{F}_{2^{d_{\text{ext}}}}$. We use a Skylake processor (LaptopS). . . . 168
9.14 Impact of $d = \mathcal{D}(H)$ on both the cost of Algorithm 37 and $D_{\text{reg}}^{\text{Exp}}$ the degree of regularity of the corresponding HFE algebraic system of 30 equations in 30 variables over $\mathbb{F}_2$. . . . 172
9.15 Degree of regularity in the case of HFE algebraic systems of m equations in m variables over $\mathbb{F}_2$, in function of s. . . . 172

9.16 Number of megacycles to compute the remainder of the Euclidean division of a degree-$(2D-2)$ polynomial by a degree-D monic polynomial over $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS). 174
9.17 Number of megacycles to compute the modular composition of D-coefficient polynomials modulo a monic HFE polynomial of degree D over $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS). 175
9.18 Number of megacycles to compute the Frobenius map modulo a HFE polynomial over $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS). 176
9.19 Number of megacycles to compute the GCD of two polynomials of degree D and $D-1$ over $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS). 176
9.20 Number of megacycles to find the roots of a HFE polynomial of degree D over $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS). 177
9.21 Number of kilocycles to evaluate a MQ system of m equations in m variables over $\mathbb{F}_2$. We use a Haswell processor (ServerH) with the AVX2 instruction set. 186
9.22 Number of kilocycles to evaluate a MQ system of m equations in m variables over $\mathbb{F}_2$. We use a Skylake processor (DesktopS) with the AVX2 instruction set. 186
9.23 Performance of the evaluation of 1 MQ equation in $\mathbb{F}_q[x_1,\dots,x_{n_{var}}]$ in points. 188
9.24 Performance of the evaluation of a univariate polynomial of degree D_{MAC} over $\mathbb{F}_{2^{\ell}}$ in points. 190
9.25 Performance of the evaluation of univariate polynomial systems of degree D_{MAC} over $\mathbb{F}_{2^{\ell}}$. 192
9.26 Number of megacycles for main steps of the keypair generation with our library. We use a Skylake processor (LaptopS). 192
9.27 Number of megacycles for each cryptographic operation with our library for a Skylake processor (LaptopS), followed by the speed-up between the best implementation provided for the NIST submissions (Table 7.37) versus our implementation. 193
9.28 Performance of the inner mode in megacycles. We use a Skylake processor (LaptopS), with PCLMULQDQ and the AVX2 instruction set. 194
9.29 Performance of the dual mode in megacycles. We use a Skylake processor (LaptopS), with PCLMULQDQ and the AVX2 instruction set. 195
9.30 Performance of the dual mode in megacycles. We use a Haswell processor (ServerH), with PCLMULQDQ and the AVX2 instruction set. 195

10.1 Practical dimension of APoSSo over $\mathbb{F}_{65521}$, for $D=2$. 201
10.2 Practical dimension of APoSSo over $\mathbb{F}_{65521}$, for $D=2$. 201

A.1 Exact size of the keys and signature of the HFEv--based schemes submitted to the NIST PQC standardization process, as well as QUARTZ from the NESSIE project. . . 233
A.2 Exact size of the keys and signature of Rainbow submitted to the NIST PQC standardization process. 234

C.1 Proposed addition chains to minimize the number of multiplications in $\mathbb{F}_{2^{d_{ext}}}$. 247

List of Algorithms

1 Keypair generation of MI-based schemes. 16
2 Inverse map of the public-key. 17
3 Signing process of MI-based schemes. 17
4 Verifying process of MI-based schemes. 17
5 Construction of MAC polynomial. 24
6 Keypair generation in the dual mode. 25
7 Signing process in the dual mode. 25
8 Generation of 2 Merkle trees as a truncated Merkle tree. 27
9 Generation of an authentication path. 27
10 Verification of a leaf via its authentication path and the roots. 28
11 Verifying process in the dual mode. 28
12 MI-based signature process using the Feistel–Patarin construction. 37
13 MI-based verification process using the Feistel–Patarin construction. 38
14 Left-to-right square-and-multiply exponentiation. 47
15 Constant-time left-to-right square-and-multiply exponentiation. 47
16 Polynomial Euclidean division with remainder. 53
17 Fast Euclidean division with remainder. 54
18 Traditional Euclidean algorithm. 59
19 Traditional extended Euclidean algorithm. 60
20 Polynomial evaluation using the baby-step giant-step approach. 62
21 Frobenius map using the left-to-right square-and-multiply algorithm on q^k. 64
22 Frobenius map using a multi-squaring table. 65
23 Frobenius trace using the left-to-right square-and-multiply algorithm on $k-1$. . . . 67
24 Algorithm to find the roots of a split and squarefree monic univariate polynomial in characteristic two. 68
25 Algorithm to find the roots of a univariate polynomial. 70
26 Variable-time conditional move from a to c, with x a boolean. 78
27 Constant-time conditional swap. 79
28 Constant-time access to the j-th element from a table T of elements. 79
29 Keypair generation in GeMSS. 85
30 Inversion in GeMSS. 86
31 Signing process in GeMSS. 87
32 Verifying process in GeMSS. 87
33 Constant-time Gaussian elimination on the rows of a matrix M_n in $\mathbb{F}_2$. 92
34 Modified inverse map of the public-key for HFEv−-based signature schemes. 107

35 Generic attack against the Feistel–Patarin construction. 110
36 ITMIA for a specific addition chain. 163
37 Polynomial Euclidean division of A by a degree-D HFE polynomial over $\mathbb{F}_{q^{d_{\mathrm{ext}}}}$. 169
38 Classical 64-bit implementation in C programming language of the SWAR algorithm. 181
39 Variable-time evaluation of a MQ system of m equations in $n_{\mathtt{var}}$ variables over $\mathbb{F}_2$. . 182
40 Variable-time evaluation of a MQ system over $\mathbb{F}_2$ using the differential trick. 183
41 Constant-time evaluation of a MQ system of m equations in $n_{\mathtt{var}}$ variables over $\mathbb{F}_2$. . 184
42 Improvement of Algorithm 41 with AVX2, VPMASKMOVQ and VPBROADCASTQ. 185
43 Improvement of Algorithm 41 with AVX2, VPMASKMOVQ and VPERMQ. 185
44 Multipoint evaluation of one MQ polynomial. 188
45 Horner precomputation (with a step of 4). 191
46 Horner by block for the evaluation of a degree-d univariate polynomial G in a. 191
47 Polynomial Euclidean division without computing the remainder. 236
48 Newton iteration. 236
49 Polynomial evaluation using the FFT decomposition. 237
50 Frobenius map using the right-to-left square-and-multiply algorithm on φ^k. 238
51 Itoh–Tsujii exponentiation for a specific (left-to-right) addition chain. 239
52 ITMIA for a specific (left-to-right) addition chain. 240
53 Constant-time GCD of F and H, where d_h is public and d_f is a public upper bound on $\deg(F)$. 245

List of Definitions and Examples

Definition 1 NP-completeness 13
Definition 2 Equivalent keys 44
Definition 3 EUF-CMA security of a signature scheme in the random oracle model . . 106
Definition 4 EUF-CMA security of a MI-based function generator 106

Problem 1 Polynomial System Solving 14
Problem 2 Extended Isomorphism of Polynomials 14
Problem 3 Approximate MQ 29
Problem 4 Approximate PoSSo 43
Problem 5 MinRank 45
Problem 6 Approximate PoSSo, matrix version 197
Problem 7 Generalized MinRank 198

Example 1 Storing a binary polynomial 152
Example 2 Storing a dense polynomial 152
Example 3 Storing a HFE polynomial 152
Example 4 Square of a binary polynomial 153
Example 5 Double product via PCLMULQDQ 189
Example 6 Cubic root in $\mathbb{F}_{2^{d_{ext}}}$ 240

List of Theorems

Theorem 1 EUF-CMA security of the outer layer for $= 1$ [157] 29
Theorem 2 EUF-CMA security of the outer layer [157] 29
Theorem 3 Irreducibility conditions of an ESP [109, Theorem 3] 58
Theorem 4 Complexity of the root finding algorithm [161, Corollary 14.16] 70
Theorem 5 Choosing the number of iterations of the Feistel–Patarin construction [58] . 106
Theorem 6 EUF-CMA security of the modified HFEv--based signature scheme [148] . . 107
Theorem 7 Classical and quantum EUF-CMA security of the outer layer for $= 1$ . . 136
Theorem 8 Classical and quantum EUF-CMA security of the outer layer [157, 25] . . . 137
Theorem 9 Generic attack of signatures among signatures 138
Theorem 10 Faster classical Euclidean division of a square by a sparse HFE polynomial . 170
Theorem 11 $\mathtt{APoSSo}(q, , m, n_{var}, D, r) \leqslant \mathtt{GMR}(q, , m, \cdot n_{var}, D, r)$ 198
Theorem 12 Dimension of APoSSo (with an extra assumption) 198
Theorem 13 $\mathtt{APoSSo}(q, , m, n_{var}, D, r) \leqslant \mathtt{PoSSo}(q, m - r, n_{var}, D)$ 202
Theorem 14 $\mathtt{PoSSo}(q, m - r, n_{var}, D) \leqslant \mathtt{APoSSo}(q, r + 1, m, n_{var}, D, r)$ 203
Theorem 15 $\mathtt{APoSSo}(q, , m, n_{var}, D, r) \equiv \mathtt{PoSSo}(q, m - r, n_{var}, D)$ 204

Corollary 1 Safe MI-based outer layer for ultra-short inner signatures 139
Corollary 2 Dimension of APoSSo 200
Corollary 3 Reduction from APoSSo to the minus variant of PoSSo 203

Lemma 1 Irreducibility conditions of an AOP [109, Lemma 1] 58
Lemma 2 Cost of generating a multi-squaring table 65
Lemma 3 Cost of the Frobenius map with a precomputed table 66
Lemma 4 Complexity of the generic attack against the Feistel–Patarin construction . . 110
Lemma 5 minimizing C_G in the lower bound (7.4) of Lemma 4 113
Lemma 6 minimizing the size public-key plus signature of the outer layer. 140
Lemma 7 Maximum number of nodes required to verify the Merkle leaves 141
Lemma 8 Impact of the odd degree terms of the divisor on the quotient 170
Lemma 9 Impact of the even degree terms of the divisor on the quotient 173
Lemma 10 APoSSo is in NP 204
Lemma 11 Index of $\mathbb{F}_2$-linearly dependent columns of M_2 for irreducible trinomials . . . 241

List of Remarks

Remark 1 . . . 57
Remark 2 . . . 58
Remark 3 . . . 79
Remark 4 . . . 84
Remark 5 . . . 86
Remark 6 . . . 91
Remark 7 . . . 94
Remark 8 . . . 94
Remark 9 . . . 94
Remark 10 . . . 95
Remark 11 . . . 97
Remark 12 . . . 99
Remark 13 . . . 108
Remark 14 . . . 108
Remark 15 . . . 133
Remark 16 . . . 144
Remark 17 . . . 180
Remark 18 . . . 187
Remark 19 . . . 189
Remark 20 . . . 195
Remark 21 . . . 203
Remark 22 . . . 203
Remark 23 . . . 237
Remark 24 . . . 244
Remark 25 . . . 245
Remark 26 . . . 245

Chapter 1

Introduction

In a world where Internet is omnipresent, the security of communications is a fundamental stake. The rise of quantum computers has shaken the modern public-key cryptography, which is based on the integer factorization problem [147] or discrete logarithm problem [149]. These problems will be solved with Shor's algorithm [152] when quantum computers are powerful enough. As a consequence, the Transport Layer Security (TLS) cryptographic protocol will become insecure. TLS is widely used nowadays, via the HTTPS protocol. To prevent this danger, National Institute of Standards and Technology (NIST) started a Post-Quantum Cryptography (PQC) standardization process. This is the first public-key cryptography standardization process for more than thirty years. In December 2016, NIST called for proposals [127]. It resulted that eighty-two proposals were submitted to the PQC standardization process on November 30, 2017. Among these proposals, sixty-nine candidates met the minimum acceptance criteria considered by NIST. These candidates were published on the NIST webpage on December 20, 2017 [128]. This was the beginning of the first round of the NIST PQC standardization process. Candidates were divided into two groups: forty-nine public-key encryption schemes and twenty digital signature schemes. With my PhD advisors, we have submitted two digital signature schemes to the PQC standardization process: GeMSS, a Great Multivariate Short Signature [48] (Chapter 7), and DualModeMS, a Dual Mode for Multivariate-Based Signature [83] (Chapter 8). The GeMSS team includes other members: A. Casanova, G. Macario-Rat and J. Patarin. GeMSS is a multivariate-based signature scheme (Chapter 2). The public-key is a boolean multivariate quadratic system, following the hash-and-sign paradigm. The verification of a signed document consists in evaluating this system in the signature, and verifying if this signature is equal to the hash value (or digest) of the signed document. The signing process consists in signing the hash value of the document. GeMSS is a HFEv--based signature scheme (Section 2.4.1). This means that the signature is obtained as the root of a secret univariate polynomial in an extension field ($\mathbb{F}_{2^{d_{\mathrm{ext}}}}$). Another HFEv--based signature scheme, Gui [62], was proposed to the PQC standardization process. Both schemes are similar and vary according to the selected security parameters. NIST selected GeMSS to move to the second round. Other multivariate-based signature schemes were also proposed. LUOV (Lifted Unbalanced Oil and Vinegar) [26] (Section 2.4.2) and Rainbow [63] (Section 2.4.3) are schemes rather similar to GeMSS. The fundamental difference is in the signing process, which consists in solving a secret linear system. The LUOV signature scheme was broken during the second round [66], because of the lifted character of this scheme, and so did not move to the third round. These four schemes share a

classical property of multivariate-based schemes. The public-key is rather large, because the latter is a multivariate quadratic system, whereas signatures are the shortest of the NIST PQC standardization process. In DualModeMS, the dual mode of GeMSS, we propose to invert this trade-off. The dual mode [157, 25] is a compression technique of public-key (Chapter 3). It is based on a proven secure dual transformation, which allows to decrease the public-key size. On the other hand, the signature size is large, but the sum of both sizes is smaller than the original. The technique used in DualModeMS was very recent (July 2017 for a submission end of November). So, we proposed the first practical implementation, based on HFEv--based signature schemes. The latter was very challenging. The keypair generation was very long, with a time of seven hundred seconds, whereas the signing process takes at least several seconds. Moreover, the secret-key size is about eighteen megabytes. All these drawbacks explain that DualModeMS did not move to the second round of the NIST PQC standardization process. We propose drastic improvements of the dual mode in the thesis (Chapter 8), with in particular the dual mode of Rainbow, so-called Dual Rainbow. The latter has a very interesting trade-off between memory and practical performance. Finally, we mention a last kind of cryptography multivariate: these which are based on the Fiat–Shamir transform [85]. MQDSS (MQ Digital Signature Scheme) [54] is a scheme of this kind. This signature size is larger than traditional multivariate schemes, but the public-key is much shorter. NIST chose to move MQDSS to the second round, but not to the third round. The latter knew new attacks, implying to increase security parameters. This decreased performance, and made it uncompetitive.

Among the sixty-nine candidates, five were withdrawn during the first round. On January 30, 2019, twenty-six candidates moved to the second round of the NIST PQC standardization process [129]. Among these candidates, nine are digital signature schemes, and four are multivariate-based (GeMSS, LUOV, Rainbow, MQDSS). NIST proposed to submit improvements and tweaks of parameters, until April 1st, 2019. To respond to the comments of NIST [2], we proposed the creation of RedGeMSS and BlueGeMSS [49]. These are faster variants of GeMSS. The Rainbow team proposed the creation of cyclic Rainbow and compressed Rainbow [64]. In cyclic Rainbow, a part of the public-key is generated from a seed. This implies a smaller public-key size. Then, the compressed version allows to generate a part of the secret-key from a seed. Instead of storing the secret-key, the seed is used to generate the secret-key again during each signature generation. We have been using this idea since April 15, 2020, last deadline to submit improvements before the selection for the third round. With this change, GeMSS is the candidate with the smallest secret-key and signature. In return, GeMSS also has the largest public-key.

On July 22, 2020, NIST announced candidates which moved to the third round [130]. Here, two kinds of candidates were considered. Seven candidates moved as finalist candidates, including among them Rainbow [65]. They will be considered for standardization at the end of the third round. Then, eight candidates moved as alternate candidates, including among them GeMSS [50]. Alternate candidates have a potential for standardization. They are still considered for standardization, but will probably not be standardized at the end of the third round. Some of these candidates will be selected for the fourth round, to keep studying them. Both, third and fourth rounds, should be achieved in twelve or eighteen months.

On October 1st, 2020, improvements and tweaks of parameters were submitted to the third round of the NIST PQC standardization process. For GeMSS, our goal was to improve performance, in particular on low-end devices.

1.1 Previous Work

GeMSS [48], Gui [62], Rainbow [63] and LUOV [26] are MI-based signature schemes [119] (Chapter 2). The public-key is a multivariate quadratic system $\mathbf{p} \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^{m'}$ having the following form:

$$\mathbf{p} = \mathcal{T} \circ \mathcal{F} \circ \mathcal{S},$$

with $\mathcal{S}, \mathcal{T} \in \mathrm{Aff}^{-1}_{n_{\mathtt{var}}}(\mathbb{F}_q) \times \mathrm{Aff}^{-1}_{m'}(\mathbb{F}_q)$ being two invertible affine transformations, and $\mathcal{F} \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^{m'}$ being a multivariate quadratic system easily invertible thanks to a special shape. The security of these schemes rely on the hardness of the PoSSo problem.

Problem 1. Polynomial System Solving (PoSSo$(q, m, n_{\mathtt{var}}, D)$). Let $q, m, n_{\mathtt{var}}$ and D be integers. Given $\mathbf{p}$ a system of m degree-D multivariate polynomials in $\mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$, the problem is to find, if any, a vector $\mathbf{x}_s$ in $\mathbb{F}_q^{n_{\mathtt{var}}}$ such that $\mathbf{p}(\mathbf{x}_s) = (0, \ldots, 0) = \mathbf{0}_m$.

In this thesis, we focus on HFE-based signature schemes [133] (Section 2.4.1). The central map $\mathcal{F}$ is generated from a univariate polynomial F defined in a degree-$d_{\mathtt{ext}}$ extension field of $\mathbb{F}_q$. The signing process requires inverting $\mathcal{S}$ and $\mathcal{T}$, as well as inverting $\mathcal{F}$. This implies finding roots of F. The verifying process requires evaluating the public-key $\mathbf{p} \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^{m'}$, a multivariate quadratic system.

QUARTZ [134], Gui [144] and GeMSS [48] are HFE-based signature schemes. The signatures require only several hundred bits, whereas the public-key is large (several hundred kilobytes (kB)). The verifying process is very fast (several hundred nanoseconds), unlike the signing process which can take up to several seconds. The root finding step is the core of the signing process. We use Berlekamp's algorithm [161, Algorithm 14.15], which is divided into three steps.

1. Computation of the Frobenius map, i.e. $R_2 = X^{q^{d_{\mathtt{ext}}}} - X \bmod F \in \mathbb{F}_{q^{d_{\mathtt{ext}}}}[X]$.
2. Computation of $G \in \mathbb{F}_{q^{d_{\mathtt{ext}}}}[X]$ the GCD of R_2 and F. G is split and squarefree, and contains exclusively the roots of F.
3. Computation of the roots of G with a split root finding algorithm.

In [157], the authors introduced a technique to invert the trade-off between public-key size and signature size, so-called SBP transformation (Chapter 3). This technique applies to MI-based schemes such as HFE [133] and Rainbow [69]. DualModeMS [83] is the direct application of the SBP transformation to GeMSS, and was submitted to the first round of the NIST PQC standardization process [128]. We also submitted the first implementation of the SBP transformation, which was rather challenging to make. For example, the keypair generation of DualModeMS is achieved in 708 seconds for a 128-bit security level. No method is presented in [157] to implement efficiently the SBP transformation. In this thesis, we introduce methods to make an efficient dual mode, and we study the SBP transformation of HFE-based schemes and Rainbow-based schemes.

Moreover, the SBP transformation relies on the hardness of solving a new problem: Approximate MQ [157].

Problem 3. Approximate MQ ($\mathtt{AMQ}(q, \ , m, n_{\mathrm{var}}, r)$).
Input. A set of quadratic polynomials $p = (p_1, \ldots, p_m) \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathrm{var}}}]^m$, vectors $y_1, \ldots, y \ \in \mathbb{F}_q^m$ and $r \in \mathbb{N}$ such that $r < \min(m, \ )$.
Question. Find $x_1, \ldots, x \ \in \mathbb{F}_q^{n_{\mathrm{var}}}$ such that

$$\dim \ \mathrm{Vec} \ p(x_1) - y_1, \ldots, p(x \ ) - y \quad \leqslant r,$$

where Vec stands for the vector space generated by $p(x_1) - y_1, \ldots, p(x \ ) - y$.

When $r = 0$, the AMQ problem is an instance of PoSSo and so is hard to solve. When $r > 0$, the authors of [157] proposed several attacks supporting that solving AMQ should be exponential in $m - r$. In particular, an instance of AMQ with m equations can be solved as an instance of PoSSo with $m - r$ equations. However, the question of the hardness of AMQ is an open question. We will demonstrate in Chapter 10 that AMQ is hard to solve (i.e. NP-complete).

1.2 Organization of the Document

The structure of this thesis is depicted in Figure 1.1. We introduce the MI-based cryptography, as well as HFE, in Chapter 2. In Chapter 3, we present the SBP transformation applied to MI-based signature schemes. Then, the attacks against HFE-based schemes and the SBP transformation are studied in Chapter 4. In Chapter 5, we describe arithmetic in polynomial rings, whereas supplemental techniques are deferred to Appendix B. This is followed by a description of important considerations about software implementations and hardware operations (Chapter 6). In Chapters 7 and 8, we present respectively the NIST submissions GeMSS and DualModeMS. These schemes are implemented efficiently thanks to MQsoft: a fast multivariate cryptography library (Chapter 9). Finally, we study the hardness of the APoSSo problem, on which the security of DualModeMS is based (Chapter 10).

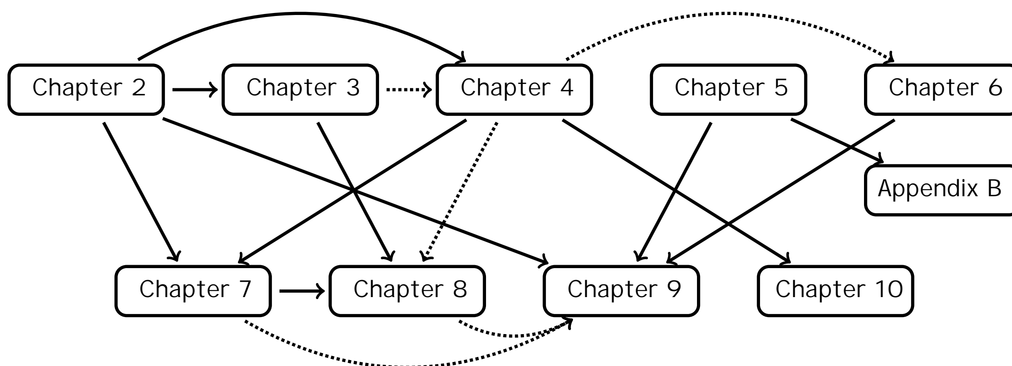

Figure 1.1: Dependencies between the chapters. The dotted arrows represent partial dependencies.

1.3 Contributions

Design and cryptanalysis. The design of secure cryptosystems such as GeMSS and DualModeMS requires studying attacks. The security of GeMSS is based on a vast state-of-the-art of twenty years of cryptanalysis on HFE (Chapter 4), including exhaustive search, direct attack and key recovery. In Section 7.7, we show how to select security parameters of GeMSS. In particular, a fundamental point is the choice of modifiers of HFE, i.e. in the case of GeMSS, the minus and vinegar modifiers. We study the impact of such modifiers on the security of HFE against Gröbner basis attacks, then we deduce an experimental rule to select them. We use the Gröbner basis solver from Magma [34] to perform these experiments. The security of DualModeMS is based on the MQ problem (Problem 3), that we generalize as APoSSo problem (Chapter 10). We introduce several new reductions. On the one hand, the APoSSo problem can be reduced to the Generalized MinRank and PoSSo problems. This allows to attack APoSSo by using known attacks against Generalized MinRank and PoSSo, i.e. Gröbner basis attacks. We also obtain the dimension of APoSSo (Corollary 2), which we permit to know the number of variables to fix to solve APoSSo in dimension zero. The Gröbner basis algorithms are more e cient in dimension zero.

Corollary 2. Let $q, m, n_{\mathtt{var}}, D,$ and $r < \min(, m)$ be integers, $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathrm{var}}}]^m$ be a system of m degree-D polynomials in $n_{\mathtt{var}}$ variables, and $p^* \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathrm{var}}}]^m$ be p without its constant terms. If the polynomials of p^* are $\mathbb{F}_q$-linearly independent, and if $\cdot n_{\mathtt{var}} \geqslant (- r)(m - r)$, then the dimension of APoSSo$(q, , m, n_{\mathtt{var}}, D, r)$ instantiated with p is $\cdot n_{\mathtt{var}} - (- r)(m - r)$.

On the other hand, we reduce PoSSo to APoSSo in polynomial time, implying an important result: APoSSo is NP-complete. The question of the hardness of APoSSo was an open question. Finally, the double reduction between PoSSo and APoSSo highlights an interesting link between these two problems. The hardness of PoSSo having m equations is similar to the hardness of APoSSo instantiated with a system of $m - r$ equations, where r is the target rank of APoSSo (Theorem 15).

Theorem 15. Let $q, m, n_{\mathtt{var}}, D,$ and $r < \min(, m)$ be integers, $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathrm{var}}}]^m$ be a system of m degree-D polynomials in $n_{\mathtt{var}}$ variables, $c \in \mathbb{F}_q^m$ be the vector of constant terms of p, and $y_1, \ldots, y \in \mathbb{F}_q^m$. If $c, y_1, \ldots, y_r$ are linearly independent, then APoSSo$(q, , m, n_{\mathtt{var}}, D, r) \equiv$ PoSSo$(q, m - r, n_{\mathtt{var}}, D)$.

The APoSSo problem can be solved as the minus variant of PoSSo. These results are confirmed by experiments using Gröbner basis.

E cient algorithms. In Chapter 5, we present a large number of state-of-the-art algorithms useful for arithmetic in $\mathbb{F}_{q^{d_{\mathrm{ext}}}}$ and univariate root finding over $\mathbb{F}_{q^{d_{\mathrm{ext}}}}$. In Chapter 9, we propose improved algorithms for the public-key generation of HFEv-, as well as for the Frobenius map which is the core of the root finding algorithm. Then, we improve the constant-time algorithm for GCD, and we propose a constant-time root finding of split polynomials.

- In Section 7.4, we propose to accelerate the public-key generation of HFE polynomials. We study two methods. The evaluation-interpolation strategy seems the most e cient on modern computers. Its e ciency is based on a fast evaluation of quadratic forms. Moreover, we improve the state-of-the-art by taking advantage of the structure of evaluation points, which have their Hamming weight less or equal to two. Then, we also improve the direct computation

of the public-key, i.e. to directly compute:

$$p(x) = (\ \circ \mathcal{T} \circ \mathcal{F} \circ \mathcal{S})(x) \in \mathbb{F}_q[x]^m.$$

In particular, we note that for HFEv-, it is faster to compute firstly $\circ \mathcal{T} \circ \mathcal{F}$, because the minus variant decreases the number of equations of the system. This computation requires the knowledge of $\mathcal{F}$, which is the multivariate representation of an HFEv polynomial. We improve this step with a smart use of matrix representations of quadratic forms.

- In Section 9.3.3, we study different methods to improve the modular reduction of polynomials over $\mathbb{F}_{q^{d_{ext}}}$. We start by highlighting that modular reduction by HFE polynomials requires only $O\left(D \log_q(D)^2\right)$ field operations, due to the HFE structure. Then, we introduce a method to modify HFE polynomials which accelerates the modular reduction when the dividend is a square. We improve it by a factor at most two. We also make some experiments to show that this change is secure against direct attack. This technique directly improves the Frobenius map based on repeated squaring.

- In Section 7.4.12, we propose a constant-time implementation of root finding algorithm over $\mathbb{F}_{2^{d_{ext}}}$, when the operand is a split polynomial of degree one, two or three. To do it, we study the existing solvers of degree-two and degree-three polynomials. For degree-two split polynomials, the so-called half trace can be used if d_{ext} is odd. Else, we compute the roots with just one vector-matrix product over $\mathbb{F}_2$. For degree-three split polynomials, we build a matrix whose kernel is a basis of solutions.

Design of a MI-based library. During this thesis, we have elaborated then improved MQsoft [84] (Chapter 9). MQsoft is an efficient C library using vector instruction sets via intrinsics. This library outperforms the state-of-the-art: NTL [153] and Magma for arithmetic in $\mathbb{F}_{2^{d_{ext}}}$ and $\mathbb{F}_{2^{d_{ext}}}[X]$, and the Gui implementation [62] submitted to the first round of the NIST PQC standardization process for HFEv--based signature schemes.

- MQsoft is based on an efficient arithmetic in $\mathbb{F}_{2^{d_{ext}}}$ (Section 9.2). We present our implementation choices about squaring, multiplication, field modular reduction and inversion. The arithmetic is state-of-the-art, and is on average four times faster than NTL. For the square and the multiplication, the selected method depends on the processor. For the modular reduction, we study a large number of ways to improve the modular reduction by the field polynomial. We use irreducible trinomials or pentanomials.

- Based on the previous arithmetic, MQsoft proposes an efficient algorithm of root finding over $\mathbb{F}_{2^{d_{ext}}}$ (Section 9.3). Our implementation of the root finding of HFEv- polynomials is very efficient: between six to thirteen times faster than NTL. The core of the root finding is the Frobenius map. On the one hand, the use of sparse Euclidean division improves the Frobenius map. On the other hand, the modular composition based on an efficient implementation of Karatsuba's multiplication algorithm allows more speed-ups.

- In Section 9.5, we implement efficiently the multipoint evaluation of univariate polynomials over a small extension of $\mathbb{F}_2$. This operation is important because used in all operations of the dual mode. In function of the instances of the multipoint evaluation, we adapt our way to implement it.

- In Section 9.4, we study efficient evaluations of multivariate quadratic systems over $\mathbb{F}_2$. Our variable-time evaluation is state-of-the-art, whereas we generate speed-ups of 10% for the constant-time evaluation, on Skylake processors. In practice, evaluations are more efficient when data are 8-bit aligned. This is not always possible, in particular when the data is packed. So, we study the hybrid representation of the public-key. We store a large number of equations from the public-key with an optimal format for evaluation, without loosing one bit. Then, we study the optimal way to store the remaining equations. We take into account the time to unpack these equations, plus the time to perform the evaluations for each equation. When $q \in \{16, 256\}$, we propose an efficient implementation of the multipoint evaluation of a multivariate polynomial over $\mathbb{F}_q$ (Section 9.4.5). To do it, we use a monomial representation of the points.

- In Section 9.6, we conclude by the performance of `MQsoft`. We study the obtained performance of `GeMSS` and `Gui`, with a detailed cost of the main steps of the keypair generation. Since the beginning of this thesis, we have obtained large speed-ups on `GeMSS`. We also have large speed-ups on the first round implementation of `Gui`, which seems to be the best efficient implementation of `HFEv-`-based signature schemes in C language. The keypair generation of `MQsoft` is between thirty and ninety times faster than the round 1 implementation. For the signing and verifying processes, we obtain respectively factors 2.5 and 1.8. Then, we study the performance of `DualModeMS` and `Dual Rainbow`. The performance of `DualModeMS` becomes reasonable (some seconds for the keypair generation and signature generation). Our implementation of `Dual Rainbow` is very competitive with NIST digital signature proposals.

The structure of `MQsoft` is depicted in Figure 1.2, and summarizes the main tasks required for each cryptographic operation. The critical part of an operation is represented by a plain arrow, whereas less important operations are represented by dotted arrows.

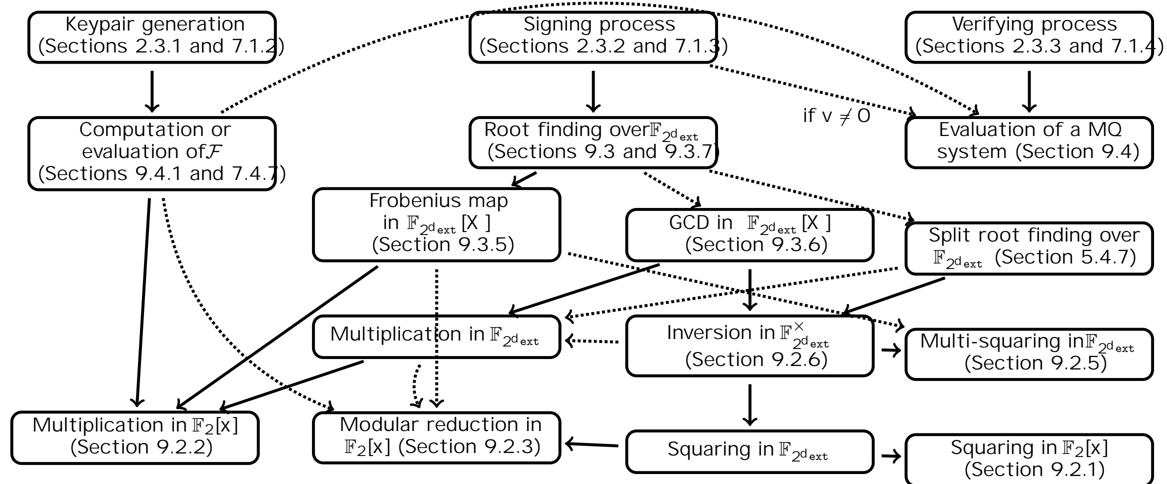

Figure 1.2: Dependencies between the different operations performed in `MQsoft`.

We show in Table 1.1 (respectively Table 1.2 and 1.3) the evolution of the performance of GeMSS128 (respectively RedGeMSS128 and BlueGeMSS256). The slow-downs between NIST round 2 and NIST round 2 (V2) are due to an improvement of the implementation about the size of the public-key, as well as the compression of a secret-key in a seed. For the signing process of GeMSS128, the round 3 implementation of the signing process is slightly slower, due to an enhancement of the security of the implementation.

operation	NIST round 1	NIST round 2	NIST round 2 (V2)	NIST round 3
keypair generation	118 Mc	$\times$ 3.07	$\times$ 3.05	$\times$ 6.03
signing process	1270 Mc	$\times$ 1.69	$\times$ 2.39	$\times$ 2.09
verifying process	0.166 Mc	$\times$ 2.03	$\times$ 1.57	$\times$ 1.57

Table 1.1: Performance of GeMSS128 at the first round of the NIST PQC standardization process, followed by the speed-up between the other rounds and the first round. For each round, we use the corresponding MQsoft version with a Skylake processor (LaptopS). Mc stands for megacycles. The results have three significant digits. For example, $\times 3.05$ means a performance of $118/3.05 = 38.7$ Mc with the NIST round 2 (V2) implementation of MQsoft, for the keypair generation.

operation	NIST round 2	NIST round 2 (V2)	NIST round 3
keypair generation	39.2 Mc	$\times$ 0.992	$\times$ 2.41
signing process	2.79 Mc	$\times$ 1.20	$\times$ 1.36
verifying process	0.109 Mc	$\times$ 0.772	$\times$ 0.774

Table 1.2: Performance of RedGeMSS128 at the second round of the NIST PQC standardization process, followed by the speed-up between the other rounds and the second round. For each round, we use the corresponding MQsoft version with a Skylake processor (LaptopS). Mc stands for megacycles. The results have three significant digits. For example, $\times 2.41$ means a performance of $39.2/2.41 = 16.3$ Mc with the NIST round 3 implementation of MQsoft, for the keypair generation.

operation	NIST round 2	NIST round 2 (V2)	NIST round 3
keypair generation	529 Mc	$\times$ 0.998	$\times$ 3.47
signing process	545 Mc	$\times$ 1.37	$\times$ 2.20
verifying process	0.583 Mc	$\times$ 0.852	$\times$ 0.857

Table 1.3: Performance of BlueGeMSS256 at the second round of the NIST PQC standardization process, followed by the speed-up between the other rounds and the second round. For each round, we use the corresponding MQsoft version with a Skylake processor (LaptopS). Mc stands for megacycles. The results have three significant digits. For example, $\times 3.47$ means a performance of $529/3.47 = 152$ Mc with the NIST round 3 implementation of MQsoft, for the keypair generation.

In Table 1.4, we show the evolution of the performance of GeMSS and Gui since the first round of the NIST PQC standardization process. The speed-ups are obtained with the final version of MQsoft. For GeMSS, the scheme has evolved during the NIST PQC standardization process. The obtained factors are larger in the conditions of the original scheme, but we decrease the performance to minimize the keys size.

scheme	key gen.	sign	verify
GeMSS128	19.6 $\times$ 6.03	608 $\times$ 2.09	0.106 $\times$ 1.57
GeMSS192	69.4 $\times$ 7.9	1760 $\times$ 1.83	0.304 $\times$ 1.47
GeMSS256	158 $\times$ 9.32	2490 $\times$ 2.16	0.665 $\times$ 1.76
FGeMSS(266)	53.7 $\times$ 8.22	44 $\times$ 2.9	0.0365 $\times$ 2.64
Gui-184	23.5 $\times$ 31.7	28.5 $\times$ 2.6	0.0712 $\times$ 1.89
Gui-312	116 $\times$ 41.9	308 $\times$ 2.53	0.161 $\times$ 1.85
Gui-448	356 $\times$ 91.7	5710 $\times$ 3.44	0.562 $\times$ 1.62

Table 1.4: Number of megacycles (Mc) for each cryptographic operation with our library for a Skylake processor (LaptopS), followed by the speed-up between the best implementation provided for the NIST submissions (Table 7.37) versus our implementation. For example, 19.6 $\times$ 6.03 means a performance of 19.6 Mc with MQsoft, and a performance of 19.6 $\times$ 6.03 = 118 Mc for the NIST implementations.

Finally, we present the evolution of DualModeMS since [84] (Table 1.5). We obtain large speed-ups thanks to MQsoft.

scheme	key gen.	sign	verify
DualModeMS128	3710 $\times$ 512	2800 $\times$ 1.97	0.643 $\times$ 15.6
DualModeMS192	6770 $\times$ 1010	8470 $\times$ 2.23	1.73 $\times$ 10.5
DualModeMS256	12700 $\times$ 1360	38000 $\times$ 2.53	3.95 $\times$ 7.69

Table 1.5: Number of megacycles (Mc) for each cryptographic operation of DualModeMS with our implementation, for a Skylake processor (LaptopS), followed by the speed-up between the Skylake implementation from [84] versus our implementation. For example, 3710 $\times$ 512 means a performance of 3710 Mc with our implementation, and a performance of 3710 $\times$ 512 = 1 900 000 Mc for [84]. Note that [84] is similar to the first round implementation of DualModeMS, except for the signing process which is faster.

Part I

Preliminaries

Chapter 2

Multivariate Cryptography

In this chapter, we introduce basic concepts on the public-key cryptography and its security (Section 2.1), then we introduce the public-key multivariate cryptography based on Matsumoto and Imai [119] (Section 2.2). In Section 2.3, we describe the MI-based digital signature schemes. Finally, we present the most important MI-based signature schemes in Section 2.4.

2.1 Public-Key Cryptography and NP-Completeness

In public-key cryptography, we distinguish two important kinds of algorithms: the public-key encryption and the digital signature. The encryption of a plaintext and the verification of the signature of a document are public operations, which require the knowledge of a public-key. The decryption of a ciphertext and the generation of the signature of a document are secret operations, which require the knowledge of a secret-key. The security of a cryptosystem is based on the hardness of performing the secret operation without the knowledge of the secret-key. In general, this is equivalent to solving a specific problem. On the one hand, solving this problem has to be infeasible in practice. On the other hand, we need to perform efficiently the cryptographic operations for concrete uses. The class of NP-complete problems [94] satisfies both properties.

Definition 1 (NP-completeness). A problem of decision is NP-complete if:

- is in NP, i.e. a candidate solution of can be verified in polynomial time,
- is NP-hard: all problems in NP can be reduced into in polynomial time.

On the one hand, NP-complete problem is in NP. We can verify a solution quickly (i.e. in polynomial time). On the other hand, the problem is NP-hard. This implies that the problem is hard to solve. Indeed, if we can solve it in polynomial time, then we will be able to solve all problems in NP in polynomial time. For the moment, the best algorithms solve NP-complete problems in exponential time.

The modern cryptography is based on the integer factorization problem [147] or discrete logarithm problem [149]. Nowadays, no algorithm is known to solve them in polynomial time on classical computers, but we do not know if these problems are NP-complete. In contrast, these

problems can be solved in polynomial time on quantum computers [152], making them insecure. In this thesis, we consider cryptosystems based on NP-complete problems, which are in particular more secure against quantum computers.

2.2 Introduction to Multivariate Cryptography

Introduced by T. Matsumoto and H. Imai in 1988 [119], MI-based multivariate cryptography is based on the hardness of solving a system of m non-linear polynomials in $n_{\mathtt{var}}$ variables. This problem is called PoSSo (Problem 1).

Problem 1. Polynomial System Solving (PoSSo$(q, m, n_{\mathtt{var}}, D)$). Let $q, m, n_{\mathtt{var}}$ and D be integers. Given p a system of m degree-D multivariate polynomials in $\mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$, the problem is to find, if any, a vector x_s in $\mathbb{F}_q^{n_{\mathtt{var}}}$ such that $p(x_s) = (0, \ldots, 0) = 0_m$.

The PoSSo problem was demonstrated NP-complete [135] when solved over a finite field with $D \geqslant 2$. The NP-completeness implies that PoSSo is hard to solve for worst-case instances (Section 2.1). However, this does not necessarily make secure a cryptosystem, because easy-to-solve instances of PoSSo could be used. So, we need a careful study of best attacks against concrete instances of PoSSo. We study them in Sections 4.3 and 4.4.

Based on the PoSSo problem, the public-key of MI-based cryptography is a system $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$ of m equations in $n_{\mathtt{var}}$ variables. When $q = 2$, the public operation (encryption or signature verification) is the evaluation of p in the bits of a plaintext or a signature. Then, the secret operation (decryption or signature generation) cannot be performed e ciently (in polynomial time) because this is equivalent to solving an instance of PoSSo. Therefore, we need to introduce a secret which will allow to invert e ciently p.

For the moment, let $m' = m$. The idea of T. Matsumoto and H. Imai [119] is to generate $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$ with the following structure:

$$p = \mathcal{T} \circ \mathcal{F} \circ \mathcal{S}, \tag{2.1}$$

with $\mathcal{S}, \mathcal{T} \in \mathrm{A}^{-1}_{n_{\mathtt{var}}}(\mathbb{F}_q) \times \mathrm{A}^{-1}_{m'}(\mathbb{F}_q)$ being two invertible a ne transformations, and $\mathcal{F} \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^{m'}$ being a multivariate system easily invertible thanks to a special shape. $\mathcal{F}$ is called the central map of p. The a ne transformations are used to hide the structure of the central map. $\mathcal{S}$ corresponds to a change of variables, whereas $\mathcal{T}$ mixes equations. There exist few choices in the literature for constructing $\mathcal{F}$. For example, $\mathcal{F}$ can be generated from a univariate polynomial $F \in \mathbb{F}_{q^{m'}}[X]$ (Section 2.4.1), or $\mathcal{F}$ can be a multivariate system with an oil and vinegar structure (Section 2.4.2), which can be coupled to a triangular structure (Section 2.4.3).

The structure of p induces another category of attacks. Given the public-key p, an adversary can try to recover its secret structure. This problem is called Extended Isomorphism of Polynomials [133], and is NP-complete [133].

Problem 2. Extended Isomorphism of Polynomials (EIP). Let $q, m', n_{\mathtt{var}}$ and D be integers. Given p a system of m' degree-D multivariate polynomials in $\mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^{m'}$, and a target shape, the problem is to find $\mathcal{S}, \mathcal{T} \in \mathrm{A}^{-1}_{n_{\mathtt{var}}}(\mathbb{F}_q) \times \mathrm{A}^{-1}_{m'}(\mathbb{F}_q)$ and $\mathcal{F} \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^{m'}$ having the target shape, such that $p = \mathcal{T} \circ \mathcal{F} \circ \mathcal{S}$.

As for PoSSo, the hardness of this problem depends on the target shape. We evaluate the complexity of solving it for specific shapes in Section 4.6.

In this thesis, we often consider $\mathcal{S}$ and $\mathcal{T}$ in $GL_{n_{var}}(\mathbb{F}_q) \times GL_{m'}(\mathbb{F}_q)$ as being secret linear transformations, instead of a ne transformations. The concept of equivalent keys (Section 4.6.1) allows this change without any impact on the security. We represent $\mathcal{S}$ and $\mathcal{T}$ respectively by invertible matrices S and T in $GL_{n_{var}}(\mathbb{F}_q) \times GL_{m'}(\mathbb{F}_q)$. Thus, we can write Equation (2.1) as:

$$p(x) = \mathcal{F}(x \cdot S) \cdot T\,.$$

Moreover, we consider the possibility to use the minus variant [151]. Let $m' \geqslant m$. This consists in discarding the last $= m' - m \geqslant 0$ equations from $\mathcal{T} \circ \mathcal{F} \circ \mathcal{S}$. So, $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$ has the following form:

$$p = \circ\, \mathcal{T} \circ \mathcal{F} \circ \mathcal{S}, \qquad (2.2)$$

where $: \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^{m'} \to \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$ is the projection map keeping the first m equations. The name of schemes using the minus variant (> 0) are followed by the $-$ symbol (e.g. HFE-).

We start by presenting the MI-based digital signature in Section 2.3, then we present some trapdoors for $\mathcal{F}$ in Section 2.4.

2.3 MI-Based Public-Key Cryptography

Here, we present the digital signature based on Matsumoto–Imai. The digital signature schemes are divided into three operations: keypair generation (Section 2.3.1), signing process (Section 2.3.2) and verifying process (Section 2.3.3).

2.3.1 Keypair Generation

We summarize the public-key/secret-key generation in Algorithm 1. This algorithm takes the unary representation of , i.e. 1 , and returns a couple secret-key public-key of MI-based schemes. This unary representation is only used in theory, to explain that cryptography attacks have to be exponential in . The linear transformations $\mathcal{S} \in GL_{n_{var}}(\mathbb{F}_q)$ and $\mathcal{T} \in GL_{m'}(\mathbb{F}_q)$ are respectively represented as invertible matrices S and T in $GL_{n_{var}}(\mathbb{F}_q) \times GL_{m'}(\mathbb{F}_q)$. Their generation is explained in Section 7.4.3. Then, let $\mathcal{H}$ be the set of elements allowing to generate $\mathcal{F} \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^{m'}$ and to easily invert it. The nature of $\mathcal{H}$ depends on the trapdoor of $\mathcal{F}$ (Section 2.4). We randomly sample F in $\mathcal{H}$, and the final secret-key corresponds to $(F, S^{-1}, T^{-1}) \in \mathcal{H} \times GL_{n_{var}}(\mathbb{F}_q) \times GL_{m'}(\mathbb{F}_q)$. Here, the matrices S and T are only used during the generation of the public-key. After, we are only using the inverse of these matrices, it is why we store them instead of S and T. Several strategies are possible to generate $p = (p_1, \ldots, p_m) \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$ and are described in Sections 7.4.6 and 7.4.7. In particular, Steps 5 and 6 can be merged by removing the last columns of T during the vector-matrix product.

Algorithm 1 Keypair generation of MI-based schemes.

1: **function** origin.KeyGen 1
2: Randomly sample $(S, T) \in GL_{n_{var}}(\mathbb{F}_q) \times GL_{m'}(\mathbb{F}_q)$.
3: Randomly sample $F \in \mathcal{H}$ in function of the shape of $\mathcal{F}$.
4: $sk \leftarrow (F, S^{-1}, T^{-1}) \in \mathcal{H} \times GL_{n_{var}}(\mathbb{F}_q) \times GL_{m'}(\mathbb{F}_q)$ ▷ Sections 7.4.3 and 7.4.4.
5: $(p_1, \ldots, p_{m'}) \leftarrow \mathcal{T} \circ \mathcal{F} \circ \mathcal{S} \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^{m'}$ ▷ Sections 7.4.6, 7.4.7 and 9.4.1.
6: $pk \leftarrow p = (p_1, \ldots, p_m) \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$ ▷ Take the first $m = m' -$ polynomials computed in Step 5.
7: **return** (sk, pk)
8: **end function**

The secret-key size is given by the size of invertible matrices $(n_{var}^2 + m'^2) \cdot \log_2(q)$ bits, plus the size of F. The latter depends on the trapdoor (Section 2.4). Then, the public-key size is $mN \log_2(q)$ bits, where N is the number of monomials of a quadratic multivariate polynomial in $\mathbb{F}_q[x_1, \ldots, x_{n_{var}}]/\langle x_i^q - x_i \rangle_{1 \leqslant i \leqslant n_{var}}$. When $q > 2$, N is given by the classical formula:

$$N = \sum_{i=0}^{d} \binom{n_{var} + i - 1}{i} = \binom{n_{var} + d}{d}, \qquad (2.3)$$

where $\binom{n_{var}+i-1}{i}$ corresponds to the number of monomials of degree exactly i, and d corresponds to the degree of the multivariate polynomial (which is two here). When $q = 2$ and $d \geqslant 2$, the terms x_i^2 have to be removed. We obtain:

$$N = \binom{n_{var} + d}{d} - n_{var}. \qquad (2.4)$$

2.3.2 Signing Process

The main step of the signing process requires inverting the public-key polynomials $p = (p_1, \ldots, p_m)$ in $\mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$, i.e. solving:

$$p_1(x_1, \ldots, x_{n_{var}}) - d_1 = 0, \ldots, p_m(x_1, \ldots, x_{n_{var}}) - d_m = 0$$

for $d = (d_1, \ldots, d_m) \in \mathbb{F}_q^m$.

To do so, we take advantage of the structure of p. We randomly sample $r = (r_1, \ldots, r\ ) \in \mathbb{F}_q$ and append it to d. This gives $d' = (d, r) \in \mathbb{F}_q^{m'}$. Thus, we can compute $y = \mathcal{T}^{-1}(d') \in \mathbb{F}_q^{m'}$. Then, we randomly sample $v = (v_1, \ldots, v_{n_{var}-m'}) \in \mathbb{F}_q^{n_{var}-m'}$, and if $\mathcal{F}(x_1, \ldots, x_{m'}, v) = y$ has at least one solution, we randomly choose $z \in \mathbb{F}_q^{m'}$ one of the solutions and we return $\mathcal{S}^{-1}(z, v) \in \mathbb{F}_q^{n_{var}}$. Else, we can choose to try again to solve $\mathcal{F}(x_1, \ldots, x_{m'}, v) = y$ with a new random v, or we can choose to randomly sample another r by keeping the same v. We can also randomly sample both r and v. In Algorithm 2, we choose the latter possibility.

Algorithm 2 Inverse map of the public-key.

1: **function** Inv_p $\left(d \in \mathbb{F}_q^m, \mathrm{sk} = (\mathcal{F}, S^{-1}, T^{-1}) \in \mathcal{H} \times GL_{n_{\mathrm{var}}}(\mathbb{F}_q) \times GL_{m'}(\mathbb{F}_q)\right)$
2: **repeat**
3: $r \in_R \mathbb{F}_q$ ▷ The notation $\in_R$ stands for randomly sampling.
4: $d' \leftarrow (d, r) \in \mathbb{F}_q^{m'}$
5: $y \leftarrow d' \cdot T^{-1} \in \mathbb{F}_q^{m'}$
6: $v \in_R \mathbb{F}_q^{n_{\mathrm{var}}-m'}$
7: $L_{\mathrm{Sol}} \leftarrow \mathrm{Solve}(\mathcal{F}(x_1, \dots, x_{m'}, v) = y)$ ▷ Use the special shape of $\mathcal{F}$ with the knowledge of $\mathcal{F}$ to solve efficiently this system. The result is the list of solutions.
8: **until** $L_{\mathrm{Sol}} \neq \emptyset$
9: $z \in_R L_{\mathrm{Sol}}$
10: **return** $(z, v) \cdot S^{-1} \in \mathbb{F}_q^{n_{\mathrm{var}}}$
11: **end function**

We can now present a way to define the signature algorithm (Algorithm 3), leading to a signature of size $n_{\mathrm{var}} \log_2(q)$ bits. The signature is obtained by signing the hash value of a document, $H_1 \colon \{0,1\}^* \to \mathbb{F}_q^m$ being the used hash function. In Section 4.2, we present the Feistel–Patarin construction which generalizes Algorithm 3 with an iterative process.

Algorithm 3 Signing process of MI-based schemes.

1: **function** origin.Sign1$\left(M \in \{0,1\}^*, \mathrm{sk} \in \mathcal{H} \times GL_{n_{\mathrm{var}}}(\mathbb{F}_q) \times GL_{m'}(\mathbb{F}_q), \mathrm{Inv}_p\right)$
2: $D_1 \leftarrow H_1(M)$ ▷ $D_1 \in \mathbb{F}_q^m$.
3: $(S_1, X_1) \leftarrow \mathrm{Inv}_p(D_1, \mathrm{sk})$ ▷ $S_1 \in \mathbb{F}_q^m$ and $X_1 \in \mathbb{F}_q^{n_{\mathrm{var}}-m}$.
4: **return** $\mathrm{sm} = (S_1, X_1) \in \mathbb{F}_q^{n_{\mathrm{var}}}$
5: **end function**

2.3.3 Verifying Process

The corresponding verification process is described in Algorithm 4.

Algorithm 4 Verifying process of MI-based schemes.

1: **function** origin.Verify1$\left(M \in \{0,1\}^*, \mathrm{sm} \in \mathbb{F}_q^{n_{\mathrm{var}}}, \mathrm{pk} = \mathbf{p} \in \mathbb{F}_q[x_1, \dots, x_{n_{\mathrm{var}}}]^m\right)$
2: $(S_1, X_1) \leftarrow \mathrm{sm}$
3: $D_1 \leftarrow H_1(M)$ ▷ $D_1 \in \mathbb{F}_q^m$.
4: **return** VALID if $\mathbf{p}(S_1, X_1) = D_1$ and INVALID otherwise ▷ Sections 9.4.2, 9.4.3 and 9.4.4.
5: **end function**

2.4 Families of Trapdoor

Here, we study the main classes of trapdoors. For each of them, the size of the secret $\mathcal{F} \in \mathcal{H}$ is computed in Appendix A.

2.4.1 Big Field Family

In this part, we consider $m' = d_{\mathtt{ext}}$, where $d_{\mathtt{ext}}$ is the degree of an extension field of $\mathbb{F}_q$. We start by assuming that $n_{\mathtt{var}} = d_{\mathtt{ext}}$. The idea of the big field family is to build $\mathcal{F}$ from a map $\mathcal{F}^* \in \mathbb{F}_{q^{d_{\mathrm{ext}}}}[X]$. Let $= \ _1, \dots, \ _{d_{\mathrm{ext}}} \in \mathbb{F}_{q^{d_{\mathrm{ext}}}}{}^{d_{\mathrm{ext}}}$ be a basis of $\mathbb{F}_{q^{d_{\mathrm{ext}}}}$ over $\mathbb{F}_q$. We set

$$: E = \sum_{k=1}^{d_{\mathrm{ext}}} e_k \cdot \ _k \in \mathbb{F}_{q^{d_{\mathrm{ext}}}} \longmapsto \ (E) = (e_1, \dots, e_{d_{\mathrm{ext}}}) \in \mathbb{F}_q^{d_{\mathrm{ext}}} \tag{2.5}$$

an isomorphism between $\mathbb{F}_{q^{d_{\mathrm{ext}}}}$ and $\mathbb{F}_q^{d_{\mathrm{ext}}}$. Then, we can write $\mathcal{F} = \ \circ \mathcal{F}^* \circ \ ^{-1}$. For such a family, $\mathcal{F}^*$ is represented by a degree-D univariate polynomial over the extension field $\mathbb{F}_{q^{d_{\mathrm{ext}}}}$. Its inversion is the problem of finding the roots of a degree-D univariate polynomial, which can be solved in quasi-linear time in D (Section 5.4.8).

The C^* scheme. The first scheme using a univariate polynomial, C^*, was introduced by T. Matsumoto and H. Imai [119]. They considered the univariate polynomial $X^{q^{\ }+1}$ $(0 \leqslant \ < d_{\mathtt{ext}})$. The integer is chosen such that $\mathrm{GCD}\left(q^{\ } + 1, q^{d_{\mathrm{ext}}} - 1\right) = 1$. Thus, $q^{\ } + 1$ is invertible modulo $q^{d_{\mathrm{ext}}} - 1$, allowing to compute a $(q^{\ } + 1)$-th root in $\mathbb{F}_{q^{d_{\mathrm{ext}}}}$ by raising an element to the power of $(q^{\ } + 1)^{-1} \bmod q^{d_{\mathrm{ext}}} - 1$. As a consequence, we can efficiently invert $\mathcal{F}$ (i.e. in polynomial time in) with this special choice of univariate polynomial, and in particular when is close to $d_{\mathtt{ext}}$. However, C^* was broken by J. Patarin [132], who introduced its generalization: Hidden Field Equations (HFE) [133].

Hidden Field Equations. In the HFE scheme, the univariate polynomial has the following form:

$$\sum_{\substack{0 \leqslant j \leqslant i < d_{\mathrm{ext}} \\ q^i + q^j \leqslant D}} A_{i,j} X^{q^i + q^j} + \sum_{\substack{0 \leqslant i < d_{\mathrm{ext}} \\ q^i \leqslant D}} B_i X^{q^i} + C \in \mathbb{F}_{q^{d_{\mathrm{ext}}}}[X], \tag{2.6}$$

with $A_{i,j}, B_i, C \in \mathbb{F}_{q^{d_{\mathrm{ext}}}}$, and $A_{i,i} = 0$ when $q = 2$. The choice of non-zero terms directly impacts the degree of $\mathcal{F}$. Since $\mathcal{F} = \ \circ \mathcal{F}^* \circ \ ^{-1}$, we replace X by $\ ^{-1}(x_1, \dots, x_{d_{\mathrm{ext}}})$ in F and we obtain a multivariate polynomial. In particular, we have that

$$X^{q^i} = \left(\sum_{k=1}^{d_{\mathrm{ext}}} \ _k \cdot x_k\right)^{q^i} = \sum_{k=1}^{d_{\mathrm{ext}}} \ _k^{q^i} \cdot x_k \bmod \langle x_1^q - x_1, \dots, x_{n_{\mathrm{var}}}^q - x_{n_{\mathrm{var}}}\rangle \text{ for } i \in \mathbb{N}, \tag{2.7}$$

is linear. Thus,

$$\begin{aligned} X^{q^i + q^j} &= \left(\sum_{k=1}^{d_{\mathrm{ext}}} \ _k \cdot x_k\right)^{q^i} \cdot \left(\sum_{k=1}^{d_{\mathrm{ext}}} \ _k \cdot x_k\right)^{q^j} \\ &= \sum_{\substack{1 \leqslant k \leqslant d_{\mathrm{ext}} \\ 1 \leqslant k' \leqslant d_{\mathrm{ext}}}} \ _k^{q^i} \ _{k'}^{q^j} \cdot x_k x_{k'} \bmod \langle x_1^q - x_1, \dots, x_{n_{\mathrm{var}}}^q - x_{n_{\mathrm{var}}}\rangle \text{ for } i, j \in \mathbb{N}, \end{aligned} \tag{2.8}$$

is quadratic (or linear if $q = 2$ and $i = j$). By using (2.8), we know that the HFE polynomial has a multivariate quadratic representation if $D \geqslant 2$ when $q \neq 2$, or if $D \geqslant 3$ otherwise. In the HFE

scheme, the inversion of $\mathcal{F}$ is equivalent to finding a root of a HFE polynomial. This can be computed efficiently with a root finding algorithm (Section 5.4.8) when the degree is small enough. We will see in Chapter 7 how to select D.

Vinegar variant. Now, we assume $n_{\mathrm{var}} \geqslant d_{\mathrm{ext}}$. In [110], the authors proposed the HFEv variant of HFE, based on the idea of UOV (Section 2.4.2). In this variant, we consider a HFEv polynomial defined by

$$\sum_{\substack{0 \leqslant j \leqslant i < d_{\mathrm{ext}} \\ q^i + q^j \leqslant D}} A_{i,j} X^{q^i + q^j} + \sum_{\substack{0 \leqslant i < d_{\mathrm{ext}} \\ q^i \leqslant D}} \beta_i(v_1, \ldots, v_v) X^{q^i} + \gamma(v_1, \ldots, v_v) \in \mathbb{F}_{q^{d_{\mathrm{ext}}}}[X, v_1, \ldots, v_v], \tag{2.9}$$

where $A_{i,j} \in \mathbb{F}_{q^{d_{\mathrm{ext}}}}$, with $A_{i,i} = 0$ when $q = 2$, each $\beta_i : \mathbb{F}_q^v \to \mathbb{F}_{q^{d_{\mathrm{ext}}}}$ is linear and $\gamma : \mathbb{F}_q^v \to \mathbb{F}_{q^{d_{\mathrm{ext}}}}$ is quadratic modulo $v_i^q - v_i$. The variables $v_1, \ldots, v_v$ are called *vinegar variables*. We shall say that a polynomial $F \in \mathbb{F}_{q^{d_{\mathrm{ext}}}}[X, v_1, \ldots, v_v]$ with the form of (2.9) has *HFEv-shape*. The particularity of a polynomial $F(X, v_1, \ldots, v_v)$ with HFEv-shape is that for any specialization of the vinegar variables, the polynomial F becomes a HFE polynomial (Equation (2.6)). By abuse of notation, we will refer to D as the degree of the HFEv polynomial. We also make the correspondence between $(x_{d_{\mathrm{ext}}+1}, \ldots, x_{n_{\mathrm{var}}})$ and $(v_1, \ldots, v_v)$.

The special structure of (2.9) is chosen such that its multivariate representation over the base field $\mathbb{F}_q$ is composed of quadratic polynomials in $\mathbb{F}_q[x_1, \ldots, x_{n_{\mathrm{var}}}]$. When the exponents chosen in X have a decomposition in base q of Hamming weight equal to two, its multivariate representation is quadratic in $x_1, \ldots, x_{d_{\mathrm{ext}}}$. When the Hamming weight is equal to one, the monomials are multiplied by linear terms in $v_1, \ldots, v_v$, thus its multivariate representation is quadratic in $x_1, \ldots, x_{n_{\mathrm{var}}}$. Finally, when the exponent is zero, the monomial is multiplied by quadratic terms in $v_1, \ldots, v_v$, implying its multivariate representation is quadratic in $v_1, \ldots, v_v$.

Security and modifiers. The original HFE scheme is broken [96]. A fundamental element in the design of secure HFE-based schemes is the introduction of perturbations. Here, we give some of them. More perturbations can be found in [164].

- The vinegar modifier [110] consists in using a HFEv polynomial (Equation (2.9)) with v vinegar variables. The obtained scheme is HFEv.
- The minus modifier [151, 133] consists in removing $\Delta = d_{\mathrm{ext}} - m$ equations from the public-key. This change corresponds to the projection map in Equation (2.2). The obtained scheme is HFE-, and the Δ removed equations are called *minus equations*. Here, $d_{\mathrm{ext}} > m$.
- The plus modifier [133, 162] consists in adding $m - d_{\mathrm{ext}}$ random quadratic equations in $\mathbb{F}_q[x_1, \ldots, x_{n_{\mathrm{var}}}]$ to $\mathcal{F} \circ \mathcal{S}$, then mixing equations with $\mathcal{T} \in \mathrm{GL}_m(\mathbb{F}_q)$. The obtained scheme is HFE+. Here, $m > d_{\mathrm{ext}}$.

The minus and vinegar variants of HFE, i.e. HFE-, HFEv and HFEv-, are still secure. Their security has been extensively studied for more than twenty years. Currently, the HFEv--based signature scheme GeMSS (Chapter 7) is an alternate candidate of the third round of the NIST PQC standardization process [130].

About the probability of finding s roots from a HFE polynomial. The cost of the inversion of the public-key (Algorithm 2) depends on the distribution of the number of roots of a univariate polynomial. A HFE polynomial has a HFE-shape (Equation (2.6)). As a consequence, its roots correspond to the zeros of a system of d_{ext} equations in d_{ext} variables. In [88], the authors studied the distribution of the number of zeros of algebraic systems. In particular, a random system of d_{ext} degree-d equations in d_{ext} variables in $\mathbb{F}_q$ has exactly s solutions with probability $\exp(-1)\cdot\frac{1}{s!}$, when q is prime and $d \geqslant 2$ [88, Corollary 2]. Although $\mathcal{F}$ is not random, experiments for $d = 2$, $D = 4097$, $d_{\text{ext}} = 13$ and $q = 2$ (Table 2.1) show that the probability of finding s roots seems to coincide with $\exp(-1)\cdot\frac{1}{s!}$. We also obtain this result for dense polynomials.

s	0	1	2	3	4	5	34
$\exp(-1)\cdot\frac{1}{s!}$	36.79%	36.79%	18.39%	6.13%	1.53%	0.31%	$2^{-129.2}$
dense polynomial	37.04%	37.09%	18.10%	6.09%	1.27%	0.35%	0
HFE polynomial	36.47%	36.71%	18.69%	6.28%	1.51%	0.28%	0

Table 2.1: Theoretical probability of finding s distinct roots for a random HFE polynomial in an extension field ($s \leqslant D$), followed by experimental tests. We study the distribution of the number of roots of 10000 random polynomials generated with Magma (Section 6.2.1).

2.4.2 Unbalanced Oil and Vinegar (UOV)

The UOV scheme [110] is based on the use of two types of variables: oil and vinegar. The secret map $\mathcal{F}$ corresponds to a multivariate system of m' equations in the $n_{\text{var}} = m' + v$ variables $o_1, \ldots, o_{m'}, v_1, \ldots, v_v = x_1, \ldots, x_{n_{\text{var}}}$. This system is quadratic in the v vinegar variables but linear in the m' oil variables. Each equation is as follows:

$$\sum_{1\leqslant j\leqslant i\leqslant v} \alpha_{i,j} v_i v_j + \sum_{\substack{1\leqslant i\leqslant v\\ 1\leqslant j\leqslant m'}} \beta_{i,j} v_i o_j + \sum_{1\leqslant i\leqslant v} \mu_i v_i + \sum_{1\leqslant i\leqslant m'} B_i o_i + C, \tag{2.10}$$

with $\alpha_{i,j}$, $\beta_{i,j}$, μ_i, B_i and C in $\mathbb{F}_q$, and $\alpha_{i,i} = 0$ when $q = 2$. Then, the structure of $\mathcal{F}$ is hidden with the use of $\mathcal{S} \in \mathrm{GL}_{n_{\text{var}}}(\mathbb{F}_q)$ (Section 2.2). The latter mixes the oil variables with the vinegar variables. The security of the UOV scheme is based on the assumption that in $\mathbf{p}$ (Equation (2.1)), the oil variables are indistinguishable from the vinegar variables. For this reason, $\mathcal{T} \in \mathrm{GL}_{m'}(\mathbb{F}_q)$ is set to the identity map and we have $\mathbf{p} = \mathcal{F} \circ \mathcal{S}$.

The map $\mathcal{F}$ is quadratic. The strategy of UOV is to randomly sample the vinegar variables. Thus, the new (square) system becomes linear, and the oil variables can be efficiently computed with a Gaussian elimination. However, all linear systems do not necessarily admit a solution. When the set of solutions is empty, the vinegar variables have to be changed (similarly to Algorithm 2).

Lifted Unbalanced Oil and Vinegar (LUOV). In [24], the authors proposed LUOV: a variant of the UOV scheme. That is also a NIST candidate [26]. The public and secret maps are generated in $\mathbb{F}_q$, but are used as maps in a degree-r extension of $\mathbb{F}_q$. Compared to a classical UOV scheme in $\mathbb{F}_{q^r}$, this transformation allows to divide by r the size of the public and secret maps. However, this scheme was broken in [66], and was not selected to the third round. The authors were able to take advantage of the structure to speed up known attacks.

2.4.3 Rainbow

The `Rainbow` scheme [69] is a generalization of the `UOV` scheme, based on the use of several `UOV` layers. In the `UOV` scheme, we have one quadratic system, which once specified in the vinegar variables, becomes linear and so the oil variables can be found. In the `Rainbow` scheme, we have a lower triangular system by block. Each of the u blocks is a `UOV` system. They share the same vinegar variables $x_1, \ldots, x_{v_1}$. Then, the oil variables are divided into u groups of variables. Let o_k be the size of the k-th group of variables, and let $v_{k+1} = v_k + o_k$ be the number of variables of the k-th block. The k-th block is a system of o_k equations in $v_k + o_k$ variables, which has the first k groups of oil variables. We consider that the first v_k variables are vinegar, whereas the last o_k variables are oil. Each equation of the k-th block is given by:

$$\sum_{1 \leqslant j \leqslant i \leqslant v_k} \alpha_{i,j} x_i x_j + \sum_{\substack{1 \leqslant i \leqslant v_k \\ v_k+1 \leqslant j \leqslant v_k+o_k}} \beta_{i,j} x_i x_j + \sum_{1 \leqslant i \leqslant v_k} \mu_i x_i + \sum_{v_k+1 \leqslant i \leqslant v_k+o_k} B_i x_i + C,$$

with $\alpha_{i,j}$, $\beta_{i,j}$, μ_i, B_i and C in $\mathbb{F}_q$, and $\alpha_{i,i} = 0$ when $q = 2$. The coefficients $\alpha_{i,j}$ and μ_i correspond to the vinegar variables, whereas $\beta_{i,j}$ is a mix between oil and vinegar variables, and B_i corresponds to the oil variables.

The public map of `Rainbow` is a multivariate quadratic system of $m' = v_{u+1} - v_1$ equations in $n_{\texttt{var}} = v_{u+1}$ variables. Here, we consider $\mathcal{F}(x_{v_1+1}, \ldots, x_{n_{\texttt{var}}}, x_1, \ldots, x_{v_1})$ to be consistent with Algorithm 2 which randomly samples the last v_1 variables. When $u = 1$, we obtain the `UOV` scheme, but unlike `UOV`, the affine transformation $\mathcal{T} \in \mathrm{Aff}^{-1}_{m'}(\mathbb{F}_q)$ in `Rainbow` is not set to the identity transformation. When $u = 2$, we obtain the `Rainbow` signature scheme [69, 65], which is a finalist candidate of the third round of the NIST PQC standardization process [130].

We will use `Rainbow` in Chapter 8, as inner layer of the so-called `Dual Rainbow` signature scheme.

Chapter 3

Public-Key Compression of Signature Schemes

In Chapter 2, we presented MI-based signature schemes. For typical parameters (Chapter 7), the public-key of such schemes is large whereas the signatures are very short. In this chapter, we study a transformation which provides a dual trade-o leading to a small public-key but a large signature size. This transformation is composed of two distinct layers. The first one (inner layer) is a classical MI-like multivariate scheme (Chapter 2). The second part (outer layer) is the core of the method proposed by A. Szepieniec, W. Beullens and B. Preneel (SBP) in [157].

The Szepieniec, Beullens and Preneel (SBP) technique [157] allows to transform the public-key origin.pk $= \mathbf{p} \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$ of the inner layer into a new public-key pk that is going to be the root of one (or several) binary tree (Section 3.1). The new secret-key sk will include the original secret-key and public-key of the inner layer. The new signature process (Section 3.2) will require generating $\geqslant 1$ signatures from the inner layer. A (new) signature from the outer layer will also include a set of random linear combinations $\mathbf{h} \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]$ from $\mathbf{p}$, together with a set of nodes allowing to check that $\mathbf{h}$ has been correctly derived from $\mathbf{p}$ (Section 3.4).

The security of the SBP transformation is based on the security of the inner layer, as well as on the hardness of a new problem: Approximate MQ (AMQ) (Section 3.5).

3.1 Dual Keypair Generation

We describe the generation of the secret-key and public-key in the SBP transformation. This process uses the function origin.KeyGen (Algorithm 1) that returns the secret-key and public-key of the inner MI-based scheme. That is (origin.sk, origin.pk) $\leftarrow$ origin.KeyGen(1) with:

$$\text{origin.sk} \leftarrow (F, S^{-1}, T^{-1}) \in \mathcal{H} \times GL_{n_{\mathtt{var}}}(\mathbb{F}_q) \times GL_m(\mathbb{F}_q),$$
$$\text{origin.pk} \leftarrow \mathbf{p} = (p_1, \ldots, p_m) \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m.$$

The signature process in the outer layer requires generating signatures from origin.Sign1 (Algorithm 3). In addition, a signature will also include a set of random linear combinations

$$\mathbf{h} = (h_1, \ldots, h) \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}], 1 \leqslant \quad \leqslant m$$

of origin.pk. A key point in the construction of [157] is a mechanism allowing to check that $h \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]$ has been indeed correctly derived from origin.pk. To do so, SBP introduced the concept of MAC (Message Authentication Code) polynomial. The MAC polynomial is a univariate polynomial obtained from a multivariate polynomial by taking its N coefficients (Equation (2.3)) by block of elements to build $\frac{N}{}$ elements in $\mathbb{F}_q$. This yields a polynomial in $\mathbb{F}_q[y]$ of degree D_{MAC}, with $D_{MAC} = \frac{N}{} - 1$. This process is described in Algorithm 5.

Algorithm 5 Construction of MAC polynomial.

1: **function** MacPoly $f \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]$
2: $\quad L_{coefs} \leftarrow$ list of all coefficients of f, sorted according to a fixed monomial order
3: $\quad D_{MAC} \leftarrow \frac{N}{} - 1$
4: $\quad$ **for** i from 0 to D_{MAC} **do**
5: $\quad\quad c_i \leftarrow$ cast $L_{coefs}[i \cdot + 1, \ldots, (i+1) \cdot]$ to $\mathbb{F}_q$
6: $\quad$ **end for**
7: $\quad \hat{f} \leftarrow \sum_{i=0}^{D_{MAC}} c_i \cdot y^i \in \mathbb{F}_q[y]$
8: $\quad$ **return** $\hat{f}$
9: **end function**

The MAC polynomials have an important property. The transformation is invariant under linear combinations:

$$h = p \cdot t \Rightarrow \hat{h} = \hat{p} \cdot t, \text{ for all } t \in \mathcal{M}_{m,} \quad \mathbb{F}_q ,$$

where $\hat{p} \in \mathbb{F}_q[y]^m$ and $\hat{h} \in \mathbb{F}_q[y]$ are respectively the MAC polynomials of p and h. So, if h is derived from p, then $\hat{p}$ and $\hat{h}$ have to coincide. This is verified by evaluating the MAC polynomials on a random subset $\subseteq Z$, where $Z \subseteq \mathbb{F}_q$ is a set of points whose evaluations in the polynomial system $\hat{p}$ are publicly known. The evaluations of $\hat{p}$ on Z could be the public-key, but its size would be larger than the inner public-key. Thus, this set of points is compressed in a Merkle tree [122]. Let $\in \mathbb{N}$ be a parameter. When $= 0$, the function Merkle.generate_tree (Algorithm 8, Section 3.3) takes a set of elements in $\mathbb{F}_q^m$ and constructs the corresponding Merkle tree. Else, the function Merkle.generate_tree returns 2 Merkle subtrees, each corresponding to a Merkle tree of $/2$ elements. These trees can be considered as an incomplete Merkle tree, or truncated Merkle tree. The leaves of the truncated Merkle tree are elements in $\mathbb{F}_q^m$ whilst the inner nodes are digests. The root of the 2 Merkle trees is published instead of the evaluations of the MAC polynomials of p, whereas the other nodes are stored in the secret-key.

We now have all the tools to describe the public-key/secret-key generation process in the dual mode (Algorithm 6). The public-key is the root of 2 Merkle trees, which implies storing 2 digests for a small $\geqslant 0$. On the other hand, the secret is large since it will include – in particular – the truncated Merkle tree containing $2 - 2^{+1}$ digests (typically, $\geqslant 2^{18}$), but also the inner public-key.

3.2 Dual Signing Process

The signature process of the dual mode is derived from origin.Sign1 (Algorithm 3, Chapter 2), which returns the signature of the inner layer. The novelty in new.Sign (Algorithm 7) is the use of a Merkle tree that provides an authentication tag. In particular, the function Merkle.path takes a

truncated Merkle tree and a leaf of this tree, and returns a list of nodes allowing to compute the corresponding node of the public-key from its leaf (Algorithm 9, Section 3.3).

Algorithm 6 Keypair generation in the dual mode.

1: **function** new.KeyGen(1)
2: $\quad$ (origin.sk, origin.pk) $\leftarrow$ origin.KeyGen(1)
3: $\quad$ $p \leftarrow$ origin.pk $= (p_1, \ldots, p_m) \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$
4: $\quad$ **for** i from 1 to m **do**
5: $\quad\quad$ $\hat{p}_i \leftarrow$ MacPoly(p_i) $\qquad$ ▷ Algorithm 5.
6: $\quad$ **end for**
7: $\quad$ $\hat{p} \leftarrow (\hat{p}_1, \ldots, \hat{p}_m)$
8: $\quad$ $Z \leftarrow$ choose a set of distinct points $z_1, \ldots, z \in \mathbb{F}_q$
9: $\quad$ mt $\leftarrow$ Merkle.generate_tree($\{\hat{p}(z)\}_{z \in Z}$) $\qquad$ ▷ Algorithm 8.
10: $\quad$ pk $\leftarrow$ Merkle.roots(mt) $\qquad$ ▷ The (new) public-key is the root of 2 Merkle trees.
11: $\quad$ sk $\leftarrow$ (origin.sk, origin.pk, mt) $\qquad$ ▷ The truncated Merkle tree is stored without its 2 roots.
12: $\quad$ **return** (sk, pk)
13: **end function**

The signature process is described in Algorithm 7. Inner signatures are computed from the digest of small variations of a message $M \in \{0,1\}^*$. Then, the derived system $h \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]$ is obtained by linear combinations of the polynomials of $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$. The corresponding map is represented by a full-rank matrix $t \in \mathcal{M}_{m,}$ $\mathbb{F}_q$, which can be generated from a deterministic random bytes generator such as SHAKE [126]. This generator is also used to create a random set O of points from Z. Finally, the evaluations of $\hat{p}$ in the points of O are computed and stored in the signature, as well as the corresponding authentication paths.

Algorithm 7 Signing process in the dual mode.

1: **function** new.Sign($M \in \{0,1\}^*$, sk $=$ (origin.sk, origin.pk, mt))
2: $\quad$ $p \leftarrow$ origin.pk $= (p_1, \ldots, p_m) \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$
3: $\quad$ **for** i from 0 to -1 **do**
4: $\quad\quad$ $s_i \leftarrow$ origin.Sign(1, $M \| i$, origin.sk, Inv$_p$) $\qquad$ ▷ Get signatures from the inner layer.
5: $\quad$ **end for**
6: $\quad$ $t \leftarrow$ cast $m \log_2(q)$ bits of SHAKE($M \| s_0 \| \cdots \| s_{-1}$) into a full-rank matrix in $\mathcal{M}_{m,}$ $\mathbb{F}_q$
7: $\quad$ $h = (h_1, \ldots, h) \leftarrow p \cdot t \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]$
8: $\quad$ $i_1, \ldots, i \leftarrow$ cast $\log_2()$ bits of SHAKE($M \| s_0 \| \cdots \| s_{-1} \| h$) into a set of integers, each in $[\![0, -1]\!]$, not necessarily distinct
9: $\quad$ $O \leftarrow \{z_{i_1}, \ldots, z_{i}\} \subseteq Z$ $\qquad$ ▷ A random subset of $Z \subseteq \mathbb{F}_q$ of size .
10: $\quad$ $\hat{p} \leftarrow$ (MacPoly(p_1), ..., MacPoly(p_m))
11: $\quad$ **for** j from 1 to **do** $\qquad$ ▷ Get the list of digests allowing to generate a Merkle path from each of the leaves.
12: $\quad\quad$ $L_{mp}[j] \leftarrow$ Merkle.path(mt, $\hat{p}(z_{i_j})$) $\qquad$ ▷ Algorithm 9.
13: $\quad$ **end for**
14: $\quad$ **return** sm $=$ ($s_0, \ldots, s_{-1}, h, \hat{p}(z_{i_1}), \ldots, \hat{p}(z_{i}), L_{mp}$)
15: **end function**

The size of a signature is then:

$$\cdot |\mathrm{originsm}| + (\quad \cdot N + \quad \cdot m \cdot \quad) \cdot \log_2(q) + \quad \cdot (\log_2(\quad) - \quad) \cdot 2 \quad \text{bits}, \qquad (3.1)$$

where $|\mathrm{originsm}|$ corresponds to the size in bits of an inner signature. In general, $|\mathrm{originsm}|$ is equal to $n_{\mathtt{var}} \log_2(q)$ bits, or $n_{\mathtt{var}} \log_2(q) + 128$ bits when a 128-bit salt is added (Section 7.6.2).

3.3 Merkle Tree

As mentioned above, the SBP transformation requires the use of Merkle trees [122] to guarantee the validity of $\mathtt{MAC}$ polynomials. Merkle trees are compression trees. Given $d_0, \ldots, d_{\ -1}$ in $\{0,1\}^*$ and a power of two, the corresponding Merkle tree is built by setting the i-th leaf to the hash value of d_i (Figure 3.1). Then, each parent node is the hash of the concatenation of these two child nodes. The root of the Merkle tree is a single digest which contains information about all digests.

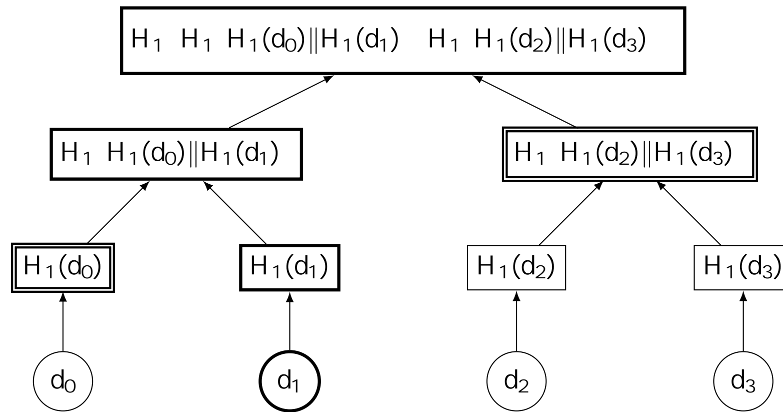

Figure 3.1: Merkle tree. H_1 is a hash function.

To check if a bit sequence d_i' for $0 \leqslant i <$ corresponds to d_i, the Merkle tree can be used as follows. We compute the i-th leaf of the Merkle tree with d_i' instead of d_i. Then, we build a path in the tree until the root node. This requires the knowledge of all sibling nodes. Finally, we check that the computed root corresponds to the public-key.

We present this principle in Figure 3.1 for $i = 1$. The receiver needs to receive d_1, as well as the authentication path composed of $H_1(d_0)$ and $H_1(H_1(d_2)\|H_1(d_3))$. Then, he can compute $H_1(d_1)$, $H_1(H_1(d_0)\|H_1(d_1))$ and $H_1(H_1(H_1(d_0)\|H_1(d_1))\|H_1(H_1(d_2)\|H_1(d_3)))$. Finally, he verifies that $H_1(H_1(H_1(d_0)\|H_1(d_1))\|H_1(H_1(d_2)\|H_1(d_3)))$ is the public-key.

We can also truncate the Merkle tree, and publish the floor containing 2 nodes instead of the root, for $\in \mathbb{N}$. For example, the public-key corresponding to Figure 3.1 and $= 1$ is the floor having 2^1 nodes, i.e. $H_1\ H_1(d_0)\|H_1(d_1)$, $H_1\ H_1(d_2)\|H_1(d_3)$. This increases the size of the public-key, but decreases the size of paths in the truncated Merkle tree. We present the process of generation of a truncated Merkle tree in Algorithm 8. The truncated Merkle tree is a tree of $2 \ - 2$ digests of size 2 bits.

Algorithm 8 Generation of 2 Merkle trees as a truncated Merkle tree.

1: **function** Merkle.generate_tree$\{d_0, \ldots, d_{-1}\}$
2: **for** i from 0 to -1 **do**
3: $mt[i] \leftarrow H_1(d_i)$
4: **end for**
5: $j \leftarrow 0$
6: **for** i from to $2 - 1 - 2$ **do** ▷ Each node is the hash of the concatenation of these leaves.
7: $mt[i] \leftarrow H_1\ mt[j]\ mt[j+1]$
8: $j \leftarrow j + 2$
9: **end for**
10: **return** mt ▷ Return a truncated Merkle tree with leaves and 2 roots.
11: **end function**

In Algorithm 9, we generate an authentication path. From a leaf of the Merkle tree, the user has to be able to generate a branch until the 2 -node floor. To do it, at each floor of the tree, the user has to know the couple of digests allowing to generate the digest of the next floor. At the end, he will be able to verify if the obtained digest at the 2 -node floor corresponds to the public-key. This process is described in Algorithm 10.

Algorithm 9 Generation of an authentication path.

1: **function** Merkle.path(mt, $leaf$)
2: $ind \leftarrow$ index of $leaf$ in mt
3: $\leftarrow 0$ ▷ Index of the beginning of the -th floor.
4: **for** i from 0 to $\log_2(\ ) - \ - 1$ **do** ▷ For each floor of the truncated Merkle tree (without the 2 roots).
5: $mp[i] \leftarrow mt[\ + (ind \texttt{ XOR } 1)]$ ▷ Sibling node of the current node.
6: $\leftarrow \ + \frac{}{2}$ ▷ Add the size of the current floor.
7: $ind \leftarrow \lfloor ind/2 \rfloor$
8: **end for**
9: **return** mp
10: **end function**

3.4 Dual Verifying Process

The verification process in the outer layer is described in Algorithm 11. We need to verify the validity of signatures from the inner layer, the validity of $h \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]$, as well as the validity of evaluations of $\hat{p} \in \mathbb{F}_q[y]^m$. To do so, we verify each point step by step. We start by verifying the inner signatures by assuming that h is derived from linear combinations of $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$. In a similar way, we verify h by evaluating $\hat{h} \in \mathbb{F}_q[y]$ in random points, then by comparing the result to the evaluations of $\hat{p}$. Finally, we verify the previous evaluations by using the function Merkle.verify (Algorithm 10), that takes the root of 2 Merkle trees and a set of nodes, and checks that the root can be indeed generated from the nodes.

Algorithm 10 Verification of a leaf via its authentication path and the roots.

```
1: function Merkle.verify(pk, mp, d_ind)
2:     h_{-1} ← H_1(d_ind)
3:     ind ← index of h_{-1} in mt
4:     for i from 0 to log_2( ) −  − 1 do      For each floor of the truncated Merkle tree (without the 2  roots).
5:         if ind mod 2 = 0 then
6:             h_i ← H_1 h_{i−1}‖mp[i]
7:         else
8:             h_i ← H_1 mp[i]‖h_{i−1}
9:         end if
10:        ind ← ⌊ind/2⌋
11:    end for
12:    if h_{log_2( )− −1} = pk[ind] then
13:        return VALID
14:    else
15:        return INVALID
16:    end if
17: end function
```

Algorithm 11 Verifying process in the dual mode.

```
1: function new.Verify M ∈ {0,1}*, sm, pk ∈ {0,1}^{2 ·2}
2:      s_0, . . . , s_{−1}, h, p̂(z_{i_1}), . . . , p̂(z_i ), L_mp ← sm
3:     t ← cast m  log_2(q) bits of SHAKE(M‖s_0‖ · · · ‖s_{−1}) into a full-rank matrix in M_{m,}  F_q
4:     for i from 0 to  − 1 do
5:         D_i ← H_1(M‖i) ∈ F_q^m       The field element has to be generated according the method used in origin.Sign1.
6:         if h(s_i) ≠ D_i · t then
7:             return INVALID
8:         end if          We use the verification process of the inner mode with h as a public-key.
9:     end for
10:    ĥ ← MacPoly(h_1), . . . , MacPoly(h )                Algorithm 5.
11:    i_1, . . . , i  ← cast  log_2( ) bits of SHAKE(M‖s_0‖ · · · ‖s_{−1}‖h) into a set of integers, each in ⟦0,  − 1⟧, not necessarily distinct
12:    O ← {z_{i_1}, . . . , z_i } ⊆ Z
13:    for j from 1 to  do
14:        mp ← L_mp[j]
15:        if Merkle.verify pk, mp, p̂(z_{i_j})  = INVALID or ĥ(z_{i_j}) ≠ p̂(z_{i_j}) · t then    Algorithm 10.
16:            return INVALID
17:        end if
18:    end for
19:    return VALID
20: end function
```

3.5 Security

The SBP transformation is proven secure, and has the Existential UnForgeability against Chosen Message Attack (EUF-CMA) property (Chapter 4). This property is given by Theorem 1 [157], when the number of inner signatures $= 1$. The authors of [157] considered $= 0$.

Theorem 1. Let $= 1$ be the number of inner signatures included in a signature of the outer layer. If there is an adversary A against the EUF-CMA property of the outer layer of the SBP transformation in time T with Q random oracle queries and with success probability , then there exists an adversary B^A against the EUF-CMA property of the inner layer in time $O(T)$ with success probability at least

$$- (Q+1) \cdot q^{-} - (Q+1) \cdot \frac{\frac{N}{\ } - 1}{\ } - (Q+1) \cdot \frac{2 - 1}{2^2}.$$

When > 1, the security of the SBP transformation relies then on a new hard problem, so-called Approximate MQ ($\mathtt{AMQ}$) problem, which is defined below.

Problem 3. Approximate MQ ($\mathtt{AMQ}(q, , m, n_{\mathtt{var}}, r)$).
Input. A set of quadratic polynomials $p = (p_1, \ldots, p_m) \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$, vectors $y_1, \ldots, y \in \mathbb{F}_q^m$ and $r \in \mathbb{N}$ such that $r < \min(m,)$.
Question. Find $x_1, \ldots, x \in \mathbb{F}_q^{n_{\mathtt{var}}}$ such that

$$\dim \operatorname{Vec} p(x_1) - y_1, \ldots, p(x) - y \leqslant r,$$

where Vec stands for the vector space generated by $p(x_1) - y_1, \ldots, p(x) - y$.

Under the assumption that $\mathtt{AMQ}$ is hard for a fixed and r, we obtain Theorem 2 [157].

Theorem 2. If there is an adversary A against the EUF-CMA property of the outer layer of the SBP transformation in time T with Q random oracle queries and with success probability , and if $\mathtt{AMQ}(q, , m, n_{\mathtt{var}}, r)$ is hard, then there exists an adversary B^A against the EUF-CMA property of the inner layer in time $O(T)$ with success probability at least

$$- (Q+1) \cdot q^{- \cdot (r+1)} - (Q+1) \cdot \frac{\frac{N}{\ } - 1}{\ } - (Q+1) \cdot \frac{2 - 1}{2^2}.$$

In Figure 3.2, we propose a simplified vision of the security model of the SBP transformation (Theorem 2). We do not consider the EUF-CMA property, but we provide more details than in [157, 25] about why the transformation is secure. In particular, we explain how an adversary can perform a signature forgery in practice. In the security model of SBP, we assume that an adversary does not know the inner secret-key, but knows the inner public-key and so the Merkle tree. This implies that he can always provide a correct $(h', pz_{i_1}, \ldots, pz_{i}, L')$. This assumption is important, because each valid signature reveals information about the inner public-key. In particular, an adversary can recover p by multivariate interpolation with at least $\frac{N}{\ }$ distinct valid signatures, thanks to the inner signatures. He can also recover $\hat{p}$ (and so p) by univariate interpolation with at least $\frac{D_{\mathtt{MAC}}+1}{\ }$ distinct valid signatures, thanks to the evaluations of $\hat{p}$.

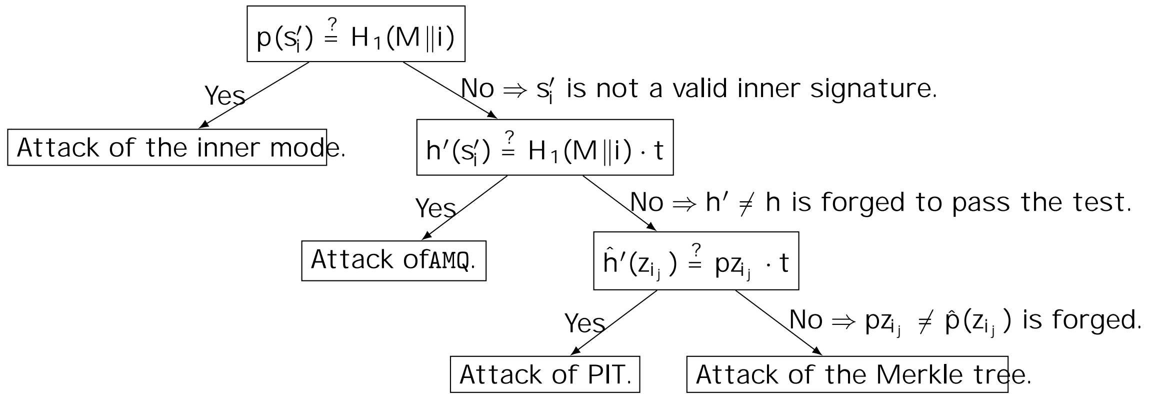

Figure 3.2: Security model of the dual mode. We verify $= (s'_0, \ldots, s'_{-1}, h', pz_{i_1}, \ldots, pz_{i}, L')$ provided by an adversary. PIT means Polynomial Identity Testing (Equations (3.2) and (3.3)).

Now, we present possible attacks with this assumption.

- The adversary forges correct inner signatures, and provides a correct $(h', pz_{i_1}, \ldots, pz_{i}, L')$. This forgery requires attacking the inner mode, which should have a -bit security level. Automatically, the three tests of the verifying process (Algorithm 11) are passed with success.

- The adversary provides incorrect random inner signatures and a correct $(h', pz_{i_1}, \ldots, pz_{i}, L')$. In this case, the probability of passing the test $h'(s'_i) \stackrel{?}{=} H_1(M\|i) \cdot t$ is q^{-}. The adversary can also choose a non-negative integer r min$(\ ,m)$ and forges $s'_0, \ldots, s'_{-1}$ by solving the instance AMQ$(q, \ , m, n_{\mathtt{var}}, r+1)$ associated to p and $y = H_1(M\|i) \cdot t$. This implies that each forged inner signature can be generated from a basis of $r+1$ signatures. As a consequence, if these $r+1$ signatures pass the test, then the signatures pass the test. This occurs with probability $q^{-(r+1)}$.

- The adversary provides incorrect inner signatures, forges h' by multivariate interpolation to pass the first test, provides random values $pz_{i_1}, \ldots, pz_{i}$ and a correct L'. In this case, $h' \neq p \cdot t$ implies that at least one univariate polynomial of $\hat{h}' - \hat{p} \cdot t$ is di erent from zero. Its maximum number of roots is bounded by its degree. So, the maximum probability that $\hat{h}' - \hat{p} \cdot t$ vanishes on distinct random points is

$$\prod_{i=1} \frac{\frac{N}{} - i}{+1-i}, \tag{3.2}$$

which is bounded by the maximum probability that $\hat{h}' - \hat{p} \cdot t$ vanishes on random points (not necessarily distinct), i.e.

$$\frac{\frac{N}{} - 1}{}. \tag{3.3}$$

Since L' is correct, the last test is always passed with success.

- The adversary provides incorrect inner signatures, and forges $(h', pz_{i_1}, \dots, pz_{i} , L')$. As previously, a multivariate interpolation gives h'. Then, he can forge $pz_{i_1}, \dots, pz_{i}$ as a solution of the classical linear algebra problem $\hat{h}'(z_{i_j}) = pz_{i_j} \cdot t$ for $1 \leqslant j \leqslant$. Finally, the authentication paths have to be forged to lead to a root equal to the public-key. This implies attacking the Merkle tree (Section 3.3) and so the underlying hash function. The latter should have a 2 -bit security level against the second preimage search.

We deduce from the previous attacks (Theorem 2) a method to choose the parameters. Firstly, , and have to be chosen such that Equation (3.3) is lower bounded by 2 . Secondly, and have to be chosen such that $q^{\ }$ is lower bounded by 2^{-}, assuming the instance of AMQ is hard to solve for $r < \min(, m)$ (Section 4.5). Naturally, this assumption is true for $=1$ since AMQ with $r = 0$ becomes an instance of PoSSo (Problem 1).

3.6 SBP Transformation of MI-Based Signature Schemes

In Table 3.1, we summarize the practical sizes of the SBP transformation [157, 25] on MI-based schemes.

Dual mode of HFEv-. Gui [144] is an HFEv--based signature scheme (Section 2.4.1). Its dual transformation allows to obtain a dual signature nine or ten times smaller than the inner public-key. However, this result relies on stronger assumptions since $=1$ (Theorem 2). The security of Gui is based on a new parameter, the _ite, that we will study in Section 4.2. However, we note that the designers of Gui set this parameter to four, whereas SBP [157] set this parameter to one (this parameter is not mentioned in [157], but we can deduce its value). This makes insecure the inner layer.

Public-key compression from a seed. Here, we present the method to compress a part of the public-key from a seed, for UOV (Section 2.4.2) and Rainbow (Section 2.4.3). This method is used by SBP [157, 25].

In [141, 142], the authors proposed cyclic UOV, which is based on a compression trick to reduce the public-key size of the UOV-based schemes. The main idea[1] is to generate a part of the public-key from a public seed. Then, we can deduce $\mathcal{F}$ by evaluation-interpolation principle (Equation (2.1)), and finally compute the remaining part of the public-key. By using this method on the quadratic terms of p, up to $\frac{v_1(v_1+1)}{2} + v_1 o_1$ monomials can be saved. This bound is exactly the number of quadratic terms of $\mathcal{F}$: the evaluation-interpolation principle implies that each fixed term in $p = \mathcal{F} \circ \mathcal{S}$ implies fixing a term in $\mathcal{F}$.

Then, this principle was extended for $u = 2$ with cyclic Rainbow [143]. Based on the previous compression trick, this method saves about $\frac{v_1(v_1+1)}{2} + v_1 o_1$ elements of $\mathbb{F}_q^m$ (similarly to cyclic UOV) and $\frac{o_1(o_1+1)}{2}$ elements of $\mathbb{F}_q^{o_2}$. The authors noted that this compression trick can be generalized for $u > 2$.

[1] In fact, a partially cyclic public-key was used. A part of the terms from the first equation were randomly sampled and stored in the public-key. Then, the corresponding terms in other equations were computed via a circular shift of the previous equation (and were not stored). Unlike the partially cyclic public-key, the generation of a non structured public-key from a public seed [64], sometimes called circumzenithal variant [65], does not have any impact on the security.

Dual UOV and dual Rainbow. In [157], the authors noted an interesting fact: the SBP transformation is perfectly compatible with the cyclic version of `UOV`. We can use it to generate $\frac{v_1(v_1+1)}{2} + v_1 o_1$ monomials from a public seed, whereas the remaining part of the public-key can be compressed with the SBP transformation. In this case, we can apply Equations (3.1), (3.2) and (3.3) with $N - \frac{v_1(v_1+1)}{2} - v_1 o_1$ instead of N, which generates smaller parameters and signature sizes. As mentioned in [25], this is also true for `Rainbow`. The only difference is that the SBP transformation is (partially) compatible with cyclic `Rainbow`. The compression trick allows to compress more than $\frac{v_1(v_1+1)}{2} + v_1 o_1$ monomials, but this would not be compatible with the SBP transformation. This is due to the dual verifying process (Algorithm 11). The monomials which are not present in h have to be generated from the public seed. We need to know the monomials from each equation, in order to complete h. In the case of `Rainbow`, the public seed allows to generate the $\frac{o_1(o_1+1)}{2}$ monomials of the last o_2 equations. This is not enough to generate the corresponding monomials in h.

In Table 3.1, the dual transformation is applied to `UOVrand` [139], which is cyclic. The dual transformation of `UOVrand` allows to obtain a dual signature two times smaller than the inner public-key. Unlike dual `Gui`, this transformation is proven secure without assumptions relied on the hardness of `AMQ` (Theorem 1). For dual `Rainbow-IIIc`, we have the same result. We note that the compression trick was not used in the original `Rainbow-IIIc` scheme [63] from the first round of the NIST PQC standardization process. Finally, another version of dual `Rainbow-IIIc` is proposed, based on stronger security assumptions. Here, the factor between the inner public-key size and the dual signature size is eleven, which is similar to dual `Gui`.

Public-key compression of constrained linear signature schemes. In [25], the authors generalize the SBP transformation for constrained linear signature schemes. The SBP transformation can be applied to other categories of signature schemes such as code-based schemes and lattices-based schemes, under the assumption that the public-key can be represented as an affine system. Of course, this assumption is true for the `MI`-based schemes since the public-key is the sum of N monomials. In [25], the SBP transformation is applied to `Rainbow`. The proposed parameters target a 128-bit quantum security level (we refer to Section 8.5.1 about quantum EUF-CMA). The inner scheme used is `Rainbow-IIIc`, which targets at least a 192-bit classical security level, and at least a 128-bit quantum security level. However, the authors did not have an efficient implementation permitting to adapt the parameters to a practical application. So, we propose an implementation of the dual mode of `Rainbow` in `MQsoft` (Chapter 9) to have practical running times, and we use them to set the parameters.

Smaller signatures. In Section 8.5.5, we show that by removing redundant digests in the authentication paths, we can slightly decrease the signature size. By applying this idea on Table 3.1, we obtain the following signature sizes (via Equation (8.8)):

- 5.48 kB and 11.6 kB respectively for dual `Gui-94` and dual `Gui-127`,
- 24.1 kB, 76.6 kB and 179.3 kB for dual `UOVrand`,
- 179.2 kB and 29.0 kB for dual `Rainbow-IIIc`.

scheme	security parameters	sec. lvl.	|pk|	|sign|
Gui-94 (HFEv-)	$q = 2, d_{ext} = 94, D = 17$ $= 4, v = 4$, nb_ite = 1 (1)	80	54.6 kB (2)	98 bits
dual Gui-94	= 80, = 1, = 21, = 7	80?	160 bits	6.04 kB (3)
Gui-127 (HFEv-)	$q = 2, d_{ext} = 127, D = 9$ $= 4, v = 6$, nb_ite = 1 (1)	120	137.0 kB (2)	133 bits (4)
dual Gui-127	= 120, = 1, = 21, = 11	120?	240 bits	13.3 kB (3)
UOVrand	$q = 256, v_1 = 90, o_1 = 45$	128	46.6 kB	1080 bits
dual UOVrand	= 1, = 16, = 3, = 12	128	384 bits	26.0 kB (3)
UOVrand	$q = 256, v_1 = 140, o_1 = 70$	192	174.0 kB	1680 bits
dual UOVrand	= 1, = 24, = 3, = 19	192	576 bits	82.1 kB (3)
UOVrand	$q = 256, v_1 = 190, o_1 = 95$	256	433.2 kB	2280 bits
dual UOVrand	= 1, = 32, = 3, = 28	256	768 bits	190.0 kB (3)
Rainbow-IIIc	$q = 256, v_1 = 68, o_1 = o_2 = 36$	128	365.5 kB	1120 bits (5)
dual Rainbow-IIIc	= 1, = 32, = 3, = 25	128	512 bits	184.0 kB
dual Rainbow-IIIc	= 16, = 2, = 3, = 25	128?	512 bits	33.8 kB

(1) The original scheme [144] considers nb_ite = 4, but the signature size from [157] indicates that the authors consider nb_ite = 1.
(2) The public-key sizes of [144, Table 9] are wrong. We have corrected them.
(3) The formula of signature size in [157] is wrong, and was corrected in [25]. We have updated the signature sizes accordingly (cf. Equation (3.1)).
(4) We have corrected the signature size from [157], which was 123 bits.
(5) The 128-bit salt is not considered here.

Table 3.1: Comparison of the inner and outer layers of MI-based schemes, as proposed in [157, 25]. For all parameter sets, $= 2^{20}$ and $= 0$. We note that for UOVrand [139] and Rainbow-IIIc [63], the authors of [157, 25] only considered the quadratic terms of the inner public-key, unlike original schemes. Rainbow-IIIc targets a 192-bit classical security level and a 128-bit quantum security level. It is used by compressing $\frac{v_1(v_1+1)}{2} + v_1 o_1$ monomials from a public seed of 32 bytes, and without the 128-bit salt in the signature. We have also corrected some errors from [157]. In particular, we have replaced the digest size by 2 bits instead of bits.

Chapter 4

Cryptanalysis Techniques

This chapter provides a summary of the main attacks against the HFE-based signature schemes (Section 2.3), as well as the known attacks against the SBP transformation (Chapter 3). In cryptography, attacks against signature schemes are traditionally divided into four categories. An adversary who can perform one of them can perform all the previous ones.

- Existential forgery. An adversary can generate a valid couple message-signature.
- Selective forgery. An adversary can select a message prior to the attack, then forge its signature.
- Universal forgery. An adversary can sign any message.
- Total break. An adversary recovers the secret-key for a given public-key, allowing to sign any message.

In Section 4.1, we present several metrics to evaluate the cost of attacks. Then, we consider existential forgery for signature schemes having a very short signature (Section 4.2). In Sections 4.3, 4.4 and 4.5, we study selective forgery and universal forgery by inverting the public-key for a fixed message. This includes, in particular, the analysis of known quantum attacks (Sections 4.3.1 and 4.4.2) and Gröbner basis attacks (Section 4.4). In Section 4.6, we consider the total break with key-recovery attacks, including solving the MinRank problem (Problem 5). Finally, the physical information leakages of an implementation can lead to side-channel attacks (Section 4.7), going to the total break.

The attacks against the HFE-based signature schemes (Section 2.4.1) that we here present will allow to evaluate the security and set precisely the parameters of our NIST submissions: GeMSS (Chapter 7) and DualModeMS (Chapter 8).

4.1 Metrics and NIST Security Strength Categories

The security of a cryptosystem is evaluated by counting the number of operations required to forge a valid signature. These operations can be for example evaluations of a certain map, arithmetic operations or binary operations. In general, the security of a cryptosystem is quantified by the

number of bit operations, or logical gates, required to forge a valid signature. A cryptosystem reaches a -bit security level if the best attack requires at least 2 bit operations. The cost of an attack is also quantified by the maximal quantity of memory used to perform it, and by the number of calls to an oracle of signature in the case of existential forgery under chosen message attack.

The security against quantum computers is slightly different from classical (binary) computers. The cost of quantum algorithms is measured by counting the number of quantum logical gates (similarly to classical gates on binary architectures). It can also be important to count the qubits (quantum bits) required by these algorithms. The number of available qubits on quantum architectures is so far limited, due to questions about instabilities and quantum entanglement.

In the NIST PQC standardization process [127], six categories of security strength are introduced. NIST defines these categories with respect to existing NIST standards in symmetric cryptography (Table 4.1).

security strength	security description
I	At least as hard to break as AES128 (exhaustive key search)
II	At least as hard to break as SHA3-256 (collision search)
III	At least as hard to break as AES192 (exhaustive key search)
IV	At least as hard to break as SHA3-384 (collision search)
V	At least as hard to break as AES256 (exhaustive key search)
VI	At least as hard to break as SHA3-512 (collision search)

Table 4.1: NIST security strength categories.

The definition of these levels requires specifying the security level of AES and SHA-3. We summarize in Table 4.2 the estimations given by NIST [127], for both classical security and quantum security.

security strength	underlying standard	classical attack	quantum attack
I	AES128	2^{143} gates	2^{170}/MAXDEPTH gates
II	SHA3-256	2^{146} gates	
III	AES192	2^{207} gates	2^{233}/MAXDEPTH gates
IV	SHA3-384	2^{210} gates	
V	AES256	2^{272} gates	2^{298}/MAXDEPTH gates
VI	SHA3-512	2^{274} gates	

Table 4.2: Estimation of NIST security strength categories. MAXDEPTH is the circuit depth.

For the classical security, exhaustive key search (respectively collision search) requires 2^{128} calls to AES128 (respectively SHA3-256) for the level I (respectively II). Then, the number of gates is obtained by multiplying 2^{128} by the cost of evaluating AES128 (respectively SHA3-256). The method is similar for other levels. The cost of evaluating is estimated to 2^{15} (classical) gates for AES128 and AES192, 2^{16} gates for AES256, and 2^{18} gates for SHA-3.

For the quantum security, the cost of exhaustive search against AES is given in [98]. In Table 4.2, the number of quantum gates is multiplied by the circuit depth (both from [98, Table 5]), then the result is divided by MAXDEPTH. NIST takes into account that too large circuit depths may not be usable in practice, due to quantum instabilities. Instead, several smaller circuits can be run in parallel. The parameter MAXDEPTH is introduced to quantify a fixed circuit depth, or running time. NIST considers that this value can be chosen between 2^{40} and 2^{96}, in function of certain assumptions on the running time.

4.2 Generic Attack and Feistel–Patarin Construction

The MI-based signature schemes presented in Section 2.3 follow the hash-and-sign paradigm. Let $G : \mathbb{F}_q^{n_{\mathtt{var}}} \to \mathbb{F}_q^m$ be a trapdoor function and $H_1 : \{0,1\}^* \to \mathbb{F}_q^m$ be a hash function. Roughly, the signature $sm \in \mathbb{F}_q^{n_{\mathtt{var}}}$ of a message $d \in \{0,1\}^*$ is obtained as

$$sm = G^{-1} H_1(d) .$$

When $m \log_2(q)$ is strictly less than 2 , these schemes are vulnerable to collision attacks on the hash function. As proposed in [58], the attack requires generating from random inputs, a first hash table of $q^{\frac{1}{2}m}$ hash values and a second hash table of $q^{\frac{1}{2}m}$ evaluations of G. Thanks to the birthday paradox, the intersection of these tables should be non-empty, creating an existential forgery since we obtain $(x, y) \in \mathbb{F}_q^{n_{\mathtt{var}}} \times \{0,1\}^*$ such that $G(x) = H_1(y)$. This method requires computing $q^{\frac{1}{2}m}$ evaluations of G and H_1. The memory cost has the same order of magnitude (note that only one hash table really needs to be stored). To circumvent this issue, we can choose $m \log_2(q) \geqslant 2$, but this choice implies lower bounding the public-key sizes which are large, as well as the signature sizes which could be even smaller.

In order to avoid both the attack and such a countermeasure, J. Patarin introduced an iterative construction [133]. The latter was extended by [58] and called Feistel–Patarin construction. Based on the Feistel scheme, the inversion process is repeated nb_ite times as explained in Algorithm 12. The signature size is $m + nb_ite \cdot (n_{\mathtt{var}} - m) \cdot \log_2(q)$ bits.

Algorithm 12 MI-based signature process using the Feistel–Patarin construction.

```
1: function sign_Feistel_Patarin M ∈ {0,1}*, G^{-1} : F_q^m → F_q^{n_var}
2:     S_0 ← 0_m
3:     for i from 0 to nb_ite − 1 do                Iterations with the Feistel–Patarin scheme.
4:         D_{i+1} ← H_1(M‖i)                                              D_{i+1} ∈ F_q^m.
5:         S_{i+1}, X_{i+1} ← G^{-1} S_i − D_{i+1}       S_{i+1} ∈ F_q^m, X_{i+1} ∈ F_q^{n_var − m}.
6:     end for
7:     return sm = S_{nb_ite}, X_{nb_ite}, X_{nb_ite−1}, . . . , X_1
8: end function
```

The corresponding verification process is given in Algorithm 13.

Algorithm 13 MI-based verification process using the Feistel–Patarin construction.

1: **function** verify_Feistel_Patarin$\left(M \in \{0,1\}^*, sm \in \mathbb{F}_q^{m+nb_ite\cdot(n_{var}-m)}, G: \mathbb{F}_q^{n_{var}} \rightarrow \mathbb{F}_q^m\right)$
2: $\quad\left(S_{nb_ite}, X_{nb_ite}, X_{nb_ite-1}, \ldots, X_1\right) \leftarrow sm$
3: $\quad$ **for** i **from** nb_ite-1 **to** 0 **by** -1 **do**
4: $\quad\quad D_{i+1} \leftarrow H_1(M\|i)$ $\quad\quad$ ▷ $D_{i+1} \in \mathbb{F}_q^m$.
5: $\quad\quad S_i \leftarrow G\left(S_{i+1}, X_{i+1}\right) + D_{i+1}$
6: $\quad$ **end for**
7: $\quad$ **return** VALID if $S_0 = 0_m$ and INVALID otherwise
8: **end function**

In [58], the author proved that an attack can be mounted in

$$O\left(q^{\frac{nb_ite}{nb_ite+1}m}\right) \text{ evaluations of } G \text{ and } H_1,$$

with a memory cost of

$$O\left(q^{\frac{nb_ite}{nb_ite+1}m} n_{var} \log_2(q)\right).$$

This generic attack was firstly proposed by J. Patarin for $nb_ite = 2$ [133, Remark 2]. He also noted the possibility to increase the number of iterations [133, Remark 3], and its impact on the generic attack.

In HFEv--based signature schemes such as QUARTZ, Gui and GeMSS (Chapter 2), the Feistel–Patarin construction is used because the signature size is very short. In Section 7.7.2, we propose a more accurate study of the generic attack, slightly decreasing its cost. Our results allow to improve the performance of GeMSS.

4.3 Direct Signature Forgery Attacks

The public-key of MI-based signature schemes is given by a set of non-linear equations $\mathbf{p} = (p_1, \ldots, p_m) \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$ (Chapter 2). Given a digest $(d_1, \ldots, d_m) \in \mathbb{F}_q^m$, the problem of forging a signature (Section 2.3.2) is equivalent to solving the following system of non-linear equations:

$$\begin{aligned} &p_1(x_1, \ldots, x_{n_{var}}) - d_1 = 0, \ldots, p_m(x_1, \ldots, x_{n_{var}}) - d_m = 0, \\ &x_1^q - x_1 = 0, \ldots, x_{n_{var}}^q - x_{n_{var}} = 0, \end{aligned} \tag{4.1}$$

where $x_i^q - x_i = 0$ for $1 \leqslant i \leqslant n_{var}$ are called field equations, because it is equivalent to $x_i \in \mathbb{F}_q$. Stated differently, the task is to invert the public map without the knowledge of the secret-key.

For HFE-based schemes (Section 2.4.1) and Rainbow (Section 2.4.3), the system is under-defined, i.e. $n_{var} > m$. As a consequence, we can randomly fix $n_{var} - m$ variables in (4.1) (let $\mathbf{r} = (r_1, \ldots, r_{n_{var}-m}) \in \mathbb{F}_q^{n_{var}-m}$ be these random values) and try to solve for the remaining variables. Note that this is similar to the (legitimate) signature process which requires randomly fixing variables.

Thus, the problem of forging a signature is reduced to solving a system of m quadratic equations in m variables over $\mathbb{F}_q$:

$$\begin{aligned} &p_1(x_1,\ldots,x_m,r) - d_1 = 0,\ldots,p_m(x_1,\ldots,x_m,r) - d_m = 0,\\ &x_1^q - x_1 = 0,\ldots,x_m^q - x_m = 0. \end{aligned} \tag{4.2}$$

4.3.1 Exhaustive Search

Classical Exhaustive Search. Equation (4.2) can be solved by evaluating the system in the q^m possible inputs, then by verifying if each evaluation is equal to the digest. This requires $O(q^m)$ operations in $\mathbb{F}_q$.

In [38], the authors describe a faster exhaustive search for solving systems of boolean quadratic equations. They also provide a detailed cost analysis of their approach. To recover a solution of (4.2), the approach from [38, Theorem 2] requires:

$$4\log_2(m)\cdot 2^m \text{ binary operations} \tag{4.3}$$

Quantum Exhaustive Search. Exhaustive search can be improved with quantum computers. Grover's algorithm [102] is a quantum algorithm which, given a function having a domain of size k, and a target output, returns an input producing this output with only $\sqrt{k}$ evaluations of this function. Grover's algorithm requires $O(q^{\frac{m}{2}})$ evaluations of the public polynomials p to solve (4.2).

[150] demonstrated that we can solve a system of $m-1$ binary quadratic equations in $n_{\mathtt{var}}-1$ binary variables using $m+n_{\mathtt{var}}+2$ qubits and evaluating a circuit of:

$$2^{\frac{n_{\mathtt{var}}}{2}}\cdot\left(2m\cdot\left(n_{\mathtt{var}}^2+2n_{\mathtt{var}}\right)+1\right) \text{ quantum gates.} \tag{4.4}$$

They also describe a variant using less qubits, i.e., $3+n_{\mathtt{var}}+\lceil\log_2(m)\rceil$ qubits, but requiring to evaluate a larger circuit, i.e. with:

$$\approx 2\times 2^{\frac{n_{\mathtt{var}}}{2}}\cdot\left(2m\cdot\left(n_{\mathtt{var}}^2+2n_{\mathtt{var}}\right)+1\right) \text{ quantum gates.} \tag{4.5}$$

4.3.2 Approximation Algorithm

The authors of [117] proposed an algorithm for solving systems of non-linear equations that is faster than a direct exhaustive search. The technique from [117] uses an approximation of a non-linear system, such as (4.2), by a single high-degree multivariate polynomial $P \in \mathbb{F}_q[x_1,\ldots,x_{m_0}]$ with $m_0 < m$. The polynomial P is constructed such that it vanishes on the same zeros as the original non-linear system with high probability. We then perform an exhaustive search on P to recover, with high probability, the zeros of the non-linear system. This leads to an algorithm for solving (4.2) whose asymptotic complexity is

$$O^*\left(2^{0.8765m}\right) \tag{4.6}$$

when $q=2$ [117, Theorem 1.1]. The O^* notation omits polynomial factors.

4.4 Gröbner Bases

To date, the best methods for solving non-linear equations, including the signature forgery (System (4.1)), utilize Gröbner bases [45, 46]. The historical method for computing such bases – known as Buchberger's algorithm – has been introduced by B. Buchberger in his PhD thesis [45, 46]. Many improvements on Buchberger's algorithm have been done leading – in particular – to more efficient algorithms such as the F4 and F5 algorithms of J.-C. Faugère [76, 77]. The F4 algorithm, for example, is the default algorithm for computing Gröbner bases in the computer algebra software `Magma` [34]. The F5 algorithm, which is available through the `FGb` [79] software[1], provides today the state-of-the-art method for computing Gröbner bases.

Besides F4 and F5, there is a large literature of algorithms computing Gröbner bases. We mention for instance `PolyBory` [43] which is a general framework to compute Gröbner basis in $\mathbb{F}_2[x_1, \ldots, x_{n_{var}}]/\langle x_i^2 - x_i \rangle_{1 \leqslant i \leqslant n_{var}}$. It uses a specific data structure – dedicated to the Boolean ring – for computing Gröbner basis on top of a tweaked Buchberger's algorithm[2]. Another technique proposed in cryptography is the `XL` algorithm [60]. It is now clearly established that `XL` is a special case of Gröbner basis algorithm [7]. More recently, a zoo of algorithms such as `G2V` [91], `GVW` [92], ... flourished building on the core ideas of F4 and F5. This literature is vast and we refer to [75] for a recent survey of these algorithms.

Despite this important algorithmic literature, it is fair to say that `Magma` and `FGb` remain the reference softwares for polynomial system solving over finite fields. We have chosen `Magma` to perform practical experiments requiring Gröbner basis computations.

4.4.1 Practically Fast Algorithms

The direct attack described in [78, 82] provides reference tools for evaluating the security of `HFE` and `HFEv-` against a direct message-recovery attack. This attack uses the F5 algorithm [77, 12] and has the following complexity:

$$O\left(\mathrm{poly}(m, n_{var})^{\omega \cdot D_{reg}}\right), \tag{4.7}$$

with $2 \leqslant \omega < 3$ being the so-called *linear algebra constant* [161], *i.e.* the smallest constant ω such that two matrices of size $N \times N$ over a field $\mathbb{F}$ can be multiplied in $O(N^\omega)$ arithmetic operations over $\mathbb{F}$. The best current bound is $\omega < 2.3728639$ [90]. The $\mathrm{poly}(m, n_{var})$ notation means the complexity is polynomial in m and n_{var}.

Complexity (4.7) is exponential in the *degree of regularity* D_{reg} [9, 14, 11]. However, this degree of regularity can be difficult to predict in general; as difficult as computing a Gröbner basis. Fortunately, there is a particular class of systems for which this degree can be computed efficiently and explicitly: *semi-regular sequences* [9, 14, 11]. This notion is supposed to capture the behavior of a random system of non-linear equations. In order to set the parameters for `HFE` and variants as well as for performing meaningful experiments on the degree of regularity, we can assume that no algebraic system has a degree of regularity higher than a semi-regular sequence.

In $\mathbb{F}_2$, the degree of regularity of a semi-regular system of m quadratic boolean equations in n_{var} variables is the smallest index i such that the term z^i of the Hilbert series G (Equation (4.8)) is non-positive (*i.e.* negative or null) [14, Proposition 10].

$$G(z) = \frac{(1+z)^{n_{var}}}{(1+z^2)^m}. \tag{4.8}$$

[1] http://www-polsys.lip6.fr/~jcf/FGb/index.html

[2] http://polybori.sourceforge.net

In Table 4.3, we provide these degrees of regularity when $n_{var} = m$, for various values of m.

m	D_{reg}
$4 \leqslant m \leqslant 8$	3
$9 \leqslant m \leqslant 15$	4
$16 \leqslant m \leqslant 23$	5
$24 \leqslant m \leqslant 31$	6
$32 \leqslant m \leqslant 40$	7
$41 \leqslant m \leqslant 48$	8
$49 \leqslant m \leqslant 57$	9
$58 \leqslant m \leqslant 66$	10
$154 \leqslant m \leqslant 163$	20
$234 \leqslant m \leqslant 243$	28
$316 \leqslant m \leqslant 325$	36

Table 4.3: Degree of regularity of m semi-regular quadratic boolean equations in m variables. The field equations are used.

In the case of HFE, when $q = 2$, the degree of regularity for solving (4.2) has been experimentally shown to be smaller than $\log_2(D)$ [78, 82]. This behavior has been further demonstrated in [96, 74]. In particular, [96, Theorem 1] claims that the degree of regularity reached in HFE is asymptotically upper bounded by:

$$2 + \quad \cdot \quad 1 - \overline{3/4} \quad \cdot \min \ m, \log_2(D) \ , \text{ for all } \ > 0. \qquad (4.9)$$

This bound is obtained by estimating the degree of regularity of a semi-regular system of $3\lceil\log_2(D)\rceil$ quadratic equations in $2\lceil\log_2(D)\rceil$ variables. We emphasize that an asymptotic bound such as (4.9) is not necessarily tight for specified values of the parameters. Thus, (4.9) cannot be directly used to derive actual parameters but still provides a meaningful asymptotic trend.

Indeed, the behavior of HFE algebraic systems is then much different from a semi-regular system of m quadratic boolean equations in m variables where the degree of regularity increases linearly with m. Roughly, D_{reg} grows like $m/11.11$ in the semi-regular case [9, 14, 11].

We report below the degree of regularity D_{reg}^{Exp} observed in practice for HFE. These bounds are only meaningful for a sufficiently large m which is given in the first column. Indeed, as we already explained, we can assume that the values from Table 4.3 are upper bounds on the degree of regularity of any algebraic system of boolean equations.

m	HFE degree D	D_{reg}^{Exp}
$\geqslant 4$	$3 \leqslant D \leqslant 16$	3
$\geqslant 9$	$17 \leqslant D \leqslant 128$	4
$\geqslant 16$	$129 \leqslant D \leqslant 512$	5
$\geqslant 24$	$513 \leqslant D \leqslant 4096$	6
$\geqslant 32$	$4097 \leqslant D$	$\geqslant 7$

Table 4.4: Degree of regularity in the case of HFE algebraic systems.

Following [82], we lower bound the complexity of F5 against HFE, i.e. for solving System (4.2). The principle [14] is to only consider the cost of performing a row-echelon computation on a full-rank sub-matrix of the biggest matrix occurring in F5. At the degree of regularity, this sub-matrix has $\binom{m}{D_{reg}}$ columns and (at least) $\binom{m}{D_{reg}}$ rows. Thus, we can bound the complexity of a Gröbner basis computation against HFE by:

$$O\left(\binom{m}{D_{reg}}\right) \text{ operations in } \mathbb{F}_2. \tag{4.10}$$

This is a conservative estimate on the cost of solving (4.2). This represents the minimum computation that has to be done in F5. We can also assume that the linear algebra constant is 2, the smallest possible value.

From Table 4.4, we can see that HFE has a degree of regularity sufficiently large to be used in practical applications. For $D_{reg} = 7$, more than 2^{20} equations and variables are required to reach a 128-bit level of security, implying a public-key size of 2^{58} bits. So, we need to use modifiers of HFE (Section 2.4.1) for increasing the degree of regularity.

In particular, the practical effect of the minus and vinegar modifiers have been considered in [78, 82]. This has been further investigated in [67, 71] who presented a theoretical upper bound on the degree of regularity arising in HFEv-. Let be the number of minus equations, v the number of vinegar variables and $R_q = \lfloor \log_q(D-1) \rfloor + 1$. Then, the degree of regularity for HFEv- is bounded from above by

$$\begin{cases} \frac{(q-1)(R_q + \;\; + \;\; v - 1)}{2} + 2, & \text{if } q \text{ is even and } R_q + \;\; \text{ is odd,} \\ \frac{(q-1)(R_q + \;\; + \;\; v)}{2} + 2, & \text{otherwise} \end{cases} \tag{4.11}$$

We observe that the degree of regularity seems to increase linearly with $n_{var} - m$. This is the sum of the modifiers: number of minus equations plus number of vinegar variables.

[140] derived an experimental lower bound on the degree of regularity of HFEv-. The authors [140] obtained that the degree of regularity for HFEv- when $q = 2$ should be at least:

$$\left\lfloor \frac{R_2 + \;\; + \;\; v + 1}{3} \right\rfloor + 2. \tag{4.12}$$

4.4.2 Asymptotically Fast Algorithms

BooleanSolve [13] is the fastest asymptotic algorithm for solving systems of non-linear boolean equations. BooleanSolve is a hybrid approach that combines exhaustive search and Gröbner bases techniques. For a system with the same number of equations and variables m, the deterministic variant of BooleanSolve has complexity bounded by $O(2^{0.841m})$, while a Las Vegas variant has expected complexity

$$O\left(2^{0.792m}\right). \tag{4.13}$$

It is mentioned in [13] that BooleanSolve is better than exhaustive search when $m \geq 200$. This is due to the fact that large constants are hidden in the big O notation. As a conservative choice, we lower bound the cost of this attack by $2^{0.792m}$ in GeMSS (Section 7.7.1).

`QuantumBooleanSolve`. In [81], the authors present a quantum version of `BooleanSolve` that takes advantages of Grover's quantum algorithm [102]. `QuantumBooleanSolve` is a Las Vegas quantum algorithm allowing to solve a system of m boolean equations in m variables. It uses $O(m)$ qubits and requires the evaluation of, on average,

$$O\left(2^{0.462m}\right) \text{ quantum gates.} \tag{4.14}$$

This complexity is obtained under certain algebraic assumptions. Note that [18] also proposed a new (Gröbner-based) quantum algorithm for solving quadratic equations with a complexity comparable to `QuantumBooleanSolve` (we refer to [81] for further details).

4.5 Direct Attack against Approximate `PoSSo`

The security of the SBP transformation (Chapter 3) depends on the `AMQ` problem (Problem 3). Here, we generalize `AMQ` by considering degree-D equations instead of quadratic equations. We call `APoSSo` this new problem.

Problem 4. Approximate `PoSSo` (`APoSSo`$(q, \ , m, n_{\mathtt{var}}, D, r)$). Let $q, m, n_{\mathtt{var}}, D,$ and r be non-negative integers such that $r \leqslant \min(\ , m)$. Given p a degree-D multivariate polynomial system in $\mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$ and $y_1, \ldots, y$ in $\mathbb{F}_q^m$, the problem is to find vectors $x_1, \ldots, x$ in $\mathbb{F}_q^{n_{\mathtt{var}}}$ such that the dimension of the vector space generated by $p(x_1) - y_1, \ldots, p(x \ ) - y$ is less or equal to r.

When $r = 0$, it is clear that `APoSSo` corresponds to independent instances of `PoSSo` (Problem 1), and so can be solved as such. When $r > 0$, the authors of [157] present three attacks against `AMQ`, that we generalize for `APoSSo`.

1. Exhaustive search. Randomly choose the values of $x_1, \ldots, x_r$ in $\mathbb{F}_q^{n_{\mathtt{var}}}$. Then, for $r < i \leqslant \ $, randomly sample $x_i \in \mathbb{F}_q^{n_{\mathtt{var}}}$ until $p(x_i) - y_i$ is a linear combination of $p(x_j) - y_j$ for $1 \leqslant j \leqslant r$. A correct value x_i is found with probability $q^{-(m-r)}$. Therefore, this strategy requires roughly $r + (\ - r)q^{m-r}$ evaluations of p and $c \cdot (\ - r)q^{m-r} + O\left(r^2 \cdot (n_{\mathtt{var}} - r)\right)$ operations in $\mathbb{F}_q$, for $c = O\left(r \cdot (n_{\mathtt{var}} - r)\right)$.

2. Quantum exhaustive search. As in Section 4.3.1, the exhaustive search can be accelerated with Grover's algorithm. This strategy requires roughly $\sqrt{\ - r} \cdot q^{\frac{1}{2}(m-r)}$ evaluations of p.

3. Algebraic attack. Here, we write the rank condition for each variable x_i for $r < i \leqslant \ $. For any i, we introduce r new variables $z_1, \ldots, z_r$. Then, we have that $p(x_i) - y_i$ is a linear combination of $p(x_j) - y_j$ means

$$\sum_{j=1}^{r} z_j \left(p(x_j) - y_j\right) + p(x_i) - y_i = 0_m. \tag{4.15}$$

The authors proposed two strategies. We can directly solve the system generated by Equation (4.15) for $r < i \leqslant \ $. This system is an instance of `PoSSo`$(q, (\ - r)m, n_{\mathtt{var}} + (\ - r)r, D + 1)$ (Problem 1). We can also randomly fix the values of $x_1, \ldots, x_r$. After applying a linear transformation, we obtain values of $z_1, \ldots, z_r$ such that r elements of Equation (4.15) vanish

on them. In this case, Equation (4.15) becomes a system of $m-r$ equations in only $n_{\mathtt{var}}$ variables. Applying this strategy for each variable x_i for $r < i \leqslant$, we obtain $-r$ independent instances of PoSSo$(q, m-r, n_{\mathtt{var}}, D)$.

These attacks show that solving PoSSo should be exponential in $m-r$. In Chapter 10, we study this problem more accurately.

4.6 Key-Recovery Attacks against HFE

We conclude attacks against HFE by covering key-recovery attacks. This part discusses the so-called Kipnis–Shamir attack [111] (Section 4.6.3) based on the MinRank problem (Section 4.6.2). The Section 4.6.1 deals with equivalent keys. The existence of equivalent keys does not make dangerous an exhaustive search, but this concept is a practical tool to simplify attacks and implementations via the use of normal forms of the secret-key.

4.6.1 Exhaustive Search and Equivalent Keys

The secret-key of an MI-based scheme can be attacked with an exhaustive search. At first glance, this attack is very inefficient because the size of the key space is very large. Just by considering the affine transformations $\mathcal{S} \in A^{-1}_{n_{\mathtt{var}}}(\mathbb{F}_q)$ and $\mathcal{T} \in A^{-1}_{m'}(\mathbb{F}_q)$, and by using that the number of invertible matrices in $\mathcal{M}_n(\mathbb{F}_q)$ [116] is

$$q^{n^2}\prod_{j=1}^{n}\left(1-q^{-j}\right),$$

we obtain a lower bound on the number of secret-keys, which is slightly less than $q^{n_{\mathtt{var}}^2+n_{\mathtt{var}}+m'^2+m'}$. However, the authors of [165] introduced the notion of equivalent keys. Indeed, we can remark that several distinct secret-keys can lead to the same public-key. In this case, these secret-keys are called equivalent keys.

Definition 2 (Equivalent keys). Two secret-keys $(\mathcal{S},\mathcal{F},\mathcal{T})$ and $(\mathcal{S}',\mathcal{F}',\mathcal{T}')$ in $A^{-1}_{n_{\mathtt{var}}}(\mathbb{F}_q^{n_{\mathtt{var}}}) \times (\mathbb{F}_q^{n_{\mathtt{var}}} \to \mathbb{F}_q^{m'}) \times A^{-1}_{m'}(\mathbb{F}_q^{m'})$ are equivalent if

$$p = \mathcal{T}\circ\mathcal{F}\circ\mathcal{S} = \mathcal{T}'\circ\mathcal{F}'\circ\mathcal{S}'.$$

By using this concept, the authors of [165] demonstrated that for HFEv-based schemes, each secret-key has at least

$$d_{\mathtt{ext}}\cdot q^{+2\ d_{\mathtt{ext}}+vd_{\mathtt{ext}}}\cdot\left(q^{d_{\mathtt{ext}}}-1\right)^2\cdot\prod_{i=0}^{v-1}\left(q^v-q^i\right)\cdot\prod_{i=m-1}^{d_{\mathtt{ext}}-1}\left(q^{d_{\mathtt{ext}}}-q^i\right)\ \text{equivalent keys.}$$

The size of the key space without these equivalent keys is too large to make dangerous an exhaustive search. However, the use of equivalent keys allows to restrict attacks to a specific type of keys. In particular, for each couple of affine transformations $\mathcal{S}'$ and $\mathcal{T}'$, there exists an equivalent secret-key where $\mathcal{S}$ and $\mathcal{T}$ are linear transformations (cf. the additive sustainer in [165]). For this reason, the affine part of $\mathcal{S}'$ and $\mathcal{T}'$ are not considered in both implementation and attack. Moreover, we can use the concept of equivalent keys to choose a monic HFEv- polynomial, as explained in Section 7.4.2.

4.6.2 MinRank

The `MinRank` problem was firstly studied in [47] where authors demonstrated its NP-completeness in $\mathbb{F}_q$. The problem is as follows: given integers q, n, k, r and a matrix M in $\mathcal{M}_n\left(\mathbb{F}_q \cup \{t_1, \ldots, t_k\}\right)$, find, if any, $\tilde{t}_1, \ldots, \tilde{t}_k \in \mathbb{F}_q$ such that

$$\operatorname{rank}\left(M\left(\tilde{t}_1, \ldots, \tilde{t}_k\right)\right) \leqslant r.$$

Then, `MinRank` was re-stated in [57], as follows.

Problem 5. `MinRank` (`MinRank`(q, m, n, k, r)). Let q, m, n, k and r be integers. Let $M_0, M_1, \ldots, M_k$ be matrices in $\mathcal{M}_{m,n}\left(\mathbb{F}_q\right)$. Find, if any, elements $\tilde{t}_1, \ldots, \tilde{t}_k \in \mathbb{F}_q$ such that the rank of

$$M = \sum_{i=1}^{k} \tilde{t}_i M_i - M_0,$$

is at most r.

The `MinRank` problem is crucial for key-recovery attacks against `HFE`. We present in the next section the best known attacks when r is very small compared to $\min(m, n)$.

4.6.3 Kipnis–Shamir Attack

In [111], A. Kipnis and A. Shamir demonstrated that the key-recovery in `HFE` is essentially equivalent to the problem of finding a low-rank linear combination of a set of m matrices of size $n_{\mathtt{var}} \times n_{\mathtt{var}}$ in $\mathbb{F}_q$. This is a particular instance of the `MinRank` problem (Problem 5).

We briefly review the principle of this attack for `HFE`. In the context of this attack, we can assume, without loss of generality, that the `HFE` polynomial has a simpler form:

$$\sum_{\substack{0 \leqslant j \leqslant i < d_{\mathtt{ext}} \\ q^i + q^j \leqslant D}} A_{i,j} X^{q^i + q^j} \in \mathbb{F}_{q^{d_{\mathtt{ext}}}}[X]. \tag{4.16}$$

Let $\underline{X} = \left(X, X^q, X^{q^2}, \ldots, X^{q^{d_{\mathtt{ext}}-1}}\right)$. We can then write (4.16) in matrix form, which is:

$$\underline{X} \cdot F \cdot \underline{X}^T,$$

where $\underline{X}^T$ stands for transpose of $\underline{X}$ and $F \in \mathcal{M}_{d_{\mathtt{ext}}}\left(\mathbb{F}_{q^{d_{\mathtt{ext}}}}\right)$ is a symmetric matrix if q is odd, and is triangular otherwise. Since the degree of F is bounded by D, it is easy to see that F has rank at most $\lceil \log_q(D) \rceil$. This implies that there exists a linear combinations of rank $\lceil \log_q(D) \rceil$ of the public matrices representing the public quadratic forms [22]. Then, the secret-key can be easily recovered from a solution of `MinRank`$(q, d_{\mathtt{ext}}, d_{\mathtt{ext}}, d_{\mathtt{ext}}, \lceil \log_q(D) \rceil)$ [111, 22].

In [22], the authors evaluated the cost of the Kipnis–Shamir key-recovery attack with the best known tools [76, 77] for solving the instance of `MinRank`$(q, d_{\mathtt{ext}}, d_{\mathtt{ext}}, d_{\mathtt{ext}}, \lceil \log_q(D) \rceil)$ that occurs in `HFE`. Following [22], the cost of the Kipnis–Shamir attack against `HFE` can be estimated to:

$$O\left(\binom{d_{\mathtt{ext}} + \lceil \log_q(D) \rceil + 1}{\lceil \log_q(D) \rceil + 1}^{\omega}\right) \simeq O\left(d_{\mathtt{ext}}^{\omega(\lceil \log_q(D) \rceil + 1)}\right),$$

where is the linear algebra constant and D is the degree of the secret univariate polynomial.

When one equation is removed from the public-key, there is a method of reconstructing the removed equation during the Kipnis–Shamir key-recovery attack [160]. Until recently, it was not clear how to apply the key-recovery attack from [111, 22] to HFE- when at least two equations are removed. In [160], the authors explained how to extend MinRank-based key-recovery for all parameters of HFE-. Their results can be summarized as follows. From key-recovery point of view, HFE- with d_{ext} variables based on a secret univariate polynomial of degree D is equivalent to a HFE with m variables based on a secret univariate polynomial of degree $D \cdot 2$. Combining with [22], we obtain a MinRank$(q, d_{ext}, d_{ext}, m, \; + \; \lceil \log_q(D) \rceil)$-based key-recovery attack against HFE- whose cost is then:

$$O\left(\binom{m + \; + \; \lceil \log_q(D) \rceil + 1}{\; + \; \lceil \log_q(D) \rceil + 1}\right) \simeq O\left(m^{(\; + \; \lceil \log_q(D) \rceil + 1)}\right).$$

For MinRank-based key-recovery, the minus modifier has then a strong impact on the security.

In the case of HFEv, one can see that the rank of the corresponding matrix (see, for example [144]) will be increased by the number of vinegar variables. Combining with the previous result, the cost of solving MinRank in the case of HFEv- is then:

$$O\left(\binom{m + v + \; + \; \lceil \log_q(D) \rceil + 1}{v + \; + \; \lceil \log_q(D) \rceil + 1}\right) \simeq O\left(m^{(v + \; + \; \lceil \log_q(D) \rceil + 1)}\right), \tag{4.17}$$

where D is the degree of the secret univariate polynomial. Here, we consider an instance of MinRank$(q, d_{ext} + v, d_{ext} + v, m, v + \; + \; \lceil \log_q(D) \rceil)$.

4.7 Side-Channel Attacks

In the previous sections, we study the hardness of solving mathematical problems. This is not enough to guarantee security because hardware and software implementations can have physical information leakages. Side-channel attacks are based on these leakages. They can be leaded, for example, by exploiting power consumption measurements, or by generating fault injections [32, 8]. A famous side-channel attack [113] breaks the implementation of the square-and-multiply exponentiation algorithm (Algorithm 14) used in the RSA cryptosystem. Let K be a secret exponent. At each step of Algorithm 14, a square is computed. But if i-th bit of K is one, an extra multiplication is performed. Thus, the number of operations depends on the secret-key. We can then try to exploit time measurements or power consumption to recover the secret-key. For the power consumption attack, the adversary requires a physical access to obtain measurements. For a timing attack, these measurements can be obtained remotely, for example by measuring the response time of a server. This attack can seem unrealistic because of different instabilities (e.g. fluctuation of the Internet connection, swap of the current process). In 2005, D. Brumley and D. Boneh [44] succeed to remotely attack the RSA implementation of the famous OpenSSL library.

In Algorithm 14, a timing attack is based on the fact that a conditional statement depends on a secret integer, generating a difference in the time used to perform one step. Another famous type of timing attack is the cache-timing attack. Here, the timing leakage is due to the delay of a memory access (Section 6.1.2), which can be used to recover the index of a lookup table. If the memory access depends on secret data, then the secret can be compromise. This type of attacks was used to break the `OpenSSL` library in 2005 [16].

Algorithm 14 Left-to-right square-and-multiply exponentiation. $\mathcal{R}$ is a ring.

```
1:  function Exp(A ∈ R, K ∈ ℕ*)
2:      A_K ← A
3:      for i from ⌊log_2(K)⌋ − 1 to 0 by −1 do
4:          A_K ← A_K^2                                         ▷ Square.
5:          if ⌊K / 2^i⌋ mod 2 = 1 then                         ▷ Extraction of the i-th bit of K.
6:              A_K ← A_K × A                                   ▷ And multiply.
7:          end if
8:      end for
9:      return A_K                                              ▷ A^K.
10: end function
```

Nowadays, the timing attacks are considered as very dangerous, and are still present. In 2019, the authors of [3] broke the keypair generation of the `RSA` implementation from the `OpenSSL` library. To do it, they used the fact that the operations performed during the binary GCD algorithm depend on the binary representation of the inputs. They motivated the danger of their attack by a typical application: the generation of certificates for the websites using the `TLS` protocol. On well-known and widely deployed services, the keys are often generated on shared cloud environments. Malicious adversaries could be present and measure the time of keypair generation to recover the secret-key. All in all, it is mandatory that all operations involving secret data are performed in constant-time (Section 6.3). This can generate an additional cost, as in Algorithm 15 where we replace the conditional statement of Algorithm 14 by a mathematical formula. Our library `MQsoft` (Chapter 9) takes into consideration these criteria about security and performance evaluation. We minimize the penalty generated by the protection of secret data.

Algorithm 15 Constant-time left-to-right square-and-multiply exponentiation. $\mathcal{R}$ is a ring.

```
1:  function CstExp(A ∈ R, K ∈ ℕ*)
2:      A' ← A − 1
3:      A_K ← A
4:      for i from ⌊log_2(K)⌋ − 1 to 0 by −1 do
5:          A_K ← A_K^2                                         ▷ Square.
6:          b ← ⌊K / 2^i⌋ mod 2                                 ▷ Extraction of the i-th bit of K.
7:          A_K ← A_K × (A' × b + 1)                            ▷ Conditional multiplication by A.
8:      end for
9:      return A_K                                              ▷ A^K.
10: end function
```

We conclude by noting that other attacks, and in particular fault attacks [8], can really be dangerous in some contexts. With a simple laser, fault attacks can be generated and used to break the use of cryptography on smart cards. The first fault attacks were introduced by the authors of [32], demonstrating the major impact of fault attacks. The authors presented several attacks on RSA:

- on a typical implementation of the RSA-based signature schemes, the complete secret-key can be recovered with only a correct signature and a faulty signature, both from the same document,
- with a fairly large number of faulty encryptions, the secret exponent of RSA implementations using Algorithm 14 can be recovered.

No fault attacks were reported against our implementations submitted to the NIST PQC standardization process [128].

Chapter 5

Arithmetic

In this chapter, we present arithmetic in polynomial rings, as well as the representation of finite field extensions. Arithmetic is crucial for the performance of MI-based schemes (Chapter 2) and their dual mode (Chapter 3). In particular, the HFE-based schemes (Section 2.4.1) require arithmetic operations in $\mathbb{F}_{q^{d_{ext}}}$ and a root finding algorithm in $\mathbb{F}_{q^{d_{ext}}}[x]$. In Section 5.1, we present arithmetic operations in $\mathbb{F}_q[x]$. Then, we study different representations of extension fields in Section 5.2. We use the polynomial basis in `MQsoft` (Chapter 9), whose operations are these described in Section 5.1. When the finite field is $\mathbb{F}_{2^{d_{ext}}}$ (Section 5.3), we present some classical optimizations to improve arithmetic from Section 5.1. This field is used in `GeMSS` (Chapter 7) and `DualModeMS` (Chapter 8), and we will study an efficient implementation of field operations in Chapter 9. Once arithmetic in $\mathbb{F}_{q^{d_{ext}}}$ is defined, we can define basic operations in $\mathbb{F}_{q^{d_{ext}}}[x]$ as in Section 5.1 (by replacing q by $q^{d_{ext}}$). In Section 5.4, we define advanced operations in $\mathbb{F}_{q^{d_{ext}}}[x]$. These operations are parts of the root finding algorithm (Section 5.4.8), except for the polynomial evaluations which are used in the dual mode.

Let p be a prime number, the prime field $\mathbb{F}_p$ is defined as:

$$\mathbb{F}_p = \mathbb{Z}/(p\mathbb{Z}).$$

$\mathbb{F}_p$ is obtained from the integer ring by adding the relationship $p = 0$. Then, we can define a degree-k extension field of $\mathbb{F}_p$ by adding   a root of an irreducible polynomial f of degree k in $\mathbb{F}_p[x]$, i.e. we add the relationship $f(\ ) = 0$. We obtain:

$$\mathbb{F}_{p^k} = \mathbb{F}_p[\ ]/(f(\ )),\ k > 1.$$

In this chapter, we present arithmetic in $\mathbb{F}_q[x]$ for $q = p^{\ }$, $\ \geqslant 1$, or in $\mathbb{F}_{q^{d_{ext}}}[x]$ when the extension degree has a major role in the studied operation (e.g. the Frobenius map, Section 5.4.5). The prime p corresponds to the characteristic of this field. The dependencies between the different arithmetic operations are depicted in Figure 5.1. Then, we summarize the complexity of main polynomial operations in $\mathbb{F}_{q^{d_{ext}}}[x]$ (Table 5.1).

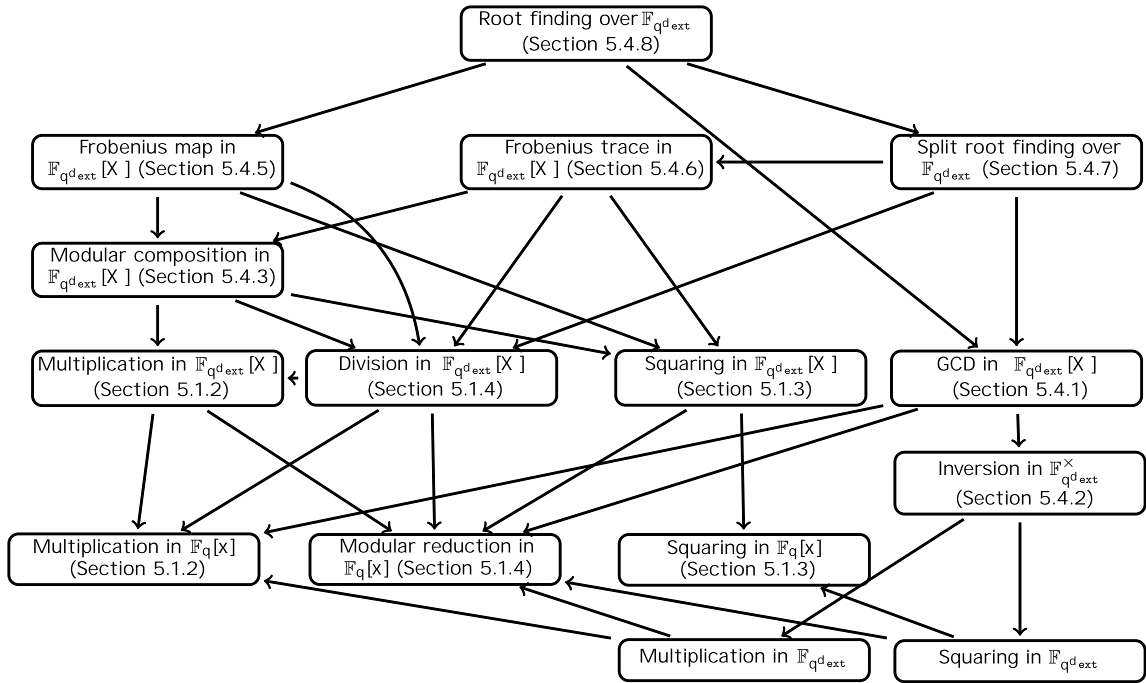

Figure 5.1: Dependencies between the di erent arithmetic operations in characteristic two.

operation	Section	method	complexity
addition	5.1.1	classical	$d+1$ (field additions)
multiplication	5.1.2	classical	$M(d) = O(d^2)$
		Karatsuba's	$M(d) = O(d^{1.585})$
		fast	$M(d) = O(d\log(d))$
square	5.1.3	classical	$O(d^2)$
		q is even	$d+1$ (field squarings)
Euclidean division	5.1.4	classical	$O(d^2)$
		fast	$O(M(d))$
GCD	5.4.1	classical	$O(d^2)$
		fast	$O(M(d)\log(d))$
modular composition	5.4.3	Horner	$\mathcal{C}_C(d) = O(M(d)\cdot d)$
		Brent–Kung	$\mathcal{C}_C(d) = O(d^{0.5}M(d) + d^{\frac{+1}{2}})$
Frobenius map	5.4.5	classical	$\mathcal{C}_F = O((d_{ext} - \lceil\log_q(d)\rceil)\log_2(q)M(d))$
		mod. comp.	$\mathcal{C}_F = O(\log_2(d_{ext})\mathcal{C}_C(d) + d_{ext}\log_2(q)\cdot d)$
		multi-squaring	$\mathcal{C}_F = O(q^{k'}\cdot d^2 + \frac{d_{ext}}{k'}\cdot d\cdot(d + k'\log_2(q)))$
root finding	5.4.8	classical	$O(\mathcal{C}_F + d^2)$ (on average)
		fast	$O(M(d)(d_{ext}\log_2(q) + \log(d)))$ (on average)

Table 5.1: Complexity of main operations on degree-d elements of $\mathbb{F}_{q^{d_{ext}}}[x]$, in number of operations in $\mathbb{F}_{q^{d_{ext}}}$. The integer k' is a parameter such that $1 \leqslant k' \leqslant d_{ext}$.

5.1 Basic Arithmetic in the Polynomial Ring $\mathbb{F}_q[x]$

In this section, we present elementary operations in $\mathbb{F}_q[x]$, for q a prime number or a power of a prime number. We note that arithmetic in $\mathbb{F}_q$ for $q = p^k$, with p a prime number and $k > 1$, can be built recursively thanks to arithmetic in $\mathbb{F}_p[x]$ (Section 5.2).
Let A and B be polynomials in $\mathbb{F}_q[x]$, with $n_a \geqslant n_b \geqslant 1$, such that:

$$A = \sum_{i=0}^{n_a-1} a_i x^i, \quad B = \sum_{i=0}^{n_b-1} b_i x^i.$$

In the next parts, we study operations between A and B. For convenience, let $d_a = n_a - 1$ and $d_b = n_b - 1$ be respectively the degree of A and B. We introduce the same notations for the quotient Q and the remainder R of the Euclidean division (Section 5.1.4):

$$Q = \sum_{i=0}^{n_q-1} q_i x^i, \quad R = \sum_{i=0}^{n_r-1} r_i x^i.$$

5.1.1 Addition and Subtraction

The polynomial addition [161, Algorithm 2.2] of A and B is the termwise addition of the coefficients, which has a cost of n_b field additions.

$$A + B = \left(\sum_{i=0}^{n_b-1} (a_i + b_i) x^i\right) + \left(\sum_{i=n_b}^{n_a-1} a_i x^i\right).$$

The principle is similar for the subtraction, which costs n_b field subtractions, with $n_a - n_b$ extra negations for the computation of $B - A$.

5.1.2 Multiplication

With the addition, the polynomial multiplication is the most crucial operation. The complexity of all algorithms used after, such as the fast Euclidean division algorithm, the fast GCD algorithms and the polynomial modular composition (respectively Sections 5.1.4, 5.4.1 and 5.4.3), relies on it. We denote by $M(d)$ the number of field operations to multiply two degree-d polynomials in $\mathbb{F}_q[x]$.

Classical multiplication. The classical multiplication [161, Algorithm 2.3] of A by B consists in computing the sum of all $a_i b_j x^{i+j}$ for $0 \leqslant i < n_a$ and $0 \leqslant j < n_b$. Here, we propose a formulation which allows to compute all terms of the result from x^0 to $x^{n_a+n_b-2}$.

$$A \times B = \left(\sum_{i=0}^{n_b-1} \sum_{j=0}^{i} a_{i-j} b_j x^i\right) + \left(\sum_{i=n_b}^{n_a-1} \sum_{j=0}^{n_b-1} a_{i-j} b_j x^i\right) + \left(\sum_{i=1}^{n_b-1} \sum_{j=1}^{n_b-i-1} a_{n_a-1-j} b_{i+j} x^{n_a-1+i}\right). \quad (5.1)$$

The cost of the classical multiplication is quadratic: exactly $n_a n_b$ field multiplications and $(n_a - 1)(n_b - 1)$ field additions are required. When $\mathbb{F}_q$ is a degree-extension field represented in polynomial basis (Section 5.2.1), we can accumulate the products, then only perform $n_a + n_b - 1$ modular reductions by the irreducible field polynomial. Therefore, the number of modular reductions is linear. We apply the same idea on Karatsuba's algorithm and the Euclidean division (Section 5.1.4). In this way, the modular reduction is completely negligible compared to the multiplication.

Karatsuba's multiplication algorithm. The cost of the classical multiplication is quadratic. Historically, Karatsuba's multiplication algorithm [161, Algorithm 8.1] is the first algorithm sub-quadratic in the degree. The idea is the following. Let:

$$A = A_\ell + A_h x^{\left\lfloor \frac{n_a}{2} \right\rfloor}, B = B_\ell + B_h x^{\left\lfloor \frac{n_a}{2} \right\rfloor},$$

with A_ℓ, B_ℓ $\left\lfloor \frac{n_a}{2} \right\rfloor$-coefficient polynomials in $\mathbb{F}_q[x]$ and A_h, B_h $\left\lceil \frac{n_a}{2} \right\rceil$-coefficient polynomials in $\mathbb{F}_q[x]$. A classical approach would be to compute $A \times B$ with four products:

$$A \times B = A_\ell B_\ell + (A_\ell B_h + A_h B_\ell) x^{\left\lfloor \frac{n_a}{2} \right\rfloor} + A_h B_h x^{2\left\lfloor \frac{n_a}{2} \right\rfloor}.$$

Karatsuba computes it with only three multiplications, which are:

$$P_1 = A_\ell \times B_\ell,\ P_2 = A_h \times B_h \text{ and } P_3 = (A_\ell + A_h) \times (B_\ell + B_h).$$

In particular, he remarks that $P_3 = P_1 + P_2 + (A_\ell B_h + A_h B_\ell)$. The latter product is exactly the middle term of $A \times B$. So, we obtain that $A \times B$ can be written with P_1, P_2, P_3 and some additions:

$$A \times B = P_1 + \left(P_3 - (P_1 + P_2)\right) x^{\left\lfloor \frac{n_a}{2} \right\rfloor} + P_2 x^{2\left\lfloor \frac{n_a}{2} \right\rfloor}.$$

In one step of Karatsuba, three half-degree multiplications are required to multiply A by B. Now, this process can be applied recursively to compute P_1, P_2 and P_3. At each step, the degree of each operand is divided by two. When the degree is small enough (*i.e.* strictly smaller than a fixed threshold $t > 0$), we can choose to stop the recursive calls and use the classical multiplication to multiply the operands. This case is commonly called base case. When $t = 1$, this approach requires $O\left(n_a^{\log_2(3)}\right) \simeq O\left(n_a^{1.585}\right)$ field operations [161, Theorem 8.3].

Karatsuba-like formulae. When the number of coefficients is not a power of two, the number of multiplications is not minimal. For a power of three, we can use the three-term Karatsuba-like formulae [124] (Section B.1), which allows to multiply two degree-two polynomials with only six multiplications (instead of seven with the previous method). This method allows to improve multiplications in $\mathbb{F}_2[x]$ on certain processors (Section 9.2.2).

Faster multiplications. Other methods, such as the Toom–Cook multiplication [112], allow to decrease the exponent of the complexity of the multiplication. With Toom–Cook algorithm, we can multiply two degree-two polynomials with five multiplications instead of six for a three-term Karatsuba-like formula. Nowadays, the best multiplication algorithm is the fast convolution algorithm [161, Algorithm 8.16] that uses Fast Fourier Transform (FFT). The complexity is quasi-linear in the number of coefficients. However, this method is based on the existence of a primitive 2^k-th root of unity. In binary fields, such roots do not exist. A variant based on a primitive 3^k-th root of unity can be used [161, Algorithm 8.30], but is considerably slower than the standard FFT. For this reason, we do not consider the FFT-based multiplications in our implementation.

5.1.3 Squaring

The computation of a square is a particular case of the multiplication. The symmetry of the operands allows to divide by two (approximately) the number of field multiplications. From Equation (5.1) with $A = B$, we obtain

$$A^2 = \sum_{i=0}^{n_a-1} a_i^2 x^{2i} + 2 \sum_{i=1}^{n_a-1} \sum_{j=0}^{\lfloor \frac{i}{2} \rfloor} a_{i-j} a_j x^i + 2 \sum_{i=1}^{n_a-2} \sum_{j=0}^{\lfloor \frac{n_a-1-i}{2} \rfloor} a_{n_a-1-j} a_{i+j} x^{n_a-1+i} . \quad (5.2)$$

In characteristic two, the computation of the square is linear, costing n_a field squarings (cf. Equation (5.2) by specifying $2 = 0$). This property is due to the linearity of the Frobenius endomorphism:

$$(a + b)^q = a^q + b^q, \text{ with } a, b \in \mathbb{F}_{q^{d_{\text{ext}}}}, \ d_{\text{ext}} \geqslant 1. \quad (5.3)$$

5.1.4 Euclidean Division

Euclidean division of A by B [161, Algorithm 2.5] consists in finding Q, R in $\mathbb{F}_q[x]$ such that $A = BQ + R$ with $\deg(R) < \deg(B)$. The polynomial Q corresponds to the quotient of the Euclidean division, whereas R corresponds to the remainder. Euclidean division is crucial to compute the GCD and perform the extended Euclidean algorithm (Section 5.4.1). In particular, the computation of the remainder is crucial to perform the modular multiplication in the extension fields represented with the polynomial basis (Section 5.2.1), as well as to compute the modular composition (Section 5.4.3), and the Frobenius map and trace (Sections 5.4.5 and 5.4.6) during the root finding algorithm (Section 5.4.8).

We present in Algorithm 16 the Euclidean division with remainder. We repeat the process of adding to A a multiple of B such that the current leading term of A will vanish. Since this term vanishes, we do not compute it (i.e. we replace B by $B - b_{d_b} x^{d_b}$ in Algorithm 16), but we directly set it to zero. Algorithm 16 is strictly equivalent to computing $R = A - BQ$, by taking into account that Q is computed as $b_{d_b}^{-1} \times Q'$, for any $Q' \in \mathbb{F}_q[x]$. In the worst case, Algorithm 16 requires computing one inverse in $\mathbb{F}_q^{\times}$, $(d_a - d_b + 1)(d_b + 1)$ field multiplications and $(d_a - d_b + 1) d_b$ field subtractions.

Algorithm 16 Polynomial Euclidean division with remainder.

1: function EuclideanDivRem($A \in \mathbb{F}_q[x], B \in \mathbb{F}_q[x]^*$)
2: $\quad c \leftarrow b_{d_b}^{-1}$
3: $\quad Q \leftarrow 0$
4: $\quad R \leftarrow A$
5: $\quad$ for i from d_a to d_b by -1 do
6: $\quad\quad q_{i-d_b} \leftarrow r_i \times c$ $\qquad$ ▷ Update of the quotient.
7: $\quad\quad R \leftarrow R - q_{i-d_b} \times (B - b_{d_b} x^{d_b}) x^{i-d_b}$
8: $\quad\quad r_i \leftarrow 0$ $\qquad$ ▷ The new R has a degree at most $i - 1$.
9: $\quad$ end for
10: $\quad$ return (Q, R)
11: end function

Naturally, the computation of the quotient can be separated from the computation of the remainder, because the quotient depends only on the higher degree terms of A. We can firstly compute the quotient without updating the d_b smallest degree terms of A, then secondly compute the remainder as $(A - B \times Q) \bmod x^{d_b}$. The remainder can be computed with $M(d_b)$ field operations and accelerated by using a fast multiplication (Section 5.1.2). The computation of the quotient can be performed without the computation of the remainder (Section B.2). However, Algorithm 16 does not allow to use fast multiplications. Thus, we present the principle of the fast Euclidean division [161, Algorithm 9.5].

Let $\mathrm{Rec}_i(A) = x^i \cdot A(1/x)$ be the reciprocal polynomial of A. The fast Euclidean division (Algorithm 17) of A by B consists in writing $A = BQ + R$ with $Q, R \in \mathbb{F}_q[x]$, and remarking that:

$$\mathrm{Rec}_{d_a}(A) = \mathrm{Rec}_{d_b}(B)\,\mathrm{Rec}_{d_a - d_b}(Q) + x^{d_a - d_b + 1}\,\mathrm{Rec}_{d_b - 1}(R). \tag{5.4}$$

Because the degree of Q is at most $d_a - d_b$, we can compute Equation (5.4) modulo $x^{d_a - d_b + 1}$. So, we obtain the following formula for Q:

$$\mathrm{Rec}_{d_a - d_b}(Q) = \mathrm{Rec}_{d_a}(A)\,\mathrm{Rec}_{d_b}(B)^{-1} \bmod x^{d_a - d_b + 1}. \tag{5.5}$$

Equation (5.5) requires computing $\mathrm{Rec}_{d_b}(B)^{-1}$ modulo a power of x. This can be performed efficiently with Newton iteration (Section B.3), requiring at most $3M(d_a - d_b) + O(d_a - d_b)$ field operations [161, Exercise 9.6]. Note that $\mathrm{Rec}_{d_b}(B)^{-1}$ can be precomputed one time for several Euclidean division by B. Then, Q is obtained with one polynomial multiplication and by reversing the order of coefficients, costing $M(d_a - d_b)$ field operations. Finally, the fast Euclidean division with remainder can be computed with $4M(d_a - d_b) + M(d_b) + O(d_a - d_b)$ field operations.

Algorithm 17 Fast Euclidean division with remainder.

1: **function** FastEuclideanDivRem($A \in \mathbb{F}_q[x], B \in \mathbb{F}_q[x]^*$)
2: $\quad$ recB_inv $\leftarrow \mathrm{Rec}_{d_b}(B)^{-1} \bmod x^{d_a - d_b + 1}$ $\quad$ ▷ Newton iteration [161, Algorithm 9.3].
3: $\quad$ recQ $\leftarrow \mathrm{Rec}_{d_a}(A) \cdot$ recB_inv $\bmod x^{d_a - d_b + 1}$ $\quad$ ▷ $\mathrm{Rec}_{d_a - d_b}(Q)$.
4: $\quad$ $Q \leftarrow \mathrm{Rec}_{d_a - d_b}$(recQ)
5: $\quad$ $R \leftarrow A - B \cdot Q \bmod x^{d_b}$ $\quad$ ▷ The modular reduction saves computations.
6: $\quad$ **return** (Q, R)
7: **end function**

5.2 Representation of Finite Field Extensions

Each element C of $\mathbb{F}_{q^{d_{ext}}}$ can be represented by using a basis $\mathcal{B} = \ {}_0, \dots, \ {}_{d_{ext}-1}$ of $\mathbb{F}_{q^{d_{ext}}}$ over $\mathbb{F}_q$. We can write:

$$C = \sum_{i=0}^{d_{ext}-1} c_i \ {}_i, \tag{5.6}$$

and $v_C = (c_0, c_1, \dots, c_{d_{ext}-1}) \in \mathbb{F}_q^{d_{ext}}$ is a representation of C over $\mathcal{B}$. Here, we present the common representations of $\mathbb{F}_{q^{d_{ext}}} = \mathbb{F}_q[\ ]/(f(\ ))$ with $f \in \mathbb{F}_q[x]$. When $q = 2$, the polynomial representation (Section 5.2.1) seems optimal on Haswell processors and later. The normal representation (Section 5.2.2) may be faster when the available vector instructions are limited (Section 6.1.3).

5.2.1 Polynomial Representation

Let $\in \mathbb{F}_{q^{d_{ext}}}$ be a root of . Then, 1, , ^{2}, $^3, \ldots,$ $^{d_{ext}-1}$ are linearly independent over $\mathbb{F}_{q^{d_{ext}}}$ and $\mathcal{B} =$ $_0, \ldots,$ $_{d_{ext}-1}$ with $_i =$ ^{i} is the canonical basis. The representation of any $C \in \mathbb{F}_{q^{d_{ext}}}$ over $\mathcal{B}$ is called polynomial representation because C is a degree-$(d_{ext} - 1)$ polynomial in $\mathbb{F}_q[\ ]$.

With such a representation, the operations in $\mathbb{F}_{q^{d_{ext}}}$ can be naturally performed with the operations in $\mathbb{F}_q[x]$ presented in Section 5.1, Moreover, when a product in $\mathbb{F}_{q^{d_{ext}}}$ is computed, the result C can be represented as $(c_0, c_1, \ldots, c_{2d_{ext}-2})$. Then, an unlimited number of products can be accumulated with this representation, before performing the modular reduction by f to obtain a d_{ext}-coe cient representation. The accumulation of coe cients allows to perform the main operations in $\mathbb{F}_{q^{d_{ext}}}[x]$ with a complexity linear in the number of field modular reductions in $\mathbb{F}_{q^{d_{ext}}}$.

5.2.2 Representation using Normal Bases

Let $\in \mathbb{F}_{q^{d_{ext}}}$. When , $^q, \ldots,$ $^{q^{d_{ext}-1}}$ are linearly independent over $\mathbb{F}_{q^{d_{ext}}}$, $\mathcal{B} =$ $_0, \ldots,$ $_{d_{ext}-1}$ with $_i =$ $^{q^i}$ is a normal basis of $\mathbb{F}_{q^{d_{ext}}}$ over $\mathbb{F}_q$. Let $C, D \in \mathbb{F}_{q^{d_{ext}}}$, and $v_C, v_D \in \mathbb{F}_q^{d_{ext}}$ be respectively their normal representation over $\mathcal{B}$. The operations in $\mathbb{F}_{q^{d_{ext}}}$ can be performed as follows:

- $C + D$ is represented by $v_C + v_D$,
- $C \times D$ is represented by $v_C \cdot M \cdot v_D^T$ for a specific matrix $M \in \mathcal{M}_{d_{ext}}$ $\mathbb{F}_q$,
- $C^{q^k} =$ $^{d_{ext}-1}_{i=0}$ $c_{i-k \bmod d_{ext}} \cdot$ $_i$ is represented by the right circular shift by k positions of v_C.

The q-exponentiation is e cient in the normal basis, because it corresponds to a circular shift of the coe cients. The multiplication can be performed as a vector-matrix-vector product, once M is known. The matrix M depends only on f and can be precomputed [106]. In fact, the computation of $v_C \cdot M \cdot v_D^T$ is the evaluation of a MQ polynomial $p_M \in \mathbb{F}_q[c_0, \ldots, c_{d_{ext}-1}, d_0, \ldots, d_{d_{ext}-1}]$. It is well-known that for $q = 2$, the number of ones in M, which is also the number of non-zero terms of p_M, is greater or equal to $2d_{ext} - 1$ [125]. When this bound is reached, $\mathcal{B}$ is called optimal normal basis [125], since it minimizes the cost of the sparse evaluation of p_M.

5.3 Arithmetic in Binary Fields

In this section, we study how to perform e ciently arithmetic in $\mathbb{F}_{2^{d_{ext}}}$ when the polynomial representation is used (Section 5.2.1). The elements are represented as coe cient vectors in $\mathbb{F}_2$, and can be easily added in parallel (Section 5.3.1). Once elements are multiplied as degree-$(d_{ext}$-1) binary polynomials (via Sections 5.1.2 or 5.1.3), the modular reduction can be performed e ciently by using sparse field polynomials (Section 5.3.3) or cyclotomic field polynomials (Section 5.3.4). For an e cient arithmetic in small binary fields, we refer to Sections 7.4.9 and 9.5.1. For a complete arithmetic in $\mathbb{F}_{2^{d_{ext}}}$ with implementation techniques and timings, we refer to Section 9.2.

5.3.1 Boolean Arithmetic

We summarize the main operations in $\mathbb{F}_2$ in Table 5.2. The addition corresponds to the XOR boolean operator, whereas the multiplication corresponds to the AND boolean operator. The squaring does not require any computation. These boolean operations are computed e ciently in parallel on all classical architectures (Section 6.1.3).

a	b	$a \oplus b$	$a \times b$	a^2
0	0	0	0	0
0	1	1	0	0
1	0	1	0	1
1	1	0	1	1

Table 5.2: Main operations in $\mathbb{F}_2$.

5.3.2 Irreducibility Conditions of Binary Polynomials

The choice of the irreducible polynomial f of degree d_{ext} defining $\mathbb{F}_{2^{d_{ext}}}$ is important for the performance of the modular reduction, when the polynomial representation is used (Section 5.2.1). It is well-known that sparse polynomials are more e cient (Section 5.3.3). In this section, we study properties on these irreducible polynomials.

Let $f = \sum_{i=0} x^{k_i} \in \mathbb{F}_2[x]$ for $0 \leqslant k_0 < k_1 < \cdots < k = d_{ext}$. For $d_{ext} > 1$, we know that f is irreducible only if $k_0 = 0$ and is even (respectively because zero and one are not roots of f). This implies that the most sparse irreducible polynomials over $\mathbb{F}_2$ are trinomials. When it exits, we can choose $k_1 \leqslant \frac{d_{ext}}{2}$ (since $\mathrm{Rec}_{d_{ext}}(f)$ is irreducible). Else, we can choose an irreducible pentanomial. For $d_{ext} \geqslant 4$, it is conjectured that an irreducible pentanomial of degree d_{ext} exists. Therefore, the use of irreducible trinomials and pentanomials is enough to represent $\mathbb{F}_{2^{d_{ext}}}$ for all $d_{ext} > 1$.

In cryptography, when a cryptosystem is being designed, the designer can choose d_{ext} which optimizes the performances. In particular, he can choose d_{ext} such that an irreducible trinomial f of degree d_{ext} exists. But between di erent trinomials, some values of d_{ext} and k_1 are more e cient in practice. Therefore, we propose to study what are the possible values. If a trinomial f is irreducible over $\mathbb{F}_2$, then the following properties are verified:

1. $d_{ext} \neq 0 \bmod 8$
2. If $k_1 = 0 \bmod 2$, then $d_{ext} = 1 \bmod 2$
3. If $k_1 = 0 \bmod 4$, then $d_{ext} = \pm 1 \bmod 8$
4. If $d_{ext} - k_1 = 0 \bmod 4$, then $d_{ext} = \pm 1 \bmod 8$
5. If $d_{ext} - 2k_1 = 0 \bmod 8$, then $k_1 = 1 \bmod 8$ or $k_1 = 3 \bmod 8$

These properties are obtained from a reformulation of [156, Corollary 5]. The latter enumerates all cases where f has an even number of factors (and hence is reducible) over $\mathbb{F}_2$. We will show in Section 9.2.3 that the modular reduction can be accelerated when d_{ext}, k_1, $d_{ext} - k_1$ or $d_{ext} - 2k_1$ are multiple of eight.

5.3.3 Modular Reduction by Sparse Polynomials over $\mathbb{F}_2$

In this section, we want to reduce $R = \sum_{i=0}^{2d_{ext}-2} r_i x^i$ the result of a multiplication/squaring in $\mathbb{F}_{2^{d_{ext}}}$, when the polynomial representation is used (Section 5.2.1). The choice of the irreducible polynomial f defining $\mathbb{F}_{2^{d_{ext}}}$ (Section 5.3.2) is important for the modular reduction: it is faster when f is a trinomial or pentanomial [101, 6, 5]. The classical method to perform the sparse modular reduction is to specify the classical Euclidean division (Algorithm 16) for a sparse polynomial. Naturally, the computation of the remainder is equivalent to computing $R - fQ$ for any $Q \in \mathbb{F}_2[x]$ with a classical polynomial multiplication (Section 5.1.4). The multiplication of f by Q is a multiplication by a sparse polynomial, which requires only $O(d_{ext})$ operations in $\mathbb{F}_2$ instead of $O(d_{ext}^2)$ for a dense polynomial. Then, we remark that when $2(k_{\ell-1} - 1) < d_{ext}$, the order of operations can be easily changed to optimize the computation with the polynomial representation (Section 5.2.1). This is due to the fact that $\mathrm{Rec}_{d_{ext}}(f)^{-1} \bmod x^{d_{ext}-1}$ is very sparse (Section 5.1.4), since $\mathrm{Rec}_{d_{ext}}(f)^{-1} = \mathrm{Rec}_{d_{ext}}(f) = 1 \bmod x^{d_{ext}-k_{\ell-1}}$. We specify this behavior for trinomials.

Modular reduction by a trinomial. Here, we explain the principle of modular reduction by a trinomial. Let $f_3 = x^{d_{ext}} + x^{k_1} + 1$ such that $0 < k_1 \leqslant \left\lfloor \frac{d_{ext}}{2} \right\rfloor$. Let $R_0 = \sum_{i=0}^{d_{ext}-1} r_i x^i$, $R_{k_1} = \sum_{i=d_{ext}}^{2d_{ext}-k_1-1} r_i x^{i-d_{ext}}$ and $S_{k_1} = \sum_{i=2d_{ext}-k_1}^{2d_{ext}-2} r_i x^{i-2d_{ext}+k_1}$, we have:

$$R = R_0 + \left(R_{k_1} + S_{k_1} x^{d_{ext}-k_1}\right) x^{d_{ext}}. \tag{5.7}$$

We perform a first step of reduction by f_3 by replacing $x^{d_{ext}}$ by $f_3 - x^{d_{ext}}$ in (5.7). We obtain:

$$R = R_0 + \left(R_{k_1} + S_{k_1} x^{d_{ext}-k_1}\right) + \left(R_{k_1} x^{k_1} + S_{k_1} x^{d_{ext}}\right) \bmod f_3.$$

We iterate a new step of reduction:

$$R = R_0 + R_{k_1} + S_{k_1} x^{d_{ext}-k_1} + R_{k_1} x^{k_1} + S_{k_1}\left(f_3 - x^{d_{ext}}\right) \bmod f_3. \tag{5.8}$$

In Equation (5.8), the degree of R is $\max\left(d_{ext} - 1, 2(k_1 - 1)\right)$. So, R is reduced modulo f_3 if and only if $2(k_1 - 1) < d_{ext}$. In two steps of reduction, we have then a method to compute the modular reduction for all trinomial such that $k_1 \leqslant \left\lfloor \frac{d_{ext}}{2} \right\rfloor$.

Remark 1. When $k_1 = 1$, $S_{k_1} = 0$ and R is reduced in only one step.

Modular reduction by a sparse polynomial. Similarly to trinomials, we explain the principle of modular reduction by f for $k_0 = 1$ and $k_{\ell-1} \leqslant \left\lfloor \frac{d_{ext}}{2} \right\rfloor$. Let $R_0 = \sum_{i=0}^{d_{ext}-1} r_i x^i$, $R_{k_j} = \sum_{i=d_{ext}}^{2d_{ext}-k_j-1} r_i x^{i-d_{ext}}$ and $S_{k_j} = \sum_{i=2d_{ext}-k_j}^{2d_{ext}-2} r_i x^{i-2d_{ext}+k_j}$ for $j \in [\![1, \ell-1]\!]$, we have:

$$R = R_0 + \left(R_{k_j} + S_{k_j} x^{d_{ext}-k_j}\right) x^{d_{ext}}. \tag{5.9}$$

We perform a first step of reduction by f by replacing $x^{d_{ext}}$ by $f - x^{d_{ext}}$ in Equation (5.9). We compute $\left(R_{k_j} + S_{k_j} x^{d_{ext}-k_j}\right) \times \left(1 + \sum_{j=1}^{\ell-1} x^{k_j}\right)$ by multiplying the left operand by x^{k_j} for $j \in [\![1, \ell-1]\!]$, and we choose $j = 1$ when we multiply the left operand by 1. We obtain:

$$R = R_0 + \left(R_{k_1} + S_{k_1} x^{d_{ext}-k_1}\right) + \sum_{j=1}^{\ell-1} \left(R_{k_j} x^{k_j} + S_{k_j} x^{d_{ext}}\right) \bmod f.$$

We iterate a new step of reduction:

$$R = R_0 + R_{k_1} + S_{k_1} x^{d_{ext}-k_1} + \sum_{j=1}^{\ell-1} R_{k_j} x^{k_j} + \left(f - x^{d_{ext}}\right) \sum_{j=1}^{\ell-1} S_{k_j} \bmod f. \tag{5.10}$$

In Equation (5.10), the degree of R is $\max\left(d_{ext} - 1, 2(k_{\ell-1} - 1)\right)$. So, R is reduced modulo f if and only if $2(k_{\ell-1} - 1) < d_{ext}$. In two steps of reduction, we have then a method to compute the modular reduction for all polynomial such that $k_{\ell-1} \leqslant \frac{d_{ext}}{2}$.

Remark 2. When $k_1 = 1$, $S_{k_1} = 0$ and Equation (5.10) can be simplified accordingly.

5.3.4 Modular Reduction by Cyclotomic Polynomials over $\mathbb{F}_2$

In the previous section, we study how to take advantage of the sparse polynomials to improve the field modular reductions. But we can also take advantage of structured polynomials. In this section, we deal with irreducible cyclotomic polynomial, also called All One Polynomial (AOP) [109, 154], and s-Equally Spaced Polynomial (s-ESP) [109] its generalization ($s \in \mathbb{N}^*$). An AOP is a one-ESP.
In polynomial basis, the cyclotomic polynomials allow to perform modular reductions faster than by using trinomials. However, the conditions on d_{ext} for that the degree-d_{ext} cyclotomic polynomial is irreducible are strong.

Straightforward modular reduction. The s-ESP of degree d_{ext} is:

$$f = \sum_{i=0}^{\frac{d_{ext}}{s}} x^{s \cdot i} = \frac{x^{d_{ext}+s} - 1}{x^s - 1}.$$

Thus, we obtain that:

$$x^{d_{ext}+s} - 1 = \left(x^s - 1\right) \cdot f = 0 \bmod f.$$

So, the modular reduction is straightforward in polynomial basis. All terms greater or equal to $x^{d_{ext}+s}$ can be reduced with a right shift by $d_{ext} + s$ of the coefficients. Finally, the terms greater or equal to $x^{d_{ext}}$ are just the product of a s-coefficient polynomial by the $(d_{ext} - s)$-degree s-ESP.

Irreducibility condition. Wah and Wang introduced the following lemma [109, Lemma 1] to know if an AOP is irreducible, then a theorem [109, Theorem 3] for the irreducibility of s-ESP.

Lemma 1. An AOP of degree d_{ext} is irreducible over $\mathbb{F}_2$ if and only if $(d_{ext} + 1)$ is a prime and 2 is the generator of $\mathbb{F}^{\times}_{d_{ext}+1}$, where $\mathbb{F}^{\times}_{d_{ext}+1}$ is the multiplicative group in $\mathbb{F}_{d_{ext}+1}$.

Theorem 3. A s-ESP of degree $d_{ext} = sn$ is irreducible over $\mathbb{F}_2$ if and only if the n-degree AOP is irreducible over $\mathbb{F}_2$ and for some integer t, $s = (n + 1)^{t-1}$ and $2^{n(n+1)^{t-2}} \not\equiv 1 \bmod (n + 1)^t$.

So, we obtain that the degree-d_{ext} AOP is reducible when $d_{ext} > 1$ is odd. Moreover, the quadratic reciprocity theorem [156] implies that 2 is a square modulo $d_{ext}+1$ if and only if $d_{ext}+1 = \pm 1 \bmod 8$. In this case, 2 is not a generator of $\mathbb{F}^{\times}_{d_{ext}+1}$. We deduce that if the degree-d_{ext} AOP is irreducible, then $d_{ext} = 2 \bmod 8$ or $d_{ext} = 4 \bmod 8$. This condition is clearly stronger than for irreducible trinomials. For d_{ext} from 2 to 576, only 44 AOPs and 55 s-ESPs are irreducible (with 52 distinct degrees), whereas there exist irreducible trinomials for 320 values of d_{ext} (cf. Section B.9).

5.4 Advanced Arithmetic in $\mathbb{F}_q[x]$

In this section, we study arithmetic operations which will allow to introduce a root finding algorithm in $\mathbb{F}_{q^{d_{ext}}}[x]$ (Section 5.4.8), as for example the Euclidean algorithm (Section 5.4.1), the modular composition (Section 5.4.3) and the Frobenius map (Section 5.4.5). We also study the evaluation of univariate polynomials in one or several points (Sections 5.4.3 and 5.4.4), which are crucial for the performance of the dual mode (Chapter 3). For each operation, we present di erent strategies to perform them, and we study the number of field operations.

Similarly to Section 5.1, let F, G and H be respectively degree-d_f, degree-d_g and degree-d_h polynomials in $\mathbb{F}_q[x]$, such that:

$$F = \sum_{i=0}^{d_f} f_i x^i,\ G = \sum_{i=0}^{d_g} g_i x^i,\ H = \sum_{i=0}^{d_h} h_i x^i.$$

In Sections 5.4.5, 5.4.6, 5.4.7 and 5.4.8, we study the impact of the extension degree on the current operation. Therefore, we consider F, G and H in $\mathbb{F}_{q^{d_{ext}}}[x]$ instead of $\mathbb{F}_q[x]$ for these sections.

5.4.1 Extended Euclidean Algorithm

The computation of the Greatest Common Divisor (GCD) of F and H in $\mathbb{F}_q[x]$, via Euclidean algorithm [161, Section 3], is a central operation in $\mathbb{F}_q[x]$. This is particularly useful for root finding (Section 5.4.8) and split root finding algorithms (Section 5.4.7). Moreover, the extended version of the Euclidean algorithm allows to build a Bezout relationship between F and H, i.e. finding U and V in $\mathbb{F}_q[x]$ such that:

$$FU + HV = \mathrm{GCD}(F, H).$$

The Bezout relationship permits to compute the inverse of F modulo H when the latter exists (Section 5.4.2). We present in this section Euclidean algorithm and its extended version. In particular, we remark that less popular methods are e cient for small degree inputs.

Euclidean algorithm. Euclidean algorithm [161, Algorithm 3.5] is based on the following relationship:

$$\mathrm{GCD}(F, H) = \mathrm{GCD}(H, F \bmod H).$$

By repeating this process (Algorithm 18), we compute the remainders of the successive Euclidean divisions of F by H, until $\mathrm{GCD}(G, 0) = G$. The last non-null remainder is $\mathrm{GCD}(F, H)$.

Algorithm 18 Traditional Euclidean algorithm.

1: **function** EA($F \in \mathbb{F}_q[x], H \in \mathbb{F}_q[x]$)
2: $\quad R_0, R_1 \leftarrow F, H$
3: $\quad i \leftarrow 1$
4: $\quad$ **while** $R_i \neq 0$ **do**
5: $\quad\quad R_{i+1} \leftarrow R_{i-1} \bmod R_i$
6: $\quad\quad i \leftarrow i + 1$
7: $\quad$ **end while**
8: $\quad$ **return** R_{i-1} $\quad$ ▷ $\mathrm{GCD}(F, H) = \mathrm{GCD}(R_{i-1}, 0) = R_{i-1}$.
9: **end function**

The core of Algorithm 18 is the computation of $R_{i-1} \bmod R_i$. This step can be optimized or modified by using properties on GCD. Assume we want to compute $F \bmod H$. We can consider the following methods:

1. directly compute the Euclidean division of F by H,
2. use $\mathrm{GCD}(F,H) = \mathrm{GCD}\left(F, h_{d_h}^{-1}H\right)$ to obtain a monic divisor,
3. use the Euclid–Stevin strategy [155, 19]. Given F and H such that $d_f \geqslant d_h$, we compute $F = h_{d_h}F - f_{d_f}Hx^{d_f - d_h}$ until $d_f < d_h$. Let F_1 be the last computed value of F. We have:

$$\mathrm{GCD}(F,H) = \mathrm{GCD}(H,F_1).$$

By using the classical Euclidean algorithm (Algorithm 16), Methods 1 and 2 require one inversion in $\mathbb{F}_q^\times$, then Method 1 multiplies the quotient by the inverse, whereas Method 2 multiplies the divisor by the inverse. So, Method 1 requires $(d_f - d_h + 1)(d_h + 1)$ field multiplications whereas Method 2 requires $(d_f - d_h + 2)d_h$ field multiplications. Method 2 is strictly better when $2d_h - 1 > d_f$, which is rare in practice. Method 3 requires at most $(d_f - d_h + 1)(d_f + d_h - 1) - \frac{(d_f - d_h - 1)(d_f - d_h)}{2}$ field multiplications, but no inversion in $\mathbb{F}_q^\times$. This method is the best when the inversion in $\mathbb{F}_q^\times$ is expensive compared to the field multiplication.

Extended Euclidean algorithm. It is well-known that Euclidean algorithm can be extended to compute Bezout coefficients [161, Algorithm 3.6]. We present it in Algorithm 19.

Algorithm 19 Traditional extended Euclidean algorithm.

```
1: function EEA(F ∈ F_q[x], H ∈ F_q[x])
2:     R_0, R_1 ← F, H
3:     U_0, U_1 ← 1, 0
4:     V_0, V_1 ← 0, 1
5:     i ← 1
6:     while R_i ≠ 0 do
7:         Q_i ← R_{i-1}/R_i                    ▷ Quotient of the Euclidean division.
8:         R_{i+1} ← R_{i-1} − Q_i R_i           ▷ Remainder of the Euclidean division.
9:         U_{i+1} ← U_{i-1} − Q_i U_i
10:        V_{i+1} ← V_{i-1} − Q_i V_i
11:        i ← i + 1
12:    end while
13:    return (R_{i-1}, U_{i-1}, V_{i-1})      ▷ F U_{i-1} + H V_{i-1} = R_{i-1} = GCD(F, H).
14: end function
```

Fast extended Euclidean algorithm. The extended Euclidean algorithm can be performed by doing *giant steps* in the list of successive remainders, by using the so-called half-GCD algorithm ([161, Algorithm 11.6], [36, Algorithm 6.9]). Coupled to a divide-and-conquer approach, we obtain a fast extended Euclidean algorithm ([161, Algorithm 11.8], [36, Algorithm 6.8]). Its complexity is $O(\mathsf{M}(d_f)\log(d_f))$ field multiplications and additions plus at most $d_f + 2$ inversions in $\mathbb{F}_q^\times$ [161,

Corollary 11.9]. In 2019, the authors of [19] introduced a fast constant-time GCD. Its complexity is $(M(d_f)\log(d_f))$ field multiplications and additions. Unlike the previous fast GCD which is based on polynomial multiplications and Euclidean divisions, the method of [19] is only based on the polynomial multiplication.

5.4.2 Modular Inversion

The inversion of $F \bmod H$ is the problem of finding, if any, a polynomial G such that $F \times G = 1 \bmod H$, i.e. $G = F^{-1} \bmod H$. This operation is critical to compute the inverse in the extension fields. There are mainly two strategies to compute it: the extended Euclidean algorithm [161, Theorem 4.1] and the modular exponentiation via Fermat [161, Section 4.4]. These methods can be easily adapted to compute inverses in prime fields (by taking $F \in \mathbb{F}_p$ and $H = p$).

Euclid-based inversion. By definition of the inverse, we search G such that $FG + HV = 1$, with $V \in \mathbb{F}_q[x]$. In other words, we want to write 1 with a Bezout relationship between F and H. This relationship exists if and only if $\mathrm{GCD}(F, H) = k$ for $k \in \mathbb{F}_q^{\times}$. When F is invertible, the Bezout relationship between F and H is:

$$FU + HV = \mathrm{GCD}(F, H) = k, \text{ with } U, V \in \mathbb{F}_q[x].$$

In this case, $k^{-1}FU = 1 \bmod H$ and $k^{-1}U$ is the inverse of $F \bmod H$. Naturally, the Euclidean algorithm (Section 5.4.1) applied to F and H allows to know if the inverse exists, and can be computed with its extended version. The latter can be improved by computing only the first Bezout coefficient.

Fermat-based inversion. When H is irreducible, $\mathbb{F}_q[x]/(H)$ is the field having q^{d_h} elements. Based on Fermat's little theorem, we have:

$$F^{q^{d_h}} = F \bmod H,$$

and when $F \neq 0 \bmod H$,

$$F^{q^{d_h}-2} = F^{-1} \bmod H. \tag{5.11}$$

So, F^{-1} can be computed with a modular exponentiation. The latter can be performed efficiently by using the Itoh–Tsujii Multiplicative Inversion Algorithm (Sections 9.2.6 and B.6).

5.4.3 Univariate Evaluation and Modular Composition

In this section, we present the polynomial evaluation of $G \in \mathbb{F}_q[x]$ in an element $a \in \mathcal{R}$, for $\mathcal{R}$ a ring. We denote by $1_{\mathcal{R}}$ the zero element of the multiplicative group of $\mathcal{R}$. This ring can be $\mathbb{F}_q$, $\mathbb{F}_q[x]/(H)$, $\mathcal{M}(\mathbb{F}_q)$, ... When $\mathcal{R} = \mathbb{F}_q[x]/(H)$, the computation of $G(a) \bmod H$ is called modular composition. The latter is useful to compute $x^{k} \bmod H$ for $k \in \mathbb{N}$, which appears during the Fermat-based modular inversion (Section 5.4.2) and during the Frobenius map (Section 5.4.5).

Dot product strategy. A classical strategy to perform the evaluation is to compute the vector of the powers of a, then compute its dot product with the vector of coefficients of G. The powers of a can be computed with $\frac{d_g-1}{2}$ multiplications in $\mathcal{R}$ and $\frac{d_g}{2}$ squarings in $\mathcal{R}$. Then, the dot product requires $d_g - 1$ additions in $\mathcal{R}$, one scalar addition and d_g scalar multiplications.

Horner's rule. The polynomial evaluation can be performed without the computation of the powers of a, by using Horner's rule:

$$G(a) = ((\cdots((g_{d_g} \cdot a + g_{d_g-1} \cdot 1_{\mathcal{R}}) \times a + g_{d_g-2} \cdot 1_{\mathcal{R}}) \cdots) \times a + g_1 \cdot 1_{\mathcal{R}}) \times a + g_0 \cdot 1_{\mathcal{R}}.$$

This rule is well-known to minimize the number of additions and multiplications, which is d_g. However, $d_g - 1$ multiplications (denoted by $\times$) are computed in $\mathcal{R}$, whereas the other operations are scalar. When $\mathcal{R}$ is not a subset of $\mathbb{F}_q$, the operations in $\mathcal{R}$ are more expensive than the scalar operations. So, we study other strategies which minimize the number of these operations.

Baby-Step Giant-Step approach. The number of operations in $\mathcal{R}$ can go down to $O\left(\sqrt{d_g}\right)$. To do it, M. Paterson and L. Stockmeyer proposed to use a Baby-Step Giant-Step approach [136, Algorithm B]. The idea is to split G into b blocks of s-coefficient polynomials, evaluate each block, then evaluate G which is become a b-coefficient polynomial in $\mathcal{R}[x]$. In Algorithm 20, we present this strategy by introducing a matrix-vector product. This idea is an adaptation of the Brent and Kung algorithm [42, Algorithm 2.1]. R. Brent and H. Kung remarked that when $\mathcal{R} = \mathbb{F}_q[x]/(H)$, the matrix-vector product can be written as a matrix product over $\mathbb{F}_q$, allowing to use fast matrix products during the modular composition. Algorithm 20 is a generalization of the previous approaches. Algorithm 20 with $b = 1$ corresponds to the dot product strategy, whereas $b = d_g$ corresponds to the direct use of Horner's rule.

Algorithm 20 Polynomial evaluation using the baby-step giant-step approach.

Input: $G \in \mathbb{F}_q[x], a \in \mathcal{R}, b \in \mathbb{N}^*$ such that $b \leqslant d_g$.
Output: $G(a)$.

0. Let $s = \left\lceil \frac{d_g}{b} \right\rceil$ and $G(x) = \sum_{i=0}^{b-1} G_i(x) \cdot \left(x^s\right)^i + g_{bs} \cdot x^{bs}$, with $G_i \in \mathbb{F}_q[x]$, $\deg(G_i) < s$.

1. Compute $a^2, a^3, \ldots, a^s$ (e.g. with Algorithm 45). These are the baby steps.

2. Re-use them to evaluate $G_0, \ldots, G_{b-1}$ in a. These evaluations can be performed with the following matrix-vector product:

$$\begin{pmatrix} g_0 & \cdots & g_{s-1} & 0 \\ g_s & \cdots & g_{2s-1} & 0 \\ \vdots & \ddots & \vdots & \vdots \\ g_{(b-2)s} & \cdots & g_{(b-1)s-1} & 0 \\ g_{(b-1)s} & \cdots & g_{bs-1} & g_{bs} \end{pmatrix} \cdot \begin{pmatrix} 1_{\mathcal{R}} \\ a \\ \vdots \\ a^{s-1} \\ a^s \end{pmatrix} = \begin{pmatrix} G_0(a) \\ G_1(a) \\ \vdots \\ G_{b-2}(a) \\ G_{b-1}(a) + g_{bs} \cdot a^s \end{pmatrix}.$$

3. Compute $G(a)$ as:

$$\sum_{i=0}^{b-2} G_i(a) \times \left(a^s\right)^i + \left(G_{b-1}(a) + g_{bs} \cdot a^s\right) \times \left(a^s\right)^{b-1}. \tag{5.12}$$

   Equation (5.12) can be performed with Horner's rule (Algorithm 46), which allows to save the computation of the powers of a^s. These are the giant steps.

By choosing $b = \lceil\sqrt{d_g}\rceil$, we can minimize the complexity, with approximately $\frac{3}{2}\sqrt{d_g}$ multiplications in $\mathcal{R}$, and $\frac{1}{2}\sqrt{d_g}$ squarings in $\mathcal{R}$. Independently, Step 2 requires roughly d_g multiplications between elements of $\mathbb{F}_q$ and $\mathcal{R}$. In Section B.4, we propose a variant with a similar complexity.

About evaluation of polynomial systems. In the case where several polynomials are evaluated in a, some optimizations can be applied to Algorithm 20. Step 1 should be performed only one time. Step 3 can be optimized by precomputing one time the powers of $(a^s)^i$ for $i < b$, then by replacing Horner's rule by a dot product.

5.4.4 Multipoint Evaluation of Univariate Polynomial Systems

In all cryptographic operations of the dual mode (Chapter 3), we have to evaluate a univariate polynomial system whose coefficients live in $\mathbb{F}_{q^{\eta}}$, a small extension of $\mathbb{F}_q$. The number of equations and evaluation points depends on the cryptographic operations, so we need to adapt the strategies used accordingly. Here, we present several strategies.

Classical polynomial and multipolynomial evaluation. The simplest method is to use the strategies described in Section 5.4.3. We note that:

- The powers of a point should be computed only one time for all polynomials of the system, and can be performed in parallel.
- During the matrix-vector product, the multiplications do not require modular reductions and can be performed in parallel. For $q = 2$, the size of $\mathbb{F}_{q^{\eta}}$ is small enough to directly compute the multiplication with one call to `PCLMULQDQ` (Section 6.1.3). So, avoiding the modular reduction improves drastically the implementation.

Fast multipoint evaluation of polynomial systems. Let $M(d)$ be the cost of multiplying two degree-d polynomials. The evaluation of a degree-d polynomial in d points can be achieved in $O(M(d)\log(d))$ field multiplications with the fast multipoint evaluation algorithm [161, Algorithm 10.7]. For a polynomial system, the subproduct tree can be computed one time for all polynomials. When the point set can be chosen, the previous algorithm can be improved. With a geometric sequence, this can be solved in $O(M(d))$ field multiplications [37].

Additive FFT in binary fields. When the evaluation point set is large, it can be interesting to evaluate polynomials with the Fast Fourier Transform. The classical FFT uses a divide-and-conquer approach by splitting a polynomial $f \in \mathbb{F}_{q^{\eta}}[X]$ into $f_0(X^2) + Xf_1(X^2)$, with f_0 and f_1 half-degree polynomials. So, computing $f(a)$ and $f(-a)$ for a a field element is reduced to computing $f_0(a^2)$ and $f_1(a^2)$. However, this classical approach cannot be applied in characteristic two, since a and $-a$ are the same element. So, S. Gao and T. Mateer [93] introduced the additive FFT. The main idea is to split $f \in \mathbb{F}_{2^{\eta}}[X]$ into $f_0(X^2 - X) + Xf_1(X^2 - X)$, because $X^2 - X$ is the field equation of $\mathbb{F}_2$. Then, computing $f(a)$ and $f(a+1)$ is reduced to computing $f_0(a^2 - a)$ and $f_1(a^2 - a)$. For the classical FFT, f_0 (respectively f_1) is generated from the even (respectively odd) degree terms of f. For the additive FFT, the computation is a bit more complicated, but is performed efficiently with the so-called *radix conversion*. The complexity of the additive FFT [17] is $(\ell - 1)2^{\ell+1} + 1$ multiplications in $\mathbb{F}_{q^{\eta}}$ and $(\ell + 3)\ell 2^{\ell-2} - 2^{\ell-1}$ additions in $\mathbb{F}_{q^{\eta}}$. Since 2010, several tricks have been proposed to decrease the number of operations [17].

5.4.5 Frobenius Map

An important step during the univariate root finding (Section 5.4.8) is to compute $x^{q^k} \bmod H$, for $k > 1$. This problem is so-called Frobenius map. Several strategies are possible to compute it. Note that for each of them, when the fast Euclidean division (Section 5.1.4) is used, $\mathrm{Rev}_{d_h}(H)^{-1} \bmod x^{d_h-1}$ can be computed one time for all modular reductions.

Classical repeated squaring algorithm. We can compute $x^{q^k} \bmod H$ by using the square-and-multiply algorithm (Algorithm 14). Let $d = q^{k-\lceil \log_q(d_h) \rceil}$. By remarking that $x^{q^{\lceil \log_q(d_h) \rceil}-1}$ is already reduced modulo H, this method requires one modular reduction to compute $x^{q^{\lceil \log_q(d_h) \rceil}} \bmod H$, then $\lfloor \log_2(d) \rfloor$ modular squarings and $\mathtt{HW}(d)-1$ modular multiplications in $\mathbb{F}_{q^{d_{ext}}}[x]/(H)$, where $\mathtt{HW}$ stands for Hamming weight. We can also just repeat $k - \lceil \log_q(d_h) \rceil$ times the modular exponentiation to the power q. Note that these methods are different only in odd characteristic.

Use of the modular composition. We can compute $x^{q^k} \bmod H$ by using the following property:

$$G(x)^{q^j} \bmod H = \left(\sum_{i=0}^{d_h-1} g_i x^i\right)^{q^j} = \sum_{i=0}^{d_h-1} g_i^{q^j} \left(x^{q^j}\right)^i = G^{(j)}\left(x^{q^j}\right) \bmod H, \tag{5.13}$$

where $G^{(j)}(x) = \sum_{i=0}^{d_h-1} g_i^{q^j} x^i \in \mathbb{F}_{q^{d_{ext}}}[x]$. Thus, we can use the square-and-multiply algorithm to compute q^k instead of $x^{q^k} \bmod H$. Once k is decomposed in base two with the square-and-multiply method, the power k' in $x^{q^{k'}}$ can be multiplied by two by using $j = k'$ and $G(x) = x^{q^{k'}} \bmod H$ in Equation (5.13), whereas the addition with k'' is obtained by using $j = k''$ and $G(x) = x^{q^{k'}} \bmod H$ in Equation (5.13). This idea was already proposed in [161, Algorithm 14.55], and recently used in [120] to improve the root finding of $\mathtt{HFE}$ polynomials. Both use a right-to-left square-and-multiply algorithm on q^k (Section B.5). However, this strategy does not minimize the number of modular compositions. Therefore, we perform a left-to-right square-and-multiply exponentiation on q^k. We present this method in Algorithm 21, which is currently used in $\mathtt{NTL}$.

Algorithm 21 Frobenius map using the left-to-right square-and-multiply algorithm on q^k.

1: **function** FrobMapLR($H \in \mathbb{F}_{q^{d_{ext}}}[x], k \in \mathbb{N}^*$)
2: $\quad r \leftarrow \lfloor \log_2(k) \rfloor$
3: $\quad X_0 \leftarrow x^q \bmod H$ ▷ $X_0 = x^{q^{2^0}} \bmod H$.
4: $\quad$**for** i from 1 to r **do**
5: $\quad\quad X_i \leftarrow X_{i-1}^{(\lfloor \frac{k}{2^{r-i+1}} \rfloor)} \circ X_{i-1} \bmod H$ ▷ Square: $X_i = x^{q^{2\lfloor \frac{k}{2^{r-i+1}} \rfloor}} \bmod H$.
6: $\quad\quad$**if** $\lfloor \frac{k}{2^{r-i}} \rfloor \bmod 2 = 1$ **then**
7: $\quad\quad\quad X_i \leftarrow X_i^q \bmod H$ ▷ And multiply: $X_i = x^{q^{2\lfloor \frac{k}{2^{r-i+1}} \rfloor+1}} \bmod H$.
8: $\quad\quad$**end if** ▷ Here, $X_i = x^{q^{\lfloor \frac{k}{2^{r-i}} \rfloor}} \bmod H$.
9: $\quad$**end for**
10: $\quad$**return** X_r ▷ $i = r$ and $X_r = x^{q^k} \bmod H$.
11: **end function**

Algorithm 21 requires $\lfloor \log_2(k) \rfloor$ modular compositions in $\mathbb{F}_{q^{d_{ext}}}[x]$, $(k - \mathtt{HW}(k)) \cdot d_h$ exponentiations to the power q in $\mathbb{F}_{q^{d_{ext}}}$ (or $\lfloor \log_2(k) \rfloor \cdot d_h$ calls to Algorithm 22, via $\lfloor \log_2(k) \rfloor$ multi-squaring tables) and $\mathtt{HW}(k) - 1$ modular exponentiations to the power q in $\mathbb{F}_{q^{d_{ext}}}[x]/(H)$. Note that these exponentiations can also be computed with the modular composition, by writing $X_i^q = X_i^{(1)} \circ X_0 \bmod H$.

Repeated squaring algorithm using multi-squaring tables. Here, the idea is to perform k' times the q-exponentiation in one step, as presented in Algorithm 22. This can be computed efficiently by remarking that the linearity of the Frobenius endomorphism (Equation (5.3)) implies that a d_h-coefficient polynomial raised to the power of $q^{k'}$ has only k' non-zero terms. The corresponding monomials can be computed once and for all modulo H (Step 1). This is the multi-squaring table of H [33, 159]. Then, each $q^{k'}$-exponentiation can be performed as the dot product of the vector of $q^{k'}$-powers of the input coefficients by the vector of monomials modulo H (Step 2).

Algorithm 22 Frobenius map using a multi-squaring table.

Input: $H \in \mathbb{F}_{q^{d_{ext}}}[x], k \in \mathbb{N}^*, k' \leqslant k$.

Output: $x^{q^k} \bmod H$.

0. Let $= \frac{d_h - 1}{q^{k'}}$, $j_0 = \log_q(d_h - 1)$, $l_q = \frac{k - j_0}{k'}$ and $l_r = k - j_0 \bmod k'$.

1. Compute $T_1 = x^{q^{k'}} \bmod H$, then $T_j = x^{jq^{k'}} \bmod H$ for $1 < j < d_h$. Note that $T_j = x^{jq^{k'}}$ without the modular reduction by H for $j \leqslant$.

2. Set $X_{j_0+k'} = T_{q^{j_0}} = x^{q^{j_0+k'}} \bmod H$.
   For $2 \leqslant i \leqslant l_q$, let $X_{j_0+(i-1)k'} = \sum_{j=0}^{d_h-1} c_j x^j$ and compute:

$$X_{j_0+ik'} = X_{j_0+(i-1)k'}^{(k')} \circ x^{q^{k'}} \bmod H = c_0^{q^{k'}} + \sum_{j=1}^{d_h-1} c_j^{q^{k'}} T_j .$$

   Note that $\deg(X_{j_0+ik'}) < d_h$ without computing the modular reduction by H.

3. Return $X_k = (X_{j_0+l_q k'})^{q^{l_r}} \bmod H$, computed with the classical repeated squaring algorithm.

We start by measuring the cost of Step 1 in Lemma 2. This step is quadratic in d_h and polynomial in q, so only small values of k' are interesting in practice. However, this step can be precomputed when H is known, which is the case for the field modular reductions. From a memory point of view, the multi-squaring table costs $(d_h - 1 - \ )d_h$ field elements, which is quadratic in d_h.

Lemma 2 (Cost of generating a multi-squaring table). *Step 1 of Algorithm 22 takes at most one $q^{k'}$-exponentiation in $\mathbb{F}_{q^{d_{ext}}}[x]/(H)$, $(d_h - 1 - \ )(d_h + 1)q^{k'}$ field multiplications, $(d_h - 1 - \ )d_h q^{k'}$ field subtractions and one inverse in $\mathbb{F}_{q^{d_{ext}}}^{\times}$.*

Proof. Step 1 costs one $q^{k'}$-exponentiation for the computation of $T_\ $ if $\ = 0$, else the cost is null because $x^{q^{k'}}$ is already reduced modulo H. Else, the modular reduction is performed to compute T_j for $j >\ $. In this case, T_j can be computed as $T_{j-1} x^{q^{k'}}$, which is a shift of coe cients, then we reduce a degree $d_h - 1 + q^{k'}$ polynomial by a degree d_h polynomial. This can be computed with the classical Euclidean division (Algorithm 16), requiring one inverse in $\mathbb{F}^{\times}_{q^{d_{ext}}}$, $q^{k'}(d_h + 1)$ field multiplications and $q^{k'} d_h$ field subtractions. Finally, the number of these Euclidean divisions is $d_h - 1 -\ $, implying $(d_h - 1 -\ )(d_h + 1)q^{k'}$ field multiplications, $(d_h - 1 -\ )d_h q^{k'}$ field subtractions but only one inverse in $\mathbb{F}^{\times}_{q^{d_{ext}}}$ since the divisor H does not change. □

Then, we evaluate the cost of Steps 2 and 3, which corresponds to the cost of Algorithm 22 when the multi-squaring table of H is precomputed.

Lemma 3 (Cost of the Frobenius map with a precomputed table)**.** *Steps 2 and 3 of Algorithm 22 take at most $(l_q - 1)d_h$ field exponentiations to the power $q^{k'}$, l_r exponentiations to the power q in $\mathbb{F}_{q^{d_{ext}}}[x]/(H)$, $(l_q - 1)(d_h - 1 -\ )d_h$ field multiplications and $(l_q - 1)\ $ field additions.*

Proof. Step 2 requires $l_q - 1$ steps of multi-squarings. Each step costs d_h field exponentiations to the power $q^{k'}$, $d_h - 1 -\ $ multiplications of a scalar by a degree $d_h - 1$ polynomial and field additions. Then, Step 3 requires l_r exponentiations to the power q in $\mathbb{F}_{q^{d_{ext}}}[x]/(H)$. By multiplying $l_q - 1$ by the cost of one multi-squaring, and by adding the cost of Step 3, we obtain the announced result. □

When d_h is small enough compared to k, the cost of generating the multi-squaring table is small enough to make attractive the steps of size k'. Otherwise, doing steps of size one with the classical repeated squaring algorithm is the most interesting method. For both methods, we can use the modular composition to reduce the number of steps. Each modular composition divides by two the number of steps that we will perform with one of both previous methods.

5.4.6 Frobenius Trace

The problem of the Frobenius trace is to compute

$$Tr_k(x) = \sum_{i=0}^{k-1} x^{q^i} \bmod H,$$

or more generally $Tr_k(X)$ for $X = r_0 x$ and $r_0 \in \mathbb{F}^{\times}_{q^{d_{ext}}}$. The Frobenius trace is particularly useful to compute the roots of split and squarefree polynomials in characteristic two (Section 5.4.7). Moreover, we have a direct link between Frobenius map and Frobenius trace:

$$x^{q^k} - x = Tr_k(x)^q - Tr_k(x).$$

We can adapt the strategies of Section 5.4.5 to compute the trace.

Repeated squaring algorithm. The powers q^i of x can be computed with $k - 1 - \lceil \log_q(d_h) \rceil$ modular exponentiations to the power q in $\mathbb{F}_{q^{d_{ext}}}[x]/(H)$. Then, the trace is obtained by summing them.

Big step in the repeated squaring algorithm. $Tr_k(x)$ is a polynomial whose all non-zero coe cients are one. As in Algorithm 21, we can double the degree of $Tr_k(x)$ by using the modular composition [161, Algorithm 14.55]. To do it, the following properties based on the linearity of the Frobenius endomorphism (Equation (5.3)) are useful:

$$\begin{aligned} Tr_{2k}(x) &= Tr_k(x)^{q^k} + Tr_k(x) \bmod H \\ Tr_{k+1}(x) &= Tr_k(x)^q + x \bmod H. \end{aligned} \tag{5.14}$$

We propose Algorithm 23 which is an adaptation of [161, Algorithm 14.55] for computing the trace. This version computes the Frobenius map to perform q^k-th exponentiation with the modular composition. The latter can be replaced by repeated q-exponentiations (cf. Section B.6) or by using a multi-squaring approach (Section 5.4.5).

Algorithm 23 Frobenius trace using the left-to-right square-and-multiply algorithm on $k-1$. This requires $2r-2$ modular compositions.

1: function FrobTraceLR $r_0 \in \mathbb{F}^{\times}_{q^{d_{ext}}}$, $H \in \mathbb{F}_{q^{d_{ext}}}[x]$, $k \in \mathbb{N}^*$
2: $r \leftarrow \lfloor \log_2(k-1) \rfloor$
3: $y \leftarrow r_0 x$
4: $Y_0 \leftarrow y$
$i = 1$.
5: $X_0 \leftarrow x^q \bmod H$
6: $Y_1 \leftarrow y^q + y \bmod H$ — Square step.
7: $b_0 \leftarrow 1$
8: $b_1 \leftarrow \frac{k-1}{2^{r-1}}$
9: if $b_1 \bmod 2 = 1$ then — Multiply step.
10: $Y_1 \leftarrow Y_1^q + y \bmod H$ — $Tr_{b_1}(y) = Tr_{b_1-1}(y)^q + y$.
11: end if
12: for i from 2 to r do
13: $X_{i-1} \leftarrow X_{i-2}^{(b_{i-2})} \circ X_{i-2} \bmod H$ — Square step from $i-1$.
14: if $b_{i-1} \bmod 2 = 1$ then — Multiply step from $i-1$.
15: $X_{i-1} \leftarrow X_{i-1}^q \bmod H$
16: end if — Square step.
17: $Y_i \leftarrow Y_{i-1}^{(b_{i-1})} \circ X_{i-1} + Y_{i-1}$ — $Tr_{2b_{i-1}}(y) = Tr_{b_{i-1}}(y)^{q^{b_{i-1}}} + Tr_{b_{i-1}}(y)$.
18: $b_i \leftarrow \frac{k-1}{2^{r-i}}$
19: if $b_i \bmod 2 = 1$ then — Multiply step.
20: $Y_i \leftarrow Y_i^q + y \bmod H$ — $Tr_{b_i}(y) = Tr_{b_i-1}(y)^q + y$.
21: end if
22: end for
23: return Y_r — $i = r$ and $Y_r = Tr_k(r_0 x) \bmod H$.
24: end function

5.4.7 Split Root Finding in Characteristic Two

In this section, the root finding algorithms are dedicated to split and squarefree polynomials in characteristic two [161, Exercise 14.16]. Algorithms in odd characteristic can be found in [161, Algorithms 14.8 and 14.10]. Algorithm 24 finds the roots of a degree-s polynomial with $O((d_{ext} + \log(s))M(s)\log(s))$ operations in $\mathbb{F}_{2^{d_{ext}}}$ [161, Theorem 14.11 adapted for $r = s$ and $d = 1$].

Algorithm 24 Algorithm to find the roots of a split and squarefree monic univariate polynomial in characteristic two.

1: **function** FindRootsSplit($H \in \mathbb{F}_{2^{d_{ext}}}[x]$)
2:     **if** $\deg(H) < 1$ **then**
3:         **return** $\emptyset$
4:     **else if** $\deg(H) = 1$ **then**
5:         **return** List(h_0) ▷ We create a list with the root of H and return it.
6:     **else**
7:         **repeat**
8:             $r \in_R \mathbb{F}^{\times}_{2^{d_{ext}}}$ ▷ The notation $\in_R$ stands for randomly sampling.
9:             $T \leftarrow \mathrm{Tr}_{d_{ext}}(rx) \bmod H$
10:            $P \leftarrow \mathrm{GCD}(H, T)$ ▷ We can assume that P is chosen monic.
11:        **until** P is a non trivial divisor of H
12:        $Q \leftarrow H/P$ ▷ Quotient of the Euclidean division.
13:        **return** Concat(FindRootsSplit(P),FindRootsSplit(Q)) ▷ The concatenation of the lists is returned.
14:    **end if**
15: **end function**

Solving $X^2 + X + A = 0$ with the half-trace. Algorithm 24 can be accelerated by optimizing the base cases. When d_{ext} is odd, the degree-two polynomials can be solved more efficiently [159]. To do it, note that solving an equation $x^2 + Bx + C = 0$, with $B \in \mathbb{F}^{\times}_{2^{d_{ext}}}$ and $C \in \mathbb{F}_{2^{d_{ext}}}$, is equivalent to solving $X^2 + X + A = 0$ with $BX = x$ and $A = CB^{-2}$. If R is a solution of this new equation, then $R + 1$ is the other root by linearity of the Frobenius endomorphism. When d_{ext} is odd, R is the so-called half-trace [159]:

$$R = \mathrm{HTr}_{d_{ext}}(A) = \sum_{i=0}^{\frac{d_{ext}-1}{2}} A^{2^{2i}} = \sum_{i=0}^{\frac{d_{ext}-1}{2}} A^{4^i}. \tag{5.15}$$

The latter can be computed efficiently with the strategies of Section 5.4.6 (by considering $R = \mathrm{Tr}_{\frac{d_{ext}+1}{2}}(A)$ with $q = 4$). Finally, the roots of the initial equation are BR and $BR + B$.

Solving $X^2 + X + A = 0$ as a linear system over $\mathbb{F}_2$. Here, the idea is to search the binary coefficients of (R) (Equation (2.5)). Let:

$$R = \sum_{i=0}^{d_{ext}-1} r_i \;{}_i, \quad A = \sum_{i=0}^{d_{ext}-1} a_i \;{}_i, \quad \text{with } r_i, a_i \in \mathbb{F}_2.$$

We search R such that $R^2 + R = A$. This implies that:

$$\sum_{i=0}^{d_{ext}-1} r_i \,{}_i + r_i^2 \,{}_i^2 = \sum_{i=0}^{d_{ext}-1} r_i \,\, {}_i + {}_i^2 \,\, = \sum_{i=0}^{d_{ext}-1} a_i \,{}_i .$$

So, let M_2 be the following matrix:

$$M_2 = \begin{pmatrix} {}_0^2 + {}_0 \\ {}_1^2 + {}_1 \\ \vdots \\ {}_{d_{ext}-1}^2 + {}_{d_{ext}-1} \end{pmatrix} = I_{d_{ext}} + \begin{pmatrix} {}_0^2 \\ {}_1^2 \\ \vdots \\ {}_{d_{ext}-1}^2 \end{pmatrix} \in \mathcal{M}_{d_{ext}}\left(\mathbb{F}_2\right). \quad (5.16)$$

Thus, we can find R by solving the following classical linear algebra problem:

$$\begin{pmatrix} r_0 & r_1 & \cdots & r_{d_{ext}-1} \end{pmatrix} \cdot M_2 = \begin{pmatrix} a_0 & a_1 & \cdots & a_{d_{ext}-1} \end{pmatrix}.$$

With this method R is obtained by solving a linear system whose matrix depends only on the field representation (Section 5.2) and so can be precomputed. Note that $\mathrm{rank}(M_2) = d_{ext} - 1$ since $R + 1$ is also solution.

Then, we can slightly modify M_2 to obtain an invertible matrix $M_2 \in GL_{d_{ext}}\left(\mathbb{F}_2\right)$. Its inverse can be precomputed, then re-used to find R with a vector-matrix product over $\mathbb{F}_2$. In practice, $\mathcal{B}$ is the canonical basis. This implies that the first row of M_2 is null. To obtain an invertible matrix which gives R from A, we set one of the coefficients of this row to one. We remark that for the irreducible trinomials or pentanomials used to define $\mathbb{F}_{2^{d_{ext}}}$ (Section 5.3.3), if d_{ext} is odd then we can set the coefficient $(0, 0)$ of M_2 to one. Else, we can set the coefficient $(0, d_{ext} - k_1)$ of M_2 to one. We refer to Section B.7 for more details.

Solving $X^3 + sX + p = 0$ as a linear system over $\mathbb{F}_2$. Now, we study how to find the roots of a degree-three split polynomial $x^3 + Bx^2 + Cx + D \in \mathbb{F}_{2^{d_{ext}}}[x]$. By using the classical change of variable $X = x + B$, we obtain $X^3 + sX + p = 0$ with $s = B^2 + C$ and $p = D + BC$. Then, we multiply $X^3 + sX + p$ by X to obtain $X^4 + sX^2 + pX$. The result is $\mathbb{F}_2$-linear: all powers of X are powers of two. Similarly to the previous method, we can solve $X^4 + sX^2 + pX$ with linear algebra. Here, we obtain:

$$M_3 = \begin{pmatrix} {}_0^4 + s\,{}_0^2 + p\,{}_0 \\ {}_1^4 + s\,{}_1^2 + p\,{}_1 \\ \vdots \\ {}_{d_{ext}-1}^4 + s\,{}_{d_{ext}-1}^2 + p\,{}_{d_{ext}-1} \end{pmatrix} \in \mathcal{M}_{d_{ext}}\left(\mathbb{F}_2\right).$$

Then, a solution R of $X^4 + sX^2 + pX$ is given by

$$\begin{pmatrix} r_0 & r_1 & \cdots & r_{d_{ext}-1} \end{pmatrix} \cdot M_3 = \begin{pmatrix} 0 & 0 & \cdots & 0 \end{pmatrix},$$

i.e. R is in the kernel of M_3. On the one hand, the extra solution $X = 0$ is absorbed by the kernel of M_3. On the other hand, zero cannot be solution of $X^3 + sX + p$. Indeed, $p = 0$ would imply that $X^3 + sX$ admits a double (or triple) root, which is impossible since $x^3 + Bx^2 + Cx + D$ is assumed split. So, the non-zero solutions of $X^4 + sX^2 + pX$ are the solutions of $X^3 + sX + p$. This implies that the dimension of the kernel of M_3 is two, and its basis allows to generate the three (distinct) roots of $X^3 + sX + p$.

5.4.8 Root Finding

Finally, we have all tools to introduce a root finding algorithm of $H \in \mathbb{F}_{q^{d_{ext}}}[x]$. Algorithm 25 describes Berlekamp's algorithm [161, Algorithm 14.15]. The main idea is to remark that $x^{q^{d_{ext}}} - x$ vanishes on all elements of $\mathbb{F}_{q^{d_{ext}}}$. We can then compute G the GCD of H and $x^{q^{d_{ext}}} - x$. G has the same roots as H but with a minimum degree (which is the number of roots). In general, the degree of G is small. The strategy is then to apply the so-called equal-degree factorization, which returns the factorization in irreducible polynomials of equal degree. Used in Berlekamp's algorithm, we obtained degree-one factors, and so all roots of G. This is turned to be cheap. Indeed, let s be the degree of G, the equal-degree factorization costs $O((d_{ext}\log(q) + \log(s))M(s)\log(s))$ operations in $\mathbb{F}_{q^{d_{ext}}}$ [161, Theorem 14.11 adapted for $r = s$ and $d = 1$]. Because the degree of $x^{q^{d_{ext}}} - x$ is big, we reduce $x^{q^{d_{ext}}} - x$ by H by using methods described in Section 5.4.5, before computing the GCD (Section 5.4.1).

Algorithm 25 Algorithm to find the roots of a univariate polynomial.

1: **function** FindRoots($H \in \mathbb{F}_{q^{d_{ext}}}[x]$)
2: $\quad R_2 \leftarrow x^{q^{d_{ext}}} - x \bmod H$ ▷ Computation of the Frobenius map.
3: $\quad G \leftarrow \mathrm{GCD}(H, R_2)$ ▷ Computation of the GCD.
4: $\quad s \leftarrow \deg(G)$ ▷ Number of roots of H.
5: $\quad$ **if** $s > 0$ **then**
6: $\quad\quad L_{roots} \leftarrow$ list of all roots of G, computed by the equal-degree factorization algorithm described in [161, Section 14.3] ▷ In characteristic two, call to Algorithm 24.
7: $\quad\quad$ **return** (s, L_{roots})
8: $\quad$ **end if**
9: $\quad$ **return** $(s, \emptyset)$
10: **end function**

The complexity of Algorithm 25 is given by the following general result [161, Corollary 14.16].

Theorem 4. *Let $\mathbb{F}_q$ be a finite field, and $M_q(D)$ be the number of operations in $\mathbb{F}_q$ to multiply two polynomials of degree less or equal to D. Given $f \in \mathbb{F}_q[x]$ of degree D, we can find all roots of f in $\mathbb{F}_q$ using an expected number of*

$$O\left(M_q(D)\log(D)\log(Dq)\right)$$

or $O\left(D\log(q)\right)$ operations in $\mathbb{F}_q$.

Chapter 6

Software Implementation

In this chapter, we present important considerations [104] about the processor and memory caches, allowing to write an e cient code (Section 6.1). Then, we describe the methodology of benchmarking, as well as state-of-the-art libraries (Section 6.2). Finally, we present classical techniques to make constant-time our implementation (Section 6.3). We use secret data in constant-time to be immune against timing attacks (Section 4.7).

6.1 Hardware Considerations

In this section, we study the processor and the cache hierarchy (Sections 6.1.1 and 6.1.2). We highlight crucial points which impact security and performance. Then, we introduce vector instructions that we use to improve the performance by significant speed-ups (Section 6.1.3). We conclude by the use of the compiler (Section 6.1.4). The compiler flags directly impact security and performance.

6.1.1 Processor

Here, we give a simplified view of how processors work. A processor has a fixed number of -bit registers, depending on the architecture (typically, $= 64$ on modern computers). These registers can be filled with a `load` instruction, and saved with a `store` instruction, both by using a memory access. Then, operations between registers are hardware instructions. These instructions flow in a pipeline, which processes them one by one like an assembly line. The latter is divided into several stages. For example, the classical RISC pipeline [104] contains the following five stages:

1. Instruction fetch: the instruction is loaded from the L1 instruction cache (Section 6.1.2),
2. Instruction decode: the instruction is decoded and the registers corresponding to operands are read from the register file (register fetch),
3. Execute: a logical or arithmetic instruction is executed here, or a memory address is computed for the `load` and `store` instructions,
4. Memory access: the memory access of a `load` or `store` instruction is executed here,

5. Register write back: the result of the instruction is written in the register file. This stage does not have effect for a `store` instruction, because the latter does not have result.

At each cycle, one instruction is performed for each stage. In practice, the cost of one stage can be slower than one cycle. In particular, the number of cycles to perform Stage 3 depends on the instruction. This number is commonly called Cycle Per Instruction (CPI) or throughput. Then, once one instruction is executed, the result is not instantly available. The required number of cycles to obtain the result is called latency, and is specific to each instruction. If an instruction requires the result of a previous instruction, these instructions have a data dependency. In this case, the instruction is not launched while the result is not available, and `nop` (no operation) instructions are placed before this instruction. So, data dependency instructions can penalize performance, particularly if the first dependent instruction has a large latency.

6.1.2 Memory Cache

In the previous section, we do not consider the number of cycles to load and store data. The memory is managed according a complex system of memory hierarchy.

Concept and use of the memory hierarchy. Here again, we give a simplified view, assuming that the processor has a memory cache.

- When a `load` instruction is performed, the data is sought in the first level of cache (L1 data cache). If the data is present, it is a cache hit. The data is loaded, costing a small number of cycles. If the data is missing, it is a cache miss. In this case, the data is sought in the upper memory level (in general, it is the L2 cache). If the data is present in the L2 cache, it is a cache hit. Then the sought data, as well as some contiguous data, are transferred in the L1 cache. This generates a larger cycle penalty than a L1 cache memory access. If the data is missing in the L2 cache, it is a cache miss and the principle described here is applied recursively, increasing each time the penalty cache to transfer data from a level +1 to a level .

- When instructions have to be fetched, they are sought in the L1 instruction cache. If the instruction is missing, the previous process is applied. Note that for certain processors, data and instructions are stored is the same L1 cache.

- When a `store` instruction is performed, the process used to update different memory levels is slightly more complex [104], and is not described here.

Derived principles. All in all, these lead to the following general rules. Firstly, we know that the memory complexity of an algorithm has an impact on the performance, depending on the highest memory level which is required to load and store all data used. The size of data and caches directly impacts the performance. Secondly, we know that the data are loaded by contiguous data block. So, contiguous memory accesses are really efficient and have to be preferred. Finally, the time of accessing a table from a secret index depends on this secret data, and so reveals information. Cache-timing attacks exploit this kind of information leakage (Sections 4.7 and 6.3.3).

Multicore processor and cache. Modern processors often have three levels of cache. On modern Intel processors, including these of our experimental platform (Section 6.2.2), all cores have a dedicated L1 instruction cache, L1 data cache and L2 cache, whereas the L3 cache is unique and shared by all cores.

6.1.3 Vector Instructions

On a -bit architecture (typically, = 64 on modern computers), the hardware instructions are performed on -bit registers. However, almost all modern processors have Single Instruction on Multiple Data (SIMD) instruction sets. These instructions compute independent instructions in parallel on larger registers, allowing to accelerate the implementations. In `MQsoft` (Chapter 9), we exploit vector instructions. We use `C` intrinsics which generate vector instructions during the compilation step. Unlike assembly code, the use of intrinsics does not depend on the targeted architecture. This makes the code portable.

Intel processors. Nowadays, Intel processors are widely used and propose a large number of SIMD instruction sets [107]. The main instruction sets are:

- The Streaming SIMD Extensions (SSE) family. The computations are performed on 128-bit registers. The SSE instruction set only supports floating-point data whereas the SSE2 instruction set supports floating-point and integer data. The SSE3, SSSE3, SSE4.1 and SSE4.2 instruction sets provide new instructions to improve performances of certain operations.
- The Advanced Vector Extensions (AVX) family. The computations are performed on 256-bit registers, or on 128-bit registers thanks to the backward compatibility. As for the SSE instruction set, the AVX instruction set only supports floating-point data whereas the AVX2 instruction set supports floating-point and integer data.
- The AVX-512 family. As indicated by its name, the computations are performed on 512-bit registers. The AVX-512 instruction set is divided into subsets of instructions. Each of them has a specific feature.

In `MQsoft`, we exploit the SSE and AVX families of instruction set, as well as certain non parallel instructions (e.g. `POPCNT` and `PCLMULQDQ`).

Main vector instructions. All classical operations on 64-bit registers, such as the logical bit shifts, the mathematical operators `ADD`, `SUB`, `MUL`, the bitwise logical `AND`, `OR`, `XOR` operators, and the equality and comparison tests, can be performed in parallel on larger registers. Here, we explain other vector instructions[1] that `MQsoft` exploits.

- `POPCNT`[2]: this instruction counts the number of bits set to one in 64-bit integers. It is used to speed up the dot product of vectors in $\mathbb{F}_2$.
- `PCLMULQDQ`[2] (carry-less multiplication): this instruction computes the product of two binary polynomials such that their degree is strictly less than 64. Inputs and output are 128-bit registers.

[1] For further information, we refer to the Intel Intrinsics Guide (https://software.intel.com/sites/landingpage/IntrinsicsGuide/#).

[2] This instruction is available in a instruction set of one instruction.

- PSLLDQ[3] and PSRLDQ[3]: these instructions from SSE2 compute respectively the logical left shift and logical right shift of a 128-bit register by a multiple of eight bits.

- PALIGNR[3]: this instruction from SSSE3 concatenates two 128-bit registers, shifts the concatenation to the right by a multiple of eight bits, then returns the 128 lower bits of the result.

- PSHUFB[3]: this instruction from SSSE3 takes sixteen indices on four bits, and looks up the corresponding 8-bit elements in a 16-byte lookup table. In fact, each index is on eight bits, but only the four lower bits are considered. However, if the highest bit is set, then the corresponding output will be null.

- VPERMQ: this instruction from AVX2 permutes the 64-bit parts of a 256-bit register. In particular, it can duplicate one 64-bit part four times.

- VPMASKMOVQ: when called with two operands, this instruction from AVX2 loads four contiguous 64-bit integers from an array, then applies a mask which permits to set to zero some of these 64-bit integers. VPMASKMOVQ can also be called with three operands. In this case, the third operand is stored as four contiguous 64-bit integers in an array. Here, the mask allows to choose what integers are stored. The VPMASKMOVQ instruction may not be constant-time, in particular if the execution of the load instruction depends on the mask.

The prefix P (respectively VP) of an instruction means that it is a part of the SSE (respectively AVX) family. The suffixes B, W, D, Q and DQ, respectively for Byte, Word, Doubleword, Quadword and Double Quadword, mean that registers are respectively used as 8-bit, 16-bit, 32-bit, 64-bit and 128-bit data blocks. When two suffixes are used, the first corresponds to the input, whereas the second corresponds to the output. For example, PCLMULQDQ computes the carry-less multiplication (CLMUL) of two quadwords, which generates a double quadword. So, the suffix QDQ is used.

The PCLMULQDQ instruction is a crucial instruction for multiplying in $\mathbb{F}_2[x]$. This instruction can be performed two or four times in parallel with the VPCLMULQDQ[2] instruction. Unlike the previous instructions, the VPCLMULQDQ instruction requires the use of AVX-512. This instruction is recent (September 2019 on Ice Lake Intel processors) and we do not consider it in MQsoft.

Performance of vector instructions. Most of instructions require one cycle per instruction, then the result is available after one cycle of latency. In Table 6.1, we highlight faster and slower instructions on Intel processors. The processors Ivy Bridge, Haswell, Broadwell and Skylake are respectively the third, fourth, fifth and sixth generations of Intel processor. The bitwise logical operators are the fastest instructions. They cost one third of cycle per instruction. This means that in one cycle, the same instruction can be computed up to three times in parallel. This also means that one instruction alone costs one complete cycle per instruction. In order to improve the performances of a software, we can take advantage of this behavior to decrease the practical number of cycles. Then, we remark that the cost of certain instructions (e.g. PCLMULQDQ) varies in function of the processors. In this case, the implementation should be specialized for each kind of processor. Finally, the large latency should be covered by calling instructions independent of the result, when it is possible.

[3] This instruction can be performed two times in parallel in AVX (and four times in AVX-512). The corresponding name takes the letter V at the beginning of the instruction name.

instruction	processor	latency	CPI
PAND, POR, PXOR	all	1	1/3
PSLL*, PSRL* $* \in \{$W, D, Q$\}$	Skylake	1	1/2
	Haswell	1	1
PSLLDQ, PSRLDQ	Skylake	1	1
	Haswell	1	1
PCMPEQ{B, W, D, Q}	all	1	1/2
PCLMULQDQ	Skylake	6	1
	Broadwell	5	1
	Haswell	7	2
	Ivy Bridge	14	8
POPCNT	all	3	1
VPERMQ	all	3	1
VPMASKMOVQ (load)	Skylake	8	1/2
	Haswell	8	2
VPMASKMOVQ (store)	Skylake	6	1
	Haswell	-	2

Table 6.1: Cost of vector instructions on Intel processors. The PSLL* and PSRL* instructions correspond to logical shifts by block, whereas PCMPEQ{B, W, D, Q} is the equality test by block.

6.1.4 Compiler Flags

In the previous sections, we studied how to optimize the instructions on a processor. In MQsort, we use a C implementation. Therefore, the generation of these instructions is done by the compiler. In our tests, we use the GNU Compiler Collection (GCC). Here, we study the options which firstly, allow the compiler to interpret the intrinsics, then secondly, improve the efficiency of the generated code. We also note that the optimization options can impact the security of the implementation.

- -msse, -msse2, -msse3, -mssse3, -msse4.1, -msse4.2, -mavx, -mavx2, -mpclmul and -mpopcnt: these flags enable the vector instruction sets described in Section 6.1.3.
- -march=native: this flag enables all available vector instruction sets on the current machine.
- -mtune=native: this flag optimizes the use of instructions in function of the targeted architecture. The target native can be replaced by haswell, skylake or other architectures.
- -O1, -O2, -O3 and -O4: these flags provide four levels of optimization of the code by the compiler. Each of them combines a set of options to accelerate the generated code. The use of these options modifies the initial implementation, and can generate insecure instructions against timing attacks.
- -funroll-loops: this flag unrolls loops whose number of iterations is known. The code becomes larger and may or may not run faster.

6.2 Experimental Platform and Tools

6.2.1 Library and Software

The MQsoft library [84] requires external supports for random bytes generation and hash functions. Natively, we use the following C libraries.

- XKCP (Extended Keccak Code Package) [89]. XKCP is a software interface for Keccak-based functions. This implementation is immune against timing attacks (Section 4.7). MQsoft uses it mainly for the SHA-3 hash functions and SHAKE extendable-output hash functions [126].

- OpenSSL [146]. OpenSSL is a cryptographic toolkit implementing, in particular, the Transport Layer Security (TLS) protocol. We use OpenSSL for compatibility with the deterministic random bytes generator provided by NIST during the PQC standardization process (which is based on AES from OpenSSL). We also propose options to MQsoft to use the SHA-1, SHA-2, SHA-3 and SHAKE functions from OpenSSL.

The performance of MQsoft is based on an e cient arithmetic (Chapter 5). In the next chapters, we will benchmark our software toolkit against the following softwares or libraries, which are state-of-the-art about computer algebra.

- Magma [34, version 2.23-6]. Magma is a computer algebra software, well-known to be very e cient over finite fields. Magma is also a reference to compute Gröbner bases (Section 4.4). In Section 9.3, we use the version 2.24-2 for benchmarks.

- NTL [153, version 10.5.0], installed with GMP [97, version 6.1.2]. NTL is one of the best libraries for number theory. It is implemented in C++ programming language. In Section 9.3, we use the last versions of NTL and GMP, which are respectively 11.4.3 and 6.2.0. We will conclude that our library MQsoft is still faster than NTL.

- gf2x[4] [41, version 1.2]. gf2x is a C implementation of the state-of-the-art multiplications in $\mathbb{F}_2[x]$. The multiplication algorithm depends on the degree of the operands and the available vector instruction sets. An option of MQsoft allows to use the gf2x multiplication instead of the native multiplication. This is especially useful to accelerate the implementation when the PCLMULQDQ instruction is not available. However, some multiplications of the software are not implemented in constant-time. For example, the generic64 implementation is vulnerable to cache-timing attacks (Section 4.7).

We have also studied the performance of FLINT [103, version 2.5.3], which is a fast C library for number theory. However, its use is irrelevant in our context, because the implementation is not optimized for binary fields.

Some of the softwares used can be optimized during the installation process. We use the Haswell implementation of the XKCP to improve performances of MQsoft by using the AVX2 instruction set. Magma is used with magma.avx64.dyn.exe to take advantage of vector instructions. In our tests, the avx64 version mainly optimizes the performance of multiplication in binary fields.

[4] https://gitlab.inria.fr/gf2x/gf2x

6.2.2 Platform and Benchmarking Methodology

computer	processor	cores	frequency	max freq.	architecture
LaptopS	Intel Core i7-6600U CPU	2	2.6 GHz	3.4 GHz	Skylake
ServerH	Intel Xeon CPU E3-1275 v3	4	3.5 GHz	3.9 GHz	Haswell
DesktopS	Intel Core i7-6700 CPU	4	3.4 GHz	4.0 GHz	Skylake

Table 6.2: Processors.

<table>
<tr><th>computer</th><th>operating system</th><th>L1d</th><th>L1i</th><th>L2</th><th>L3</th><th>RAM</th></tr>
<tr><td>LaptopS</td><td>Ubuntu 16.04.5 LTS</td><td rowspan="3">32 KiB</td><td rowspan="3">32 KiB</td><td rowspan="3">256 KiB</td><td>4 MiB</td><td rowspan="2">32 GB</td></tr>
<tr><td>ServerH</td><td>CentOS Linux 7 (Core)</td><td rowspan="2">8 MiB</td></tr>
<tr><td>DesktopS</td><td>Debian GNU/Linux 9</td><td>16 GB</td></tr>
</table>

Table 6.3: OS and memory. All cores have the same cache size. The L3 cache is shared.

Tables 6.2 and 6.3 summarize the main information about the platform used in the experimental measurements. The measurements used one core of the CPU, and the `C` code was compiled with `gcc -O3 -mavx2 -mpclmul -mpopcnt -funroll-loops`. We used the version 6.4.0 of GCC. Turbo Boost and Enhanced Intel Speedstep Technology were disabled to have more accurate measurements, except when we used DesktopS. Turbo Boost allows the processor to reach its maximum frequency while its temperature is less than its physical limit. As soon as this limit is reached, the frequency decreases drastically to permit its cooling. Enhanced Intel Speedstep Technology allows the operating system to modify the frequency, in order to optimize the threshold between power consumption and performance. Disabling Turbo Boost decreases performances whilst disabling Enhanced Intel Speedstep Technology increases power consumption. On the other hand, the frequency of the processor remains constant, which stabilizes running time measurements. In practice, Turbo Boost generates a speed-up of 1.2 on LaptopS and 1.1 on ServerH.

6.3 Constant-Time Implementation

Here, we present classical methods to circumvent some timing attacks (Section 4.7). We use similar techniques to make `MQsoft` constant-time. In particular, we explain in Section 7.4.4 how to remove the information leakages of the Gaussian elimination (operation used, for instance, in `GeMSS` during the keypair generation, Chapter 7), due to the null pivots. In Section 7.4.11, we present the constant-time GCD algorithm (key step in `GeMSS` signing process) which was only introduced in 2019 [19]. Our adaptation of this GCD algorithm for `GeMSS` is detailed in Section B.8.

6.3.1 Variable-Time Instructions

In Section 6.1.1, we remarked that the number of spent cycles depends on the instruction used. On certain architectures and for certain instructions, the number of cycles depends on the inputs. This behavior is dangerous when secret data are used, and so these instructions should be prohibited in this context. Here, we list some instructions which are not always implemented in constant-time:

- Mathematical functions, such as $\cos$, $\sin$, $\tan$, ... but in general, these instructions are not used in cryptography.
- Integer multiplications. This operation is crucial for arithmetic in odd prime fields and their extension fields [51].
- Integer and floating-point divisions. These operations can be used for arithmetic in odd prime fields and their extension fields.

In `MQsoft`, we avoid using these instructions with secret data.

6.3.2 Constant-Time Conditional Statements

In Algorithm 15, we removed the information leakage of the square-and-multiply algorithm via a constant-time conditional multiplication. Here, we study a typical example of insecure algorithm against timing attacks on conditional statements. Let Algorithm 26 be a possible implementation of the conditional move (`cmov`) instruction.

Algorithm 26 Variable-time conditional move from a to c, with x a boolean.

```
1: function cmov_var(c, a, x)
2:     if x = 1 then
3:         c ← a
4:     end if
5:     return c
6: end function
```

Algorithm 26 generates an information leakage on x because of the conditional statement. The latter can be easily avoided with the following formula:

$$c = a \cdot x + c \cdot (1 - x).$$

When x is one, the right operand is null and c is replaced by a, whereas when x is zero, the left operand is null and c is replaced by c. This method allows to choose one operand in constant-time. It can be accelerated by replacing the addition by the bitwise `OR`, and by replacing the multiplication by the bitwise `AND` and by using a mask. If x is a boolean, and if x is stored with two's complement, then the memory representation of $-x$ is the boolean duplicated on each bit. We obtain:

$$c = \big(a \texttt{ AND } (-x)\big) \texttt{ OR } \big(c \texttt{ AND } (x - 1)\big).$$

In practice, it seems slightly better to implement the `cmov` instruction as follows:

$$c = c \texttt{ XOR } \big((a \texttt{ XOR } c) \texttt{ AND } (-x)\big). \tag{6.1}$$

When x is one, c is replaced by $c \texttt{ XOR } a \texttt{ XOR } c$ which is equal to a, whereas when x is zero, c is replaced by c.

The strategy of Equation (6.1) can be used to implement a conditional swap of two elements (Algorithm 27). We use it for constant-time implementations of sort [17] required in Section 7.1.3 and GCD required in Section 7.4.11 and detailed in Section B.8.

Algorithm 27 Constant-time conditional swap. Swap a and c if x is one.

```
1: function cswap(a, c, x)
2:     s ← (a XOR c) AND (−x)
3:     a ← a XOR s
4:     c ← c XOR s
5:     return (a, c)
6: end function
```

Remark 3. We often generate a mask $m \in \{-1, 0\}$ from the i-th bit of a -bit word c, for $0 \leqslant i <$.
This can be achieved in constant-time as follows:

- $m \leftarrow -\ (c \gg i)$ AND 1 , where $\gg$ stands for logical right shift,
- $m \leftarrow\ \ c \ll (\ \ -1-i)\ \ \gg_a (\ \ -1)$, where $\gg_a$ stands for arithmetic right shift.

The latter is particularly e cient to generate a mask from the sign bit of a signed integer, since this is equivalent to $c \gg_a (\ \ -1)$. In this thesis, we only use logical shifts.

6.3.3 Constant-Time Use of Tables

Accessing a table reveals information about the index, which can be exploited by cache-timing attacks (Section 4.7). This leakage can be avoided by reading all elements of the table, and by selecting the targeted element with a conditional move. By optimizing the conditional move, we obtain Algorithm 28. Here, == is the equality operator of the C programming language, which returns 1 for true and 0 for false. This instruction is constant-time, but the compiler can replace it by a dangerous conditional statement. So in practice, this instruction (as well as the compiler) should be used carefully.

Algorithm 28 Constant-time access to the j-th element from a table T of elements.

```
1: function accessTable(T, , j)
2:     a ← 0
3:     for i from 0 to  − 1 do
4:         a ← a OR (T[i] AND −(j == i))          ▷ Optimized conditional move.
5:     end for
6:     return a
7: end function
```

In a variable-time implementation, the use of lookup tables allows to obtain the result in only one memory access. A constant-time implementation such as Algorithm 28 makes this strategy slow, with memory accesses instead of one.

Part II

Main Contributions

Chapter 7

GeMSS – a Great Multivariate Short Signature

The purpose of this chapter is to present GeMSS: a Great Multivariate Short Signature. GeMSS [50] is an alternate candidate of the third round of the NIST PQC standardization process [130]. As suggested by its name, GeMSS is a multivariate-based (Chapter 2) signature scheme producing small signatures. It has a fast verification process, and a medium/large public key. GeMSS is in direct lineage from QUARTZ [134] and borrows some design rationale of the Gui multivariate signature scheme [144]. The former schemes are built from the Hidden Field Equations cryptosystem (HFE) [133, published in 1996] by using the so-called minus and vinegar modifiers, i.e. HFEv- [110] (Section 2.4.1). It is fair to say that HFE, and its variants, are the most studied schemes in multivariate cryptography. QUARTZ produces signatures of 128 bits for a security level of 80 bits and was submitted to the NESSIE competition [145] for public-key signatures. In contrast to many multivariate schemes, no practical attack has been reported against QUARTZ. This is remarkable knowing the intense activity in the cryptanalysis of multivariate schemes, e.g. [132, 111, 78, 82, 96, 95, 73, 86, 70, 29, 39, 22, 137, 160, 68] (Chapter 4). The best known attack remains [82] (Section 4.4) that serves as a reference to set the parameters for GeMSS.

GeMSS is a fast variant of QUARTZ that incorporates the latest results in multivariate cryptography to reach higher security levels than QUARTZ whilst improving efficiency. The main sections of this chapter follow the algorithm specification and supporting documentation from the call for proposals [127].

7.1 General Algorithm Specification (Round 3)

7.1.1 Parameter Space

The main parameters involved in GeMSS are:

- , the security level of GeMSS,
- D, a positive integer that is the degree of a secret polynomial. D is such that $D = 2^i$ for $i \geqslant 0$, or $D = 2^i + 2^j$ for $i \neq j$ and $i, j \geqslant 0$,

- m, the number of equations in the public-key,
- $n_{\texttt{var}}$, the number of variables in the public-key,
- $\texttt{nb_ite} > 0$, the number of iterations in the signature and verification processes (Section 4.2),
- $d_{\texttt{ext}}$, the degree of an extension field of $\mathbb{F}_2$,
- v, the number of vinegar variables (the number of variables in the public-key is $n_{\texttt{var}} = d_{\texttt{ext}} + v$),
- Δ, the number of minus equations (the number of equations in the public-key is $m = d_{\texttt{ext}} - \Delta$).

In Section 7.2, we specify precisely these parameters to achieve a security level $\lambda \in \{128, 192, 256\}$.

7.1.2 Secret-Key and Public-Key

The public-key in $\texttt{GeMSS}$ is a set $p_1, \ldots, p_m \in \mathbb{F}_2[x_1, \ldots, x_{n_{\texttt{var}}}]$ of m quadratic equations in $n_{\texttt{var}}$ variables. These equations are derived from a multivariate polynomial $F \in \mathbb{F}_{2^{d_{\texttt{ext}}}}[X, v_1, \ldots, v_v]$ with a specific form – as recalled in (7.1) – such that generating a signature is essentially equivalent to finding the roots of F (Section 2.4.1).

Secret-key. It is composed of a couple of invertible matrices $(S, T) \in GL_{n_{\texttt{var}}}(\mathbb{F}_2) \times GL_{d_{\texttt{ext}}}(\mathbb{F}_2)$ and a polynomial $F \in \mathbb{F}_{2^{d_{\texttt{ext}}}}[X, v_1, \ldots, v_v]$ with the following structure:

$$\sum_{\substack{0 \leqslant j < i < d_{\texttt{ext}} \\ 2^i + 2^j \leqslant D}} A_{i,j} X^{2^i + 2^j} + \sum_{\substack{0 \leqslant i < d_{\texttt{ext}} \\ 2^i \leqslant D}} \beta_i(v_1, \ldots, v_v) X^{2^i} + \gamma(v_1, \ldots, v_v) \in \mathbb{F}_{2^{d_{\texttt{ext}}}}[X, v_1, \ldots, v_v], \qquad (7.1)$$

where $A_{i,j} \in \mathbb{F}_{2^{d_{\texttt{ext}}}}$, each $\beta_i : \mathbb{F}_2^v \to \mathbb{F}_{2^{d_{\texttt{ext}}}}$ is linear and $\gamma : \mathbb{F}_2^v \to \mathbb{F}_{2^{d_{\texttt{ext}}}}$ is quadratic modulo $v_i^2 - v_i$. The variables $v_1, \ldots, v_v$ are called *vinegar variables*. A polynomial $F \in \mathbb{F}_{2^{d_{\texttt{ext}}}}[X, v_1, \ldots, v_v]$ with the form of (7.1) has *$\texttt{HFEv}$-shape*. By abuse of notation, we will call degree of F the (max) degree of its corresponding $\texttt{HFE}$ polynomial, i.e. D.

Remark 4. The particularity of a polynomial $F(X, v_1, \ldots, v_v)$ with $\texttt{HFEv}$-shape is that for any specialization of the vinegar variables, the polynomial F becomes a $\texttt{HFE}$ polynomial (Equation (2.6), Section 2.4.1).

The special structure of (7.1) is chosen such that its *multivariate representation* over the base field $\mathbb{F}_2$ is composed of quadratic polynomials in $\mathbb{F}_2[x_1, \ldots, x_{n_{\texttt{var}}}]$. This is due to the special exponents chosen in X (Equations (2.7) and (2.8)) that have all a binary decomposition of Hamming weight at most two.

Let $\theta = (\theta_1, \ldots, \theta_{d_{\texttt{ext}}}) \in (\mathbb{F}_{2^{d_{\texttt{ext}}}})^{d_{\texttt{ext}}}$ be a basis of $\mathbb{F}_{2^{d_{\texttt{ext}}}}$ over $\mathbb{F}_2$. We set

$$\varphi : E = \sum_{k=1}^{d_{\texttt{ext}}} o_k \cdot \theta_k \in \mathbb{F}_{2^{d_{\texttt{ext}}}} \longmapsto \varphi(E) = (o_1, \ldots, o_{d_{\texttt{ext}}}) \in \mathbb{F}_2^{d_{\texttt{ext}}}.$$

We can now define a set of multivariate polynomials $\mathbf{f} = (f_1, \ldots, f_{d_{\texttt{ext}}}) \in \mathbb{F}_2[x_1, \ldots, x_{n_{\texttt{var}}}]^{d_{\texttt{ext}}}$ derived from a $\texttt{HFEv}$ polynomial $F \in \mathbb{F}_{2^{d_{\texttt{ext}}}}[X, v_1, \ldots, v_v]$ by:

$$F\left(\varphi^{-1}(x_1, \ldots, x_{d_{\texttt{ext}}}), v_1, \ldots, v_v\right) = F\left(\sum_{k=1}^{d_{\texttt{ext}}} x_k \cdot \theta_k, v_1, \ldots, v_v\right) = \sum_{k=1}^{d_{\texttt{ext}}} f_k \cdot \theta_k = \varphi^{-1}(\mathbf{f}). \qquad (7.2)$$

To ease notations, we now identify the vinegar variables $(v_1, \ldots, v_v) = (x_{d_{ext}+1}, \ldots, x_{n_{var}})$. Also, we shall say that the polynomials $f_1, \ldots, f_{d_{ext}} \in \mathbb{F}_2[x_1, \ldots, x_{n_{var}}]$ are the *components* of F over $\mathbb{F}_2$.

Public-key. It is given by a set of m square-free quadratic polynomials in n_{var} variables over $\mathbb{F}_2$. That is, the public-key is $p = (p_1, \ldots, p_m) \in \mathbb{F}_2[x_1, \ldots, x_{n_{var}}]^m$. It is obtained from the secret-key by taking the first $m = d_{ext} -$ polynomials of:

$$f_1\ (x_1, \ldots, x_{n_{var}}) \cdot S\ , \ldots, f_{d_{ext}}\ (x_1, \ldots, x_{n_{var}}) \cdot S \quad \cdot T\ ,$$

and reducing it modulo the field equations, modulo $\langle x_1^2 - x_1, \ldots, x_{n_{var}}^2 - x_{n_{var}} \rangle$. We denote these polynomials by $p = (p_1, \ldots, p_m) \in \mathbb{F}_2[x_1, \ldots, x_{n_{var}}]^m$.

Keypair generation. We summarize the public-key/secret-key generation in Algorithm 29. It takes the unary representation of the security parameter as input. As we will see in Section 7.7, the security level of GeMSS will be a function of D, $, v$ and m. For more details on the implementation, we refer to Section 7.4.

Algorithm 29 Keypair generation in GeMSS.

1: **function** GeMSS.KeyGen 1
2: Randomly sample $(S, T) \in GL_{n_{var}}\ \mathbb{F}_2\ \times GL_{d_{ext}}\ \mathbb{F}_2$. ▷ Section 7.4.3.
3: Randomly sample $F \in \mathbb{F}_{2^{d_{ext}}}[X, v_1, \ldots, v_v]$ with HFEv-shape of degree D. ▷ Section 7.4.5.
4: $sk \leftarrow\ F, S^{-1}, T^{-1} \in \mathbb{F}_{2^{d_{ext}}}[X, v_1, \ldots, v_v] \times GL_{n_{var}}\ \mathbb{F}_2\ \times GL_{d_{ext}}\ \mathbb{F}_2$ ▷ Section 7.4.4.
5: Compute $f = (f_1, \ldots, f_{d_{ext}}) \in \mathbb{F}_2[x_1, \ldots, x_{n_{var}}]^{d_{ext}}$ such that:

$$F \sum_{k=1}^{d_{ext}} x_k \cdot \ _k, v_1, \ldots, v_v \quad = \sum_{k=1}^{d_{ext}} f_k \cdot \ _k.$$

▷ See Section 9.4.1 for details on Step 5.

6: Compute $(p_1, \ldots, p_{d_{ext}}) =$

$$f_1\ (x_1, \ldots, x_{n_{var}}) \cdot S\ , \ldots, f_{d_{ext}}\ (x_1, \ldots, x_{n_{var}}) \cdot S \quad \cdot T \bmod \langle x_1^2 - x_1, \ldots, x_{n_{var}}^2 - x_{n_{var}} \rangle$$

which lives in $\mathbb{F}_2[x_1, \ldots, x_{n_{var}}]^{d_{ext}}$. ▷ Sections 7.4.6 and 7.4.7.
7: $pk \leftarrow p = (p_1, \ldots, p_m) \in \mathbb{F}_2[x_1, \ldots, x_{n_{var}}]^m$ ▷ Take the first $m = d_{ext} -$ polynomials computed in Step 6.
8: **return** (sk, pk) ▷ The format is further detailed in Sections 7.4.1 and 7.4.8.
9: **end function**

7.1.3 Signing Process

The main step of the signature process requires solving:

$$p_1(x_1, \ldots, x_{n_{var}}) - d_1 = 0, \ldots, p_m(x_1, \ldots, x_{n_{var}}) - d_m = 0$$

for $d = (d_1, \ldots, d_m) \in \mathbb{F}_2^m$. To do so, we randomly sample $r = (r_1, \ldots, r_{d_{ext}-m}) \in \mathbb{F}_2^{d_{ext}-m}$ and append it to d. This gives $d' = (d, r) \in \mathbb{F}_2^{d_{ext}}$. We then compute $Y = \ ^{-1}\ d' \cdot T^{-1} \in \mathbb{F}_{2^{d_{ext}}}$ and try to find a root $(Z, v) \in \mathbb{F}_{2^{d_{ext}}} \times \mathbb{F}_2^v$ of the multivariate equation:

$$F(X, x_{d_{ext}+1}, \ldots, x_{n_{var}}) - Y = 0.$$

To solve this equation, we take advantage of the special HFEv-shape. That is, we randomly sample $v \in \mathbb{F}_2^v$ and consider the univariate polynomial $F(X, v) \in \mathbb{F}_{2^{d_{ext}}}[X]$. This yields a HFE polynomial according to Remark 4. We then find the roots of the univariate equation:

$$F(X, v) - Y = 0.$$

If there is a root $Z \in \mathbb{F}_{2^{d_{ext}}}$, we return $\ (Z), v \ \cdot S^{-1} \in \mathbb{F}_2^{n_{var}}$.

The core part of the signature generation is to compute the roots of $F_Y(X) = F(X, v) - Y$. To do so, we use Berlekamp's algorithm as described in Algorithm 25, Section 5.4.8. For $q = 2^{d_{ext}}$, we get that finding all the roots of a polynomial of degree D can be done in (expected) quasi-linear time (Theorem 4, Section 5.4.8), i.e.:

$$O\ D \cdot d_{ext}\ .$$

We can now present the inversion function (Algorithm 30).

Algorithm 30 Inversion in GeMSS.

1: **function** GeMSS.Inv$_p$ $d \in \mathbb{F}_2^m, sk = \ F, S^{-1}, T^{-1} \ \in \mathbb{F}_{2^{d_{ext}}}[X, v_1, \ldots, v_v] \times GL_{n_{var}}\ \mathbb{F}_2 \ \times GL_{d_{ext}}\ \mathbb{F}_2$
2: **repeat**
3: $r \in_R \mathbb{F}_2$ ▷ The notation $\in_R$ stands for randomly sampling.
4: $d' \leftarrow (d, r) \in \mathbb{F}_2^{d_{ext}}$
5: $Y \leftarrow \ ^{-1}\ d' \cdot T^{-1} \ \in \mathbb{F}_{2^{d_{ext}}}$
6: $v \in_R \mathbb{F}_2^v$
7: $F_Y(X) \leftarrow F(X, v) - Y$
8: $(n, L_{roots}) \leftarrow$ FindRoots(F_Y) ▷ Sections 5.4.8, 7.4.12 and 9.3.
9: **until** $n \neq 0$ ▷ F_Y has n roots.
10: $Z \in_R L_{roots}$
11: **return** $\ (Z), v \ \cdot S^{-1} \in \mathbb{F}_2^{n_{var}}$
12: **end function**

Remark 5. We always sample a root at Step 10 in the same way. Firstly, we sort the elements of L_{roots} as unsigned integers (i.e. $_i < _{i+1}$ in the chosen representation of $\mathbb{F}_{2^{d_{ext}}}$ (Section 5.2)) in ascending order. We then compute SHA-3(Y), and take H_{64} the first 64 bits of this digest. We consider H_{64} as an unsigned integer, and finally return the $(H_{64} \bmod n)$-th element in L_{roots}.

Let $d \in \mathbb{F}_2^m$ and $x_s \leftarrow$ GeMSS.Inv$_p$ $(d, sk = (F, S^{-1}, T^{-1})) \in \mathbb{F}_2^{n_{var}}$. By construction, we have $p(x_s) = d$, where p is the public-key associated to sk. Thus, $x_s \in \mathbb{F}_2^{n_{var}}$ could be directly used as a signature for the corresponding digest $d \in \mathbb{F}_2^m$ (Algorithm 3). In the case of GeMSS, m is small enough to make the cost of simple birthday-paradox attack (Section 4.2) against the hash function more efficient than all possible attacks (as those listed in Section 7.7). This problem was already identified in QUARTZ and Gui [134, 58, 59, 144]. Their authors proposed to handle this issue by using the so-called Feistel–Patarin scheme [58] (Section 4.2).

The basic principle of the Feistel–Patarin scheme is to roughly iterate Algorithm 30 several times. The number of iterations is a parameter nb_ite that will be discussed in Section 7.6.1.

Algorithm 31 Signing process in GeMSS.

1: **function** GeMSS.Sign$\left(M \in \{0,1\}^*, sk \in \mathbb{F}_{2^{d_{ext}}}[X, v_1, \ldots, v_v] \times GL_{n_{var}}(\mathbb{F}_2) \times GL_{d_{ext}}(\mathbb{F}_2)\right.$, GeMSS.Inv$_p$)
2: $\quad S_0 \leftarrow 0_m$
3: $\quad$ **for** i from 0 to nb_ite $- 1$ **do** $\quad\triangleright$ Iterations of the Feistel–Patarin scheme.
4: $\quad\quad h \leftarrow$ SHA-3$(M \| i)$ $\quad\triangleright$ Section 7.4.13.
5: $\quad\quad D_{i+1} \leftarrow$ first m bits of h $\quad\triangleright$ $D_{i+1} \in \mathbb{F}_2^m$.
6: $\quad\quad (S_{i+1}, X_{i+1}) \leftarrow$ GeMSS.Inv$_p(S_i + D_{i+1}, sk)$ $\quad\triangleright$ $S_{i+1} \in \mathbb{F}_2^m$ and $X_{i+1} \in \mathbb{F}_2^{n_{var}-m}$.
7: $\quad$ **end for**
8: $\quad$ **return** $sm = (S_{nb_ite}, X_{nb_ite}, X_{nb_ite-1}, \ldots, X_1)$
9: **end function**

7.1.4 Verification Process

Naturally, the verifying process is also iterative as shown in Algorithm 32.

Algorithm 32 Verifying process in GeMSS.

1: **function** GeMSS.Verify$\left(M \in \{0,1\}^*, sm \in \mathbb{F}_2^{m+nb_ite(n_{var}-m)}, pk = p \in \mathbb{F}_2[x_1, \ldots, x_{n_{var}}]^m\right)$
2: $\quad (S_{nb_ite}, X_{nb_ite}, X_{nb_ite-1}, \ldots, X_1) \leftarrow sm$
3: $\quad$ **for** i from nb_ite $- 1$ to 0 by -1 **do**
4: $\quad\quad h \leftarrow$ SHA-3$(M \| i)$ $\quad\triangleright$ Section 7.4.13.
5: $\quad\quad D_{i+1} \leftarrow$ first m bits of h $\quad\triangleright$ $D_{i+1} \in \mathbb{F}_2^m$.
6: $\quad\quad S_i \leftarrow p(S_{i+1}, X_{i+1}) + D_{i+1}$ $\quad\triangleright$ Sections 9.4.2, 9.4.3 and 9.4.4
7: $\quad$ **end for**
8: $\quad$ **return** VALID if $S_0 = 0_m$ and INVALID otherwise
9: **end function**

7.2 List of Parameter Sets

Following the analysis of Section 7.7, we propose several parameters for 128, 192 and 256 bits of classical security. Namely, we propose six sets of parameters: GeMSS, BlueGeMSS, RedGeMSS, WhiteGeMSS, CyanGeMSS and MagentaGeMSS. GeMSS corresponds to the same parameters that those proposed for the first round. This choice is conservative in term of security. As advised in [2] for

the second round, we explored more aggressive choice of parameters. This leaded to more e cient schemes: BlueGeMSS and RedGeMSS (especially, regarding the signing timings). The parameters were extracted from Section 7.8.10 where we propose a rather exhaustive choice of possible parameters and trade-o s between public-key size, signature size and e ciency. For the third round, [1] suggested evaluating the cost of the generic attack against the Feistel–Patarin scheme more accurately (Section 7.6.1), in order to improve performance of GeMSS. Following this analysis, we introduce WhiteGeMSS, CyanGeMSS and MagentaGeMSS. These schemes are respectively variants of GeMSS, BlueGeMSS, RedGeMSS, where nb_ite is set to three (instead of four). Compared to parameters from Section 7.8.10, we choose a smaller nb_ite thanks to a more accurate lower bound of the generic attack against the Feistel–Patarin scheme. The other security parameters are adjusted accordingly.

7.2.1 Parameter Sets for a Security of 2^{128} (Level I)

scheme	$(D, d_{ext}, , v, nb_ite)$	key gen. (Mc)	sign (Mc)	verify (kc)	|pk| (kB)	|sk| (bits)	|sign| (bits)
GeMSS128	(513, 174, 12, 12, 4)	19.6	608	106	352.19	128	258
BlueGeMSS128	(129, 175, 13, 14, 4)	18.4	67.2	134	363.61		270
RedGeMSS128	(17, 177, 15, 15, 4)	16.3	2.05	141	375.21		282
WhiteGeMSS128	(513, 175, 12, 12, 3)	20	436	91.7	358.17	128	235
CyanGeMSS128	(129, 177, 14, 13, 3)	18.5	49.8	91	369.72		244
MagentaGeMSS128	(17, 178, 15, 15, 3)	16.7	1.82	101	381.46		253

Table 7.1: Performance of GeMSS for a 128-bit security level (= 128), with MQsoft. We use a Skylake processor (LaptopS). Mc (respectively kc) stands for megacycles (respectively kilocycles). The results have three significant digits.

7.2.2 Parameter Sets for a Security of 2^{192} (Level III)

scheme	$(D, d_{ext}, , v, nb_ite)$	key gen. (Mc)	sign (Mc)	verify (kc)	|pk| (kB)	|sk| (bits)	|sign| (bits)
GeMSS192	(513, 265, 22, 20, 4)	69.4	1760	304	1237.96	192	411
BlueGeMSS192	(129, 265, 22, 23, 4)	65	173	325	1264.12		423
RedGeMSS192	(17, 266, 23, 25, 4)	57.1	5.55	335	1290.54		435
WhiteGeMSS192	(513, 268, 21, 21, 3)	73.1	1330	263	1293.85	192	373
CyanGeMSS192	(129, 270, 23, 22, 3)	68.2	131	269	1320.80		382
MagentaGeMSS192	(17, 271, 24, 24, 3)	60.3	4.53	274	1348.03		391

Table 7.2: Performance of GeMSS for a 192-bit security level (= 192), with MQsoft. We use a Skylake processor (LaptopS). Mc (respectively kc) stands for megacycles (respectively kilocycles). The results have three significant digits.

7.2.3 Parameter Sets for a Security of 2^{256} (Level V)

scheme	$(D, d_{ext}, \ , v, nb_ite)$	key gen. (Mc)	sign (Mc)	verify (kc)	$\lvert pk \rvert$ (kB)	$\lvert sk \rvert$ (bits)	$\lvert sign \rvert$ (bits)
GeMSS256	(513, 354, 30, 33, 4)	158	2490	665	3040.70	256	576
BlueGeMSS256	(129, 358, 34, 32, 4)	152	248	680	3087.96		588
RedGeMSS256	(17, 358, 34, 35, 4)	143	8.76	709	3135.59		600
WhiteGeMSS256	(513, 364, 31, 29, 3)	163	1920	516	3222.69	256	513
CyanGeMSS256	(129, 364, 31, 32, 3)	159	190	535	3272.02		522
MagentaGeMSS256	(17, 366, 33, 33, 3)	148	7.61	535	3321.72		531

Table 7.3: Performance of GeMSS for a 256-bit security level (= 256), with MQsoft. We use a Skylake processor (LaptopS). Mc (respectively kc) stands for megacycles (respectively kilocycles). The results have three significant digits.

7.3 Design Rationale

A multivariate scheme. The first design rational of GeMSS is to construct a signature scheme producing short signatures. It is well-known that multivariate cryptography [164, 29, 70] (Chapter 2) provides the schemes with the smallest signatures among all post-quantum schemes. Multivariate-based signature schemes are even competitive with ECC-based, pre-quantum, signature schemes (see, for example [31, 123]). This explains the choice of a multivariate cryptosystem for GeMSS.

A HFE-based scheme. HFE [133] (Section 2.4.1) is probably the most popular multivariate cryptosystem. Its security has been extensively studied for more than twenty years. The complexity of the best known attacks against HFE are all exponential in $O(\log_2(D))$ (Sections 4.4 and 4.6), where D is the degree of the secret univariate polynomial. When D is too small, then HFE can be broken, e.g. [111, 82, 22]. In contrast, solving HFE is NP-hard when $D = O(2^{d_{ext}})$ [111]. However, the complexity of the signature generation – which requires finding the roots of a univariate polynomial – is quasi-linear in D (Theorem 4). All in all, there is essentially one parameter, the degree D of the univariate secret polynomial, which governs the security and e ciency of HFE. The design challenge in HFE is to find a proper trade-o between e ciency and security.

Variants of HFE. A fundamental element in the design of secure signature schemes based on HFE is the introduction of perturbations. These create many variants of the scheme (Section 2.4.1). Classical perturbations include the minus modifier (HFE-, [151, 133]) and the vinegar modifier (HFEv, [110, 134]). Typically, QUARTZ is a HFEv- signature scheme where $D = 129$, $q = 2$, $d_{ext} = 103$, $v = 4$ vinegar variables and $= 3$ minus equations. The resistance, up to now, of QUARTZ against all known attacks illustrates that minus and vinegar variants permit to indeed strengthen the security of a HFE-based signature. A nude HFE, i.e. without any perturbation, with $D = 129$ and $d_{ext} = 103$ would be insecure whilst no practical attack against QUARTZ has been reported in the literature. The best known attack is [82] that serves as a reference to set the parameters for GeMSS. Besides, [68] gave new insights on how to choose the vinegar and minus modifiers.

QUARTZ has the reputation to be solid but with a rather slow signature generation process. The authors of [134] reported a signature generation process taking about a minute. Today, the same parameters will take less than one hundred milliseconds. This is partly due to the technological progress on the speed of processors. In fact, it is mostly due to a deeper understanding on algorithms finding the roots of univariate polynomials. This is further detailed in Section 9.3.

Large set of parameters. We propose a general methodology to derive parameters (Section 7.7). This permits to derive a large selection of parameters with various trade-o s between sizes and e ciency (Section 7.8.10).

7.4 Implementation

Here, we detail some of the choices we have made for implementing GeMSS.

7.4.1 Data Representation

Compressed secret-key. The size of the secret-key, as computed in Appendix A, can be drastically reduced. For that, we expand the secret-key from a random seed. This is classical and implies considering a new attack: the exhaustive search of the seed. Thus, we set the size of the seed to bits to reach a -bit security level. This change increases the cost of the signing process, since the secret-key has to be generated for each operation. However, the expansion of the seed is negligible in comparison to the cost of the root finding algorithm.

In MQsoft (Chapter 9) and in the NIST submission GeMSS [50], the use of a seed is controlled with the `ENABLED_SEED_SK` macro (set to 1 by default) from `config_HFE.h`. When enabled, the seed is expanded with SHAKE (Section 6.2.1).

Data structure for MQ systems. The first idea [56] is to see m equations of $\mathbb{F}_2[x_1,\ldots,x_{n_{var}}]$ as one element in $\mathbb{F}_{2^m}[x_1,\ldots,x_{n_{var}}]$. The second idea is to use quadratic forms. Let $\mathbf{x} = (x_1,\ldots,x_{n_{var}})$, $c \in \mathbb{F}_{2^m}$ and $\mathbf{Q}, \mathbf{Q}' \in \mathcal{M}_{n_{var}}(\mathbb{F}_{2^m})$, then a square-free quadratic polynomial in $\mathbb{F}_{2^m}[x_1,\ldots,x_{n_{var}}]$ can be written as

$$c + \mathbf{x} \cdot \mathbf{Q}' \cdot \mathbf{x}^T,$$

where $\mathbf{x}^T$ stands for transpose of $\mathbf{x}$. The coe cient $\mathbf{Q}'_{i,j}$ corresponds to the term $x_i x_j$ in the polynomial. Since $x_i^2 = x_i$, the linear term can be stored on the diagonal of $\mathbf{Q}'$. To minimize the size, $\mathbf{Q}'$ can be transformed into an upper triangular matrix $\mathbf{Q}$. By construction, $\mathbf{Q}'_{i,j}$ and $\mathbf{Q}'_{j,i}$ are the coe cients of the same term $x_i x_j$ $(i \neq j)$. The matrix $\mathbf{Q}$ is such that:

$$\mathbf{Q}_{i,j} = \begin{cases} \mathbf{Q}'_{i,j} & \text{if } i = j, \\ \mathbf{Q}'_{i,j} + \mathbf{Q}'_{j,i} & \text{if } i < j, \\ 0 & \text{otherwise.} \end{cases}$$

7.4.2 Representation of the Secret-Key

The existence of equivalent keys (Section 4.6.1) allows to choose a representation of the secret-key. Here, we present our choice for GeMSS.

Additive sustainer. In [165], the authors introduced the additive sustainer. Applied to HFEv-, we have that for any $C_s, C_t \in \mathbb{F}_{2^{d_{ext}}}$ and a HFEv polynomial $F \in \mathbb{F}_{2^{d_{ext}}}[X, v_1, \ldots, v_v]$, there exist two equivalent keys corresponding to $F(X)$ and $F(X + C_s) + C_t$. So, for any random F, we can choose C_s and C_t to have a unique representation of the secret-key. We can easily show [165] that we can choose C_s and C_t to have linear transformations $\mathcal{S}$ and $\mathcal{T}$ instead of affine transformations.

Big sustainer. Similarly to the additive sustainer, the big sustainer [165] gives the existence of equivalent keys between $F(X)$ and $B_t \cdot F(B_s \cdot X)$ for any $B_s, B_t \in \mathbb{F}^{\times}_{2^{d_{ext}}}$. So, for any random $F \in \mathbb{F}_{2^{d_{ext}}}[X, v_1, \ldots, v_v]$, we can set B_s to one and B_t to the inverse of the leading coefficient of F to obtain a monic HFEv polynomial. We generate monic HFEv polynomials in GeMSS, and so we do not store its leading term.

Remark 6. Assume $D = 2^I + 2^J$ for $0 \leqslant J < I$. With a smart choice of B_s and B_t, we could set the term X^D and one of terms $X^{D-2^{I-1}}$ ($I > 0$ and $J \neq I - 1$) or $X^{D-2^{J-1}}$ ($J > 0$) of F to one.

7.4.3 Generating Invertible Matrices

Algorithm 29 requires, at Step 2, generating a pair of invertible matrices $(S, T) \in GL_{n_{var}}(\mathbb{F}_2) \times GL_{d_{ext}}(\mathbb{F}_2)$. This problem was already discussed in [119]. The authors presented two (natural) methods to generate invertible matrices.

Trial and error. The first one (*trial and error*) samples random matrices until one is invertible. The probability that a random matrix in $\mathcal{M}_n(\mathbb{F}_q)$ is invertible [116] is given by

$$\prod_{j=1}^{n} \left(1 - q^{-j}\right).$$

This probability is especially small when q is small, implying a large number of *errors* before finding an invertible matrix. For $n = 128$ and $q = 2$, the probability of success is 28.9%.

LU decomposition. The second one uses the so-called LU decomposition [99]. This method has the advantage to directly return an invertible matrix. It is as follows.

- Generate a random lower triangular matrix $L \in GL_n(\mathbb{F}_2)$ and a random upper triangular matrix $U \in GL_n(\mathbb{F}_2)$, both with ones on the diagonal (to have a non-zero determinant).
- Return $L \cdot U \in GL_n(\mathbb{F}_2)$.

It is known [119] that this method is slightly biased. A small part of the invertible matrices cannot be generated with this method. For a square matrix of size n, the number of invertible upper triangular matrices is $2^{(n^2-n)/2}$ (the exponent is $\sum_{i=1}^{n-1} i$). So, the number of matrices that can be generated with the LU decomposition is 2^{n^2-n}. Still, this does not reduce the search space on the secret matrices sufficiently to impact the security of GeMSS.

In MQsoft (Chapter 9) and in the NIST submission of GeMSS [50], we have implemented both generation methods. The implementation gives the possibility to switch the method with the GEN_INVERTIBLE_MATRIX_LU macro, which is in the file config_HFE.h. It is set to 1 by default.

The matrices $(S, T) \in GL_{n_{var}}(\mathbb{F}_2) \times GL_{d_{ext}}(\mathbb{F}_2)$ are only used during the generation of the public-key. After, we are only using the inverse of these matrices. So, S^{-1} and T^{-1} are computed during the generation and stored in the secret-key (except when the secret-key is compressed).

Computation of $L \cdot U$ and inversion. During the keypair generation, we generate random invertible matrices and their inverse (when the secret-key is compressed, the inverses are generated during the signing process). When the LU decomposition is used, several ways to compute $L \cdot U$ and $(L \cdot U)^{-1}$ are possible.

- $L \cdot U$ can be directly computed with a matrix product. Similarly, $(L \cdot U)^{-1}$ can be computed as $U^{-1} \cdot L^{-1}$.
- By using the fact that the Gaussian elimination applied on $L \cdot U$ gives U, and that the operations used are stored in L, we can reverse this Gaussian elimination to obtain $L \cdot U$ from U. Similarly, we know that we can obtain $(L \cdot U)^{-1}$ from a Gaussian elimination on $(L \cdot U \mid I)$. So, we can use L then U to transform the identity matrix to $(L \cdot U)^{-1}$.

In our implementation, we compute $L \cdot U$ with a matrix product, whereas $(L \cdot U)^{-1}$ is computed with the Gaussian elimination. When the secret-key is compressed, we assume that the matrix product is faster than the Gaussian elimination. So, we modify the previous process by using L^{-1} and U^{-1} instead of L and U (the seed directly generates L^{-1} and U^{-1} for the signing process).

7.4.4 Constant-Time Gaussian Elimination

The trial and error method (Section 7.4.3) requires the generation and inversion of two boolean invertible matrices. The classical way to do it is to use a Gaussian elimination. Unlike the LU decomposition, the main problem is that the pivot may be null. Searching a row with a non-zero pivot reveals information on the matrix. The operation allowing to select one row of M has to be performed on all rows, in order to avoid information leakages (Section 4.7). In [17], the authors proposed a constant-time Gaussian elimination. Let $M \in \mathcal{M}_n(\mathbb{F}_2)$. The idea is to multiply (in constant-time) each row of M by a boolean, then add them to the pivot row. This boolean is used as a switch. While the pivot is null, this switch is set to one and the next row will be added to the pivot row. But as soon as the pivot is not null, the switch is definitely set to zero, implying that the pivot row will no longer be modified. We propose a possible implementation of this strategy in Algorithm 33. According to Section 6.3.2, the multiplications in $\mathbb{F}_2$ have to be replaced by the use of masks.

Algorithm 33 Constant-time Gaussian elimination on the rows of a matrix in $\mathcal{M}_n(\mathbb{F}_2)$. $M[i]$ corresponds to the i-th row of M, and $M[i][j]$ corresponds to $M_{i,j}$.

1: **function** cstGauss($M \in \mathcal{M}_n(\mathbb{F}_2)$)
2:     **for** i from 1 to $n-1$ **do**
3:         **for** j from $i+1$ to n **do** ▷ We search a non-zero pivot for the pivot row.
4:             $M[i] \leftarrow M[i] + (1 - M[i][i]) \cdot M[j]$ ▷ $M[i][i]$ is the pivot.
5:         **end for**
6:         **for** j from $i+1$ to n **do** ▷ We eliminate the coefficients of the i-th column with the pivot row.
7:             $M[j] \leftarrow M[j] - M[j][i] \cdot M[i]$
8:         **end for** ▷ Note that both loops could be merged.
9:     **end for**
10:     **return** M
11: **end function**

7.4.5 Generating HFEv Polynomials

Algorithm 29 requires, at Step 3, the generation of a polynomial $F \in \mathbb{F}_{2^{d_{\text{ext}}}}[X, v_1, \ldots, v_v]$ with a HFEv-shape of degree D (Equation (7.1)). The polynomial F can be considered as a polynomial in X whose coefficients live in $\mathbb{F}_{2^{d_{\text{ext}}}}[v_1, \ldots, v_v]$. We randomly generate and store the $N_{D,v,2}$ terms of F (Equation (A.1)). In fact, we only store $N_{D,v,2} - 1$ terms. The polynomial F is chosen monic and so the leading coefficient is not stored (Section 7.4.2). This choice makes easier the root finding part (Algorithm 25).

7.4.6 Public-Key Generation via Quadratic Forms

According to Section 7.4.1, $f \in \mathbb{F}_2[x_1, \ldots, x_{n_{\text{var}}}]^{d_{\text{ext}}}$ is stored as $c + x \cdot Q \cdot x^T \in \mathbb{F}_{2^{d_{\text{ext}}}}[x_1, \ldots, x_{n_{\text{var}}}]$. We first compute $f_1(x_1, \ldots, x_{n_{\text{var}}}) \cdot S, \ldots, f_{d_{\text{ext}}}(x_1, \ldots, x_{n_{\text{var}}}) \cdot S$ (Step 6, Algorithm 29) with our representation. To do so, we just replace x by $x \cdot S$. The linear change of variables $\mathcal{S}$ can be represented as:

$$c + x \cdot Q' \cdot x^T \in \mathbb{F}_{2^{d_{\text{ext}}}}[x_1, \ldots, x_{n_{\text{var}}}]$$

for $Q' = S \cdot Q \cdot S^T$. We then symmetrize the matrix Q' as in Section 7.4.1 to get an upper triangular matrix. To obtain the public-key, we now need to perform linear combinations with the matrix T. With our representation, this is equivalent to multiplying of each coefficient by T (by removing its last columns) to obtain the public-key in the form:

$$c_{pk} + x \cdot Q_{pk} \cdot x^T \in \mathbb{F}_{2^m}[x_1, \ldots, x_{n_{\text{var}}}],$$

with $c_{pk} \in \mathbb{F}_{2^m}$ and $Q_{pk} \in \mathcal{M}_{n_{\text{var}}}(\mathbb{F}_{2^m})$.

In this form, the evaluation of the public-key is reduced to a matrix-vector product and a dot product. However, the practical use of this representation is not optimal in memory when m is not a multiple of eight. So, we have to pack the bits of the public-key (Section 7.4.8).

7.4.7 Public-Key Generation by Evaluation-Interpolation

As mentioned in the seminal paper of Matsumoto–Imai [119], the public-key polynomials $p = (p_1, \ldots, p_m) \in \mathbb{F}_2[x_1, \ldots, x_{n_{\text{var}}}]^m$ can be generated by evaluation-interpolation. p is evaluated in N distinct evaluation points in $\mathbb{F}_2^{n_{\text{var}}}$, then a multivariate interpolation allows to find the coefficients of p. Since p is not yet known, each evaluation of p in $a \in \mathbb{F}_2^{n_{\text{var}}}$ is computed from the secret-key as follows:

$$p(a) = (\ \circ \mathcal{T} \circ \mathcal{F} \circ \mathcal{S})(a) = \mathcal{F}(a \cdot S) \cdot T \in \mathbb{F}_2^m,$$

where T is T without its last columns (cf. Equation (2.2)). This method can be easily simplified with a smart choice of the evaluation points. In [119], the authors consider points having their Hamming weight less or equal to two. Let e_i be the i-th row of $I_{n_{\text{var}}}$ the identity matrix in $\mathcal{M}_{n_{\text{var}}}(\mathbb{F}_2)$, and let p_k be as follows:

$$p_k = \sum_{i=1}^{n_{\text{var}}} \sum_{j=1}^{i-1} c_{i,j} x_i x_j + \sum_{i=1}^{n_{\text{var}}} c_i x_i + c \in \mathbb{F}_2[x_1, \ldots, x_{n_{\text{var}}}],$$

with $c_{i,j}, c_i, c \in \mathbb{F}_2$, for all i, j, $1 \leqslant j < i \leqslant n_{\text{var}}$.

Then, we have:

1. $p_k(0_{n_{\mathtt{var}}}) = c$,
2. $p_k(e_i) = c + c_i$, for all i, $1 \leqslant i \leqslant n_{\mathtt{var}}$,
3. $p_k(e_i + e_j) = c + c_i + c_j + c_{i,j}$, for all i, j, $1 \leqslant j < i \leqslant n_{\mathtt{var}}$.

We easily deduce the coefficients of the public-key. This method is adapted for low-end devices (Section 7.8.7), since quadratic terms of the public-key can be computed one by one.

Remark 7. As mentioned in [163], the multiplication by S is trivial since the Hamming weight of the vectors is less or equal to two. We just sum i-th and j-th rows of S. Note that the evaluation points are public and so do not require a constant-time use.

Remark 8. In the implementation, we compute $\mathcal{F} \circ \mathcal{S}$ by evaluation-interpolation instead of $\mathbf{p}$. Then, we apply T on each coefficient to obtain the public-key.

Evaluation of a **HFEv** polynomial. The evaluation-interpolation strategy requires evaluating N times the map $\mathcal{F}$. To do it, we need an efficient multipoint evaluation of HFEv polynomial. We use the matrix representation of F (Equation (9.3)) that we will present in Section 9.4.1.

Remark 9. When a quadratic form (e.g. $\mathcal{F}$ and F) is evaluated in several points, some computations can be re-used to accelerate its evaluation in a linear combination of these points.

7.4.8 Packed Representation of the Public-Key

The implementation of GeMSS submitted to the second round of the NIST PQC standardization process used the monomial representation of the public-key (Section 7.4.1). However, the latter does not reach the minimum theoretical size. We have solved this problem in our implementation for the third round. We use a public-key format allowing to pack the bits of the public-key, while maintaining a quick use during the verifying process. On one hand, we save up to 18% of the public-key size. On the other hand, the verifying process is slightly slower (up to 31%).

This format is based on the so-called hybrid representation [84]. Let $m = 8 \times k + r$ be the Euclidean division of m by 8. We store the first $8k$ equations with the monomial representation (Section 9.4.4), then we store the last r equations one by one (Section 9.4.2). This process is illustrated by Figure 7.1. Firstly, we pack monomial by monomial the coefficients of the $8k$ first equations. This corresponds to take the vertical rectangles from left to right, then take coefficients from up to down. Secondly, we pack equation by equation the coefficients of the r last equations. This corresponds to take the horizontal rectangles from up to down, then take coefficients from left to right.

Our aim is to decrease the cost of unpacking the bits of the public-key during the verifying process. With our format, a big part of the public-key uses the monomial representation. At the beginning of the second round, this representation was used to store the m equations (instead of $8k$ equations). So, the evaluation of the first $8k$ equations is performed as efficiently as before. They do not need to be unpacked. This implies that only the last r equations generate an additional cost, which is slight ($r \leqslant 7$ is small compared to $8k$). These equations can be evaluated packed, but when $nb_ite > 1$, unpacking them permits to accelerate the evaluation (which is repeated nb_ite times).

$$
\begin{array}{l}
c^{(1)} + p^{(1)}_{1,1}x_1^2 + p^{(1)}_{1,2}x_1x_2 + p^{(1)}_{1,3}x_1x_3 + p^{(1)}_{2,2}x_2^2 + p^{(1)}_{2,3}x_2x_3 + p^{(1)}_{3,3}x_3^2 \\
c^{(2)} + p^{(2)}_{1,1}x_1^2 + p^{(2)}_{1,2}x_1x_2 + p^{(2)}_{1,3}x_1x_3 + p^{(2)}_{2,2}x_2^2 + p^{(2)}_{2,3}x_2x_3 + p^{(2)}_{3,3}x_3^2 \\
c^{(3)} + p^{(3)}_{1,1}x_1^2 + p^{(3)}_{1,2}x_1x_2 + p^{(3)}_{1,3}x_1x_3 + p^{(3)}_{2,2}x_2^2 + p^{(3)}_{2,3}x_2x_3 + p^{(3)}_{3,3}x_3^2 \\
c^{(4)} + p^{(4)}_{1,1}x_1^2 + p^{(4)}_{1,2}x_1x_2 + p^{(4)}_{1,3}x_1x_3 + p^{(4)}_{2,2}x_2^2 + p^{(4)}_{2,3}x_2x_3 + p^{(4)}_{3,3}x_3^2 \\
c^{(5)} + p^{(5)}_{1,1}x_1^2 + p^{(5)}_{1,2}x_1x_2 + p^{(5)}_{1,3}x_1x_3 + p^{(5)}_{2,2}x_2^2 + p^{(5)}_{2,3}x_2x_3 + p^{(5)}_{3,3}x_3^2 \\
c^{(6)} + p^{(6)}_{1,1}x_1^2 + p^{(6)}_{1,2}x_1x_2 + p^{(6)}_{1,3}x_1x_3 + p^{(6)}_{2,2}x_2^2 + p^{(6)}_{2,3}x_2x_3 + p^{(6)}_{3,3}x_3^2 \\
c^{(7)} + p^{(7)}_{1,1}x_1^2 + p^{(7)}_{1,2}x_1x_2 + p^{(7)}_{1,3}x_1x_3 + p^{(7)}_{2,2}x_2^2 + p^{(7)}_{2,3}x_2x_3 + p^{(7)}_{3,3}x_3^2 \\
c^{(8)} + p^{(8)}_{1,1}x_1^2 + p^{(8)}_{1,2}x_1x_2 + p^{(8)}_{1,3}x_1x_3 + p^{(8)}_{2,2}x_2^2 + p^{(8)}_{2,3}x_2x_3 + p^{(8)}_{3,3}x_3^2 \\
c^{(9)} + p^{(9)}_{1,1}x_1^2 + p^{(9)}_{1,2}x_1x_2 + p^{(9)}_{2,2}x_2^2 + p^{(9)}_{1,3}x_1x_3 + p^{(9)}_{2,3}x_2x_3 + p^{(9)}_{3,3}x_3^2 \\
c^{(10)} + p^{(10)}_{1,1}x_1^2 + p^{(10)}_{1,2}x_1x_2 + p^{(10)}_{2,2}x_2^2 + p^{(10)}_{1,3}x_1x_3 + p^{(10)}_{2,3}x_2x_3 + p^{(10)}_{3,3}x_3^2
\end{array}
$$

Figure 7.1: Example of hybrid representation of a MQ system of 10 equations in 3 variables. Each row corresponds to one equation, and the $c^{(k)}$ and $p^{(k)}_{i,j}$ are in $\mathbb{F}_2$ for $1 \leqslant k \leqslant 10$.

Implementation details. An important point in our implementation is the memory alignment. All data have to be 8-bit aligned. This permits to have simpler and more e cient implementations. In the previous implementation, we used a zero padding when necessary. However, this implied that the theoretical size was not reached.

Firstly, the first $8k$ equations are stored without loss. Since for each monomial, $8k$ coe cients in $\mathbb{F}_2$ are packed, we obtain that k bytes are required to store them. So, we do not require padding to align data to 8 bits. The monomials are stored in the graded lexicographic order (as on Figure 7.1). Secondly, the last r equations are stored in the graded reverse lexicographic order (as on Figure 7.1). Each equation requires storing N elements of $\mathbb{F}_2$. The alignment of the equations requires using a zero padding when N is not multiple of eight. In this case, the padding size is $N_p = 8 - (N \bmod 8)$ bits. We solve this problem by using the last $(r-1)N_p$ bits of the last equation to fill the padding of the $r-1$ other equations. In particular, we take these last bits by pack of N_p, and the i-th pack is used to fill the padding of the $(8k+i)$-th equation. For example, on Figure 7.1, the 9-th equation contains 7 coe cients. So, with our process, we would remove $p^{(10)}_{3,3}$ from the 10-th equation to store it just after $p^{(9)}_{3,3}$. Thus, the 10-th equation would be 8-bit aligned.

Remark 10. If $rN_p \leqslant ((8-(rN \bmod 8)) \bmod 8)$, then the minimum size of the public-key (rounded up to the next byte) is also reached by using the zero padding of each equation.

7.4.9 Parallel Arithmetic in $\mathbb{F}_2$, $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$

Here, we study the classical strategies to perform operations in parallel in $\mathbb{F}_q$, for $q \in \{2, 16, 256\}$. We use $\mathbb{F}_2$ for `GeMSS`, whereas $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ are used for low-end devices (Section 7.8.7). The fields $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ are also used in `Rainbow` (Section 7.10).

Tower fields defining $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$. Here, we present how we define $\mathbb{F}_4$, $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$. These representations [56] are the same that these used in `Rainbow` [64], which allows a compatibility between the `Rainbow` implementations and our implementation of its dual mode (Section 8.6.3).

- $\mathbb{F}_4$ is (necessarily) $\mathbb{F}_2[e_1]/\left(e_1^2 + e_1 + 1\right)$,
- $\mathbb{F}_{16}$ is $\mathbb{F}_4[e_2]/\left(e_2^2 + e_2 + e_1\right)$,
- $\mathbb{F}_{256}$ is $\mathbb{F}_{16}[e_3]/\left(e_3^2 + e_3 + e_2 e_1\right)$.

We can perform efficiently arithmetic operations in parallel via vector instructions (Section 6.1.3). By defining $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ with tower fields, we can take advantage of subfields to accelerate field operations, in particular the (general) multiplication in $\mathbb{F}_{256}$. Other operations that we present here do not require defining fields with tower fields. In fact, we can use any representation of $\mathbb{F}_q$ ($q \in \{2, 16, 256\}$ here), if the representation of all its elements corresponds to the same basis $\mathcal{B}$ of $\mathbb{F}_q$ over $\mathbb{F}_2$ (Section 5.2). This is true for tower fields respecting Equation (5.6) ($\mathcal{B} = (1, e_1, e_2, e_1 e_2, e_3, e_1 e_3, e_2 e_3, e_1 e_2 e_3)$ if $\mathbb{F}_4$, $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ are represented with the canonical basis). Except for the (general) multiplication in $\mathbb{F}_{256}$, these parallel operations can also be used in $\mathbb{F}_{2^k}$ for $1 \leqslant k \leqslant \log_2(q)$ (with a relevant data alignment).

Packed representation. The elements of $\mathbb{F}_2$, $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ require respectively one bit, four bits and one byte. Naturally, we store elements as a vector of bytes, with in particular two elements of $\mathbb{F}_{16}$ by byte. By using 128-bit registers available in SSE, we can store $\frac{128}{\log_2(q)}$ elements of $\mathbb{F}_q$, which makes 128, 32 and 16 elements respectively for $\mathbb{F}_2$, $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$.

Addition. The addition in $\mathbb{F}_q$ corresponds to the bitwise logical `XOR`, since each element is a binary polynomial. With the `PXOR` instruction, $\frac{128}{\log_2(q)}$ additions can be performed in parallel.

Squaring. The square in $\mathbb{F}_q$ can be computed with one or two calls to the `PSHUFB` instruction.

Parallel multiplication by a scalar. In $\mathbb{F}_2$, the multiplication of 128 elements $a_0, \ldots, a_{127}$ by a scalar $c \in \mathbb{F}_2$ can be easily implemented. To do it, we duplicate c in a 128-bit register, then the multiplication is performed via the `PAND` instruction. The duplication of c is performed by duplicating it firstly on each 64-bit block, then by computing $-c$ in parallel on each block. This can be computed as $0 - c$ via the `PSUB` instruction from SSE2. However, we remark that the equality test $c == 1$ via the (constant-time) `PCMPEQ` instruction from SSE4.1 seems faster.

In $\mathbb{F}_{16}$, the idea [51] is to use multiplication tables coupled to the `PSHUFB` instruction. This instruction takes 16 indices on 4 bits, and looks up the corresponding 8-bit elements in a 16-byte lookup table. Since each result only uses 4 bits, we can multiply by one or two scalars in one call to `PSHUFB` [51]. Let $a_0, \ldots, a_{15} \in \mathbb{F}_{16}$, and let T_c (respectively $T_{c,d}$) be the lookup table whose i-th index corresponds to the i-th element of $\mathbb{F}_{16}$ multiplied by c (respectively c and d) in $\mathbb{F}_{16}$.

The PSHUFB instruction permits to perform the 16 multiplications by c (respectively c and d) in one instruction (once the coefficients and T_c (respectively $T_{c,d}$) are loaded in 128-bit registers). Here, note that the loaded table depends on $c, d \in \mathbb{F}_{16}$. So, this implementation is variable-time about c and d (Section 6.3.3). Naturally, the same method applied to $\mathbb{F}_4$ permits to perform 32 multiplications by one or two scalars, in one instruction (and 64 multiplications for $\mathbb{F}_2$).

In $\mathbb{F}_{256}$, the idea [51] is similar to $\mathbb{F}_{16}$. We just split the memory representation of each element of $\mathbb{F}_{256}$ into two packs of four bits, then we use one lookup table by pack. Let $a_0, \dots, a_{15} \in \mathbb{F}_{256}$, and let $\mathcal{P}$ be the map which generates an element of $\mathbb{F}_{256}$ from its memory representation: an integer in $[\![0, 255]\!]$. For $0 \leqslant i \leqslant 15$, let $b_i = \mathcal{P}^{-1}(a_i) \in [\![0, 255]\!]$ be the memory representation of a_i. We have $a_i = \mathcal{P}(b_i \bmod 16) + \mathcal{P}(16 \cdot \lfloor \frac{b_i}{16} \rfloor)$. Let T_c (respectively U_c) be the lookup table whose i-th index corresponds to $\mathcal{P}(i)$ (respectively $\mathcal{P}(16 \cdot i)$) multiplied by c in $\mathbb{F}_{256}$. The 16 multiplications by $c \in \mathbb{F}_{256}$ can be performed by splitting b_i into $\lfloor \frac{b_i}{16} \rfloor$, $b_i \bmod 16$ for $0 \leqslant i \leqslant 15$, by multiplying each $\mathcal{P}(b_i \bmod 16)$ (respectively $\mathcal{P}(16 \cdot \lfloor \frac{b_i}{16} \rfloor)$) by c with T_c (respectively U_c) and one call to the PSHUFB instruction, then by summing both results with the PXOR instruction.

General multiplication. In $\mathbb{F}_2$, the multiplication corresponds to the bitwise logical AND. The PAND instruction permits to compute 128 multiplications in parallel.

In $\mathbb{F}_{16}$, the strategy [56] is to remark that for $a, b \in \mathbb{F}_{16}^{\times}$, $a \cdot b = g^{\log_g(a) + \log_g(b)}$, where g is a generator of the multiplicative group $\mathbb{F}_{16}^{\times}$. The Zech logarithm representation [35] consists in writing any non-zero element of a finite field $\mathbb{F}_q$ as the power of a generator of the multiplicative group, and allows to perform the multiplication as the addition of powers modulo $q-1$. Therefore, the multiplication can be parallelized with logarithm and exponential tables, coupled to the PSHUFB instruction. Here, we just set $\log_g(0)$ to $256 - 42$, and we put $s + \lfloor \frac{s}{16} \rfloor$ as input of the PSHUFB instruction, for $s = (\log_g(a) + \log_g(b)) \bmod 256$. If a and b are not zero, then PSHUFB only considers the first four bits of this 8-bit block in input, and returns the corresponding block from the exponential table. Otherwise, the last bit of the 8-bit block is one and the peculiar property of PSHUFB implies that a null block is returned. This strategy is more expensive than the previous methods, but is constant-time.

In $\mathbb{F}_{256}$, the strategy [56] is to perform Karatsuba's multiplication algorithm (Section 5.1.2) on degree-one elements of $\mathbb{F}_{16}[e_3]$ via the tower field representation. The three products in $\mathbb{F}_{16}$ are computed via logarithm and exponential tables, and the resulting degree-two term is multiplied by $e_3 + e_2e_1$. By using notations from Section 5.1.2, the result is $P_1 + P_2 \cdot e_2e_1 + (P_1 + P_3)e_3$. The multiplication of $P_2 \in \mathbb{F}_{16}$ by the public scalar $e_2e_1 \in \mathbb{F}_{16}$ can be performed with one call to the PSHUFB instruction, but it is more efficient to directly compute $P_2 \cdot e_2e_1$ instead of computing P_2 then $P_2 \cdot e_2e_1$. To do it, let $P_2 = a \cdot b$ for $a, b \in \mathbb{F}_{16}$, and use a logarithm table returning $\log_g(a \cdot e_2e_1)$ from a during the computation of P_2 in order to directly compute $P_2 \cdot e_2e_1$.

Remark 11. By using the AVX2 instruction set, all previous 128-bit instructions can be performed two times in parallel, improving the performance of arithmetic operations in $\mathbb{F}_q$.

7.4.10 Choice of the Field Polynomial for $\mathbb{F}_{2^{d_{\text{ext}}}}$

Keypair generation and signing process require arithmetic operations in $\mathbb{F}_{2^{d_{\text{ext}}}}$ (Section 9.2). In the implementation, the field $\mathbb{F}_{2^{d_{\text{ext}}}}$ is defined as $\mathbb{F}_2[x]/(f(x))$ with f being an irreducible trinomial of degree d_{ext} in $\mathbb{F}_2[x]$ (Section 9.1). In Table 7.4, we present our implementation choices for GeMSS. This choice does not impact the security.

d_{ext}	174	175	177	178	265	266	268	270	271	354	358	364	366	537
k_1	13	16	8	31	42	47	25	53	58	99	57	9	29	94

Table 7.4: Irreducible trinomials $x^{d_{\text{ext}}} + x^{k_1} + 1$ defining $\mathbb{F}_{2^{d_{\text{ext}}}}$ for GeMSS.

7.4.11 Constant-Time GCD of Polynomials

During the GCD algorithm (Section 5.4.1), the potential jumps in the degree of the successive remainders generate an information leakage (Section 4.7). In [19], the authors introduced a constant-time GCD. The latter is based on the Euclid–Stevin relationship [155] (Section 5.4.1). Given F and H in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$ of degrees $d_f \geqslant d_h$ respectively, we compute $F = h_{d_h} F - f_{d_f} H X^{d_f - d_h}$ until $d_f < d_h$. Let F_1 be the last computed value of F. Thus, we have $\text{GCD}(F, H) = \text{GCD}(H, F_1)$. The idea of [19] is to use this relationship in constant-time. This implies computing $d_f + d_h$ Euclid–Stevin relationship, by swapping F and H in constant-time when $d_f < d_h$. In practice, the constant-time version of the Euclid–Stevin algorithm is a bit more tricky and we refer to [19] for more details.

7.4.12 Constant-Time Root Finding

We propose to modify the GeMSS implementation to obtain a constant-time signing process. The root finding algorithm (Algorithm 25) is currently implemented with a constant-time Frobenius map and a constant-time GCD (Section 7.4.11), but the root finding of split polynomials (Section 5.4.7) has a time that depends on the degree (which is the number of roots). In QUARTZ [134] and Gui (round 1 candidate), the strategy is to select one root only if the latter is unique. However, this method generates a theoretical slow-down factor of 1.72 [88] (Table 2.1), compared to the selection of roots when they exist.

So, we propose to extend this strategy by introducing constant-time solvers for degree-two and degree-three split polynomials in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$. By selecting roots only if there are one or two roots, the theoretical slow-down factor drops to 1.15. Then, an additional slow-down is due to the fact that solving degree-one polynomials has to be as slow as solving degree-two polynomials. This process can also be performed by selecting roots only if there are one, two or three roots. In this case, the theoretical slow-down factor drops to 1.03. In return, the cost of solving degree-three polynomials in constant-time is larger than degree-two polynomials, which can give worse performance than the previous method. Therefore, the best strategy between allowed one or two roots, or one, two or three roots, should be chosen in function of the available implementation and security parameters. Note that the rate of signature failure is still very small (Section 7.6.3).

Solvers of degree-two and degree-three split polynomials can be implemented with linear algebra, respectively by using that $X^2 + X + A$ and $X^4 + sX^2 + pX = X \cdot (X^3 + sX + p)$ are $\mathbb{F}_2$-linear (Section 5.4.7). For degree-two split polynomials, we can also use the half-trace (Equation (5.15)) when d_{ext} is odd.

7.4.13 About the Use of Hash Functions

In GeMSS, we use the SHA3-256 function for security levels I and II (Section 4.1), the SHA3-384 function for security levels III and IV, and the SHA3-512 function for security levels V and VI. When the secret-key is generated from a seed (Section 7.4.1), we use SHAKE128 (Section 6.2.1) for security levels I and II, and SHAKE256 for security levels III, IV, V and VI.

Remark 12. For $\text{nb_ite} > 1$ and a large message M, the repeated computation of SHA-3$(M\|i)$ penalizes Algorithms 31 and 32. In this case, the implementation should store relevant results during the computation of SHA-3(M), then use them to compute SHA-3$(M\|i)$ for $0 \leqslant i < \text{nb_ite}$. If this option is not suitable (for example, because a third library is used for SHA-3), then the classical strategy consists in precomputing $h_M \leftarrow$ SHA-3(M), then computing SHA-3$(h_M\|i)$ instead of SHA-3$(M\|i)$.

7.5 Detailed Performance Analysis

7.5.1 Experimental Platform

The measurements in Sections 7.5.3, 7.5.4, 7.5.5 and 7.5.6 are for GeMSS. For signature, it signs and verifies a document of 32 bytes. For the measurements, it runs a number of tests such that the elapsed time is greater than 2 seconds, and this time is divided by the number of tests. For the signature, the lower bound of the number of tests is 256. The times of the signing process are unstable, since it depends on the probability of finding a root of a univariate polynomial (Table 2.1). So, we have taken a large number of signatures.

The measurements used one core of the CPU, and the reference implementation was compiled with `gcc -O2 -msse2 -msse3 -mssse3 -msse4.1 -mpclmul`. The SIMD is enabled only to inline the (potential) vector multiplication functions from the gf2x library (Section 6.2.1). The reference implementation does not exploit these instruction sets. For the optimized and additional implementations, the code was compiled with `gcc -O3 -mavx2 -mpclmul -mpopcnt -funroll-loops`. Turbo Boost and Enhanced Intel Speedstep Technology are disabled to have more accurate measurements (Section 6.2.2).

7.5.2 Third-Party Open Source Library

For all implementations, we have used the SHA-3 and SHAKE functions from the Extended Keccak Code Package (Section 6.2.1). The HFE-based schemes require using arithmetic in $\mathbb{F}_{2^{d_{ext}}}[X]$ (Section 9.3). In particular, the multiplication in $\mathbb{F}_{2^{d_{ext}}}$ is the most critical operation. In the optimized and additional implementations, we have implemented this operation by using the PCLMULQDQ instruction (Section 6.1.3). This instruction computes the product of two binary polynomials such that their degree is strictly less than 64. In the reference implementation, we use the fast multiplications of binary polynomials implemented in the gf2x library. In all implementations, the use of the gf2x library can be enabled (or disabled) by setting to 1 (or 0) the ENABLED_GF2X macro from arch.h.

7.5.3 Reference Implementation

In Table 7.5, we summarize timings of the reference implementation.

scheme	(, D, d_{ext}, , v, nb_ite)	key gen. (Mc)	sign (Mc)	verify (kc)
GeMSS128	(128, 513, 174, 12, 12, 4)	140	2420	211
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	108	473	236
RedGeMSS128	(128, 17, 177, 15, 15, 4)	89.2	49.4	242
GeMSS192	(192, 513, 265, 22, 20, 4)	600	6310	591
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	506	1290	603
RedGeMSS192	(192, 17, 266, 23, 25, 4)	413	121	596
GeMSS256	(256, 513, 354, 30, 33, 4)	1660	10600	1140
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	1500	2060	1180
RedGeMSS256	(256, 17, 358, 34, 35, 4)	1290	200	1170
WhiteGeMSS128	(128, 513, 175, 12, 12, 3)	138	1810	163
CyanGeMSS128	(128, 129, 177, 14, 13, 3)	117	383	172
MagentaGeMSS128	(128, 17, 178, 15, 15, 3)	92.4	37	170
WhiteGeMSS192	(192, 513, 268, 21, 21, 3)	620	4940	464
CyanGeMSS192	(192, 129, 270, 23, 22, 3)	529	929	468
MagentaGeMSS192	(192, 17, 271, 24, 24, 3)	433	86	464
WhiteGeMSS256	(256, 513, 364, 31, 29, 3)	1740	8020	956
CyanGeMSS256	(256, 129, 364, 31, 32, 3)	1540	1700	990
MagentaGeMSS256	(256, 17, 366, 33, 33, 3)	1350	157	985

Table 7.5: Performance of the reference implementation. We use a Skylake processor (LaptopS). Mc (respectively kc) stands for megacycles (respectively kilocycles). The results have three significant digits.

7.5.4 Optimized (Haswell) Implementation

In Table 7.6, we summarize timings of the optimized implementation. The use of the evaluation-interpolation method (Section 7.4.7) for generating the public-key is rather e cient for small degrees such as 17. For large degrees such as 129 and 513, we use the round 2 keypair generation (Section 7.4.6). To do it, we set to zero the `INTERPOLATE_PK_HFE` macro in the file `sign_keypairHFE.c`.

scheme	(, D, d_{ext}, , v, nb_ite)	key gen. (Mc)	sign (Mc)	verify (kc)
`GeMSS128`	(128, 513, 174, 12, 12, 4)	51.6	1340	163
`BlueGeMSS128`	(128, 129, 175, 13, 14, 4)	52.1	195	168
`RedGeMSS128`	(128, 17, 177, 15, 15, 4)	40.5	4.93	179
`GeMSS192`	(192, 513, 265, 22, 20, 4)	270	3550	455
`BlueGeMSS192`	(192, 129, 265, 22, 23, 4)	268	474	468
`RedGeMSS192`	(192, 17, 266, 23, 25, 4)	228	12.8	477
`GeMSS256`	(256, 513, 354, 30, 33, 4)	816	5670	979
`BlueGeMSS256`	(256, 129, 358, 34, 32, 4)	810	736	990
`RedGeMSS256`	(256, 17, 358, 34, 35, 4)	793	20.4	1010
`WhiteGeMSS128`	(128, 513, 175, 12, 12, 3)	52.4	997	118
`CyanGeMSS128`	(128, 129, 177, 14, 13, 3)	53.5	157	125
`MagentaGeMSS128`	(128, 17, 178, 15, 15, 3)	41.3	4.03	129
`WhiteGeMSS192`	(192, 513, 268, 21, 21, 3)	281	2640	357
`CyanGeMSS192`	(192, 129, 270, 23, 22, 3)	281	370	364
`MagentaGeMSS192`	(192, 17, 271, 24, 24, 3)	231	9.93	368
`WhiteGeMSS256`	(256, 513, 364, 31, 29, 3)	844	4400	819
`CyanGeMSS256`	(256, 129, 364, 31, 32, 3)	846	553	833
`MagentaGeMSS256`	(256, 17, 366, 33, 33, 3)	770	16.4	845

Table 7.6: Performance of the optimized implementation. We use a Haswell processor (ServerH). Mc (respectively kc) stands for megacycles (respectively kilocycles). The results have three significant digits.

7.5.5 Additional (Skylake) Implementation

In Table 7.7, we summarize timings of the additional implementation. The additional and optimized implementations are based on the same implementation. We have only set the `PROC_SKYLAKE` macro to 1, whereas in the optimized implementation, we set the `PROC_HASWELL` macro to 1. These macros from `arch.h` impact mainly the multiplication in $\mathbb{F}_{2^{d_{ext}}}$.

scheme	(, D, d_{ext}, , v, nb_ite)	key gen. (Mc)	sign (Mc)	verify (kc)
GeMSS128	(128, 513, 174, 12, 12, 4)	51.9	1080	163
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	51.5	154	174
RedGeMSS128	(128, 17, 177, 15, 15, 4)	41.1	4.37	183
GeMSS192	(192, 513, 265, 22, 20, 4)	274	3170	495
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	262	445	509
RedGeMSS192	(192, 17, 266, 23, 25, 4)	221	12	514
GeMSS256	(256, 513, 354, 30, 33, 4)	915	5300	1120
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	856	658	1130
RedGeMSS256	(256, 17, 358, 34, 35, 4)	765	19.5	1140
WhiteGeMSS128	(128, 513, 175, 12, 12, 3)	52.9	815	112
CyanGeMSS128	(128, 129, 177, 14, 13, 3)	54.4	119	116
MagentaGeMSS128	(128, 17, 178, 15, 15, 3)	41.9	3.51	125
WhiteGeMSS192	(192, 513, 268, 21, 21, 3)	287	2380	388
CyanGeMSS192	(192, 129, 270, 23, 22, 3)	289	339	396
MagentaGeMSS192	(192, 17, 271, 24, 24, 3)	223	9.38	401
WhiteGeMSS256	(256, 513, 364, 31, 29, 3)	960	3910	914
CyanGeMSS256	(256, 129, 364, 31, 32, 3)	963	529	911
MagentaGeMSS256	(256, 17, 366, 33, 33, 3)	750	15.6	936

Table 7.7: Performance of the additional implementation. We use a Skylake processor (LaptopS). Mc (respectively kc) stands for megacycles (respectively kilocycles). The results have three significant digits.

7.5.6 MQsoft

MQsoft [84] (Chapter 9) is a new efficient library in C programming language for HFE-based schemes such as GeMSS, Gui and DualModeMS. In [84], we improved the complexity of several fundamental building blocks for such schemes and improved the protection against timing attacks. This gives the best implementation of the GeMSS family. Here, we give the timings (Tables 7.8 and 7.9) with the final version of MQsoft that uses the SSE2, SSSE3, SSE4.1 and AVX2 instruction sets to be faster.

scheme	(, D, d_{ext}, , v, nb_ite)	key gen. (Mc)	sign (Mc)	verify (kc)
GeMSS128	(128, 513, 174, 12, 12, 4)	19.6	608	106
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	18.4	67.2	134
RedGeMSS128	(128, 17, 177, 15, 15, 4)	16.3	2.05 (1.57)	141
GeMSS192	(192, 513, 265, 22, 20, 4)	69.4	1760	304
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	65	173	325
RedGeMSS192	(192, 17, 266, 23, 25, 4)	57.1	5.55 (3.88)	335
GeMSS256	(256, 513, 354, 30, 33, 4)	158	2490	665
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	152	248	680
RedGeMSS256	(256, 17, 358, 34, 35, 4)	143	8.76 (5.32)	709
WhiteGeMSS128	(128, 513, 175, 12, 12, 3)	20	436	91.7
CyanGeMSS128	(128, 129, 177, 14, 13, 3)	18.5	49.8	91
MagentaGeMSS128	(128, 17, 178, 15, 15, 3)	16.7	1.82 (1.27)	101
WhiteGeMSS192	(192, 513, 268, 21, 21, 3)	73.1	1330	263
CyanGeMSS192	(192, 129, 270, 23, 22, 3)	68.2	131	269
MagentaGeMSS192	(192, 17, 271, 24, 24, 3)	60.3	4.53 (2.84)	274
WhiteGeMSS256	(256, 513, 364, 31, 29, 3)	163	1920	516
CyanGeMSS256	(256, 129, 364, 31, 32, 3)	159	190	535
MagentaGeMSS256	(256, 17, 366, 33, 33, 3)	148	7.61 (4.07)	535

Table 7.8: Performance of MQsoft. We use a Skylake processor (LaptopS). Mc (respectively kc) stands for megacycles (respectively kilocycles). The results have three significant digits. The second value of the signing process corresponds to the time to sign without decompressing the secret-key.

scheme	(, D, d_{ext}, , v, nb_ite)	key gen. (Mc)	sign (Mc)	verify (kc)
GeMSS128	(128, 513, 174, 12, 12, 4)	18.8	809	95.1
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	18.6	88.2	116
RedGeMSS128	(128, 17, 177, 15, 15, 4)	15.7	2.31	121
GeMSS192	(192, 513, 265, 22, 20, 4)	69.6	2280	289
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	65.8	220	309
RedGeMSS192	(192, 17, 266, 23, 25, 4)	57.3	6.01	309
GeMSS256	(256, 513, 354, 30, 33, 4)	167	3030	617
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	159	300	620
RedGeMSS256	(256, 17, 358, 34, 35, 4)	145	8.86	636
WhiteGeMSS128	(128, 513, 175, 12, 12, 3)	18.9	589	77.2
CyanGeMSS128	(128, 129, 177, 14, 13, 3)	17.7	65.3	78.5
MagentaGeMSS128	(128, 17, 178, 15, 15, 3)	16.2	1.91	85.8
WhiteGeMSS192	(192, 513, 268, 21, 21, 3)	72.8	1740	245
CyanGeMSS192	(192, 129, 270, 23, 22, 3)	67.7	170	247
MagentaGeMSS192	(192, 17, 271, 24, 24, 3)	58.8	4.81	255
WhiteGeMSS256	(256, 513, 364, 31, 29, 3)	173	2360	487
CyanGeMSS256	(256, 129, 364, 31, 32, 3)	166	228	491
MagentaGeMSS256	(256, 17, 366, 33, 33, 3)	150	7.34	494

Table 7.9: Performance of MQsoft. We use a Haswell processor (ServerH). Mc (respectively kc) stands for megacycles (respectively kilocycles). The results have three significant digits.

7.5.7 Space

In Table 7.10, we provide the sizes of the public-key, secret-key and signature. Since the secret-key is generated from a seed (Section 7.4.1), the secret-key is very small: just several hundred bits. In contrast, the decompressed secret-key size is between 10 and 80 kB (Table A.1).

scheme	(, D, d_{ext}, , v, nb_ite)	|pk| (kB)	|sk| (B)	|sign| (B)
GeMSS128	(128,513,174,12,12,4)	352.188	16	32.25
BlueGeMSS128	(128,129,175,13,14,4)	363.609		33.75
RedGeMSS128	(128,17,177,15,15,4)	375.21225		35.25
WhiteGeMSS128	(128,513,175,12,12,3)	358.172125		29.375
CyanGeMSS128	(128,129,177,14,13,3)	369.72475		30.5
MagentaGeMSS128	(128,17,178,15,15,3)	381.46075		31.625
GeMSS192	(192,513,265,22,20,4)	1237.9635	24	51.375
BlueGeMSS192	(192,129,265,22,23,4)	1264.116375		52.875
RedGeMSS192	(192,17,266,23,25,4)	1290.542625		54.375
WhiteGeMSS192	(192,513,268,21,21,3)	1293.84775		46.625
CyanGeMSS192	(192,129,270,23,22,3)	1320.801625		47.75
MagentaGeMSS192	(192,17,271,24,24,3)	1348.033375		48.875
GeMSS256	(256,513,354,30,33,4)	3040.6995	32	72
BlueGeMSS256	(256,129,358,34,32,4)	3087.963		73.5
RedGeMSS256	(256,17,358,34,35,4)	3135.591		75
WhiteGeMSS256	(256,513,364,31,29,3)	3222.69075		64.125
CyanGeMSS256	(256,129,364,31,32,3)	3272.016375		65.25
MagentaGeMSS256	(256,17,366,33,33,3)	3321.716625		66.375

Table 7.10: Memory cost of GeMSS. 1 kB is 1000 bytes.

7.5.8 How Parameters Affect Performance

Signature generation is mainly affected by d_{ext} and the degree D of the secret univariate polynomial. According to Theorem 4, we can find the roots of $F \in \mathbb{F}_{2^{d_{ext}}}[X]$ in $O(D \cdot d_{ext})$ binary operations. So, d_{ext} and D are the main parameters which influence the efficiency. In Section 7.7, we will see how to choose these parameters in function of the other security parameters.

7.6 Expected Strength in General

We review in this part known results on the provable security of GeMSS. This includes the required number of iterations in the Feistel–Patarin scheme (Section 7.6.1) as well as the security (Section 7.6.2) in the sense of the existential unforgeability against adaptive chosen-message attack (EUF-CMA). Finally, we demonstrate in Section 7.6.3 that the failure probability of the signing process is completely negligible.

7.6.1 Number of Iterations

Here, we explain how the number of iterations $\texttt{nb_ite} > 0$ has to be chosen in Algorithms 31 and 32. Until round 3, we used the following result from $\texttt{QUARTZ}$ [134, 58].

Theorem 5. The number of iterations nb_ite has to be chosen such that

$$2^{\frac{\text{nb_ite}}{\text{nb_ite}+1} m} \geqslant 2 \,.$$

We used this result to derive the number of iterations for $\texttt{GeMSS}$, $\texttt{BlueGeMSS}$ and $\texttt{RedGeMSS}$.

The round 3 report [1] pointed that "*It is possible that there may yet be additional trade-offs to further improve performance. In particular, the consideration of the number of bit operations involved in a hash collision attack may warrant a reevaluation of the number of iterations required in the Feistel–Patarin transformation.*" We study this point in Section 7.7.2.

7.6.2 Existential Unforgeability against Chosen Message Attack

EUF-CMA of $\texttt{HFEv-}$-based signature schemes, such as $\texttt{GeMSS}$, has been mainly investigated in [148]. The authors demonstrated that a minor, but costly, modification of Inv_p (Algorithm 2), permits to achieve EUF-CMA for $\texttt{HFEv-}$. In fact, the result of [148] applies to Algorithms 3 and 4, i.e. when nb_ite is equal to one. In this case, the EUF-CMA of (modified) $\texttt{HFEv-}$-based signature schemes follows easily from [148].

We first formalize (in the random oracle model) the existential unforgeability of any signature scheme against chosen message attacks [148, Definition 2].

Definition 3. We say that a signature scheme (KeyGen, Sign, Verify) is $(\ (\ ), t(\ ), q_H(\ ), q_s(\ ))$-secure if there is no forger $\mathcal{A}$ who takes a public-key pk generated via $(\cdot, \text{pk}) \leftarrow \text{KeyGen}(1\ )$, after at most $q_H(\ )$ queries to the random oracle, $q_s(\ )$ signature queries, and $t(\ )$ processing time, then outputs a valid signature with probability at least $(\ )$.

We want to provably reduce the EUF-CMA of any $\texttt{MI}$-based signature scheme to the hardness of inverting the public-key of this one. Formally [148, Definition 6]:

Definition 4. We say that the $\texttt{MI}$-based function generator origin.KeyGen is $(\ (\ ), t(\ ))$-secure if there is no inverting algorithm that takes $\text{pk} = p$ generated via $(\cdot, \text{pk}) \leftarrow \text{origin.KeyGen}(1\ )$ and a challenge $d \in_R \mathbb{F}_q^m$, then finds a preimage $x_s \in \mathbb{F}_q^{n_{\text{var}}}$ such that

$$p(x_s) = d$$

at $t(\ )$ processing time with probability at least $(\ )$.

Following [148], we now explain how to modify $\texttt{HFEv-}$-based signing and verifying processes for proving EUF-CMA. Recall that D is degree of the secret polynomial with $\texttt{HFEv}$-shape. The main modification proposed by [148] is roughly to repeat D times the inversion step described in Algorithm 2. Let be the length (in bits) of a random salt. The modified inversion process is given in Algorithm 34.

Algorithm 34 Modified inverse map of the public-key for HFEv−-based signature schemes.

1: **function** $\mathrm{Inv}_p^*\ M \in \{0,1\}^*, sk = F, S^{-1}, T^{-1} \in \mathbb{F}_{q^{d_{ext}}}[X, v_1, \ldots, v_v] \times GL_{n_{var}}\ \mathbb{F}_q \times GL_{d_{ext}}\ \mathbb{F}_q$
2: $v \in_R \mathbb{F}_q^{n_{var}-d_{ext}}$ — The notation $\in_R$ stands for randomly sampling.
3: **repeat**
4: $r \in_R \mathbb{F}_q$
5: $salt \in_R \{0,1\}$
6: $h \leftarrow$ SHA-3$(M \| salt)$
7: $d \leftarrow$ first $\lceil m \log_2(q) \rceil$ bits of h — $d \leftarrow H_1(M \| salt) \in \mathbb{F}_q^m$.
8: $d' \leftarrow (d, r) \in \mathbb{F}_q^{d_{ext}}$
9: $Y \leftarrow {}^{-1}\ d' \cdot T^{-1} \in \mathbb{F}_{q^{d_{ext}}}$
10: $F_Y(X) \leftarrow F(X, v) - Y$
11: $(n, L_{roots}) \leftarrow$ FindRoots(F_Y) — Algorithm 25.
12: $u \in_R [\![1, D]\!]$
13: **until** $1 \leqslant u \leqslant n$ — F_Y has n roots.
14: $Z \leftarrow L_{roots}[u]$
15: **return** $salt, ((Z), v) \cdot S^{-1} \in \{0,1\} \times \mathbb{F}_q^{n_{var}}$
16: **end function**

Given Algorithm 34, we can define origin.Sign1* as the signature algorithm 3 instantiated with Inv_p^*, and including the salt in the signature. This increases the signature size by bits. Since d is computed in Algorithm 34, Algorithm 3 has to be modified by giving directly M as first argument of Inv_p^* (because $nb_ite = 1$, we obtain that origin.Sign1* is exactly Inv_p^* here). Similarly, origin.Verify1* is the verification algorithm 4, by computing d in accordance with Algorithm 34. The same modifications could be applied to the signing and verifying processes of GeMSS, with the possibility to use a unique salt for the nb_ite inversion steps.

Theorem 6. [148, Theorem 2]. *Let Sign*$_{\text{HFEv-}}$* *be the modified* HFEv−*-based signature scheme defined by* (origin.KeyGen, origin.Sign1*, origin.Verify1*). *If the function generator* origin.KeyGen *is* $'$, t' *-secure, then* Sign$_{\text{HFEv-}}$* *is* $, t, q_H, q_S$ *-secure, where:*

$$= \frac{'(q_H + q_S + 1)}{1 - (q_H + q_S)q_S 2^{-}},$$

$$t = t' - (q_H + q_S + 1)(t_{\text{HFEv-}} + O(1)),$$

and $t_{\text{HFEv-}}$ *is running time to evaluate the public-key.*

The modified scheme introduces three major changes. First, Inv_p^* is more costly than Inv_p. The expected number of calls to the root finding algorithm (Step 11) in Inv_p^* is D [148]. In Inv_p, the average number of calls to the root finding algorithm (Step 8) is $\frac{1}{1-1/e} \approx 1.58$. In GeMSS, we are typically considering D between 17 and 513. For efficiency reasons, we did not incorporate this modification in our implementation.

Remark 13. The threshold D at Step 12 of Inv_p^* corresponds to a bound on the number of roots of the univariate polynomial F_Y at Step 11. As shown in Table 2.1, the probability of finding a large number of roots is almost null. Thus, as also mentioned in [148], the threshold D at Step 12 can be theoretically much decreased without compromising the proof of Theorem 6. The authors of [148] mentioned a value around 30 for the threshold.

Remark 14. For a fairly large message M, we can use Remark 12 to decrease the cost of repeating $\texttt{SHA-3}(M\|\mathrm{salt})$ in Inv_p^*. However, we note that precomputing $h_M \leftarrow \texttt{SHA-3}(M)$, then computing $\texttt{SHA-3}(h_M\|\mathrm{salt})$ instead of $\texttt{SHA-3}(M\|\mathrm{salt})$, necessarily decreases the performance of the verifying process (because $\mathrm{nb_ite} = 1$).

The second change is the use of a salt, which increases the signature size of bits. From Theorem 6, the authors of [148] considered that the length of the salt has to be $\log_2 q_S \cdot (q_H + q_S)$ bits. Since NIST proposed to consider that any adversary has access to signatures for no more than 2^{64} chosen messages, independently of the security level [127], we consider 128-bit salts.

The last change is about the number of iterations. The treatment of [148] did not include the use of a Feistel–Patarin transform (Section 4.2). It is an interesting open problem of formally proving the EUF-CMA for $\mathrm{nb_ite} > 1$.

7.6.3 Signature Failure

This analysis is essentially similar to the one performed for $\texttt{QUARTZ}$ [134]. A failure can occur in Inv_p (Algorithm 2) if $L_{\mathrm{Sol}} = \emptyset$ for all $(r, v) \in \mathbb{F}_q \times \mathbb{F}_q^v$. For $\texttt{HFEv-}$, the probability that L_{Sol} is empty for a given $(d', v) \in \mathbb{F}_q^{d_{\mathrm{ext}}} \times \mathbb{F}_q^v$ is $\exp(-1)$ (Table 2.1). Thus, Algorithm 2 fails with probability $\exp(-q^{n_{\mathtt{var}}-m})$. Finally, Inv_p is called $\mathrm{nb_ite}$ times during the signing process. By using that $(1-a)^n \geqslant 1 - na$ for $n \in \mathbb{N}$ and $a \in \mathbb{R}$ such that $a \leqslant 2$, the probability of failure is then:

$$1 - \left(1 - \exp\left(-q^{n_{\mathtt{var}}-m}\right)\right)^{\mathrm{nb_ite}} \leqslant \mathrm{nb_ite} \cdot \exp\left(-q^{n_{\mathtt{var}}-m}\right).$$

This probability is completely negligible. For $q \geqslant 2$, $+ v \geqslant 8$ and $\mathrm{nb_ite} \leqslant 4$, this probability is less than $2^{2-256/\ln(2)} < 2^{-367}$. For $\texttt{GeMSS128}$, the failure probability is less than $2^{-24204404}$.

Similarly, the failure probability of origin.Sign*1 (Section 7.6.2) is that of Inv_p^*, which is:

$$\exp\left(-\min\left(2, q^m\right) \cdot q^{d_{\mathrm{ext}}-m}\right).$$

In Section 7.4.12, we propose (for example) to select one root only if the latter is unique. In this case, the probability of failure for the signing process is upper bounded by:

$$\mathrm{nb_ite} \cdot \left(1 - \exp(-1)\right)^{q^{n_{\mathtt{var}}-m}} < \mathrm{nb_ite} \cdot \exp\left(-q^{n_{\mathtt{var}}-m}\right) \cdot 2^{q^{n_{\mathtt{var}}-m}}.$$

7.7 Security

Here, we study the choice of security parameters in function of known attacks presented in Chapter 4, in order to achieve the NIST security levels (Section 4.1). We deduce a general protocol to design a secure $\texttt{HFEv-}$-based signature scheme. In this part, we study the security of $\texttt{GeMSS}$, but also that of $\texttt{Inner.DualModeMS}$ (Chapter 8) which is based on another set of parameters.

7.7.1 Minimum Number of Equations

We start by studying constraints on the number of equations. To do it, we evaluate the cost of all attacks implying only m (and linear in nb_ite). These attacks have to be repeated nb_ite times because we use the Feistel–Patarin construction (Section 4.2), so we take a lower bound by setting nb_ite to one. We obtain that `BooleanSolve` is the most dangerous attack. This implies that m cannot be smaller than 162, 243 and 324 respectively to reach 128-bit, 192-bit and 256-bit security level. Of course, we considered that the constant in the big Oh notation is one (Section 4.4.2). In practice, the constant is large, and in particular larger than the cost of the `SHA-3` function. Therefore, setting m to 162, 243 and 324 permits to reach respectively the security level II, IV and VI. We summarize in Tables 7.11 and 7.12 the cost of best attacks for these values of m. We also consider $m = 2$ for $\in \{128, 192, 256\}$, that we use in `Inner.DualModeMS`. The exhaustive search attacks are considered in Table 7.11. The given costs are exact, unlike Table 7.12 where the cost of attacks is a lower bound. These attacks are asymptotically better, which means they are more e cient only from a threshold value of m. We note that for $m = 2$, the security is largely satisfied, even assuming that the constant in the big Oh notation is one.

m	fast ex. search (4.3)	quantum ex. search (4.4)		quantum ex. search (4.5)	
162	$2^{166.87}$	$2^{104.56}$	328 qubits	$\approx 2^{105.56}$	174 qubits
243	$2^{247.98}$	$2^{146.80}$	490 qubits	$\approx 2^{147.80}$	255 qubits
324	$2^{329.06}$	$2^{188.54}$	652 qubits	$\approx 2^{189.54}$	337 qubits
256	2^{261}	$2^{153.52}$	516 qubits	$\approx 2^{154.52}$	269 qubits
384	$2^{389.10}$	$2^{219.27}$	772 qubits	$\approx 2^{22027}$	397 qubits
512	$2^{517.16}$	2^{28451}	1028 qubits	$\approx 2^{28551}$	526 qubits

Table 7.11: Complexity of solving a multivariate quadratic system of m equations in m variables in $\mathbb{F}_2$, with the exhaustive search. In practice, these attacks have to be repeated nb_ite times.

m	approximation (4.6)	`BooleanSolve` (4.13)	`QuantumBooleanSolve` (4.14)	
162	$2^{141.99}$	$2^{128.30}$	$2^{74.84}$	$O(m)$ qubits
243	$2^{212.98}$	$2^{192.45}$	$2^{112.26}$	$O(m)$ qubits
324	2^{28398}	2^{25660}	$2^{149.68}$	$O(m)$ qubits
256	2^{22438}	2^{20275}	$2^{118.27}$	$O(m)$ qubits
384	2^{33657}	2^{30412}	$2^{177.40}$	$O(m)$ qubits
512	2^{44876}	2^{40550}	2^{23654}	$O(m)$ qubits

Table 7.12: Lower bound on the complexity of solving a multivariate quadratic system of m equations in m variables in $\mathbb{F}_2$, with asymptotically fast algorithms. In practice, these attacks have to be repeated nb_ite times.

7.7.2 Trade-Off between Number of Equations and Number of Iterations

Now, we study the cost of the generic attack described in Section 4.2, in order to set m and nb_ite.

Slight improvement of the generic attack. Following [133, Remark 2], as well as the idea of the proof of [58], we describe a way to implement the generic attack in Algorithm 35, as well as its cost in Lemma 4.

Algorithm 35 Generic attack against the Feistel–Patarin construction.

Input: a function $G : \mathbb{F}_q^{n_{\mathtt{var}}} \rightarrow \mathbb{F}_q^m$, and an integer ℓ such that $1 \leqslant \ell \leqslant q^{n_{\mathtt{var}}}$.
Output: a message and its signature respecting the Feistel–Patarin construction associated to G.

0. Let F be a hash table such that $F[i]$ returns a value corresponding to the index i if the latter exists (note that several values can have the same index).
1. Let $s_1, \dots, s_\ell$ be random elements of $\mathbb{F}_q^{n_{\mathtt{var}}}$ and set $F[G(s_i)] = s_i$ for $1 \leqslant i \leqslant \ell$. F is an inversion table of the G function.
2. Try to sign a random message d by using Algorithm 12. Perform each inversion of G by using the inversion table. If a use of the inversion table fails to return a result, then abort and repeat this step with a new random message d.
3. Return d and the forged signature at Step 2.

Lemma 4. Let $H_1 : \{0,1\}^* \rightarrow \mathbb{F}_q^m$ be a hash function, and $G : \mathbb{F}_q^{n_{\mathtt{var}}} \rightarrow \mathbb{F}_q^m$ be a function. Then, Algorithm 35 requires storing $\ell(m + n_{\mathtt{var}}) \log_2(q)$ bits, computing ℓ evaluations of G and on average

$$\sum_{i=1}^{nb_ite} \left(\frac{q^m}{\ell}\right)^{nb_ite+1-i} \text{ evaluations of } H_1.$$

Proof. For convenience, let $x = \frac{q^m}{\ell}$. Step 1 requires generating a hash table of ℓ couples of index and evaluation, which requires computing ℓ evaluations of G and storing $\ell(m + n_{\mathtt{var}}) \log_2(q)$ bits. Then, the probability of successfully inverting nb_ite times G, for a random input, is x^{-nb_ite}. So, Step 2 requires on average x^{nb_ite} attempts to succeed the forge of a valid signature, implying to evaluate H_1 for x^{nb_ite} messages. If the inversion fails, then Step 2 aborts. The probability of successfully inverting one random message is x^{-1}. So, x^{nb_ite-1} messages succeed to continue Step 2. Repeating this process, we obtain that x^{nb_ite-1} evaluations of H_1, then x^{nb_ite-2} evaluations of H_1, ... until x evaluations of H_1 are computed. We also have that $x^0 = 1$ message succeeds the final inversion. So, Algorithm 35 requires on average $x^{nb_ite+1-i}$ evaluations of H_1 for $1 \leqslant i \leqslant nb_ite$, which concludes the proof. □

The generic attack of [58] only considers $\ell = \left(\frac{q^m}{\ell}\right)^{nb_ite} = q^{\frac{nb_ite}{nb_ite+1}m}$, because this value balances the number of evaluation of H_1 and G for $nb_ite = 1$. In practice, these evaluations have probably different costs. Moreover, multipoint evaluation can be used to minimize the practical cost (e.g. Remark 12), it is why each complexity was given in function of the number of evaluation. So, ℓ should be chosen to minimize the overall cost of Algorithm 35, and by taking account the required memory.

Theoretical estimation of the cost of G**.** Here, we consider that G is p in Algorithm 35, and H_1 is SHA-3 in Lemma 4. On the one hand, we estimate the cost of the evaluation of a multivariate quadratic system over $\mathbb{F}_2$ as follows. Let $n'_{\mathtt{var}} = \frac{n_{\mathtt{var}}}{2}$. On average, the evaluation vector has $n'_{\mathtt{var}}$ or $n'_{\mathtt{var}} + (n_{\mathtt{var}} \bmod 2)$ non-null components. We deduce the number of non-null monomials is on average:

$$N' = \frac{n'_{\mathtt{var}} + (n_{\mathtt{var}} \bmod 2)\,(n'_{\mathtt{var}} + 1)}{2} + 1 \simeq \frac{1}{4}N.$$

Thus, we can lower bound the average cost of an evaluation of p by mN' bit operations. On the other hand, the cost of the evaluation of SHA-3 is estimated to 2^{18} gates (Section 4.1). With these estimations, we can now use Lemma 4 to estimate the cost of Algorithm 35.

For $n_{\mathtt{var}} \geqslant m \geqslant 128$, the cost of the evaluation of p is greater than that of SHA-3. Therefore, a generic attack requiring 2 evaluations is necessarily more expensive against a MI-based signature scheme than against SHA-3, for $\in \{128, 192, 256\}$. This allows to achieve a security level II, IV or VI if the cost of generic attack is greater or equal to 2 evaluations. Thus, we can propose a simple way to choose m and nb_ite. We lower bound the cost of the generic attack by using $= 2^{\frac{nb_ite}{nb_ite+1}\cdot m}$, since the latter balances evaluations of p and SHA-3. Then, we can lower bound the cost of Algorithm 35 by $2^{\frac{nb_ite}{nb_ite+1}\cdot m}$ evaluations of SHA-3, which implies choosing m and nb_ite such that

$$\leqslant \frac{nb_ite}{nb_ite+1} \cdot m. \tag{7.3}$$

This method was used to set m and nb_ite in GeMSS and Inner.DualModeMS. For GeMSS, we consider the minimum value of m to obtain the smallest public-key size. This implies setting nb_ite to 4. For Inner.DualModeMS, necessarily, nb_ite has to be set to one. So, the number of equations is 2.

In Table 7.13, we propose to study the minimum cost of generic attacks more accurately. Since the number of variables depends on the underlying scheme, we give lower bounds by considering $n_{\mathtt{var}} = m$. Moreover, we lower bound the cost given by Lemma 4 by

$$\min_{\in\mathbb{N}^*} \; C_G \cdot \quad + C_{H_1} \cdot \left(\frac{2^m}{\quad}\right)^{nb_ite}, \tag{7.4}$$

where C_G is the cost of evaluating G and C_{H_1} is the cost of computing H_1.

m	nb_ite	evaluation	optimal	generic attack	memory (bits)
162	4	$2^{19.04}$	$2^{129.79}$	$2^{149.15}$	$2^{137.13}$
243		$2^{20.79}$	$2^{194.24}$	$2^{215.35}$	$2^{202.17}$
324		$2^{22.03}$	$2^{258.79}$	$2^{281.14}$	$2^{267.13}$
168	3	$2^{19.19}$	$2^{126.10}$	$2^{145.70}$	$2^{133.49}$
256	1	$2^{21.01}$	$2^{126.49}$	$2^{148.50}$	$2^{134.49}$
384		$2^{22.76}$	$2^{189.62}$	$2^{213.38}$	$2^{198.20}$
512		$2^{24.01}$	$2^{253.00}$	$2^{278.00}$	$2^{262.00}$

Table 7.13: Lower bound on the complexity of finding a collision with a generic attack (Lemma 4). Here, we consider $n_{\mathtt{var}} = m$, calls to p, $\left(\frac{2^m}{}\right)^{nb_ite}$ calls to the hash function, and a memory cost of m bits. The cost of evaluating p is lower bounded by mN' bit operations.

The optimal is given by $= \lfloor \ ' \rfloor$ or $= \lceil \ ' \rceil$, for $' = \left(\text{nb_ite} \cdot C_{H_1} \cdot C_G^{-1} \cdot 2^{m \cdot \text{nb_ite}} \right)^{\frac{1}{\text{nb_ite}+1}}$. For GeMSS and Inner.DualModeMS, the generic attack is slightly harder to perform than on SHA-3, confirming the correctness to use Equation (7.3). For GeMSS184, the authors fixed m to 168, and our lower bound is too small to achieve the level II of security. So, we consider a more accurate lower bound by considering $n_{\text{var}} = m + 32$. This implies the generic attack requires $2^{146.08}$ gates and $2^{133.36}$ bits (with $= 2^{125.97}$ and $C_G = 2^{19.69}$), which allows to reach the level II of security, and confirms the choice to set nb_ite to three, despite the fact that $\frac{\text{nb_ite}}{\text{nb_ite}+1} \cdot m = 126 < 128$.

Practical estimation of the cost of G. In Table 7.14, we summarize some experiments to estimate the ratio C_G/C_{H_1}. We compare our best and state-of-the-art (variable-time) evaluation function (in AVX2) to the best implementations of SHA-3 from XKCP (i.e. the Haswell implementation). We consider the hash value of 2-bit sequences when SHA3- is used. We use SHA3-256 (respectively SHA3-384 and SHA3-512) for level I (respectively III and V). The minimum ratio required to reach the given security level (third column) is obtained from Equation (7.4) by taking nb_ite $= 3$, $C_{H_1} = 2^{18}$ and the value of minimizing C_G (cf. Lemma 5 by taking g from Table 4.2). The experimental (exp.) ratio corresponds to the ratio of the running time of our public-key evaluation by that of SHA-3. Sequential means that C_{H_1} corresponds to the running time to compute one SHA-3 hash (in AVX2), whereas the parallel version considers the cost of computing four SHA-3 hashes (in AVX2), divided by four to obtain the cost of one SHA-3 hash.

level	m	$C_G/2^{18}$ (7.4)	experimental C_G/C_{H_1}	
			sequential SHA-3	parallel SHA-3
I	162	12	10.87	26.79
	163	6	$\geqslant 10.87$	$\geqslant 26.79$
	164	3		
	165	1.5		
	166	0.75		
III	243	241.90	24.47	60.60
	244	120.95	$\geqslant 24.47$	$\geqslant 60.60$
	245	60.48		
	246	30.24		
	247	15.12		
	248	7.56		
V	324	12288	55.20	135.15
	332	48	> 55.20	> 135.15
	333	24		
	334	12		

Table 7.14: Minimum ratio of the cost of evaluating a boolean system of m equations in m variables by that of SHA-3. For example, for $m = 163$, 6 means that SHA-3 should be at least 6 times faster than evaluating a boolean system of m polynomials in m variables to reach the first security level. We give the experimental ratio on a Skylake processor (LaptopS) using the AVX2 instruction set. We consider the sequential and parallel versions of SHA-3 from the Extended Keccak Code Package (XKCP), both using the AVX2 instruction set.

From Table 7.14, we see that slightly increasing the number of equations allows to reduce the number of iterations to three. The results are summarized in Table 7.15. We assume that $C_{H_1} = 2^{18}$, and we give the minimum ratio C_G/C_{H_1} (cf. Lemma 5) required to reach security levels I, III and V. These parameters are used in Sections 7.8.4, 7.8.5 and 7.8.6, where we introduce WhiteGeMSS, MagentaGeMSS and CyanGeMSS.

m	nb_ite	C_G/C_{H_1}	optimal	generic attack	memory (bits)
163	3	6	2^{122}	2^{143}	$2^{129.35}$
247		15.12	$2^{184+\frac{2}{3}}$	2^{207}	$2^{192.62}$
333		24	2^{249}	2^{272}	$2^{257.38}$

Table 7.15: Lower bound on the complexity of finding a collision with the generic attack, Equation (7.4). Here, we consider $C_{H_1} = 2^{18}$, calls to p, $\frac{2^m}{}^{nb_ite}$ calls to the hash function, and a memory cost of m bits.

Lemma 5. If (7.4) is equal to $g \in \mathbb{R}_+^*$, then $\in \mathbb{N}^*$ minimizing C_G is $= \lfloor ' \rfloor$ or $= \lceil ' \rceil$, for

$$' = 2^m \cdot (nb_ite + 1) \cdot C_{H_1} \cdot g^{-1} {}^{\frac{1}{nb_ite}} \in \mathbb{R}_+^*.$$

7.7.3 Experimental Results for HFEv-

The main question in the design of GeMSS is to quantify, as precisely as possible, the effect of the modifiers (Section 2.4.1) on the degree of regularity (Section 4.4.1). To do so, we performed experimental results on the behavior of a direct attack against HFEv-, i.e. computing a Gröbner basis of (4.2). We mention that similar experiments were performed in [144].

We first consider $v = 0$, and denote by the number of minus equations, i.e. $m = d_{ext} -$. According to the upper bound (4.11), the degree of regularity should increase by 1 when 2 equations are removed. In Tables 7.16 and 7.17, we report the degree of regularity D_{reg}^{Exp} reached during a Gröbner basis computation of a system of $m = d_{ext} -$ equations in m variables coming from a HFE- public-key, generated from a univariate polynomial in $\mathbb{F}_{2^{d_{ext}}}[X]$ of degree D. We also reported the degree of regularity D_{reg}^{Theo} of a semi-regular system of the same size (as in Table 4.3).

The experimental results of HFE-, no vinegar, are not completely conclusive. Whilst the degree of regularity appears to increase, it seems difficult to predict its behavior in function of the number of minus equations. This was also observed in [144] where the authors advised against using the minus modifier alone. Thus, the minus modifier should not be used alone.

We now consider the opposite situation, i.e. no minus equations and we increase the number of vinegar variables, i.e. HFEv. In Tables 7.18, 7.19 and 7.20, D_{reg}^{Exp} corresponds to the degree of regularity reached during a Gröbner basis computation of a system of $m = n_{var} - v$ equations in m variables coming from HFEv public-key, generated from a univariate polynomial in $\mathbb{F}_{2^{d_{ext}}}[X]$ of degree D.

The experimental results are more stable. In all cases, we need to add 3 vinegar variables to increase the degree of regularity by 1.

We also performed experimental results with a combination of vinegar and minus. Similarly to [144], we observed that the behavior obtained seems similar for HFEv- with $= 0$ and v vinegar variables than for a HFEv- with $= v/2$ and $v/2$ vinegar variables.

d_{ext}		m	D	D_{reg}^{Theo}	D_{reg}^{Exp}
32	0				3
33	1	32	4	7	3
34	2				3
35	3				4
36	4				4
37	5	32	4	7	4
38	6				4
39	7				4
40	8				5
41	9				5
42	10	32	4	7	5
43	11				5
44	12				5
45	13				5
46	14				6
47	15				6
48	16				6
49	17	32	4	7	6
50	18				6
51	19				6
52	20				6

d_{ext}		m	D	D_{reg}^{Theo}	D_{reg}^{Exp}
41	0				3
42	1	41	4	8	3
43	2				3
44	3				4
45	4				4
46	5	41	4	8	4
47	6				4
48	7				4

Table 7.16: HFE- with $D = 4$; 32 and 41 equations.

d_{ext}		m	D	D_{reg}^{Theo}	D_{reg}^{Exp}
32	0				4
33	1	32	17	7	4
34	2				4
35	3	32	17	7	5
36	4				5
37	5				6
38	6	32	17	7	6
39	7				6

d_{ext}		m	D	D_{reg}^{Theo}	D_{reg}^{Exp}
41	0				4
42	1	41	17	8	4
43	2				4
44	3	41	17	8	5
45	4				5

Table 7.17: HFE- with $D = 17$; 32 and 41 equations.

n_{var}	v	$m = n_{var} - v$	D	D_{reg}^{Theo}	D_{reg}^{Exp}
32	0	32	6	7	3
39	7	32	6	7	5
40	8	32	6	7	6
41	9				6
42	10				6
43	11	32	6	7	7
44	12				7
47	15	32	6	7	7

Table 7.18: HFEv with $D = 6$ and 32 equations.

n_{var}	v	$m = n_{var} - v$	D	D_{reg}^{Theo}	D_{reg}^{Exp}
25	0	25	9	6	3
26	1	25	9	6	4
27	2				4
28	3				4
29	4	25	9	6	5
30	5				5
31	6				5
32	7	25	9	6	6

Table 7.19: HFEv with $D = 9$ and 25 equations.

n_{var}	v	m	D	D_{reg}^{Theo}	D_{reg}^{Exp}
25	0	25	16	6	3
26	1	25	16	6	4
27	2				4
28	3				4
29	4	25	16	6	5
30	5				5
31	6				5
32	7	25	16	6	6

n_{var}	v	m	D	D_{reg}^{Theo}	D_{reg}^{Exp}
32	0	32	16	7	3
33	1	32	16	7	4
34	2				4
35	3				4
36	4	32	16	7	5
37	5				5

Table 7.20: HFEv with $D = 16$; 25 and 32 equations.

7.7.4 Minimum Number of Vinegar Variables

In Tables 7.21 and 7.22, we study values of v which achieve the security levels against MinRank-based attacks. In particular, we consider MinRank-based attacks with projections [68]. In Table 7.21, we take $= v$. In Table 7.22, we take $= 0$. Both behaviors seem similar, except for the project-then-MinRank attack. The latter implies upper bounding the number of vinegar variables for $= 0$, which can be dangerous for practical values of v. So, we advise against using the vinegar modifier alone. However, we note that the security can be achieved by increasing D.

(level, m, D)	MinRank (4.17)	MinRank-then-project [68]	project-then-MinRank [68]
(II, 162, 17)	$+ v \geqslant 10$	$= v \geqslant 5$	$= v \geqslant 4$
(IV, 243, 17)	$+ v \geqslant 16$	$= v \geqslant 8$	$= v \geqslant 7$
(VI, 324, 17)	$+ v \geqslant 22$	$= v \geqslant 11$	$= v \geqslant 9$
(II, 256, 17)	$+ v \geqslant 8$	$= v \geqslant 4$	$= v \geqslant 3$
(IV, 384, 17)	$+ v \geqslant 13$	$= v \geqslant 6$	$= v \geqslant 5$
(VI, 512, 17)	$+ v \geqslant 18$	$= v \geqslant 9$	$= v \geqslant 8$

Table 7.21: Values of and v which achieve the security levels against MinRank-based attacks.

(level, m, D)	MinRank (4.17)	MinRank-then-project [68]	project-then-MinRank [68]
(II, 162, 17)	$v \geqslant 10$	$v \geqslant 9$	$8 \leqslant v \leqslant 29$
(IV, 243, 17)	$v \geqslant 16$	$v \geqslant 15$	$14 \leqslant v \leqslant 32$
(VI, 324, 17)	$v \geqslant 22$	$v \geqslant 21$	$19 \leqslant v \leqslant 33$
(II, 256, 17)	$v \geqslant 8$	$v \geqslant 7$	$6 \leqslant v \leqslant 37$
(IV, 384, 17)	$v \geqslant 13$	$v \geqslant 12$	$11 \leqslant v \leqslant 42$
(VI, 512, 17)	$v \geqslant 18$	$v \geqslant 17$	$16 \leqslant v \leqslant 46$

Table 7.22: Values of v which achieve the security levels against MinRank-based attacks, for $= 0$.

In [10], the authors introduced the Support Minors technique to solve the MinRank-based problems. The complexity of this attack against parameters of round 2 proposals of GeMSS is as presented in Table 7.23. This technique is more e cient than all previous MinRank-based attacks. However, the degree of minors equations is already bigger than the degree of regularity considered in a direct attack. So, we can just choose parameters secure against the direct attack to be immune against the Support Minors technique.

7.7.5 Choice of Degree and Number of Modifiers

At this stage, we have a methodology for fixing the minimum number of equations m (Table 7.12), as well as the number of iterations nb_ite (Table 7.13). We now need to derive the number of vinegar variables v and minus equations required to achieve the degree of regularity corresponding to a given security level (Table 7.24). This is the most delicate point. According to the experiments performed in Section 7.7.3, and the insight provided by the key-recovery attacks (Section 7.7.4), we make the choice to balance v and .

(level, D, d_{ext}, , v)	Support Minors technique [10]
(II, 513, 174, 12, 12)	2^{158}
(IV, 513, 265, 22, 20)	2^{224}
(VI, 513, 354, 30, 33)	2^{304}
(II, 129, 175, 13, 14)	2^{162}
(IV, 129, 265, 22, 23)	2^{229}
(VI, 129, 358, 34, 32)	2^{305}
(II, 17, 177, 15, 15)	2^{160}
(IV, 17, 266, 23, 25)	2^{227}
(VI, 17, 358, 34, 35)	2^{305}

Table 7.23: Lower bound on the complexity of the Support Minors technique against parameters of round 2 proposals of GeMSS. Here, we consider that the constant in the big Oh notation is one.

In addition, we need to fix the degree D of the HFEv polynomial. This will give the initial degree of regularity for a nude HFE (Table 4.4). For GeMSS, we consider a secret univariate polynomial of degree $D = 513$. This corresponds to a degree of regularity of six for a HFE, i.e. without any modifier. The variants consider smaller degrees, in order to speed up the signing process.

From our experiments, we consider that 3 modifiers allow to increase the degree of regularity by one. Independently of the GeMSS submission [48], the author of [140] also derived a similar rule; as one can see from (4.12). In Table 7.25, we then derive the number of modifiers required as $+ v = 3 \times \text{Gap}$, Gap being the targeted degree of regularity minus the initial degree of regularity. We consider the number of equations m and the targeted degree of regularity as in Table 7.24. The last column of Table 7.25 gives the number of modifiers required. We present below the results for GeMSS, RedGeMSS, BlueGeMSS, WhiteGeMSS, MagentaGeMSS, CyanGeMSS and FGeMSS.

m	minimum D_{reg} required	lower bound on the cost of a Gröbner basis as given in (4.10)
162	14	$2^{131.16}$
243	20	2^{19251}
324	27	2^{26086}
333	26	2^{25607}
256	12	2^{13357}
384	17	$2^{194.17}$
512	23	2^{26364}

Table 7.24: Smallest degree of regularity required = (2). In practice, the Gröbner basis attack has to be repeated n t e times.

scheme	m	D	Gap	+ v
GeMSS128		513	14 − 6 = 8	24
BlueGeMSS128	162	129	14 − 5 = 9	27
RedGeMSS128		17	14 − 4 = 10	30
GeMSS192		513	20 − 6 = 14	42
BlueGeMSS192	243	129	20 − 5 = 15	45
RedGeMSS192		17	20 − 4 = 16	48
GeMSS256		513	27 − 6 = 21	63
BlueGeMSS256	324	129	27 − 5 = 22	66
RedGeMSS256		17	27 − 4 = 23	69
WhiteGeMSS256		513	26 − 6 = 20	60
CyanGeMSS256	333	129	26 − 5 = 21	63
MagentaGeMSS256		17	26 − 4 = 22	66
FGeMSS(266)		129	12 − 5 = 7	21
Inner.DualModeMS128	256			
Inner.RedDualModeMS128		17	12 − 4 = 8	24
FGeMSS(402)		640	17 − 6 = 11	33(1)
Inner.DualModeMS192	384	129	17 − 5 = 12	36
Inner.RedDualModeMS192		17	17 − 4 = 13	39
FGeMSS(537)		1152	23 − 6 = 17	51
Inner.DualModeMS256	512	129	23 − 5 = 18	54
Inner.RedDualModeMS256		17	23 − 4 = 19	57

(1) Here, + v ⩾ 33 is required. But the scheme uses + v = 36.

Table 7.25: Number of modifiers required in GeMSS. Except for the level V of security, the number of modifiers of WhiteGeMSS, CyanGeMSS and MagentaGeMSS are respectively obtained as these of GeMSS, BlueGeMSS and RedGeMSS.

7.8 Design

In [2, 1], NIST announced the second round and third round candidates. They also provided some recommendations for the selected candidates. The goal of this part is to address the comments from [2, 1] regarding GeMSS. The parameters proposed for GeMSS in the first round were very conservative in term of security. [2, 1] suggested exploring di erent parameters in order to improve e ciency. We address this comment as follows.

- We suggest six sets of parameters for each security level with several trade-o s. This includes the initial parameters of GeMSS proposed for the first round, and two new more aggressive parameters (BlueGeMSS and RedGeMSS). We also introduce WhiteGeMSS, CyanGeMSS and MagentaGeMSS thanks to a tighter analysis of the Feistel–Patarin construction (Section 7.7.2).

- In Section 7.8.7, we propose a slight modification of the extension degree to improve multiplications in $\mathbb{F}_{2^{d_{ext}}}$ on low-end devices.

- We then design a family of parameters that depends on only one parameter d_{ext}. We call this family FGeMSS(d_{ext}) (Section 7.8.8).

- In Section 7.8.9, we explore the use of sparse polynomials in GeMSS to improve the efficiency of the signing process.

- Finally, we present an exhaustive table including possible parameters and the corresponding timings in Section 7.8.10.

7.8.1 Set 1 of Parameters: GeMSS

The first set, that we call GeMSS (Table 7.26), was the parameters proposed for the first round.

scheme	(, D, d_{ext}, , v, nb_ite, m, n_{var})	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
GeMSS128	(128, 513, 174, 12, 12, 4, 162, 186)	352.19	13.44	16	32.25
GeMSS192	(192, 513, 265, 22, 20, 4, 243, 285)	1237.96	34.07	24	51.375
GeMSS256	(256, 513, 354, 30, 33, 4, 324, 387)	3040.70	75.89	32	72

Table 7.26: Summary of the parameters of GeMSS.

7.8.2 Set 2 of Parameters: RedGeMSS

We call RedGeMSS the schemes described in Table 7.27. The public-key of RedGeMSS128 is 1.065 times larger than GeMSS128, the time to sign with RedGeMSS128 is 296 times faster than GeMSS128. This is because we use a smaller D.

scheme	(, D, d_{ext}, , v, nb_ite, m, n_{var})	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
RedGeMSS128	(128, 17, 177, 15, 15, 4, 162, 192)	375.21	13.10	16	35.25
RedGeMSS192	(192, 17, 266, 23, 25, 4, 243, 291)	1290.54	34.79	24	54.375
RedGeMSS256	(256, 17, 358, 34, 35, 4, 324, 393)	3135.59	71.89	32	75

Table 7.27: Summary of the parameters of RedGeMSS.

7.8.3 Set 3 of Parameters: BlueGeMSS

We call BlueGeMSS the schemes described in Table 7.28. The public-key of BlueGeMSS128 is 1.032 times larger than GeMSS128, the time to sign with BlueGeMSS128 is 9.05 times faster than GeMSS128. This is because we use a smaller D.

scheme	(, D, d_{ext}, , v, nb_ite, m, n_{var})	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
BlueGeMSS128	(128, 129, 175, 13, 14, 4, 162, 189)	363.61	13.70	16	33.75
BlueGeMSS192	(192, 129, 265, 22, 23, 4, 243, 288)	1264.12	35.38	24	52.875
BlueGeMSS256	(256, 129, 358, 34, 32, 4, 324, 390)	3087.96	71.46	32	73.5

Table 7.28: Summary of the parameters of BlueGeMSS.

7.8.4 Set 4 of parameters: **WhiteGeMSS**

We call `WhiteGeMSS` the schemes described in Table 7.29.

scheme	(, D, d_{ext}, , v, nb_ite, m, n_{var})	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
`WhiteGeMSS128`	(128, 513, 175, 12, 12, 3, 163, 187)	358.17	13.56	16	29.375
`WhiteGeMSS192`	(192, 513, 268, 21, 21, 3, 247, 289)	1293.85	35.77	24	46.625
`WhiteGeMSS256`	(256, 513, 364, 31, 29, 3, 333, 393)	3222.69	70.99	32	64.125

Table 7.29: Summary of the parameters of `WhiteGeMSS`.

7.8.5 Set 5 of parameters: **MagentaGeMSS**

We call `MagentaGeMSS` the schemes described in Table 7.30.

scheme	(, D, d_{ext}, , v, nb_ite, m, n_{var})	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
`MagentaGeMSS128`	(128, 17, 178, 15, 15, 3, 163, 193)	381.46	13.22	16	31.625
`MagentaGeMSS192`	(192, 17, 271, 24, 24, 3, 247, 295)	1348.03	34.69	24	48.875
`MagentaGeMSS256`	(256, 17, 366, 33, 33, 3, 333, 399)	3321.72	70.41	32	66.375

Table 7.30: Summary of the parameters of `MagentaGeMSS`.

7.8.6 Set 6 of parameters: **CyanGeMSS**

We call `CyanGeMSS` the schemes described in Table 7.31.

scheme	(, D, d_{ext}, , v, nb_ite, m, n_{var})	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
`CyanGeMSS128`	(128, 129, 177, 14, 13, 3, 163, 190)	369.72	13.41	16	30.5
`CyanGeMSS192`	(192, 129, 270, 23, 22, 3, 247, 292)	1320.80	35.26	24	47.75
`CyanGeMSS256`	(256, 129, 364, 31, 32, 3, 333, 396)	3272.02	73.20	32	65.25

Table 7.31: Summary of the parameters of `CyanGeMSS`.

7.8.7 A Family of Parameters for Low-End Devices

The multiplication in $\mathbb{F}_{2^{d_{ext}}}$ is a crucial operation for the performance of `GeMSS`. On low-end devices, this operation is naturally very expensive. So, we design new parameters especially for these devices. It is well-known that multiplications in $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ can be efficiently computed in parallel, with vector instructions and logarithm tables (Section 7.4.9). A solution to improve performance of the multiplication in $\mathbb{F}_{2^{d_{ext}}}$ is to choose d_{ext} multiple of four or eight. This allows to use an isomorphism between $\mathbb{F}_{2^{d_{ext}}}$ and $\mathbb{F}_{16^{d_{ext}/4}}$ or $\mathbb{F}_{256^{d_{ext}/8}}$. We base our new parameters on these of `GeMSS`, by slightly modifying the balance between minus and vinegar. We obtain Table 7.32.

These parameters are a proposal to improve the performance of `GeMSS` on low-end devices. We do not have implementations exploiting the tower field representation of $\mathbb{F}_{2^{d_{ext}}}$. Therefore, we cannot estimate obtained speed-ups when `PCLMULQDQ` is not available.

level	nb_ite	m	D	+ v	$d_{ext} = 0 \bmod 4$, , v	$d_{ext} = 0 \bmod 8$, , v
I	4	162	513	24	172, 10, 14	176, 14, 10
			129	27	176, 14, 13	
			17	30	176, 14, 16	
III		243	513	42	264, 21, 21	
			129	45	264, 21, 24	
			17	48	268, 25, 23	264, 21, 27
V		324	513	63	356, 32, 31	352, 28, 35
			129	66	356, 32, 34	360, 36, 30
			17	69	360, 36, 33	

Table 7.32: Slight modification of GeMSS for low-end devices.

7.8.8 FGeMSS(d_{ext})

In multivariate schemes, we have many parameters that can be adjusted. This is an advantage since, for example, we can decrease the time to sign for a given security level if we increase the length of the public-key, i.e. some interesting trade-offs are possible. However, when a new cryptanalysis idea is found, it is not always easy for a non multivariate specialist to see how to adjust the parameters in order to maintain a given security level against the best known attacks. For example, when RSA-512 was factored, it was natural to suggest using a larger modulo n and looking at what value of n should be used from the best known attacks (instead of designing another scheme). But when an attack on QUARTZ was published with a security expected [82] to be slightly smaller than 2^{80}, it was not so easy to adjust the security parameters since we have many possibilities here. Therefore, we see that it is sometime convenient to have a dimension 1 family instead of a single point (like QUARTZ) or a many dimension family (like the variants of HFE).

Here, we present such dimension 1 family, called FGeMSS(d_{ext}). It is such that:

- nb_ite $= 1$,
- d_{ext} is again $m +$,
- $+ v = 21 + 0.11 \cdot (d_{ext} - 266)$, $= \frac{+ v}{2}$ and $v = \frac{+ v}{2}$,
- D is the largest sum of two powers of two less or equal to $129 + 4.2 \cdot (d_{ext} - 266)$.

The public-key is a system of $m = d_{ext} -$ equations in $n_{var} = d_{ext} + v$ variables over $\mathbb{F}_2$. For example, we obtain the following parameters.

scheme	(, D, d_{ext}, , v, nb_ite, m, n_{var})	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
FGeMSS(266)	(128, 129, 266, 10, 11, 1, 256, 277)	1232.13	24.55	16	34.625
FGeMSS(402)	(192, 640, 402, 18, 18, 1, 384, 420)	4243.73	62.60	24	52.5
FGeMSS(537)	(256, 1152, 537, 25, 26, 1, 512, 563)	10161.09	122.72	32	70.375

Table 7.33: Parameters of FGeMSS.

It can be emphasized that FGeMSS can be nicely combined with DualModeMS [83] (Chapter 8). In the case of FGeMSS(266), we will typically get a public-key of 512 bytes with a signature size of about 32 kB.

7.8.9 SparseGeMSS

In this section, we introduce s, a new security parameter. We propose to remove s terms in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$ from the HFEv polynomial to improve the efficiency of the signing process, as introduced in Section 9.3.3. When s is small, we think the security is not impacted by this change, whereas we can obtain a factor at most two for the signing process. This method is new and so a new analysis of security is required. We will present some experiments in Section 9.3.3.

The improvement is based on the fact that during the computation of the Frobenius map, a degree-$(2D-2)$ square in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$ is computed, then is reduced modulo H, a HFE polynomial in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$. In binary fields, all odd degree terms of a square are null, thanks to the linearity of the Frobenius endomorphism. Then, we remark that the Euclidean division of a square B by a square implies that the quotient Q is a square. H is not a square because it contains the terms X^{2^0} and X^{2^i+1} for $0 < i \leqslant k = \lfloor \log_2(D) \rfloor$. However, the gap between the odd degrees 2^i+1 and $2^{i+1}+1$ is 2^i. This gap increases quickly when i increases. So, if we take $D = 2^k+2$, then we remove the s largest odd degrees ($s \leqslant k$), we obtain a HFE polynomial $H = H_0 + X^{2^{k-s}+2} H_1$ for H_0 a degree-$(2^{k-s}+1)$ polynomial and H_1 a degree-$(2^k - 2^{k-s})$ square. By removing only one term (i.e. $s = 1$), the higher half of H is a square.

Now, we exploit the fact that H_1 is a square. This implies $Q = Q_0 + X^{2^{k-s}} Q_1$ for Q_0 a degree-$(2^{k-s}-1)$ polynomial and Q_1 a degree-$(2^k - 2^{k-s})$ square. Moreover, the classical Euclidean division algorithm (Algorithm 16) is equivalent to performing the classical multiplication of Q by H, then add it to B. So, if Q_1 is a square, we avoid the half of the multiplications for this part of Q. The size of Q_1 is $2^k - 2^{k-s} + 1$, so we avoid $2^{k-1} - \lfloor 2^{k-s-1} \rfloor$ multiplications in $\mathbb{F}_{2^{d_{\text{ext}}}}$.

With our sparse trick, all previous families of GeMSS could become more efficient by using their sparse version. To apply this transformation, we increment D if D is odd and we set $s = 3$. In this way, we avoid 43.75% of the field multiplications (for $D = 2^k + 2$) during the modular reduction by H. The speed-up for $D > 2^k + 2$ is different because our trick improves the modular reduction for $s = 0$ (because $Q = Q_0 + X^{2^k} Q_1$ with Q_1 a degree-$(D - 2^k - 2)$ square, so we avoid $\frac{D-2^k-2}{2} > 0$ multiplications in $\mathbb{F}_{2^{d_{\text{ext}}}}$). We take a small value of s to be secure, but large enough to obtain an interesting speed-up. The Frobenius map is the core of the signing process, so this factor remains approximately the same for the signing process. However, this method is not interesting for small degrees, because the Frobenius map can be computed more quickly with multi-squaring tables (Algorithm 22). Experimentally, we keep the previous speed-up for $D \geqslant 514$, we loose a part for $D = 130$ and $d_{\text{ext}} > 196$, and the method is completely useless for $D \leqslant 34$. For this reason, we give the possibility to use SparseGeMSS only for $D \geqslant 128$ (Tables 7.34, 7.35 and 7.36).

7.8.10 An Exhaustive Table for the Choice of the Parameters

Here, we propose a large number of security parameters. For different values of D and for nb_ite from 1 to 4, we take the smallest m such that $(m, \text{nb_ite})$ respects (7.3). Then, we deduce the number of modifiers, and so Δ and v. Finally, for $D \geqslant 128$, we take $s = 0$ then $s = 3$ (as described

in Section 7.8.9). In Tables 7.34, 7.35 and 7.36, we give the performance of these parameters with the final version of `MQsoft` (Chapter 9).

For nb_ite < 3, the number of equations is a multiple of eight. So, the public-key is naturally stored with the packed representation (Section 7.4.8). This implies the theoretical size of the public-key is reached without decreasing performances. For the other values of m, the performance of the verifying process decreases when $m \bmod 8$ increases.

(, D, d_{ext}, , v, nb_ite, s)	key gen. (Mc)	sign (Mc)	verify (kc)	|pk| (kB)	|sk| (B)	|sign| (bits)
(128,17, 268,12, 12, 1, 0)	47.4	2.15	38	1260	16	280
(128,17, 204,12, 15, 2, 0)	20.2	1.89	53.5	578	16	246
(128,17, 186,15, 15, 3, 0)	18.6	1.81	95.1	434	16	261
(128, 17, 177, 15, 15, 4, 0)	16.3	2.05	141	375	16	282
(128,33,268,12, 12, 1, 0)	49.3	4.59	38.4	1260	16	280
(128,33,204,12, 15, 2, 0)	20.8	4.79	53.9	578	16	246
(128,33,186,15, 15, 3, 0)	19.7	4.78	95.3	434	16	261
(128,33,177,15, 15, 4, 0)	17.1	5.78	142	375	16	282
(128,129,266,10, 11, 1, 0)	54.1	44.1	38.2	1230	16	277
(128,130,266,10, 11, 1, 3)	54.9	34.4	36.8	1230	16	277
(128,129,204,12, 12, 2, 0)	21.4	53.7	51.4	562	16	240
(128,130,204,12, 12, 2, 3)	21.4	39.1	51.6	562	16	240
(128,129,185,14, 13, 3, 0)	20.9	51.7	106	421	16	252
(128,130,185,14, 13, 3, 3)	20.9	39.3	107	421	16	252
(128, 129, 175, 13, 14, 4, 0)	18.4	67.2	134	364	16	270
(128,130,175,13, 14, 4, 3)	18.4	49	138	364	16	270
(128,513,265,9, 9, 1, 0)	57.6	442	36.2	1210	16	274
(128,514,265,9, 9, 1, 3)	57	293	36.2	1210	16	274
(128,513,202,10, 11, 2, 0)	22.2	494	50.2	547	16	234
(128,514,202,10, 11, 2, 3)	21.6	297	49.9	547	16	234
(128,513,183,12, 12, 3, 0)	22.6	452	102	408	16	243
(128,514,183,12, 12, 3, 3)	22.2	303	104	408	16	243
(128, 513, 174, 12, 12, 4, 0)	19.6	608	106	352	16	258
(128,514,174,12, 12, 4, 3)	19.8	372	107	352	16	258

Table 7.34: Performance of an exhaustive set of security parameters achieving the level I, with `MQsoft`. We use a Skylake processor (LaptopS). The results have three significant digits. The parameters in bold correspond to `RedGeMSS`, `BlueGeMSS` and `GeMSS`.

(, D, d_{ext}, , v, nb_ite, s)	key gen. (Mc)	sign (Mc)	verify (kc)	|pk| (kB)	|sk| (B)	|sign| (bits)
(192, 17, 404, 20, 19, 1, 0)	180	5.61	126	4300	24	423
(192, 17, 310, 22, 23, 2, 0)	85	4.25	159	2000	24	378
(192, 17, 279, 23, 25, 3, 0)	61.3	4.45	202	1480	24	400
(192, 17, 266, 23, 25, 4, 0)	57.1	5.55	335	1290	24	435
(192, 33, 404, 20, 19, 1, 0)	190	10.1	125	4300	24	423
(192, 33, 310, 22, 23, 2, 0)	90.1	8.87	159	2000	24	378
(192, 33, 279, 23, 25, 3, 0)	65.1	11.9	199	1480	24	400
(192, 33, 266, 23, 25, 4, 0)	61.3	15	336	1290	24	435
(192, 129, 402, 18, 18, 1, 0)	204	89.2	124	4240	24	420
(192, 130, 402, 18, 18, 1, 3)	204	73.7	125	4240	24	420
(192, 640, 402, 18, 18, 1, 0)	224	1560	122	4240	24	420
(192, 640, 402, 18, 18, 1, 3)	223	999	123	4240	24	420
(192, 129, 308, 20, 22, 2, 0)	96.1	88.2	159	1970	24	372
(192, 130, 308, 20, 22, 2, 3)	96.1	72.7	160	1970	24	372
(192, 129, 278, 22, 23, 3, 0)	70.4	139	191	1450	24	391
(192, 130, 278, 22, 23, 3, 3)	70.7	114	195	1450	24	391
(192, 129, 265, 22, 23, 4, 0)	65	173	325	1260	24	423
(192, 130, 265, 22, 23, 4, 3)	66	141	323	1260	24	423
(192, 513, 399, 15, 18, 1, 0)	219	960	119	4180	24	417
(192, 514, 399, 15, 18, 1, 3)	219	795	120	4180	24	417
(192, 513, 308, 20, 19, 2, 0)	101	864	155	1930	24	366
(192, 514, 308, 20, 19, 2, 3)	102	604	154	1930	24	366
(192, 513, 276, 20, 22, 3, 0)	75.3	1360	198	1430	24	382
(192, 514, 276, 20, 22, 3, 3)	75.3	905	191	1430	24	382
(192, 513, 265, 22, 20, 4, 0)	69.4	1760	304	1240	24	411
(192, 514, 265, 22, 20, 4, 3)	69.9	1180	304	1240	24	411

Table 7.35: Performance of an exhaustive set of security parameters achieving the level III, with `MQsoft`. We use a Skylake processor (LaptopS). The results have three significant digits. The parameters in bold correspond to `RedGeMSS`, `BlueGeMSS` and `GeMSS`.

(, D, d_{ext}, , v, nb_ite, s)	key gen. (Mc)	sign (Mc)	verify (kc)	|pk| (kB)	|sk| (B)	|sign| (bits)
(256, 17, 540, 28, 29, 1, 0)	545	10	380	10400	32	569
(256, 17, 415, 31, 32, 2, 0)	221	7.59	380	4810	32	510
(256, 17, 375, 33, 33, 3, 0)	159	7.36	605	3570	32	540
(256, 17, 358, 34, 35, 4, 0)	143	8.76	709	3140	32	600
(256, 33, 540, 28, 29, 1, 0)	569	18.7	380	10400	32	569
(256, 33, 415, 31, 32, 2, 0)	233	16.4	385	4810	32	510
(256, 33, 375, 33, 33, 3, 0)	164	17.3	609	3570	32	540
(256, 33, 358, 34, 35, 4, 0)	149	21.9	697	3140	32	600
(256, 129, 540, 28, 26, 1, 0)	607	153	379	10300	32	566
(256, 130, 540, 28, 26, 1, 3)	603	133	382	10300	32	566
(256, 129, 414, 30, 30, 2, 0)	246	185	379	4740	32	504
(256, 130, 414, 30, 30, 2, 3)	245	139	369	4740	32	504
(256, 129, 372, 30, 33, 3, 0)	171	191	579	3510	32	531
(256, 130, 372, 30, 33, 3, 3)	169	154	596	3510	32	531
(256, 129, 358, 34, 32, 4, 0)	152	248	680	3090	32	588
(256, 130, 358, 34, 32, 4, 3)	153	202	682	3090	32	588
(256, 513, 537, 25, 26, 1, 0)	651	1630	364	10200	32	563
(256, 514, 537, 25, 26, 1, 3)	645	1520	369	10200	32	563
(256, 1152, 537, 25, 26, 1, 0)	674	7430	367	10200	32	563
(256, 1152, 537, 25, 26, 1, 3)	672	4870	360	10200	32	563
(256, 513, 414, 30, 27, 2, 0)	258	1830	361	4680	32	498
(256, 514, 414, 30, 27, 2, 3)	261	1450	364	4680	32	498
(256, 513, 372, 30, 30, 3, 0)	175	1990	565	3460	32	522
(256, 514, 372, 30, 30, 3, 3)	175	1420	573	3460	32	522
(256, 513, 354, 30, 33, 4, 0)	158	2490	665	3040	32	576
(256, 514, 354, 30, 33, 4, 3)	159	1800	663	3040	32	576

Table 7.36: Performance of an exhaustive set of security parameters achieving the level V, with `MQsoft`. We use a Skylake processor (LaptopS). The results have three significant digits. The parameters in bold correspond to RedGeMSS, BlueGeMSS and GeMSS.

7.9 Advantages and Limitations

Since the first scheme of Matsumoto and Imai [119] in 1988, almost thirty years ago, multivariate-based cryptosystems have been extensively analyzed in the literature. We have designed GeMSS using this knowledge and derive a general methodology to derive parameters. We then proposed three sets of parameters: GeMSS, the most conservative, and BlueGeMSS/RedGeMSS that are more efficient (but also, more aggressive in term of security). We also performed practical experiments using the best known tools for computing Gröbner bases.

From a practical point of view, the main drawback of GeMSS is the size of the public-key. However, we mention that the generation of a (public-key, secret-key) remains rather efficient in GeMSS. The main advantages of GeMSS are the size of the signatures generated, about 2 bits, and the fast verification process.

7.10 MI-Based Cryptography in the NIST Post-Quantum Cryptography Standardization Process

Several MI-based signature schemes were proposed to the NIST PQC standardization process [128].

- GeMSS [50], a HFEv−-based scheme which is currently an alternate candidate of the third round.
- Gui [62], another HFEv−-based scheme not selected to the second round.
- DualModeMS [83], a HFEv−-based scheme using the transformation described in Chapter 3. DualModeMS can be considered as the dual mode of GeMSS. It did not go to the second round.
- LUOV [27]. This variant of UOV (Section 2.4.2) was broken in [66] and so did not go to the third round.
- Rainbow [65], a Rainbow-based scheme (Section 2.4.3) using $u = 2$. Currently, Rainbow is a finalist candidate of the third round.

We summarize security parameters and sizes of these candidate in Tables 7.38 and 7.39. The exact sizes are available in Appendix A. QUARTZ [134] is a HFEv−-based signature scheme submitted to the NESSIE project [145]. This scheme was broken because the number of minus equations and vinegar variables used was too small. Gui has large public-keys. The goal of Gui is to minimize the signature size as well as the running time of signing and verifying processes. Gui, unlike GeMSS minimizes the public-key size by setting nb_ite to four. This allows to minimize the number of equations, which also minimizes the number of variables. The original Gui-184 scheme used nb_ite = 2, which is insecure. The generic attack from Section 7.7.2 breaks it in 2^{112} evaluations of the public-key. So, the submitters incremented nb_ite to reach the announced security level. DualModeMS is a signature scheme with two layers (Chapter 3). The inner layer is a HFEv−-based signature scheme, which can be used to have a large public-key but a small signature. Because of a particularity of the outer layer, the inner layer cannot use the Feistel–Patarin construction, implying to set nb_ite to one.

Table 7.39 summarizes the parameters and sizes of Rainbow during the NIST PQC standardization process. For the first round of this process, Rainbow [63] was proposed as described in Section 2.4.3, for the fields $\mathbb{F}_{16}$, $\mathbb{F}_{31}$ and $\mathbb{F}_{256}$. The roman numbers correspond to the NIST security level

(Section 4.1), whereas a, b and c correspond respectively to $\mathbb{F}_{16}$, $\mathbb{F}_{31}$ and $\mathbb{F}_{256}$. We note that $\mathcal{S}$ and $\mathcal{T}$ were a ne transformations. Then, some changes were announced for the second round [64]. Only one field was kept by security level, then two new variants were proposed:

- cyclic Rainbow, which uses the trick described in Section 3.6 to decrease the public-key size by generating a part from a public seed,
- compressed Rainbow, which applies the technique from Section 7.4.1 to cyclic Rainbow for generating the (decompressed) secret-key from a secret seed. The public seed is also stored in the secret-key, because it is used during the signing process for generating the (decompressed) secret-key again.

We note another changes. The secret seed is also stored and used in Rainbow and cyclic Rainbow. Then, $\mathcal{S}$ and $\mathcal{T}$ are linear transformations with a special structure. Moreover, the authors only considered the quadratic terms of $\mathcal{F}$. Coupled to linear transformations, this choice leads to a public map p without linear and constant terms, implying $N = \binom{n_{var}+1}{2}$. For the third round [65], the parameters were modified to take into account new attacks [10]. The cyclic Rainbow variant becomes CZ-Rainbow (circumzenithal Rainbow).

Table 7.37 summarizes the performance measurements of GeMSS additional (best) implementation and Gui PCLMULQDQ implementation which were submitted to the NIST PQC standardization process (first round). The implementations of GeMSS have been corrected since the submission. The parameter D was mistakenly set to 512 in the implementation. Because the Frobenius map was not implemented in constant-time (Section 6.3), $D = 512$ allowed to save 27% of computations in the critical part of the signature generation (cf. Table 9.14). We refer to Section 6.2 about experimental platform and third libraries used.

scheme	(level, q, d_{ext}, D, , v, nb_ite)	key gen.	sign	verify
GeMSS128	(I, 2, 174, 513, 12, 12, 4)	118	1270	0.166
GeMSS192	(III, 2, 265, 513, 22, 20, 4)	549	3220	0.448
GeMSS256	(V, 2, 354, 513, 30, 33, 4)	1470	5380	1.17
FGeMSS(266) Inner.DualModeMS128	(I, 2, 266, 129, 10, 11, 1)	442	128	0.0964
Gui-184	(II, 2, 184, 33, 16, 16, 2)	732	24.9	0.0911
Gui-184 (updated)	(II, 2, 184, 33, 16, 16, 3)	744	74.3	0.134
Gui-312	(IV, 2, 312, 129, 24, 20, 2)	4860	781	0.298
Gui-448	(VI, 2, 448, 513, 32, 28, 2)	32600	19700	0.910

Table 7.37: Performance in megacycles of GeMSS and Gui best implementations submitted to the first round of the NIST PQC standardization process. We use a Skylake processor (LaptopS). The results have three significant digits.

GeMSS has much bigger public-key sizes and much slower signing processes than Rainbow (Table 9.28). In return, its signature size is slightly smaller. The public-key size is the main drawback of the MI-based cryptography, which gives an advantage to Rainbow. Therefore, NIST selected Rainbow as finalist candidate, whereas GeMSS is an alternate candidate. We note that new attacks would compromise the possible standardization of Rainbow and GeMSS.

scheme	(level, q, d_{ext}, D, , v, nb_ite)	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
QUARTZ	(80[1], 2, 103, 129, 3, 4, 4)	72.24[2]	3.733[3]	16	16
Gui-184	(II [4], 2, 184, 33, 16, 16, 2)	422.1[5]	14.98[5]	no seed	29[6]
Gui-184 (updated)	(II , 2, 184, 33, 16, 16, 3)				33[6]
Gui-312	(IV , 2, 312, 129, 24, 20, 2)	1990[5]	41.75[5]	no seed	47[6]
Gui-448	(VI , 2, 448, 513, 32, 28, 2)	5903[5]	94.76[5]	no seed	67[6]
GeMSS128	(I , 2, 174, 513, 12, 12, 4)	352.2	13.44	16	32.25
BlueGeMSS128	(I , 2, 175, 129, 13, 14, 4)	363.6	13.70		33.75
RedGeMSS128	(I , 2, 177, 17, 15, 15, 4)	375.2	13.10		35.25
WhiteGeMSS128	(I , 2, 175, 513, 12, 12, 3)	358.2	13.56		29.375
CyanGeMSS128	(I , 2, 177, 129, 14, 13, 3)	369.7	13.41		30.5
MagentaGeMSS128	(I , 2, 178, 17, 15, 15, 3)	381.5	13.22		31.625
GeMSS192	(III , 2, 265, 513, 22, 20, 4)	1238	34.07	24	51.375
BlueGeMSS192	(III , 2, 265, 129, 22, 23, 4)	1264	35.38		52.875
RedGeMSS192	(III , 2, 266, 17, 23, 25, 4)	1291	34.79		54.375
WhiteGeMSS192	(III , 2, 268, 513, 21, 21, 3)	1294	35.77		46.625
CyanGeMSS192	(III , 2, 270, 129, 23, 22, 3)	1321	35.26		47.75
MagentaGeMSS192	(III , 2, 271, 17, 24, 24, 3)	1348	34.69		48.875
GeMSS256	(V , 2, 354, 513, 30, 33, 4)	3041	75.89	32	72
BlueGeMSS256	(V , 2, 358, 129, 34, 32, 4)	3088	71.46		73.5
RedGeMSS256	(V , 2, 358, 17, 34, 35, 4)	3136	71.89		75
WhiteGeMSS256	(V , 2, 364, 513, 31, 29, 3)	3223	70.99		64.125
CyanGeMSS256	(V , 2, 364, 129, 31, 32, 3)	3272	73.20		65.25
MagentaGeMSS256	(V , 2, 366, 17, 33, 33, 3)	3322	70.41		66.375
FGeMSS(266)	(I , 2, 266, 129, 10, 11, 1)	1232	24.55	16	34.625
Inner.DualModeMS128					
FGeMSS(402)	(III , 2, 402, 640, 18, 18, 1)	4244	62.60	24	52.5
Inner.DualModeMS192	(III , 2, 402, 129, 18, 18, 1)		59.59		
FGeMSS(537)	(V , 2, 537, 1152, 25, 26, 1)	10161	122.7	32	70.375
Inner.DualModeMS256	(V , 2, 540, 129, 28, 26, 1)[7]	10270	116.3		70.75

[1] Original security level in bits, but QUARTZ was broken.
[2] The authors did not provide the exact size, and seemed to consider the linear terms of p. We compute the public-key size without these terms to be consistent with other schemes.
[3] The authors did not provide the exact size. However, they claimed to use 30497 random bits to generate the secret-key from the seed, when the LU decomposition is used. They considered $\mathcal{T}$ in $A^{-1}_{n_{var}}(\mathbb{F}_q)$, but in reality $\mathcal{T}$ is in $A^{-1}_{d_{ext}}(\mathbb{F}_q)$. So, the correct number is 29657 bits. We deduce the secret-key size by adding 210 bits, corresponding to the fact that the LU decomposition in $\mathbb{F}_2$ allows to save the random generation of the diagonal.
[4] The original security level was 128 bits, but the generic attack from Section 7.7.2 breaks it in 2^{112} evaluations.
[5] The sizes of [62, Table 2] are wrong. In particular, the public-key size contains the size of quadratic terms x_i^2. The sizes that we compute here are consistent with the practical implementation provided by [62] (by removing an extra useless byte from this implementation).
[6] The signature size is given without the 128-bit salt.
[7] The original scheme used $d_{ext} = 544$ and $= v = 32$. However, the correction of an error in [83, Table 13] implies $+ v = 54$ instead of 64. So, we update parameters according the philosophy of GeMSS.

Table 7.38: Size of the keys and signature of the HFEv--based schemes submitted to the NIST PQC standardization process, as well as QUARTZ from the NESSIE project [145].

round	scheme	(q, v_1, o_1, o_2)	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
	Rainbow-Ia	(16, 32, 32, 32)	152.1	100.2		64
	Rainbow-Ib	(31, 36, 28, 28)	151.6	106.2		78
	Rainbow-Ic	(256, 40, 24, 24)	192.2	143.4		104
	Rainbow-IIIb	(31, 64, 32, 48)	524.4	380.4		112
1	Rainbow-IIIc	(256, 68, 36, 36)	720.8	537.8	no seed	156
	Rainbow-IVa	(16, 56, 48, 48)	565.5	376.1		92
	Rainbow-Vc	(256, 92, 48, 48)	1724	1274		204
	Rainbow-VIa	(16, 76, 64, 64)	1351	892.1		118
	Rainbow-VIb	(31, 84, 56, 56)	1353	944.6		147
2	Rainbow-Ia	(16, 32, 32, 32)	149.0	92.99	no seed	64
	cyclic Rainbow-Ia		58.14		64[2]	
	Rainbow-Ic[1]	(256, 40, 24, 24)	188.0	132.6	no seed	104
	cyclic Rainbow-Ic[1]		58.50		64[2]	
	Rainbow-IIIc	(256, 68, 36, 36)	710.6	511.5	no seed	156
	cyclic Rainbow-IIIc		206.7		64[2]	
	Rainbow-Vc	(256, 92, 48, 48)	1706	1227	no seed	204
	cyclic Rainbow-Vc		491.9		64[2]	
3	Rainbow-Ia	(16, 36, 32, 32)	161.6	103.6	no seed	66
	CZ-Rainbow-Ia[3]		60.19		64[2]	
	Rainbow-IIIc	(256, 68, 32, 48)	882.1	626.0	no seed	164
	CZ-Rainbow-IIIc[3]		264.6		64[2]	
	Rainbow-Vc	(256, 96, 36, 64)	1931	1409	no seed	212
	CZ-Rainbow-Vc[3]		536.1		64[2]	

[1] `Rainbow-Ic` is not submitted to the second round of the NIST PQC standardization process. However, this implementation is available in the submitted implementation. We use this scheme in Section 8.6.3.
[2] The secret-key contains the 32-byte public seed. The latter is also stored in the public-key.
[3] `circumzenithal Rainbow`, new name of `cyclic Rainbow`.

Table 7.39: Size of the keys and signature of `Rainbow` submitted to the NIST PQC standardization process. We consider the evolution of these schemes during the three rounds.

Chapter 8

DualModeMS – a Dual Mode for Multivariate-Based Signatures

The purpose of this chapter is to present DualModeMS [83]; a multivariate-based signature scheme with a rather peculiar property. Its public-key is small whilst the signature is large. This is in sharp contrast with traditional multivariate signature schemes (Chapter 2) based on the so-called Matsumoto and Imai (MI) construction [119], such as QUARTZ [134] or Gui [144], that produce short signatures but have larger public-keys.

DualModeMS is based on the method proposed by A. Szepieniec, W. Beullens, and B. Preneel (SBP) in [157] (Chapter 3) who describe a generic technique permitting to transform any MI-based multivariate signature scheme into a new scheme with much shorter public-key but larger signatures. We emphasize that the technique from [157] can be viewed as a *modes of operations* that offers a new flexibility for MI-like signature schemes. Thus, DualModeMS is also useful for others multivariate-based signature candidates proposed to NIST.

DualModeMS is composed of two distinct layers. The first one (Inner.DualModeMS), that we shall call *inner layer*, is a classical MI-like multivariate scheme based on HFEv- (Section 2.4.1). The second part, *outer layer*, is the mode of operations specified in [157] (Chapter 3).

This submission is somewhat a complement to another multivariate-based signature scheme proposed to NIST: GeMSS [48] (Chapter 7). In particular, the security analysis for Inner.DualModeMS is largely similar to the one performed for GeMSS. In fact, Inner.DualModeMS is a re-parametrization of GeMSS imposed by a specificity of SBP [157].

The main sections of this chapter follow the algorithm specification and supporting documentation from the call for proposals [127].

8.1 General Algorithm Specification

8.1.1 Parameter Space

The main parameters involved in Inner.DualModeMS are:

- , the security level of Inner.DualModeMS (and DualModeMS),
- q, a prime or a power prime that is the order of the finite field $\mathbb{F}_q$,

- D, a positive integer that is the degree of a secret polynomial. D is such that $D = q^i$ for $i \geqslant 0$, or $D = q^i + q^j$ for $i, j \geqslant 0$,
- m, the number of equations in the public-key,
- n_{var}, the number of variables in the public-key,
- nb_ite = 1, the number of iterations in the signature and verification processes,
- d_{ext}, the degree of an extension field of $\mathbb{F}_q$,
- v, the number of vinegar variables (the number of variables in the public-key is $n_{var} = d_{ext} + v$),
- , the number of minus equations (the number of equations in the public-key is $m = d_{ext} -$).

Then, the main parameters involved in DualModeMS are:

- , the number of signatures of Inner.DualModeMS included in a final signature of DualModeMS,
- , the number of MQ polynomials included in a signature of DualModeMS,
- $\geqslant 1$, the size of an evaluation point set; must be a power of two,
- , the degree of the extension field for MAC polynomials (Algorithm 5); constrained by $q \geqslant$,
- , the number of Merkle paths that are opened during the signing and verifying processes,
- N, number of square-free monomials in n_{var} variables of degree $\leqslant 2$ (Equation (2.3) or (2.4)).

In Section 8.2, we specify precisely these parameters to achieve a security level $\in \{128, 192, 256\}$.

8.1.2 Cryptographic Operations

For the inner layer, we refer to Section 7.1. The keypair generation, signing process and verifying process are respectively described in Algorithms 29, 3 and 4. We set q to two and nb_ite to one.
For the outer layer, we refer to Chapter 3. The keypair generation, signing process and verifying process are respectively described in Algorithms 6, 7 and 11. H_1 is SHA-3 (we follow Section 7.4.13).

8.1.3 Implementation

For the inner layer, we refer to Section 7.4. For the outer layer, we have proposed new techniques since the submission of DualModeMS [83] to the NIST PQC standardization process. Two crucial operations are present in all cryptographic operations of the outer layer: the multipoint evaluation of MAC polynomials (Algorithm 5) and the computation of the digests for Merkle trees (Section 3.3). In Section 9.5, we improve the multipoint evaluations of MAC polynomials. Then, the computation of Merkle trees can be naturally parallelized. We use the hash functions as described in Section 7.4.13. In particular, we use a parallel implementation in AVX2 of the SHA-3 and SHAKE functions. These functions compute four hash values in parallel, and are based on the XKCP (Section 6.2.1). We note that the XKCP only provides an implementation of the core of SHA-3 and SHAKE (function KeccakP1600times4_PermuteAll_24rounds). So, we have implemented an interface allowing to use this implementation. A similar work can be found e.g. in the shake256-avx2 implementation of SPHINCS+ [105]. For DualModeMS, we have also specified our implementation for inputs of 4 bits. This further accelerates the implementation of Merkle trees, since each node of size 2 is the hash value of the concatenation of its two child nodes.

8.2 List of Parameter Sets

In `Inner.DualModeMS`, we consider $q = 2$. New schemes are proposed in Section 8.6, based on the inner layers `GeMSS` and `Rainbow`, as well as a variant of `Inner.DualModeMS` for $D = 17$.

8.2.1 Parameter Sets for a Security of 2^{128} (Level I)

We choose $D = 129$, $d_{\text{ext}} = 266$, $= 10$ and $v = 11$. This gives $m = 256$, $n_{\text{var}} = 277$ and $= 128$. The extension field is defined as $\mathbb{F}_{2^{d_{\text{ext}}}} = \mathbb{F}_2[\ ]/\ \ ^{d_{\text{ext}}} + \ ^{47} + 1$. For `Inner.DualModeMS128`, the public-key size is then 1232.128 kB and the signature size is 277 bits.

For the outer layer, we choose $= 2$, $= 64$, $= 2^{18}$, $= 21$, $= 18$ and $= 4$. The extension field is defined as $\mathbb{F}_2 = \mathbb{F}_2[\ ]/\ \ + \ ^{2} + 1$. For `DualModeMS128`, this gives a public-key of 512 bytes and a signature of 32.002 kB.

8.2.2 Parameter Sets for a Security of 2^{192} (Level III)

We choose $D = 129$, $d_{\text{ext}} = 402$, $= 18$ and $v = 18$. This gives $m = 384$, $n_{\text{var}} = 420$ and $= 192$. The extension field is defined as $\mathbb{F}_{2^{d_{\text{ext}}}} = \mathbb{F}_2[\ ]/\ \ ^{d_{\text{ext}}} + \ ^{171} + 1$. For `Inner.DualModeMS192`, this gives a public-key of 4243.728 kB and a signature of 420 bits.

For the outer layer, we choose $= 2$, $= 96$, $= 2^{18}$, $= 20$, $= 33$ and $= 5$. The extension field is defined as $\mathbb{F}_2 = \mathbb{F}_2[\ ]/\ \ + \ ^{3} + 1$. For `DualModeMS192`, this gives a public-key of 1536 bytes and a signature of 79.41475 kB.

8.2.3 Parameter Sets for a Security of 2^{256} (Level V), Version 1

We choose $D = 129$, $d_{\text{ext}} = 544$, $= 32$ and $v = 32$. This gives $m = 512$, $n_{\text{var}} = 576$ and $= 256$. The extension field is defined as $\mathbb{F}_{2^{d_{\text{ext}}}} = \mathbb{F}_2[\ ]/\ \ ^{d_{\text{ext}}} + \ ^{8} + \ ^{3} + \ + 1$. For `Inner.DualModeMS256`^{V1}, this gives a public-key of 10635.328 kB and a signature of 576 bits.

For the outer layer, we choose $= 1$, $= 256$, $= 2^{18}$, $= 20$, $= 52$ and $= 5$. The extension field is defined as $\mathbb{F}_2 = \mathbb{F}_2[\ ]/\ \ + \ ^{3} + 1$. For `DualModeMS256`^{V1}, this gives a public-key of 2048 bytes and a signature of 149.028125 kB.

Remark 15. In the final version of `MQsoft`, we use $\mathbb{F}_{2^{d_{\text{ext}}}} = \mathbb{F}_2[\ ]/\ \ ^{d_{\text{ext}}} + \ ^{128} + \ ^{3} + \ + 1$ to speed up the field modular reduction (Section 9.2.4).

8.2.4 Corrected Parameter Sets for a Security of 2^{256} (Level V)

The original `DualModeMS256` scheme used $d_{\text{ext}} = 544$ and $= v = 32$. However, the correction of an error in [83, Table 13] implies $- v = 54$ instead of 64. So, we update parameters following the methodology of Section 7.7 (Table 7.36). The lower number of monomials in the inner public-key allows to decrement (Equation (3.3)). We note that the security level of the original `DualModeMS256` scheme is not impacted by the error.

We choose $D = 129$, $d_{\text{ext}} = 540$, $= 28$ and $v = 26$. This gives $m = 512$, $n_{\text{var}} = 566$ and $= 256$. The extension field is defined as $\mathbb{F}_{2^{d_{\text{ext}}}} = \mathbb{F}_2[\ ]/\ \ ^{d_{\text{ext}}} + \ ^{9} + 1$. For `Inner.DualModeMS256`, this gives a public-key of 10269.568 kB and a signature of 566 bits.

For the outer layer, we choose $= 1$, $= 256$, $= 2^{18}$, $= 20$, $= 51$ and $= 5$. The extension field is defined as $\mathbb{F}_2 = \mathbb{F}_2[\ ]/\ \ + \ ^{3} + 1$. For `DualModeMS256`, this gives a public-key of 2048 bytes and a signature of 145.88175 kB.

8.3 Design Rationale

The main design rationale of DualModeMS is to propose SBP [157] (Chapter 3) as a mode of operations for multivariate schemes (Chapter 2). In order to demonstrate the drastic effect of SBP on public-key sizes, we tailored a specific inner multivariate-based scheme. Inner.DualModeMS is a HFEv--based scheme [133, 110, 134] since [157] identified that such a family is well suited in the context for SBP.

The design of Inner.DualModeMS follows from the analysis performed for GeMSS (Chapter 7), a HFEv--based scheme, proposed in [48]. The SBP imposes to have a scheme without the iterative procedure proposed in GeMSS (Sections 7.1.3 and 7.1.4). It is then rather natural (Equation (7.3)) to take the number of equations m equals the double of the security level of Inner.DualModeMS. We then use the methodology proposed in GeMSS (Section 7.7 and Tables 7.34, 7.35 and 7.36) to derive secure parameters. A signature of SBP requires generating many signatures from Inner.DualModeMS. This leads toward the choice of a small $D = 129$ to make the signature process of Inner.DualModeMS efficient. We detail these choices in Section 8.5.

8.4 Detailed Performance Analysis

8.4.1 Time

Here, we consider the parameters of DualModeMS128. The implementation submitted to NIST does not support DualModeMS192 and DualModeMS256 V1. We refer to Section 7.5 about experimental conditions. The only difference is that the time measurements are obtained on LaptopS with Turbo Boost enabled. Here are our performance results:

- For the reference implementation, we do not give measurements. We estimate that the keypair generation requires several days to be achieved.
- For the optimized implementation, the keypair generation takes 797 seconds, the time to sign is 2.31 seconds, and the verification takes 2.69 milliseconds.
- For the additional implementation, the keypair generation takes 552 seconds, the time to sign is 2.05 seconds, and the verification takes 2.84 milliseconds.

Since the submission of DualModeMS to the NIST PQC standardization process, we have run the additional implementation on LaptopS by disabling Turbo Boost. The keypair generation takes 1990 gigacycles (708 seconds), the time to sign is 7870 megacycles (2.80 seconds), and the verification takes 9.87 megacycles (3.51 milliseconds).

implementation	Turbo Boost	key gen.	sign	verify
optimized	enabled	797 s	2.31 s	2.69 ms
additional	enabled	552 s	2.05 s	2.84 ms
	disabled	708 s	2.80 s	3.51 ms

Table 8.1: Performance of DualModeMS128 at the first round of the NIST PQC standardization process. We use a Skylake processor (LaptopS).

8.4.2 Time (Updated)

In [84], we extended MQsoft (Chapter 9) to support DualModeMS192 and DualModeMS256^{V1}. We also improved the inner layer, which mainly impacts the performance of the dual signing process. We report the time measurements in Table 8.2. They are obtained in the conditions described in Section 6.2.

scheme	key gen.	sign	verify
DualModeMS128	1 900 000	5 530	10.0
DualModeMS192	6 860 000	18 900	18.1
DualModeMS256^{V1}	17 300 000	95 900	30.4

Table 8.2: Number of megacycles for each cryptographic operation of DualModeMS with the version of MQsoft used in [84]. We use a Skylake processor (LaptopS). Turbo Boost is disabled.

8.4.3 Time (Final Version)

Since the implementation used in [84], we have drastically improved the inner and outer layers. In Table 8.3, we report our best time measurements. For the signing process, the largest part of the speed-up is obtained by using the Frobenius map based on the modular composition (Section 9.3.5). Since d_{ext} is large compared to D, this method is very e cient. The remaining part of the speed-up is due to our new multipoint evaluation of MAC polynomials in random points (Section 9.5.3). This new evaluation also improves the verifying process. The keypair generation is drastically faster. This is due to the multipoint evaluation of MAC polynomials in structured points via the additive FFT (Section 9.5.2). Then, the parallel implementation of SHA-3 amplifies the obtained speed-up (Section 8.1.3). This implementation also improves the verifying process.

scheme	key gen.	sign	verify
DualModeMS128	3710 × 512	2800 × 1.97	0.643 × 15.6
DualModeMS192	6770 × 1010	8470 × 2.23	1.73 × 10.5
DualModeMS256	12700 × 1360	38000 × 2.53	3.95 × 7.69

Table 8.3: Number of megacycles (Mc) for each cryptographic operation of DualModeMS with our implementation, for a Skylake processor (LaptopS), followed by the speed-up between the Skylake implementation from [84] versus our implementation. For example, 3710 × 512 means a performance of 3710 Mc with our implementation, and a performance of 3710 × 512 = 1 900 000 Mc for [84]. Note that [84] is similar to the first round implementation of DualModeMS, except for the signing process which is faster.

8.4.4 Space

In Table 8.4, we present the theoretical sizes of public-key, secret-key and signature. The implementation does not optimize the size, so it explains the di erence with theoretical sizes. For the practical sizes of DualModeMS128, we have: public-key is 512 bytes, secret-key is 18038184 bytes, signatures are 32640 bytes. In Section 8.6.4, we propose a technique to decrease the secret-key size.

scheme	(, , , , log$_2$( ), ,)	|pk| (B)	|sk| (kB)	|sign| (kB)
DualModeMS128	(128,64,2,21,18,18,4)	512	18032.889625	32.002
DualModeMS192	(192,96,2,20,18,33,5)	1536	29466.09075	79.41475
DualModeMS256^{V1}	(256,256,1,20,18,52,5)	2048	44319.512	149.028125
DualModeMS256	(256,256,1,20,18,51,5)	2048	43819.936	145.88175

Table 8.4: Memory cost of DualModeMS. 1 kB is 1000 bytes.

8.5 Security and Selection of Parameters

In Section 7.7, we studied the security of the inner layer. In this part, we study the security of the outer layer. We obtain rules for selecting security parameters. In particular, we prove in Section 8.5.3 that the generic attack against the inner layer can be avoided by increasing the number of inner signatures, as if their generation was a Feistel–Patarin construction. In Sections 8.5.4 and 8.5.5, we study the trade-off between signature size, public-key size and size public-key plus signature.

8.5.1 Existential Unforgeability against Chosen Message Attack

We consider the EUF-CMA property of DualModeMS. Both fundamental theorems are derived from [157, 25]. Note that these theorems are more advanced than in Section 3.5.

Theorem 7. Let $= 1$ be the number of signatures of Inner.DualModeMS included in a signature of DualModeMS. If there is an adversary A against the EUF-CMA property of DualModeMS in time T with Q random oracle queries (respectively Q quantum random oracle queries) and with success probability then there exists an adversary B^A against the EUF-CMA property of Inner.DualModeMS in time $O(T)$ with success probability at least

$$- (Q+1)\cdot q^{-} - (Q+1)\cdot \frac{N - 1}{} - (Q+1)\cdot \frac{2 - 2}{2^2} \quad (8.1)$$

in the classical random oracle model, and respectively at least

$$- \quad Q+1 \;^2\cdot q^{-} \quad - \quad Q+1 \;^2\cdot \frac{N - 1}{} \quad - \quad Q+1 \;^2\cdot \frac{2 - 2}{2^2}$$

in the quantum random oracle model.

Here, we consider a-bit classical security level and $\frac{a}{2}$-bit quantum security level. Theorem 7 provides a guidance for choosing the various parameters involved in DualModeMS. The last term in (8.1) corresponds to the probability of finding a second preimage for one of the $(2 - 2)$ nodes of the Merkle tree. We generalize the original result of [157, 25] which have considered $= 0$. The third term in (8.1) is the probability that an invalid set of polynomials in $\mathbb{F}_q[x_1, \ldots, x_{n_{var}}]$ passes the test of the dual verifying process (Algorithm 11). Thus, we need to choose the parameters such that:

$$\frac{N - 1}{} \leqslant 2^{-} . \quad (8.2)$$

When $= 1$, Theorem 7 implies $\cdot \log_2(q) \geqslant$. Otherwise, Theorem 7 is meaningless. The security of the SBP transform relies then on a new hard problem, so-called Approximate PoSSo (APoSSo) problem (Section 4.5), that is defined below.

Problem 4. Approximate PoSSo (APoSSo$(q, \ , m, n_{var}, D, r)$). Let $q, m, n_{var}, D,$ and r be non-negative integers such that $r < \min(\ , m)$. Given p a degree-D multivariate polynomial system in $\mathbb{F}_q[x_1, \ldots, x_{n_{var}}]^m$ and $y_1, \ldots, y$ in $\mathbb{F}_q^m$, the problem is to find vectors $x_1, \ldots, x$ in $\mathbb{F}_q^{n_{var}}$ such that the dimension of the vector space generated by $p(x_1) - y_1, \ldots, p(x \ ) - y$ is less or equal to r.

Under the assumption that APoSSo is hard for a fixed and r, we obtain Theorem 8.

Theorem 8. If there is an adversary $\mathcal{A}$ against the EUF-CMA property of DualModeMS in time T with Q random oracle queries (respectively Q quantum random oracle queries) and with success probability , and if APoSSo$(q, \ , m, n_{var}, D, r)$ is hard, then there exists an adversary $\mathcal{B}^{\mathcal{A}}$ against the EUF-CMA property of Inner.DualModeMS in time $O(T)$ with success probability at least

$$- (Q + 1) \cdot q^{- \ \cdot (r+1)} - (Q + 1) \cdot \frac{N \ - 1}{} - (Q + 1) \cdot \frac{2 \ - 2}{2^2}$$

in the classical random oracle model, and respectively at least

$$- \left(Q + 1 \right)^2 \cdot q^{- \ \cdot (r+1)} - \left(Q + 1 \right)^2 \cdot \frac{N \ - 1}{} - \left(Q + 1 \right)^2 \cdot \frac{2 \ - 2}{2^2}$$

in the quantum random oracle model.

Thus, we need to choose $> r$ and $m > r$ such that APoSSo is hard for small values of $r > 0$, i.e:

$$r < \frac{}{\cdot \log_2(q)} . \qquad (8.3)$$

The hardness of APoSSo is discussed in Chapter 10. Some classical attacks were presented in Section 4.5. In Section 8.5.2, we choose and for practical parameters.

8.5.2 Approximate PoSSo and Selection of Parameters

For > 1, we need to assume the hardness of APoSSo for r respecting Equation (8.3). Indeed, if the best attack against APoSSo is the exhaustive search from Section 4.5, then we repeat this attack on average $q^{\ \cdot r}$ times to break the dual mode, requiring at least $q^{m + (\ -1) \cdot r}$ evaluations of p (or $q^{\frac{1}{2} \cdot (m + (\ -1) \cdot r)}$ for the quantum exhaustive search). This attack is more expensive than to break the inner layer. Therefore, the initial security is not impacted.

For our parameters, equals 1 or 2, which minimizes the signature size. Precisely, is strictly bounded by equals 64, 96 and 256 respectively for the three parameter sets proposed in Section 8.2. We make the assumption that APoSSo is hard for $= \frac{}{\cdot \log_2(q)}$. It can also be mentioned that APoSSo is related to the so-called Generalized MinRank ([80], Problem 7). Given a matrix whose coefficients are multivariate polynomials, the goal is to find an assignment of the variables that makes the rank of the matrix smaller than a given rank. Thus, we have a problem which is in some sense harder than the Kipnis–Shamir attack described in Section 4.6.3. APoSSo is Generalized MinRank with $\cdot n_{var}$ variables, a matrix of size $\times m$ and a target rank r (Chapter 10). [80] provides then the degree of regularity for solving the corresponding determinantal system.

8.5.3 Safe Extension of the Dual Mode for Vulnerable Inner Layers

The Feistel–Patarin construction (Section 4.2) is not compatible with the SBP transform. So, we designed `DualModeMS` without using such a construction, i.e., nb_ite = 1 in the inner layer. This implies $m \cdot \log_2(q) \geqslant 2$. In this section, we show that by slightly increasing , we can apply the SBP transform on `GeMSS` by setting nb_ite to four in the inner layer. The main idea is to remark that a generic attack on signatures is similar to an attack on nb_ite signatures. If both attacks are identical, then by taking + nb_ite − 1 inner signatures, an adversary can break at most nb_ite − 1 of them and we obtain the original security. In practice, the forgery of nb_ite inner signatures among + nb_ite − 1 is slightly easier than among nb_ite inner signatures. We introduce Theorem 9 to evaluate the cost of this attack. Then, we obtain by Corollary 1 that generating + nb_ite inner signatures instead of seems enough to be immune against this new generic attack.

Theorem 9. Let $x \in [\![1, \ ]\!]$ be an integer, and $P = \frac{}{q^m} \in \ 0, 1$. If the inner layer is a `MI`-based signature scheme (without the Feistel–Patarin construction, and during the verifying process, the evaluation of p has to be compared to a unique digest for a fixed document), and if is negligible against P^{-1}, then at least x inner signatures of the outer layer can be forged in approximately

$$= q^{\frac{x}{x+1} \cdot m} \cdot \binom{ -1}{x-1}^{-\frac{1}{x+1}} \qquad (8.4)$$

evaluations of the inner public-key, with a memory cost of $(m + n_{\mathtt{var}}) \log_2(q)$ bits. The number of digests required is $+ \cdot P \cdot \left(-x+1+(x-2)\left(1-(1-P)\right)^{-x+1}\right) \leqslant + \cdot P \cdot (-1)$. When is not negligible against P^{-1}, Equation (8.4) is only a lower bound on the cost of this generic attack. Note that the generic attack is not completely balanced since the number of digests is greater than

Proof. The proof is similar to this of Lemma 4. Here, we extend the generic attack for an adversary forging $x \leqslant$ inner signatures instead of $x =$.

Assume that an adversary uses a generic attack to generate an outer signature such that x among the inner signatures are valid. Then, he can proceed as follows. He starts by building an inversion table of elements. Since the hash function is modelized as a random oracle, this implies that the probability of inverting the public-key is $P = \frac{}{q^m}$. Then, he iterates the following process. He chooses a document, and tries to sign it as one of the inner signatures. If he fails, the document is changed. Else, the adversary knows one valid inner signature. Thus, he keeps the current document, and tries to forge at least $x-1$ inner signatures from the -1 remaining signatures. Since the hash function is modelized as a random oracle (note that once the first signature is found, the other digests are fixed and independent), this event occurs with probability $\sum_{i=x-1}^{-1} \binom{-1}{i} P^i (1-P)^{-1-i}$. We can write this probability in the form $P^{x-1} \cdot \binom{-1}{x-1} \cdot S$, for

$$S = \sum_{i=0}^{-x} \binom{x-1+i}{i}^{-1} \cdot \binom{-x}{i} \cdot P^i \cdot (1-P)^{-x-i} \leqslant \left(P + (1-P)\right)^{-x}.$$

So, this event has to be repeated $\left(P^{x-1} \cdot \binom{-1}{x-1} \cdot S\right)^{-1}$ times on average to succeed at least $x-1$ forgeries. Now, we balance the cost of the inversion table and the cost of forging at least x inner signatures, by solving:

$$= \left(\frac{q^m}{}\right)^{x-1} \cdot \binom{-1}{x-1}^{-1} \cdot S^{-1} \cdot \left(\frac{q^m}{} + -x+1+(x-2)\left(1-(1-P)\right)^{-x+1}\right). \qquad (8.5)$$

Here, we consider that if the adversary fails to forge at least one signature among $-x+1$, then he does not try to forge the $x-2$ remaining signatures. We multiply by ^{x} in Equation (8.5), then we apply the $(x+1)$-th root. Since we assume that is negligible against P^{-1}, we obtain approximately:

$$= q^{\frac{x}{x+1}\cdot m}\cdot \binom{-1}{x-1}^{-\frac{1}{x+1}}\cdot S^{-\frac{1}{x+1}},$$

which coincides with the announced result when $S\approx 1$. For $x=1$ and $x=\ $, $S=1$ and we obtain the cost of the standard generic attack (Section 4.2).

Now, we show that S is close to 1. We lower bound $S^{\frac{1}{x+1}}$ by $(1-P)^{\frac{-x}{x+1}}\geqslant 1-s$, for $s=\ \frac{-x}{x+1}\cdot P$. We obtain that:

$$1\leqslant S^{-\frac{1}{x+1}}\leqslant 1+\frac{s}{1-s}.$$

Therefore, $S^{-\frac{1}{x+1}}-1\leqslant\ $ for $\ >0$. Thus, we have that if $s<\ $ is negligible against P^{-1}, then $S^{-\frac{1}{x+1}}$ is close to 1. □

Corollary 1. Let $x\geqslant 1$ be the smallest integer such that

$$q^{\frac{x}{x+1}\cdot m}\cdot \binom{+x-2}{x-1}^{-\frac{1}{x+1}}\geqslant 2\ . \tag{8.6}$$

If the inner layer is a MI-based signature scheme, having a -bit security level except against the generic attack, and if Theorem 9 describes the best generic attack against the outer layer, then the SBP transform is safe by taking $+x-1$ inner signatures, where is the number of required signatures for a safe outer layer based on a safe inner layer (including a safe choice of nb_ite).

Proof. Assume that an adversary wants to generate at least $y\geqslant 1$ valid inner signatures among the $+x-1$ inner signatures. Since the inner layer is safe except against the generic attack, the adversary has to use the generic attack. We assume that Theorem 9 provides the best generic attack against the outer layer. By applying Equation (8.4) with y instead of x and $+y-1$ instead of , we obtain the left part of Equation (8.6) (with y instead of x). This implies that if $y\geqslant x$, the attack necessarily fails. If y is chosen strictly smaller than x, the generic attack can be performed. Then, the y forged signatures only impact the probability of passing the verification of $+x-1$ inner signatures with the derived public-key (cf. Figure 3.2). By removing these y signatures, we obtain a dual mode having $+x-1-y\geqslant\ $ inner signatures, immune against generic attacks, where the document is fixed. Moreover, directly solving APoSSo$(q,\ +x-1,m,n_{\mathtt{var}},D,r)$ is harder than solving APoSSo$(q,\ ,m,n_{\mathtt{var}},D,r)$ for $r<\ $. Therefore, we obtain a reduction to the original security level of the outer layer (based on a safe inner layer). □

Corollary 1 allows to apply the SBP transform on signature schemes having a smaller public-key, as GeMSS (Section 8.6.2). This improves drastically the performance of the dual mode. Indeed, by taking a smaller m, we decrease $n_{\mathtt{var}}$ and so N, which is quadratic in $n_{\mathtt{var}}$. This directly decreases the size of inner signatures, polynomials from the derived public-key and evaluations of $\hat{p}$, as well as the probability that an invalid set of polynomials passes the PIT (Equation (8.2)). So, can be chosen smaller for fixed and . Smaller parameters imply faster cryptographic operations.

However, Corollary 1 cannot be coupled to the use of salt (Section 7.6.2) without drastically increasing x or m. Theorem 9 requires the verification of a unique digest for a fixed document,

whereas the concatenation of the document and a (variable) salt generates a large number of possible digests. This would imply a standard generic attack (Section 4.2) independent of each inner signature, requiring evaluations and hash values for = $\sqrt{x} \cdot q^{\frac{1}{2} \cdot m}$. Corollary 1 could be used by replacing Equation (8.6) by $\geqslant 2$, in particular for small values of x (e.g. q or q^2).

8.5.4 Minimizing the Size Public-Key Plus Signature

Initially, the authors of [157] took the Merkle root as a public-key. Then, they proposed to decrease the size of the signature by taking shorter authentication paths, but by multiplying by 2 the public-key size. This process is described in Section 3.3. In this part, we show two options for minimizing the size public-key plus signature. We can choose minimizing it (option 1). Then we can modify this option to decrease the public-key size (option 2). For the first option, we introduce Lemma 6.

Lemma 6. $= \lceil \log_2( ) \rceil \in \mathbb{N}$ minimizes the size public-key plus signature of the outer layer.

Proof. On the one hand, the public-key contains 2 digests. On the other hand, the signature contains $\in \mathbb{N}^*$ authentication paths of $\log_2( ) -$ digests. The value of which minimizes the size of both is given by the minimum of $f( ) = 2 -$. We compute it by looking when the derivative of f is equal to zero, i.e. we solve:

$$\ln(2) \cdot 2 - = 0.$$

Let $u \in \mathbb{R}_+^*$ be the solution of the previous equation. We have that $\lfloor \log_2( ) \rfloor < u < \lceil \log_2( ) \rceil + 1$. Moreover, f is continuous, decreasing before u and increasing after. So, we have three integers which could minimize f. We conclude this proof by comparing $f\lfloor \log_2( ) \rfloor$, $f\lceil \log_2( ) \rceil$ and $f\lceil \log_2( ) \rceil + 1$. We have:

$$f\lceil \log_2( ) \rceil = f\lfloor \log_2( ) \rfloor + 2^{\lfloor \log_2( ) \rfloor} - ,$$
$$f\lceil \log_2( ) \rceil + 1 = f\lceil \log_2( ) \rceil + 2^{\lceil \log_2( ) \rceil} - .$$

We know that $2^{\lfloor \log_2( ) \rfloor} \leqslant$ and $2^{\lceil \log_2( ) \rceil} \geqslant$, so $= \lceil \log_2( ) \rceil$ is the integer which minimizes f. □

For the second option, we propose a technique which allows to decrease the public-key size with a slight impact on the size public-key plus signature, when $1 \leqslant \lceil \log_2( ) \rceil$. Let $'$ be an integer in $[\![0, -1]\!]$. We propose to add the public-key in the signature, then take $2^{'}$ nodes of the Merkle tree as a new public-key. Compared to the possibility to take $'$, the user traces shorter Merkle paths. In return, he verifies that the old public-key of 2 nodes leads to the $2^{'}$ nodes of the new public-key. With this process, we remark that only impacts the signature size. So, we can take the value of which minimizes the size of the new signature. As explained previously (Lemma 6), this value is $\lceil \log_2( ) \rceil$. For the new public-key of size $2^{'} \cdot 2$ bits, we can minimize its size by setting $' = 0$. However, someone who would also wish to accelerate the verifying process should choose $' > 0$.

With the first option, when we minimize the size public-key plus signature (i.e. $= \lceil \log_2( ) \rceil$), the latter is:

$$\cdot |\text{origin.sm}| + (\cdot N + \cdot m \cdot) \cdot \log_2(q) + 2^{\lceil \log_2( ) \rceil} + \cdot \log_2( ) - \lceil \log_2( ) \rceil \cdot 2 \text{ bits.} \quad (8.7)$$

With the second option, we can choose to have a smaller public-key of size $2^{'} \cdot 2$ bits. In this case, we keep $= \lceil \log_2( ) \rceil$, and we just modify the verifying process by checking if the 2-element Merkle floor leads to the new public-key. The signature size is also given by Equation (8.7).

8.5.5 Smaller Signatures

During the outer signing process, we remark that certain nodes from the Merkle tree appear several times, in particular for the nodes close to the root. These nodes are redundant. During the outer verifying process, other nodes are redundant because we compute them thanks to the evaluations.
The first traced paths provide useful nodes for verifying other evaluations. Each redundant node allows to speed up the verification, since redundant nodes should be computed only one time. The outer cryptographic operations could be easily modified to speed up the verifying process and minimize the number of necessary digests. Lemma 7 provides the largest signature size with this consideration.

Lemma 7 (Maximum number of nodes required to verify the Merkle leaves).
Let > 1 be a power of two, $\in [\![1, \ ]\!]$, $x \in [\![1, \frac{}{2}]\!]$ and $y \in [\![0, x]\!]$ be integers such that $= x + y$.
If distinct nodes in the largest floor of a Merkle tree of size $2 - 1$ are known, if y is the number of known couples of sibling nodes in this floor, and if the root of the Merkle tree is known, then the maximum number of new nodes z that requires the verification of the nodes by tracing their corresponding path until the root is less or equal to $\frac{}{2} - y$.

Proof. We use a proof by induction on $\log_2(\ ) \geqslant 1$. For $= 2$, we have two possibilities. If $= 1$, then $y = 0$ and the verification requires the sibling node. Else, $= 2$ and we know a node and its sibling node, so $y = 1$ and no new node is required. In both cases, $z = \frac{}{2} - y$.

Now, we assume that Lemma 7 is true for a tree of size $2 - 1$, and we consider a tree of size $4 - 1$. We study z in function of the -element floor. Let $x \in [\![1, \ ]\!]$ be the number of distinct nodes of this floor that we have to compute for the verification of the nodes. This implies that the 2 -element floor has $= x + y$ known nodes for $y \in [\![0, x]\!]$, and requires storing $x - y$ new nodes for this floor. Note that y is the number of known couples of sibling nodes in the 2 -element floor. Let $x' \in [\![1, \frac{}{2}]\!]$ and $y' \in [\![0, x']\!]$ be integers such that $x = x' + y'$. By induction hypothesis, we obtain $z \leqslant \frac{}{2} - y' + x - y = \frac{}{2} + x' - y$. Thus, $z \leqslant \ - y$. □

For any outer signature, Lemma 7 allows to decrease the number of digests corresponding to the $n + 1$ smallest floors (which correspond to a tree of size $2^{n} - 1$) of the Merkle tree. For $< n$, we can replace $\cdot (n - \ )$ digests by at most 2^{n-1} digests in the outer signature. The values of n minimizing the signature size are $\min \lfloor \log_2(\ ) \rfloor + 2, \log_2(\ )$ and $\min \lceil \log_2(\ ) \rceil + 1, \log_2(\ )$. For $n = \lceil \log_2(\ ) \rceil + 1 \leqslant \log_2(\ )$, we obtain the following signature size:

$$\cdot |\text{origin.sm}| + (\ \cdot N + \ \cdot m \cdot \ ) \cdot \log_2(q) + \ 2^{\lceil \log_2(\ ) \rceil} + \ \cdot \ \log_2(\ ) - \lceil \log_2(\ ) \rceil - 1 \ \cdot 2 \ \text{bits.} \quad (8.8)$$

In this case, Lemma 7 saves at least $- 2$ digests in the size public-key plus signature compared to the first option, and saves at least digests in the outer signatures compared to the second option (defined in Section 8.5.4). The size of the public-key is 2 digests for any $\in [\![0, \lceil \log_2(\ ) \rceil]\!]$. We can choose to simplify the cryptographic operations by removing redundant digests only from the $\min \lceil \log_2(\ ) \rceil + 1, \log_2(\ ) - \ - 1$ smallest floors (the $+ 1$ smallest floors are removed), or we can remove all redundant digests for decreasing the signature size on average. In both cases, the signature size is upper bounded by Equation (8.8). The verifying process requires computing at most $2^{\lceil \log_2(\ ) \rceil} - 2 + \ \cdot \ \log_2(\ ) - \lceil \log_2(\ ) \rceil$ digests from 4 -bit sequences.

8.6 Design

In [83] (Section 8.2), we proposed three sets of parameters for `DualModeMS`. In this section, we propose new parameters:

- In Section 8.6.1, we set D to 17 to accelerate the signing process. We obtain `RedDualModeMS`.
- In Section 8.6.2, we apply the technique from Section 8.5.3. We remark that for our parameters, if nb_ite > 1 is chosen to avoid the generic attack against the inner layer, then we can set nb_ite $= 1$ if we take $+$ nb_ite inner signatures instead of . Therefore, we can apply the SBP transform to `GeMSS` and improve drastically both signature size and performance.
- Then, we propose security parameters for the dual mode of `Rainbow` in Section 8.6.3.
- Finally, we introduce a new parameter to control the trade-o between secret-key size and performance of the signing process in Section 8.6.4.

In this part, we compress the inner secret-key of `HFEv-`-based schemes from a seed, as described in Section 7.4.1. We use the technique from Section 8.5.5 which minimizes the size public-key plus signature. We reach a public-key size of 2 bits by setting to zero. The signature size is given by Equation (8.8). Performance measurements are available in Sections 8.7 and 9.6.3.

8.6.1 `RedDualModeMS`

In Section 7.8.2, we proposed a red version of `GeMSS`. The idea is to set D to 17 to speed up the signing process. We propose to apply this idea on `DualModeMS`. The obtained scheme is called `RedDualModeMS`, and is described in Table 8.5. The parameters of `Inner.RedDualModeMS` are extracted from Tables 7.34, 7.35 and 7.36. For $= 18$, the field polynomial defining $\mathbb{F}_2$ is the degree- 9-ESP (i.e. $^{18} + {}^{9} + 1$), but could also be the degree- AOP (Section 5.3.4).

inner layer	$(, D, d_{\text{ext}}, , v, \text{nb_ite})$	|pk| (MB)	|sk| (B)	|sign| (bits)
`Inner.DualModeMS128`	(128,129,266,10,11,1)	1.23	16	277
`Inner.RedDualModeMS128`	(128,17,268,12,12,1)	1.26		280
`Inner.DualModeMS192`	(192,129,402,18,18,1)	4.24	24	420
`Inner.RedDualModeMS192`	(192,17,404,20,19,1)	4.30		423
`Inner.DualModeMS256`	(256,129,540,28,26,1)	10.3	32	566
`Inner.RedDualModeMS256`	(256,17,540,28,29,1)	10.4		569

outer layer	$(, , , , \log_2( ), ,)$	|pk| (B)	|sk| (MB)	|sign| (kB)
`DualModeMS128`	(128,64,2,21,18,18,0)	32	18.0	31.9
`RedDualModeMS128`	(128,128,1,18,18,19,0)		18.0	28.7
`DualModeMS192`	(192,96,2,20,18,33,0)	48	29.4	79.3
`RedDualModeMS192`	(192,192,1,18,18,34,0)		29.5	71.8
`DualModeMS256`	(256,256,1,20,18,51,0)	64	43.8	143
`RedDualModeMS256`	(256,256,1,18,18,53,0)		43.9	141

Table 8.5: Size of the keys and signature of the inner and dual modes. We compress the inner secret-key from a seed (Section 7.4.1).

8.6.2 Dual GeMSS

Here, we apply the idea from Section 8.5.3 on GeMSS. We choose nb_ite > 1 providing a safe inner layer, then we remove the Feistel–Patarin construction in the inner layer when we use it in the outer layer. For our practical values of , we obtain that when = −, x = nb_ite + 1 satisfies Corollary 1. Thus, we obtain a very e cient dual mode of GeMSS (Table 8.6). The use of the dual mode of GeMSS allows a faster signing process than DualModeMS (since $d_{\mathtt{ext}}$ is smaller), as well as smaller signature sizes (since and $n_{\mathtt{var}}$ are smaller). The signature sizes of Dual RedGeMSS are between 1.38 and 1.51 times smaller than these of RedDualModeMS. For = 19, the extension field is defined as $\mathbb{F}_2 = \mathbb{F}_2[\ ]/(\ + \ ^6 + \ ^5 + \ + 1)$. For = 18, we refer to Section 8.6.1.

scheme	(, , , , log$_2$( ), ,)	|pk| (B)	|sk| (MB)	|sign| (kB)
Dual BlueGeMSS128	(128, 68, 2, 18, 18, 16, 0)	32	17.1	20.2
Dual RedGeMSS128	(128, 68, 2, 19, 18, 16, 0)		17.2	20.7
Dual BlueGeMSS192	(192, 196, 1, 18, 18, 29, 0)	48	26.4	49.5
Dual RedGeMSS192			26.5	49.7
Dual BlueGeMSS256	(256, 260, 1, 18, 18, 44, 0)	64	36.6	93.5
Dual RedGeMSS256			36.7	93.8

Table 8.6: Size of the keys and signature of the dual mode of BlueGeMSS and RedGeMSS. We remove the Feistel–Patarin construction in the inner layer.

8.6.3 Dual Rainbow

In Table 8.7, we propose new parameters for the dual mode of Rainbow (Section 2.4.3), called Dual Rainbow. The parameters proposed in [25] (Section 3.6) target a 128-bit quantum security level (and a 192-bit classical security level), whereas we consider a $\frac{}{2}$-bit quantum security level. We have considered the Rainbow schemes [64] submitted to the second round of the NIST PQC standardization process, as well as Rainbow-Ic which is supported by the second round implementation. This choice allows to study the impact of on the dual mode of Rainbow. The parameters and sizes of the inner layer of Rainbow are described in Table 7.39 (Section 7.10). As in [25], we could also propose the dual mode of Cyclic Rainbow to reduce the size of the outer signature (Section 3.6). This possibility is not supported by MQsoft (Chapter 9). In Section 9.6.3, we present the performance of Dual Rainbow with MQsoft.

We define $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ as in Section 7.4.9. For = 5, the extension field is defined as $\mathbb{F}_{16} = \mathbb{F}_{16}[\ ]/(\ + \ ^2 + 1)$. For = 3, the extension field is defined as $\mathbb{F}_{256} = \mathbb{F}_{256}[\ ]/(\ + \ + 1)$.

scheme	(, , , , log$_2$( ), ,)	|pk| (B)	|sk| (MB)	|sign| (kB)
Dual Rainbow-Ia	(128, 32, 1, 5, 18, 16, 0)	32	17.0	14.1
Dual Rainbow-Ic	(128, 16, 1, 3, 18, 17, 0)	32	17.1	15.6
Dual Rainbow-IIIc	(192, 24, 1, 3, 18, 31, 0)	48	26.4	39.7
Dual Rainbow-Vc	(256, 32, 1, 3, 18, 47, 0)	64	36.5	75.0

Table 8.7: Size of the keys and signature of the dual mode of Rainbow.

8.6.4 Performance with a Smaller Secret-Key

In this section, we introduce $\in \mathbb{N}$, a new parameter corresponding to the number of floors, from the leaves of the Merkle tree, that we remove from the secret-key. This divides the size of the tree (approximately) by 2 . In return, we generate again these floors during the signing process. So, this process only impacts the signing process: 2 evaluations of m MAC polynomials, $\cdot(2 \;-1)$ digests from $(m \;\log_2(q))$-bit sequences and $\cdot(2 \;-1-\;)$ digests from 4 -bit sequences are required. For small values of , we obtain interesting trade-o s between size of secret-key and performance of the signing process (Table 8.8).

		0	1	2	3	4	5	6
DualModeMS128	secret-key (MB)	18.0	9.62	5.43	3.33	2.28	1.76	1.49
	signing process	1.00	1.00	1.00	1.01	1.02	1.04	1.09
RedDualModeMS128	secret-key (MB)	18.0	9.65	5.45	3.36	2.31	1.78	1.52
	signing process	1.00	1.05	1.14	1.33	1.67	2.38	3.80
Dual Rainbow-Ia	secret-key (MB)	17.0	8.63	4.44	2.34	1.29	0.766	0.504
	signing process	1.00	1.21	1.63	2.43	4.03	7.21	13.6
Dual Rainbow-Ic	secret-key (MB)	17.1	8.71	4.51	2.42	1.37	0.845	0.583
	signing process	1.00	1.27	1.81	2.85	5.00	9.22	17.7

Table 8.8: Performance of the dual mode in function of . For the signing process, we give the slow-down factor compared to equals 0. We use a Skylake processor (LaptopS), with PCLMULQDQ and the AVX2 instruction set. Turbo Boost is not used.

Remark 16. The inner secret-key of Rainbow can be generated from a small seed (Section 7.4.1), permitting to decrease the size of the dual secret-key. As for the HFE-based schemes, the secret-key can be decompressed one time for inner signatures.

8.7 Comparison of **DualModeMS** to Other Signature Schemes

In Table 8.9, we compare DualModeMS and Dual Rainbow to the other second round signature schemes of the NIST PQC standardization process. GeMSS, picnic and SPHINCS$^+$ are currently alternate candidates of the third round, whereas MQDSS [55] was not selected for the third round. Clearly, Dual Rainbow is very competitive compared to the other second round schemes. It has the fastest verifying process, and the signing process is one of the fastest. The compromise between speed and signature size seems optimal. Moreover, the inner layer of Dual Rainbow is currently a finalist candidate of the third round (except Rainbow-Ic that we take from the first round). DualModeMS would be competitive by using the dual mode of GeMSS as proposed in Section 8.6.2. The secret-key sizes are very large, but can be drastically reduced by setting > 0 (Section 8.6.4).

scheme	key gen.	sign	verify	|pk| (B)	|sk|	|sign| (kB)
DualModeMS128	3740	2800	0.684	32	18 MB	31.9
RedDualModeMS128	3790	136	0.801	32	18 MB	28.7
Dual Rainbow-Ia	998	3.20	0.296	32	17 MB	14.1
Dual Rainbow-Ic	995	5.54	0.338	32	17.1 MB	15.6
picnic-L1-FS	0.0170	5.87	4.61	32	16 B[1]	34.0
picnic-L1-UR	0.0170	7.03	5.66	32	16 B[1]	54.0
picnic2-L1-FS	0.0167	254	115	32	16 B[1]	13.8
SPHINCS$^+$-SHAKE256-128s-s	141	2260	4.53	32	32 B[1]	8.08
SPHINCS$^+$-SHAKE256-128s-r	276	4100	8.40	32	32 B[1]	8.08
SPHINCS$^+$-SHAKE256-128f-s	4.47	145	10.0	32	32 B[1]	17.0
SPHINCS$^+$-SHAKE256-128f-r	8.83	270	20.4	32	32 B[1]	17.0
MQDSS-31-48[2]	0.916	4.49	2.80	46	16 B	28.4
PKP-DSS-128	0.0775	2.67	1.01	57	16 B	20.9

[1] The public-key is used during the signing process, but its size is not included in the secret-key size.
[2] Version 2.1.

Table 8.9: Comparison of the dual mode to the second round candidates (except PKP-DSS [23]), for a 128-bit security level. Here, the dual mode minimizes the size of the public-key (cf. Section 8.5.5). The cryptographic operations are measured in megacycles. We use a Skylake processor (LaptopS), with PCLMULQDQ and the AVX2 instruction set. Turbo Boost is not used.

8.8 Advantages and Limitations

The SBP construction allows to greatly decrease the public-key size of Inner.DualModeMS and Rainbow. In return, the signature size is larger, but much smaller than the inner public-key size. The secret-key size is much bigger, but can be reduced. The time to sign/verify is larger, but this provides very interesting trade-offs for multivariate schemes. The obtained schemes are very competitive with the NIST candidates. Dual Rainbow is faster than DualModeMS and has smaller signature sizes, but the use of Dual GeMSS allows to reduce this gap. Finally, the SBP construction can be applied to other categories of signature schemes [25] such as code-based schemes and lattices-based schemes.

Chapter 9

MQsoft – a Fast Multivariate Cryptography Library

Here, we present software tools [84] that allow the efficient implementation of HFE-based schemes (using arithmetic in $\mathbb{F}_{2^{d_{ext}}}$). In particular, our software tools allow to speed up GeMSS [50] (Chapter 7), Gui [62] and DualModeMS [83] (Chapter 8) signature schemes, which are candidates submitted to the NIST post-quantum cryptography standardization process [128]. The advantage of $\mathbb{F}_{2^{d_{ext}}}$ is that each element can be represented as a vector of bits, which corresponds to the architecture of binary computers and can be naturally improved by vector instructions (Section 6.1.3). The signature generation requires arithmetic in $\mathbb{F}_{2^{d_{ext}}}[X]$, and its implementation is already provided by various libraries. Among the best, NTL [153] (Section 6.2.1) provides high-quality implementations of state-of-the-art algorithms. But these algorithms are not dedicated to sparse polynomials. Moreover, the implementations are not constant-time (Section 6.3) and so are vulnerable to timing attacks (Section 4.7). For these reasons, we need to adapt the algorithms used. We have chosen to create a new library, which is based on constant-time arithmetic in $\mathbb{F}_{2^{d_{ext}}}$. Unlike NTL, which offers a general implementation, the value of d_{ext} can be fixed in the code of our library, allowing a more efficient arithmetic. Moreover, we exploit the sparse polynomial structure to improve the performance. More generally, our implementation uses the Intel vector instructions (Section 6.1.3) to obtain interesting speed-ups.

Our library supports DualModeMS [83] (Chapter 8), which is one of the candidates of the NIST PQC standardization process. By improving the implementation of HFE-based schemes, we automatically improve the implementation of DualModeMS. The main results of this chapter were published in [84]. Compared to [84], we propose crucial improvements. Firstly, we improve our implementation of polynomial multiplication (Section 9.3.2), in order to add an efficient modular composition (Section 9.3.4), used to speed up the Frobenius map (Section 9.3.5). Secondly, we implement the constant-time GCD of [19] (Section 9.3.6). Thirdly, the performance results of GeMSS (Section 9.6.1) are based on an efficient keypair generation via evaluation-interpolation (Section 7.4.7). Fourth, we propose more implementations of arithmetic in $\mathbb{F}_{2^{d_{ext}}}$, including Haswell processors. Finally, we introduce new algorithms (Sections 9.4.5 and 9.5) to improve the performance of all cryptographic operations of DualModeMS and Dual Rainbow (Section 8.6.3).

Evaluation of multivariate quadratic systems. Many multivariate cryptosystems (Chapter 2) require evaluating a multivariate quadratic system to encrypt data or verify a signature (e.g. [62, 50, 53]). Encryption uses secret data and should be performed in constant-time (Section 6.3), whereas verification is a public process and does not have this constraint. In HFEv- signature schemes (Section 2.4.1), the evaluation step is the main part of verification. Efficient implementations of evaluation were studied in [15, 51, 53, 56]. The authors of [15] proposed different strategies for the evaluation over $\mathbb{F}_2$, $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$. In [51], the evaluation over $\mathbb{F}_{31}$, $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ is vectorized with SSSE3 instructions. In [53], the authors proposed to optimize the evaluation over $\mathbb{F}_{31}$ and $\mathbb{F}_2$ by evaluating the public-key equations one by one. Their implementation is vectorized with the AVX2 instruction set. In [56], the authors presented a faster evaluation over $\mathbb{F}_2$ with the same instruction set. To do so, they used a *monomial representation* of the public-key: for each monomial, the corresponding coefficients in each equation are stored together. We optimize the evaluation with this representation to obtain new speed records.

Root finding of a `HFE` polynomial. The main part of the signature generation in `Gui` and `GeMSS` is to find the roots of a polynomial F over $\mathbb{F}_{2^{d_{\text{ext}}}}$ with a specific form. Root finding (Section 5.4.8) is a fundamental problem in computer algebra with various applications in discrete mathematics. A survey of the main root finding methods can be found in [161]. Recently, the successive resultant algorithm (`SRA`) [138] has been proposed to find the roots of a polynomial in small characteristic. In [61], `SRA` has been extended to arbitrary finite fields. In particular, root finding is improved for split and separable polynomials, when the cardinality of the multiplicative group is smooth.

In the case of the `HFE` polynomial $F \in \mathbb{F}_{2^{d_{\text{ext}}}}[X]$, F has a sparse structure and its coefficients are in a field of small characteristic. Moreover, the number of roots is generally small (it is almost always less than ten for our parameters). The main challenge is to exploit the sparse structure of F to improve the complexity of the root finding: it should depend on the number of coefficients of F and not on its degree. In practice, Berlekamp's algorithm [161, Algorithm 14.15], which computes $\text{GCD}(F, X^{2^{d_{\text{ext}}}} - X \bmod F)$, is used. The most costly task is the computation of $X^{2^{d_{\text{ext}}}} \bmod F$, also called Frobenius map, and the `HFE` structure of F can be exploited during the modular reduction. In [144], the authors proposed a method to compute the Frobenius map (Section 5.4.5) with multi-squaring tables, which is interesting when the degree of F is (approximately) smaller than d_{ext}. We study how to implement the Frobenius map efficiently, optimizing as a function of the parameters.

Arithmetic in $\mathbb{F}_{2^{d_{\text{ext}}}}$. Arithmetic in $\mathbb{F}_{2^{d_{\text{ext}}}}$ (Section 5.3) is a critical part of the root finding algorithm, because all operations in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$ require it, and is studied in [6, 5, 159, 30]. In particular, multiplication in $\mathbb{F}_{2^{d_{\text{ext}}}}$ is the most critical operation. This is a well-known task and is studied in [41, 72, 121, 52]. Here, we choose to use the `PCLMULQDQ` instruction (Section 6.1.3) to obtain an efficient implementation. This instruction computes the product of two binary polynomials, each of degree strictly less than 64.

Organization of the Chapter and Main Results

We present `MQsoft` [84]: an efficient library in `C` programming language for `HFE`-based schemes such as `GeMSS`, `Gui` and `DualModeMS`. `MQsoft` is an improved version of the `GeMSS` additional implementation submitted to the NIST post-quantum cryptography competition [128]. Our library permits to improve the fastest known implementations of `GeMSS`, `Gui` and `DualModeMS`. The performance results are studied in Section 9.6. Table 9.1 summarizes the obtained speed-ups.

The structure of MQsoft is depicted in Figure 9.1 which summarizes the main tasks required for each cryptographic operation. The critical part of an operation is represented by a plain arrow, whereas less important operations are represented by dotted arrows.

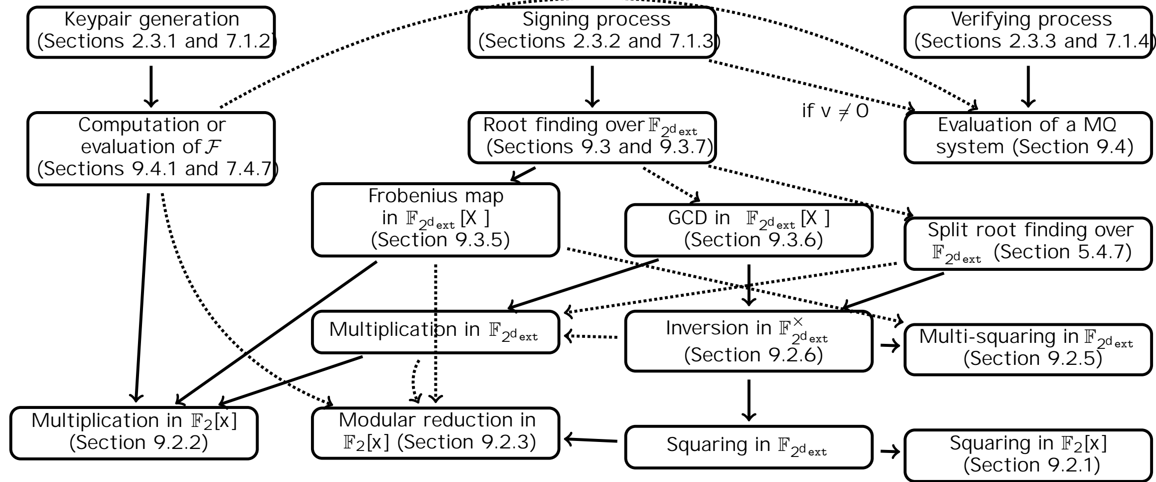

Figure 9.1: Dependencies between the different operations performed in MQsoft.

scheme	key gen.	sign	verify
GeMSS128	19.6 × 6.03	608 × 2.09	0.106 × 1.57
GeMSS192	69.4 × 7.90	1760 × 1.83	0.304 × 1.47
GeMSS256	158 × 9.32	2490 × 2.16	0.665 × 1.76
FGeMSS(266)	53.7 × 8.22	44 × 2.90	0.0365 × 2.64
DualModeMS128	3710 × 537	2800 × 2.81	0.643 × 15.3
Gui-184	23.5 × 31.7	28.5 × 2.60	0.0712 × 1.89
Gui-312	116 × 41.9	308 × 2.53	0.161 × 1.85
Gui-448	356 × 91.7	5710 × 3.44	0.562 × 1.62

Table 9.1: Number of megacycles (Mc) for each cryptographic operation with our library for a Skylake processor (LaptopS), followed by the speed-up between the best implementation provided for the NIST submissions (Table 7.37) versus our implementation. For example, 19.6 × 6.03 means a performance of 19.6 Mc with MQsoft, and a performance of $19.6 \times 6.03 = 118$ Mc for the NIST implementations.

It is clear from Figure 9.1 that HFE-based schemes require an efficient implementation of arithmetic in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$ and so in $\mathbb{F}_{2^{d_{\text{ext}}}}$. This is studied in Section 9.2. We have implemented state-of-the-art algorithms for arithmetic in $\mathbb{F}_{2^{d_{\text{ext}}}}$ that use vectorization (SSE2 and AVX2) and the `PCLMULQDQ` instruction to improve multiplication in $\mathbb{F}_{2^{d_{\text{ext}}}}$ (Section 9.2.2). The multiplication is computed with the schoolbook algorithm by block of 64 bits or with Karatsuba's algorithm, in function of the number of blocks and the processor used. When `PCLMULQDQ` is not available, `MQsoft` uses the multiplication in $\mathbb{F}_2[x]$ of the `gf2x` library (Section 6.2.1). `MQsoft` also proposes straightforward algorithms to assure a constant-time implementation, but the performance could be improved. The modular inverse is computed with the Itoh–Tsujii Multiplicative Inversion Algorithm (Section 9.2.6) together with multi-squaring tables (Section 9.2.5).

To optimize the arithmetic in $\mathbb{F}_{2^{d_{\text{ext}}}}$, the choice of d_{ext} has to be made before the compilation. This permits the specialization of the implementation. The library is flexible and allows the choice of any $d_{\text{ext}} \leqslant 576$. $\mathbb{F}_{2^{d_{\text{ext}}}}$ is built as $\mathbb{F}_2$ quotiented by an irreducible polynomial f of degree d_{ext}. When it is possible, we choose an irreducible trinomial for f to accelerate the modular reduction (Section 9.2.3). The modular reduction by f is vectorized for trinomials such that the degree of $f - X^{d_{\text{ext}}}$ is strictly less than 128, and for the parameters of studied schemes. We have vectorized the modular reduction by a pentanomial exclusively for $d_{\text{ext}} \in \{184, 312, 448, 544\}$ (Section 9.2.4), because they are the parameters of `Gui` and `DualModeMS256`^{V1}. Otherwise, the modular reduction is implemented for pentanomials such that the degree of $f - X^{d_{\text{ext}}}$ is strictly less than 33. For $d_{\text{ext}} \leqslant 576$, 56% of the finite fields can be created with an irreducible trinomial (Section B.9). Our library vectorizes modular reduction for 92% of these cases. We obtain approximately a speed-up of a factor of four compared to the arithmetic in $\mathbb{F}_{2^{d_{\text{ext}}}}$ of `NTL`.

In Section 9.4, the verifying process is accelerated via an efficient evaluation of multivariate quadratic systems using AVX2 instruction set. We obtain new speed records for the constant-time and variable-time evaluations of binary multivariate polynomials. To do it, we have chosen to use the monomial representation as in [56]. We have stored MQ systems of m equations in $\mathbb{F}_2[x_1, \ldots, x_{n_{\text{var}}}]$ as a pair $(C, Q) \in \mathbb{F}_{2^m} \times \mathcal{M}_{n_{\text{var}}}(\mathbb{F}_{2^m})$, where Q is an upper triangular matrix such that $Q_{i,j}$ corresponds to the term $x_i x_j$, and C is the constant term. Since the MQ systems will be evaluated over $\mathbb{F}_2$, $x_i^2 = x_i$ and so, the linear term x_i is stored with the term x_i^2 of Q. With this representation, the evaluation in $x \in \mathbb{F}_2^{n_{\text{var}}}$ is computed as $C + x \cdot Q \cdot x^T$. For 256 equations in 256 variables over $\mathbb{F}_2$, our variable-time evaluation is 1.38 times faster than in [56]. To obtain this, we use unrolled loops and a specific way to extract the terms x_i. For the constant-time evaluation, we obtain a performance similar to [56], which targets Haswell processors. However, on Skylake processors, the evaluation can be faster by using vector instructions in a specific way, as explained in Section 9.4.4. This method saves a factor 1.1 on Skylake (for 256 equations in 256 variables).

The core of the signing process is to find the roots of a univariate HFE polynomial F in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$, which has a special structure. In particular, F is in the following note:

$$\sum_{\substack{0 \leqslant j < i < d_{\text{ext}} \\ 2^i + 2^j \leqslant D}} A_{i,j} X^{2^i + 2^j} + \sum_{\substack{0 \leqslant i < d_{\text{ext}} \\ 2^i \leqslant D}} B_i X^{2^i} + C \in \mathbb{F}_{2^{d_{\text{ext}}}}[X].$$

Our goal is to exploit this structure to accelerate the root finding. We address this question in Section 9.3. We have been able to tweak Berlekamp's algorithm [161, Algorithm 14.15] to take advantage of the sparse structure of F.

When $D > d_{ext}$, the computation of $X^{2^{d_{ext}}} \bmod F$ is done with the repeated squaring algorithm [161, Algorithm 4.8]. The core of this algorithm is to compute the modular reduction of the square of an element $B \in \mathbb{F}_{2^{d_{ext}}}[X]$ by F. The classical Euclidean division of B^2 by F requires computing $B^2 - QF$, where Q is the quotient of this division. With a naive implementation, the multiplication of Q by F costs $O(D^2)$ field multiplications. Using a sparse representation of F, the multiplication only costs $O(D \log_2(D)^2)$ field multiplications.

So, with a sparse polynomial, the computation of the roots is faster. This suggests considering sparse HFE polynomials. In Theorem 10, we prove that making F more sparse improves the complexity. Because F is a part of the secret-key, the nature of this change requires a new analysis of security. We observe in practice that removing a small number of odd degree terms appears not to affect the security. However, the security of this method has to be studied in depth. With Theorem 10, we can save 43.75% of the computations we would have done, by removing only three terms having an odd degree in F. The general idea to make F more sparse has already been proposed in HFEBoost[1], but independently of this, the proof of Theorem 10 (Section 9.3.3) provides a concrete method to improve the complexity. It has the advantage of being in constant-time because the useless computations are known and so can be avoided.

Theorem 10. *Let H be a HFE polynomial of degree D in $\mathbb{F}_{q^{d_{ext}}}[x]$ where the s terms of highest odd degree have been removed ($0 \leqslant s \leqslant \lceil \log_q(D) \rceil$), and let $A \in \mathbb{F}_{q^{d_{ext}}}[x]$ be a square of degree at most $2D-2$. If D and q are even, then the computation of the classical Euclidean division (Algorithm 37) of A by H can be accelerated by a factor $(D-1)/\left(\frac{D}{2} + \lfloor q^{\lceil \log_q(D) \rceil - s - 2} \rfloor\right)$.*

When $D < d_{ext}$, the strategy of [144] becomes more efficient for computing the Frobenius map. The idea is to compute a lookup table of $X^{2i} \bmod F$ to accelerate the modular reduction. Thus, the squaring modulo F is computed by multiplying its i-th coefficient by the element $X^{2i} \bmod F$ from the table for $i \in [\![0, D-1]\!]$. The authors of [144] also suggest doing several squarings in one step, with multi-squaring tables. In Section 5.4.5, we describe an explicit strategy for doing this efficiently, and how to choose the number of squarings to perform before the modular reduction. In Section 9.3.5, we show the results obtained by exploiting the HFE structure of F when it is possible.

The performance of both strategies described above depends on the required number of field multiplications. The accurate number of multiplications of each method which is given in Section 5.4.5 allows to choose the best strategy as a function of the parameters. Finally, when D is small enough compared to d_{ext}, both previous strategies can be embedded with the use of modular composition (Section 9.3.4). This method generates speed-ups which are proportional to the gap between D and d_{ext}.

9.1 Data Structure

Here, we describe the data structure used to store elements of $\mathbb{F}_{2^{d_{ext}}}$ and $\mathbb{F}_{2^{d_{ext}}}[X]$. This representation is crucial for the efficiency of our implementation. This is especially true for binary fields since operations in $\mathbb{F}_2$ can be naturally vectorized (Section 5.3.1). Arithmetic in $\mathbb{F}_{2^{d_{ext}}}[X]$ is used during the root finding of a HFE polynomial. To be efficient, it is important to distinguish dense polynomials which appear during the computation of the Frobenius map and the GCD, from a HFE polynomial which is used to reduce $X^{2^{d_{ext}}} - X$. We can notice that the HFE polynomial is sparse since it only has $K = O(\log_2(D)^2)$ non-zero coefficients.

[1] https://www-polsys.lip6.fr/Links/hfeboost.html

Representation of elements in $\mathbb{F}_{2^{d_{\text{ext}}}}$. The field $\mathbb{F}_{2^{d_{\text{ext}}}}$ is defined as $\mathbb{F}_2[\ ]/(f(\ ))$ with f being an irreducible polynomial of degree d_{ext} in $\mathbb{F}_2[x]$. We have chosen the polynomial basis (Section 5.2.1). An element of $\mathbb{F}_{2^{d_{\text{ext}}}}$ is represented by a polynomial in $\mathbb{F}_2[\ ]$ of degree at most $d_{\text{ext}} - 1$. The coefficients are stored as a vector of bits, requiring $\lceil \frac{d_{\text{ext}}}{w} \rceil$ words, where w is the word size (in bits). The j-th bit of the i-th word is the coefficient of the term of degree $wi + j$, for $i \in [\![0, \lceil \frac{d_{\text{ext}}}{w} \rceil - 1]\!]$ and $j \in [\![0, w-1]\!]$. It is set to zero when $(wi + j) \geqslant d_{\text{ext}}$.

Example 1 (Storing a binary polynomial). Let $w = 64$ and $P = {}^{36} + {}^{4} \in \mathbb{F}_{2^{40}}$. To simplify the notations, we represent vectors of bits as 64-bit integers. P is stored as `0x0000001000000010`. In particular, the bits from 37 to 63 are set to zero.

Representation of dense polynomials in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$. An element of $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$ is represented by its degree d and a vector of $d+1$ coefficients. The coefficients are stored from lower to higher degree of the corresponding terms in an array. The degree is stored in a local variable, except for the implementation of the fast GCD in Section 9.3.6, because it requires matrices in $\mathcal{M}_2(\mathbb{F}_{2^{d_{\text{ext}}}}[X])$. In this case, we use a C structure to store the degree and the pointer toward the array of coefficients.

Example 2 (Storing a dense polynomial). Let $P = X^8 + \ X^7 + (\ +1)X^6 + X^5 + \ X \in \mathbb{F}_4[X]$. P is stored as $8, (0, \ , 0, 0, 0, 1, \ +1, \ , 1)$.

Representation of HFE polynomials in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$. In HFEv-based schemes, the HFEv polynomial is a part of the secret-key. During the signature generation of $\mathcal{S}$ (Section 2.3.2), the vinegar variables of the HFEv polynomial are evaluated to obtain a HFE polynomial. Its degree D is a parameter of security and is assumed to be known. It is defined by the C directive `#define`. A HFE polynomial in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$ is represented as a vector of coefficients where only the terms X^0, X^{2^i} and $X^{2^i+2^j}$ are stored. It is chosen monic and so the leading term is not stored. If P is in $\mathbb{F}_{2^{d_{\text{ext}}}}[X]$, we denote by P_{HFE} its HFE representation.

Example 3 (Storing a HFE polynomial). Let $P = X^{16} + \ X^{12} + (\ +1)X^{10} + \ X \in \mathbb{F}_4[X]$. P_{HFE} is stored as $(0, \ , 0, 0, 0, 0, 0, 0, 0, \ +1, \ )$. Only the coefficients of terms with a degree in $\{0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12\}$ are stored.

9.2 Arithmetic in $\mathbb{F}_{2^{d_{\text{ext}}}}$

Arithmetic in $\mathbb{F}_{q^{d_{\text{ext}}}}$ is the core of the signature generation (Section 7.1.3) and keypair generation (Section 7.1.2). In Section 9.4.1, we explain how to generate efficiently the inner secret polynomials of f (Equation (7.2)). This requires $O\left(d_{\text{ext}} \log_q(D) \log_2(q)\right)$ squarings in $\mathbb{F}_{q^{d_{\text{ext}}}}$ (precomputation step), $O\left(d_{\text{ext}} \log_q(D)(n_{\text{var}} + \log_q(D))\right)$ operations in $\mathbb{F}_q[x]$ and $O\left(d_{\text{ext}}(n_{\text{var}} + \log_q(D))\right)$ modular reductions. In Section 7.4.7, we generate the public-key by evaluation-interpolation. This requires $N = O\left(n_{\text{var}}^2\right)$ evaluations of a HFEv polynomial. Once the vinegar variables are evaluated, we have to evaluate a HFE polynomial, which implies an overall cost of $O\left(N \log_q(D) \log_2(q)\right)$ squarings in $\mathbb{F}_{q^{d_{\text{ext}}}}$, $O\left(N \log_q(D)^2\right)$ operations in $\mathbb{F}_q[x]$ and $O\left(N \log_q(D)\right)$ modular reductions. With a multipoint evaluation (Section 7.4.7), the computation of $\mathcal{F}$ or $\mathcal{S}$ requires only $O\left(n_{\text{var}} \log_q(D) \log_2(q)\right)$ squarings in $\mathbb{F}_{q^{d_{\text{ext}}}}$, $O\left(n_{\text{var}} \log_q(D) \cdot (n_{\text{var}} + \log_q(D))\right)$ operations in $\mathbb{F}_q[x]$ and $O\left(n_{\text{var}} \cdot (n_{\text{var}} + \log_q(D))\right)$ modular reductions, as well as $O\left(n_{\text{var}} d_{\text{ext}} v \cdot (n_{\text{var}} + v + \log_q(D))\right)$ operations in $\mathbb{F}_q$ due to the vinegar part, and the extra memory cost is approximately $n_{\text{var}} \cdot \left(v + 2 \cdot \lceil \log_q(D) \rceil + 1\right)$ elements of $\mathbb{F}_{q^{d_{\text{ext}}}}$.

In Section 9.3, the signature generation can be performed with $O(\text{nb_ite} \cdot D^2)$ field operations, $O(\text{nb_ite} \cdot d_{ext} D \log_q(D)^2 \log_2(q))$ operations in $\mathbb{F}_q[x]$, $O(\text{nb_ite} \cdot d_{ext} D \log_2(q))$ squarings in $\mathbb{F}_q[x]$, $O(\text{nb_ite} \cdot d_{ext} D \log_2(q))$ modular reductions and (optionally) $O(\text{nb_ite} \cdot D)$ field inversions.

In GeMSS, $q = 2$. We use the polynomial representation defined in Section 9.1. It is the most efficient representation when PCLMULQDQ is available [159]. To compute the square (respectively the multiplication) of B in $\mathbb{F}_{2^{d_{ext}}}$, we choose to compute the square (respectively the multiplication) of B in $\mathbb{F}_2[x]$ before performing the reduction by the univariate polynomial defining the extension.

architecture	Skylake	Broadwell	Haswell	Ivy Bridge
latency	6	5	7	14
throughput (CPI)	1	1	2	8

Table 9.2: Performance of the PCLMULQDQ instruction in function of the architecture, as presented in the Intel Intrinsics Guide. We note that the latency on Skylake was evaluated to 7 until early 2020. This explains why we find this value in older articles, as in [84, 62, 72].

Table 9.2 presents the cost of PCLMULQDQ in function of the architecture. The choice of the best algorithm of multiplication in $\mathbb{F}_{2^{d_{ext}}}$ depends on the processor. Our choices target the Skylake processors, which use only one CPI (Cycle Per Instruction) to perform PCLMULQDQ. We have also extended MQsoft to support Haswell processors, because it is the most studied architecture in the scientific world. On Haswell, PCLMULQDQ requires two CPI. This makes optimal methods using less calls to PCLMULQDQ.

This section contains mainly the results of [84]. We have added some details on the best methods to compute the square and the multiplication in $\mathbb{F}_{2^{d_{ext}}}$ on Haswell, which are currently used in the optimized implementation of GeMSS (Section 7.5.4). We have completed Table 9.3. In [84], the AVX2 implementation of the squaring was available only for $\lceil \frac{d_{ext}}{64} \rceil = 0 \bmod 4$ We also improve the modular reduction for specific trinomials (Table 9.9), and we give the performance of the reduction modular by pentanomials that we use (Table 9.10), based on several kinds of optimization.

9.2.1 Polynomial Squaring over $\mathbb{F}_2$

Squaring is used during the root finding algorithm (Section 5.4.8) which is the core of the signature generation (Section 7.1.3). It is also used during the so-called Itoh–Tsujii Multiplicative Inversion Algorithm which computes the modular inverse in $\mathbb{F}_{2^{d_{ext}}}$ (Section 9.2.6).

In binary fields, the squaring of $B = \sum_{i=0}^{d_{ext}-1} b_i \alpha^i \in \mathbb{F}_{2^{d_{ext}}}$ can be performed in linear time (Section 5.1.3). The linearity of the Frobenius endomorphism implies that $B^2 = \sum_{i=0}^{d_{ext}-1} b_i \alpha^{2i}$. Since we have stored B as a vector of bits, squaring corresponds to insert a null bit between each bit of B.

Example 4 (Square of a binary polynomial). Let $B = \alpha^3 + \alpha^2 + 1 \in \mathbb{F}_{2^4}$. B is stored as the binary integer 1101. Its square is $B^2 = \alpha^6 + \alpha^4 + 1$, which is represented as 1010001.

To compute the square of a d_{ext}-bit element on Skylake processors and newer, we divide it into words of 64 bits. For each one, the PCLMULQDQ instruction directly computes the binary polynomial multiplication of the 64-bit element by itself. This method requires $\lceil \frac{d_{ext}}{64} \rceil$ calls to PCLMULQDQ.

On Haswell processors, we use the previous method for $\frac{d_{ext}}{64} \leqslant 3$. For $\frac{d_{ext}}{64} \geqslant 4$, the use of lookup tables [5] is faster (in AVX2). We have implemented Algorithm 1 of [5] which uses SSE2 instructions and `PSHUFB` from SSSE3, and a variant in AVX2 that uses the `VPSHUFB` instruction. The `PSHUFB` instruction performs the lookup of the square of 16 elements on 4 bits in a lookup table in constant-time, and the `VPSHUFB` instruction performs two times `PSHUFB` (Section 6.1.3).

Table 9.3 summarizes the performance of squaring functions which are proposed in our library. The experimental process consists in computing the square of elements from a small array, then measuring the average cost of one operation. We have added the SSE2 column to highlight the impact of the `PSHUFB` instruction. This SSE2 implementation emulates our specific use of `PSHUFB` by using some shift, `XOR` instructions and masks (we start by inserting two null bits between each pair of bits, then we insert one null bit between each bit). For the squaring using `VPSHUFB`, the implementation for odd values of $\frac{d_{ext}}{64}$ is slower than for the even values. It is due to the `load` and `store` instructions, which are less efficient. On the Skylake processors, the best squaring is the one using the `PCLMULQDQ` instruction: this instruction costs only one cycle of throughput, but six cycles of latency (Table 9.2). However, the latency can be used to run other instructions, which improves the performance.

$\frac{d_{ext}}{64}$	Skylake				Haswell			
	SSE2	PSHUFB	VPSHUFB	PCLMULQDQ	SSE2	PSHUFB	VPSHUFB	PCLMULQDQ
1	5.9	5.7	5.9	2.2	4.9	4.9	5.1	2.1
2	8.7	5.7	5.6	5.0	8.2	5.9	5.3	4.9
3	13.0	8.0	8.1	6.2	12.2	7.0	7.8	6.9
4	16.1	9.0	7.5	6.3	14.9	8.8	7.4	8.4
5	19.7	12.2	9.3	7.2	19.5	10.6	8.9	10.5
6	23.7	13.3	9.1	7.9	21.4	12.4	9.7	12.4
7	27.3	16.6	13.8	9.4	26.4	14.5	12.9	14.6
8	30.9	17.6	11.3	10.6	29.9	16.3	12.1	16.3
9	35.7	20.8	14.7	11.8	35.3	18.5	14.2	18.7

Table 9.3: Number of cycles for computing the square of an element of $\mathbb{F}_2[x]$ of degree $d_{ext} - 1$, with `MQsoft`. We use a Skylake processor (LaptopS) then a Haswell processor (ServerH).

9.2.2 Polynomial Multiplication over $\mathbb{F}_2$

The multiplication of two distinct elements in $\mathbb{F}_{2^{d_{ext}}}$ is a central operation involved in the keypair generation and the signing process. In `MQsoft`, we adapt the multiplication algorithm in function of d_{ext}. We start by optimizing it on the Skylake processors.

When $\frac{d_{ext}}{64} \in \{1, 2, 3, 5, 6\}$, we use a schoolbook multiplication by block of 64 bits. We use the `PCLMULQDQ` instruction for multiplying each block. Then $\left(\frac{d_{ext}}{64}\right)^2$ calls to `PCLMULQDQ` are required. This method is naturally constant-time. Our implementation uses `PCLMULQDQ` which implies the use of SSE2 instructions. We also use the `PALIGNR` instruction from SSSE3 to improve the implementation. This instruction concatenates two 128-bit registers, shifts the concatenation to the right by a multiple of eight bits, then returns the 128 lower bits of the result. We use it to align the output of `PCLMULQDQ` (which is at least 64-bit aligned) to 128 bits. When `PALIGNR` is not available, we use `SHUFPD` (or `SHUFPS`) for this kind of alignment.

When $\frac{d_{\mathtt{ext}}}{64} \in \{4, 7, 8, 9\}$, Karatsuba's multiplication algorithm (Section 5.1.2) becomes faster than the schoolbook method. We use the schoolbook multiplication as base case for size words 2, 3 and 5. For $\frac{d_{\mathtt{ext}}}{64} = 8$, we call three times Karatsuba's method on four words, the latter also calling three times Karatsuba's method on two words. The number of calls to PCLMULQDQ for $\frac{d_{\mathtt{ext}}}{64} \in \{4, 8\}$ is respectively 12 and 36. For $\frac{d_{\mathtt{ext}}}{64} \in \{7, 9\}$, we split each input in two: the 256 lower bits create a degree-255 polynomial, and the remaining bits create a degree-$(d_{\mathtt{ext}} - 257)$ polynomial. Thus, Karatsuba's algorithm requires one multiplication of degree-255 polynomials, one multiplication of degree-$(d_{\mathtt{ext}} - 257)$ polynomials and one multiplication of degree-$\max(255, d_{\mathtt{ext}} - 257)$ polynomials. This method requires 33 calls to PCLMULQDQ for $\frac{d_{\mathtt{ext}}}{64} = 7$ and 62 calls[2] for $\frac{d_{\mathtt{ext}}}{64} = 9$. Here, the way to split inputs is important. The lower bits have to generate a k-bit polynomial such that k is a multiple of 128. This allows to efficiently add it to the higher part of the input, without any use of shifts. Moreover, the result of each recursive call will also be 128-bit aligned, simplifying the recombination step.

The trade-off between the schoolbook multiplication and Karatsuba's algorithm depends on the performance of PCLMULQDQ (Table 9.2). For the Skylake processors, this instruction costs one CPI, which makes schoolbook multiplication more efficient for 6-word multiplication. For the Haswell processors, PCLMULQDQ costs two CPI. This decreases the trade-off because each call to PCLMULQDQ is more penalizing. When $\frac{d_{\mathtt{ext}}}{64}$ is equal to 3, we remark that the Karatsuba multiplication is already faster on Haswell. We have compared the schoolbook multiplication to the three-term Karatsuba-like formulae described in Section B.1. The latter is slightly faster and requires only 6 calls to PCLMULQDQ.

Then, we proceed as follows for Haswell. For $\frac{d_{\mathtt{ext}}}{64} = 8$, we call Karatsuba's algorithm recursively until a base case on one word. This requires 27 calls to PCLMULQDQ. For $\frac{d_{\mathtt{ext}}}{64} \in \{4, 5, 6, 7, 9\}$, we use base cases on two words (using the schoolbook method) and three words (using the three-term Karatsuba-like formula). The case $\frac{d_{\mathtt{ext}}}{64}$ equals 4 (respectively 6) is computed with 3 calls to the 2-word (respectively 3-word) multiplication. For $\frac{d_{\mathtt{ext}}}{64} = 5$, we split each input in two: the 128 lower bits create a degree-127 polynomial, and the remaining bits create a degree-$(d_{\mathtt{ext}} - 129)$ polynomial. This method requires 2 calls to the 3-word multiplications and 1 call to the 2-word multiplication. Finally, for $\frac{d_{\mathtt{ext}}}{64} \in \{7, 9\}$, we use the same strategy that for Skylake. We just perform the base cases with Haswell multiplications instead of Skylake multiplications.

Table 9.4 compares our multiplication to gf2x. As in Section 9.2.1, we measure the average cost of multiplying elements from a small array. The multiplication of gf2x is sometimes abnormally slow. This is probably due to the fact that the implementation uses vector and no vector instructions in the same function, which penalizes it. This is the first reason to explain that our multiplication is faster. The second reason is that gf2x uses Karatsuba's algorithm, which is slower than schoolbook multiplication for 6-word multiplications on Skylake. The gf2x code may target Haswell processors. We have also remarked that installing NTL with gf2x slightly decreases the performance. For this reason, NTL is not installed with gf2x on our experimental platform.

[2]The use of a 8-word multiplication followed by a classical multiplication for each last block may be slightly faster. This requires 53 calls to PCLMULQDQ.

$\frac{d_{ext}}{64}$	Skylake		Haswell	
	`gf2x`	`MQsoft`	`gf2x`	`MQsoft`
1	3.2	3.2	2.1	2.1
2	7.7	6.8	8.5	8.5
3	37.0	15.5	34.2	18.4
4	23.1	19.9	27.6	24.6
5	47.0	34.9	52.2	43.7
6	54.3	45.7	58.1	57.3
7	142.2	55.4	136.2	73.2
8	91.1	59.9	110.6	87.4
9	131.8	91.3	146.2	117.5

Table 9.4: Number of cycles to multiply two elements of $\mathbb{F}_2[x]$ of degree $d_{ext} - 1$. We use a Skylake processor (LaptopS) then a Haswell processor (ServerH).

9.2.3 Modular Reduction by Trinomials over $\mathbb{F}_2$ and Field Product

In this section, we want to reduce $R = \sum_{i=0}^{2d_{ext}-2} r_i x^i$ the result of the previous multiplication or squaring in $\mathbb{F}_2[x]$. The choice of the irreducible polynomial $f = \sum_{i=0}^{\omega-1} x^{k_i}$ ($k_0 = 0 < k_1 < \cdots < k_{\omega-1} = d_{ext}$) defining $\mathbb{F}_{2^{d_{ext}}}$ (Section 9.1) is important for the modular reduction. In Section 5.3.3, we presented classical techniques when f is sparse. For trinomials, we optimize the computation of (5.8) by factoring by $f_3 - x^{d_{ext}}$. In this way, we can write R as:

$$R = R_0 + S_{k_1} x^{d_{ext}-k_1} + \left(R_{k_1} + S_{k_1}\right) \cdot \left(f_3 - x^{d_{ext}}\right) \bmod f_3. \tag{9.1}$$

For $k_1 \leqslant \left\lfloor \frac{d_{ext}}{2} \right\rfloor$, there are mainly two methods [30] to compute (9.1). The first is the shift-and-add strategy: $\left(R_{k_1} + S_{k_1}\right) \cdot \left(f_3 - x^{d_{ext}}\right)$ is computed as $Q + Q \cdot x^{k_1}$ with $Q = R_{k_1} + S_{k_1}$. The second is the mul-and-add strategy: the multiplication by $f_3 - x^{d_{ext}}$ is computed with the `PCLMULQDQ` instruction. In this case, it is recommended to choose $k_1 + 64 \cdot \left\lceil \frac{d_{ext}}{64} \right\rceil - d_{ext}$ strictly less than 64. In this way, $f_3 - x^{d_{ext}}$ can be shifted then used for multiplying 64-bit blocks of R via the `PCLMULQDQ` instruction. We choose the first method because it requires a small number of low-cost instructions.

Shift-and-add strategy in practice. For the shift-and-add strategy, the main operations are shifts of coefficients. This operation is not trivial. When a binary polynomial is stored on several registers, we need the bits coming out of a register are coming into the next register. Moreover, the use of parallel shifts implies that the bits coming out of a block are not coming into the next block. We study the shifts of coefficients for 128-bit registers. Firstly, the shift by a multiple of eight coefficients is native. We can do it with the `PSLLDQ` and `PSRLDQ` instructions, or with the `PALIGNR` instruction. Secondly, when we require a shift by $64k + j$ positions for $k \in \{0, 1\}$ and $0 < j < 64$, we start by using a shift by 8 bytes only if $k = 1$, then we use a j-bit shift by block of 64 bits (`PSLLQ` or `PSRLQ`). However, a j-bit shift implies loosing j bits between each block. For $k = 0$, we need to compute these j bits with a complementary shift, then we xor them to the previous result. The complementary shift requires a shift by 8 bytes followed by a shift by $64 - j$ bits in the opposite direction. For shifts of successive registers, we use `PALIGNR` instead of `PSLLDQ` and `PSRLDQ`. When `PALIGNR` is not available, we perform shifts by 8 bytes with `SHUFPD` (or `SHUFPS`). Now, we propose to study irreducible polynomials which simply the use of shifts.

Optimization for specific values of d_{ext} and k_1. The shift-and-add and mul-and-add methods can be optimized for specific values of k_1 and d_{ext}. Firstly, the case $k_1 = 1$ permits to avoid computations because $S_{k_1} = 0$. Secondly, we previously note that shifting by a multiple of eight bits is faster, thanks to the `PSLLDQ`, `PSRLDQ` and `PALIGNR` instructions. Here, we list cases where these instructions could be used:

- for the extraction of S_{k_1} from R when $2d_{ext} - k_1$ is a multiple of eight,
- to obtain $S_{k_1}x^{d_{ext}-k_1}$ from S_{k_1} when $d_{ext} - k_1$ is a multiple of eight,
- for the multiplication by x^{k_1} when k_1 is a multiple of eight,
- for the extraction of $R_{k_1} + S_{k_1}x^{d_{ext}-k_1}$ from R when d_{ext} is a multiple of eight. However, irreducible trinomials such that d_{ext} is a multiple of eight do not exist (Section 5.3.2).

For the shift-and-add strategy, we can also compute $(R_{k_1} + S_{k_1})(f_3 - x^{d_{ext}})$ as $\frac{Q'}{x^{k_1}} + Q'$ with $Q' = R_{k_1}x^{k_1} + S_{k_1}x^{k_1}$. In this case, the bytewise shifts improve the implementation in the following cases:

- for the extraction of $R_{k_1}x^{k_1}$ from R when $d_{ext} - k_1$ is a multiple of eight,
- for the extraction of $S_{k_1}x^{k_1}$ from R when $2d_{ext} - 2k_1$ is a multiple of eight,
- to obtain $S_{k_1}x^{d_{ext}-k_1}$ from $S_{k_1}x^{k_1}$ when $d_{ext} - 2k_1$ is a multiple of eight,
- for the division by x^{k_1} when k_1 is a multiple of eight.

Note that the previous shifts can be completely removed when the value is a multiple of the register size. In this case, a shift corresponds to a choice of register. When R is a square, we use similar optimizations by removing useless shifts (whose result is necessarily zero due to the square structure, as in [131, Algorithm 2]). A shift by one bit using `PSLLQ` (respectively `PSRLQ`) does not require a complementary shift if and only if the highest (respectively lowest) bit of each block is null. In this case, we improve the shifts by $8k + j$ bits for $j = 1$ or $j = 7$ (since $8k + 7 = 8(k + 1) - 1$).

Results. We have implemented the shift-and-add method for trinomials with different SIMD instruction sets. SSSE3 is used to improve the implementation with the `PALIGNR` instruction. We have also implemented the shift-and-add method for pentanomials, but it is vectorized only for $d_{ext} \in \{184, 312, 448, 544\}$, because they are the parameters of `Gui` and `DualModeMS256`^{V1}. The method used is described in Section 9.2.4. For the design of `HFE`-based schemes, we choose d_{ext} such that there exists an irreducible trinomial of degree d_{ext} (Section 7.4.10).

The performance of the modular reduction depends on the context. In Table 9.5, we reduce products from a small array, then we measure the cost of one modular reduction on average. Here (including Tables 9.5, 9.6, 9.7, 9.8 and 9.11), we do not exploit the values of d_{ext} and k_1 (as explained above) to improve the implementation. This allows to have comparable measurements in function of $\frac{d_{ext}}{64}$. In practice, they are used (for example, the SSSE3 modular reduction for $d_{ext} = 177$ and $k_1 = 8$ takes 8.7 cycles on Skylake with these optimizations). The SSSE3 version is two times faster than without vector instructions because SSE2 permits to perform two 64-bit instructions in one instruction. On Skylake, the AVX2 implementation is slightly faster than the SSSE3 version, probably because the AVX2 is faster to load and store data.

(d_{ext}, k_1)	Skylake			Haswell		
	no SIMD	SSSE3	AVX2	no SIMD	SSSE3	AVX2
(62,29)	6.6	6.6	×	5.3	5.3	×
(126,21)	10.0	7.1	×	10.6	6.9	×
(186,11)	14.5	10.5	10.7	14.8	10.4	11.1
(252,15)	18.4	11.1	10.4	18.8	10.8	11.5
(314,15)	23.9	14.9	×	23.5	13.7	×
(366,29)	28.6	15.0	×	28.3	13.9	×
(425,12)	33.5	18.4	×	32.5	18.0	×
(506,23)	37.9	19.2	17.5	36.9	18.1	20.7
(574,13)	40.6	24.0	×	38.3	23.5	×

Table 9.5: Number of cycles to compute the modular reduction of an element of $\mathbb{F}_2[x]$ of degree $2d_{ext} - 2$ by $x^{d_{ext}} + x^{k_1} + 1$, with MQsoft. We use a Skylake processor (LaptopS) then a Haswell processor (ServerH).

In Table 9.6, the modular reduction is also measured when it is used with multiplication in $\mathbb{F}_2[x]$. The performance of multiplication in $\mathbb{F}_{2^{d_{ext}}}$ depends on the context. For this reason, we measure it in two ways:

- Left value: we measure the cost of one field multiplication on average during the computation of the naive exponentiation function (x^i is computed as $x^{i-1} \cdot x$). Each result depends on the previous result, and the data are already loaded.
- Right value: we measure the cost of one field multiplication on average to compute the multiplication of elements of two arrays. The data are independent but each multiplication requires loading input and storing output.

(d_{ext}, k_1)	Skylake				Haswell			
	SSSE3		AVX2		SSSE3		AVX2	
	exp.	array	exp.	array	exp.	array	exp.	array
(62,29)	17.4	9.3	×	×	17.0	8.9	×	×
(126,21)	26.4	15.2	×	×	27.1	22.2	×	×
(186,11)	32.1	24.5	44.3	25.3	32.7	30.4	47.7	33.7
(252,15)	39.8	34.7	52.4	37.0	44.8	55.4	59.8	74.3
(314,15)	54.0	50.2	×	×	53.0	55.6	×	×
(366,29)	66.3	63.8	×	×	68.2	72.4	×	×
(425,12)	78.4	79.9	×	×	85.2	89.1	×	×
(506,23)	84.7	87.4	95.5	90.1	94.9	111.1	109.0	112.5
(574,13)	112.2	118.3	×	×	127.2	140.9	×	×

Table 9.6: Number of cycles to compute the multiplication in $\mathbb{F}_{2^{d_{ext}}}$ in function of the modular reduction, with MQsoft. We use a Skylake processor (LaptopS) then a Haswell processor (ServerH).

We remark that for $\frac{d_{\text{ext}}}{64}$ less or equal to 6, the multiplication on independent data is faster on Skylake. It is probably due to the latency of the PCLMULQDQ instruction. The field multiplication with modular reduction using SSE2 is the fastest, because the PCLMULQDQ instruction requires using 128-bit registers. When SSE2 is used with AVX2, the implementation pays a penalty. But this problem should be solved with the VPCLMULQDQ instruction on Ice Lake processors (Section 6.1.3).

(d_{ext}, k_1)	PSHUFB		VPSHUFB		PCLMULQDQ		PCLMULQDQ, AVX2	
	multi-sqr	array	multi-sqr	array	multi-sqr	array	multi-sqr	array
(62, 29)	14.1	8.8	×	×	17.4	8.0	×	×
(126, 21)	18.5	12.1	×	×	21.3	10.5	×	×
(186, 11)	21.8	17.4	32.5	19.1	21.1	14.4	35.0	15.6
(252, 15)	25.2	19.9	30.6	17.0	22.3	15.5	37.1	16.4
(314, 15)	28.1	25.5	×	×	25.9	21.1	×	×
(366, 29)	30.6	26.9	×	×	26.9	20.4	×	×
(425, 12)	35.3	33.6	×	×	27.0	24.3	×	×
(506, 23)	37.7	37.7	36.5	27.6	28.1	24.2	40.4	29.5
(574, 13)	43.4	43.2	×	×	29.8	31.0	×	×

Table 9.7: Number of cycles to compute the squaring in $\mathbb{F}_{2^{d_{\text{ext}}}}$ in function of the enabled instructions, with MQsoft. We use a Skylake processor (LaptopS).

(d_{ext}, k_1)	PSHUFB		VPSHUFB		PCLMULQDQ		PCLMULQDQ, AVX2	
	multi-sqr	array	multi-sqr	array	multi-sqr	array	multi-sqr	array
(62, 29)	13.0	8.1	×	×	17.0	8.5	×	×
(126, 21)	16.5	11.5	×	×	21.0	12.5	×	×
(186, 11)	19.8	16.6	34.1	18.2	22.9	18.0	35.5	17.5
(252, 15)	20.4	18.1	29.9	17.3	25.4	19.9	38.3	19.9
(314, 15)	23.9	24.5	×	×	28.7	27.3	×	×
(366, 29)	27.7	26.0	×	×	30.4	29.2	×	×
(425, 12)	32.9	32.3	×	×	35.1	35.4	×	×
(506, 23)	36.0	38.0	34.4	28.1	37.8	37.2	44.0	36.7
(574, 13)	40.1	41.8	×	×	43.1	43.6	×	×

Table 9.8: Number of cycles to compute the squaring in $\mathbb{F}_{2^{d_{\text{ext}}}}$ in function of the enabled instructions, with MQsoft. We use a Haswell processor (ServerH).

In Tables 9.7 and 9.8, the modular reduction is measured when it is used with squaring in $\mathbb{F}_2[x]$. As for the multiplication in $\mathbb{F}_{2^{d_{\text{ext}}}}$, the performance of squaring depends on the context. For this reason, we measure it in two ways:

- Left value: we measure the cost of one field squaring on average during the raising of an element of $\mathbb{F}_{2^{d_{\text{ext}}}}$ to the power of 2^i (x^{2^i} is computed as $(x^{2^{i-1}})^2$). Each result depends on the previous result, and the data are already loaded.

- Right value: we measure the cost of one field squaring on average to compute the squaring of elements of one array. The data are independent but each squaring requires loading input and storing output.

Table 9.7 shows the performance of squaring in $\mathbb{F}_{2^{d_{ext}}}$ on Skylake. The squaring using PCLMULQDQ is the most efficient. For the same reasons that for the multiplication in $\mathbb{F}_{2^{d_{ext}}}$, the best modular squaring is the one using only SSE2 modular reduction. This is the default setting in MQsoft. On Haswell (Table 9.8), the squaring using PSHUFB is the most efficient, and could be improved with the AVX2 instruction set.

Advanced results. Now, we exploit the values of d_{ext} and k_1 to improve the implementation. In Table 9.9, we present some results, comparing several kinds of optimization. In particular, we highlight the improvements due to the square structure ($d_{ext} = 7 \bmod 8$ or $k_1 = \pm 1 \bmod 8$).

(d_{ext}, k_1)	remsqr	square PSHUFB		square PCLMULQDQ		inv ITMIA
	array	multi-sqr	array	multi-sqr	array	array
(172, 1)	8.3	16.5	13.4	17.6	10.7	3290 [1]
(172, 7)	10.5	21.8	17.5	21.1	14.5	4160 [1]
(174, 13)	10.7	21.9	17.5	21.1	14.5	2120
(174, 57)	9.9	21.2	15.5	21.8	13.3	2110
(175, 6)	9.3	18.9	15.5	18.8	12.8	2031
(175, 16)	7.7	16.9	12.5	16.5	9.7	1890
(177, 8)	8.7	19.3	14.1	19.7	11.7	1930
(239, 36)	9.9	22.3	18.2	20.7	13.9	5540 [1]
(265, 42)	16.0	28.4	25.8	26.1	21.3	3610
(265, 127)	12.8	26.0	22.1	22.1	16.9	3440
(266, 47)	16.0	28.4	25.8	26.1	21.6	3720
(354, 99)	15.1	31.1	26.3	27.3	20.3	5088
(358, 57)	13.9	30.6	24.6	25.3	18.3	5230
(402, 171)	20.5	36.0	34.5	28.5	25.7	6470
(409, 87)	18.6	34.2	30.6	24.7	23.4	11300 [1]

[1] ITMIA without multi-squaring tables (cf. Sections 9.2.5 and 9.2.6).

Table 9.9: Number of cycles to compute the modular reduction of a square, the squaring in $\mathbb{F}_{2^{d_{ext}}}$ and the inverse in $\mathbb{F}^{\times}_{2^{d_{ext}}}$, in function of d_{ext} and k_1, with MQsoft. We use a Skylake processor (LaptopS). Here, the AVX2 instruction set is used only for the multi-squaring tables. The bold values allow a speed-up due to the division by $x^{d_{ext}}$ or the multiplication by x^{k_1}.

The modular reduction of a square is important for multi-squaring algorithm. The latter is the main algorithm where the products cannot be accumulated in $\mathbb{F}_2[x]$, justifying the importance of improving the modular reduction. In Table 9.9, we obtain that several irreducible trinomials are very efficient. Using $x^{172} + x + 1$ is naturally a bit faster since $S_{k_1} = 0$, but for squares, the computation of $R_{k_1} x^{k_1}$ is just a left shift by one bit (without 8-bit alignments), because d_{ext} is

even. Using $x^{175} + x^{16} + 1$ is naturally faster since $k_1 = 0 \bmod 8$ and the property $d_{ext} = 7 \bmod 8$ improves the division by $x^{d_{ext}}$ for squares. We also note that our strategy has applications for elliptic curves [30]. Arithmetic for $d_{ext} = 239$ can be improved independently of k_1, since $d_{ext} = 7 \bmod 8$. For $d_{ext} = 409$, the choice of $k_1 = 87$ allows a speed-up since d_{ext} is odd and $k_1 = 7 \bmod 8$. We emphasize that all functions measured in this section are implemented in constant-time.

9.2.4 Modular Reduction by Pentanomials over $\mathbb{F}_2$

Similarly to trinomials (Section 9.2.3), we start by optimizing the computation of (5.10). We compute $R_{k_1} + \sum_{j=1}^{\ell-1} R_{k_j} x^{k_j}$ as $R_{k_1} \cdot \left(f - x^{d_{ext}}\right) \bmod x^{d_{ext}}$. For $k_{\ell-1} \leqslant \frac{d_{ext}}{2}$, we can write R as:

$$R = \left(R_0 + S_{k_1} x^{d_{ext}-k_1} + \left(\left(R_{k_1} + S_{k_1} + \sum_{j=2}^{\ell-1} S_{k_j}\right) \cdot \left(f - x^{d_{ext}}\right) \bmod x^{d_{ext}}\right)\right) \bmod f.$$

The product $\left(R_{k_1} + S_{k_1} + \sum_{j=2}^{\ell-1} S_{k_j}\right) \cdot \left(f - x^{d_{ext}}\right)$ can be computed with the same methods that for trinomials: directly with several calls to the `PCLMULQDQ` instruction, or else with the shift-and-add strategy. From now, we consider $\ell = 4$, i.e. pentanomials. Our library uses the shift-and-add strategy which has the advantage to be portable since it does not require `PCLMULQDQ`.

Optimization for specific values of d_{ext} and k_1, k_2, k_3. As for trinomials, we study different choices of pentanomials. Firstly, we always choose k_3 such that the modular reduction requires two steps (Section 5.3.3). The part greater or equal to $x^{d_{ext}}$ during the first step of reduction has to be reduced during the second step. This part is $S_{k_1} + S_{k_2} + S_{k_3}$. Naturally, when we compute the highest terms strictly less than $x^{d_{ext}}$, we compute the lowest terms of $R_{k_1} + S_{k_1} + S_{k_2} + S_{k_3}$ (because we extract R_0 and R_{k_1} by keeping useless bits on the last 128-bit block to be faster). In fact, we compute exactly $(R_{k_1} + S_{k_1} + S_{k_2} + S_{k_3}) \bmod x^{128 \cdot \lceil \frac{d_{ext}}{128} \rceil - d_{ext}} \cdot x^{d_{ext}}$ if $d_{ext} = 0 \bmod 8$. So, we can save some instructions when $k_3 - 1 \leqslant 128 \cdot \lceil \frac{d_{ext}}{128} \rceil - d_{ext}$. The extraction of the last block of R_{k_1} followed by a multiplication by $f_5 - x^{d_{ext}}$ generates $R_{k_1} + S_{k_1} + S_{k_2} + S_{k_3}$ in the first block of R_{k_1}, then we conclude the first step by extracting the remaining blocks of R_{k_1} (which was transformed into $R_{k_1} + S_{k_1} + S_{k_2} + S_{k_3}$). The technique of updating R during the modular reduction is classical (Algorithm 16). Secondly, the authors of [30] noted that the multiplication by $f_5 - x^{d_{ext}}$ can be optimized. By computing one time the operand ($R_{k_1} + S_{k_1} + S_{k_2} + S_{k_3}$ here) multiplied by x^{64}, we can avoid repeating an operation of alignment for each of three shifts. We note that the use of bytewise shifts does not require this alignment. Then, we remark that the alignment to 64 bits can be replaced by an alignment to t bits (respectively to $128 - t$ bits) for $t \in \{16, 32, 64\}$, thanks to shifts by block of t bits instead of 64 bits. This is possible when $k_i \neq 0 \bmod 8$ implies $k_i \in [\![0, t]\!]$ (respectively $k_i \in [\![128 - t, 128]\!]$) for $1 \leqslant i \leqslant 3$: the multiplication by x^{k_i} is coupled to the division of the aligned data by x^{t-k_i} (respectively x^{128-k_i}), both using shifts by block of t bits. In addition, this alignment is already computed if $k_3 = t$ (respectively $k_1 = 128 - t$) or if $t = d_{ext} \bmod 128$ (respectively $t = -d_{ext} \bmod 128$). For the latter, the input provides only $R_{k_1} \cdot x^{d_{ext}}$, but the previous method allows to replace R_{k_1} by $R_{k_1} + S_{k_1} + S_{k_2} + S_{k_3}$ without any extra computation. Thirdly, multiplying by $f_5 - x^{d_{ext}} = 1 + x^{k_1} + x^{k_2} + x^{k_3}$ can be accelerated when $k_3 = k_1 + k_2$. The authors of [4] remarked that $f_5 - x^{d_{ext}} = (1 + x^{k_1})(1 + x^{k_2})$, which requires two multiplications (or shifts) instead of three. This method is not compatible with the previous one, except for $t = \pm d_{ext} \bmod 128$ which saves only one alignment. Fourth, the methods used for trinomials can be adapted for pentanomials, by taking into account the previous techniques.

Selection of irreducible pentanomials. The parameters of Gui and $\mathtt{DualModeMS256^{V1}}$ require the use of irreducible pentanomials, because $d_{\mathtt{ext}}$ is a multiple of 8. This constraint allows to speed up the extraction of R_{k_1} from R by using bytewise shifts (PSRLDQ and PALIGNR instructions). We have chosen $x^{184} + x^{16} + x^9 + x^7 + 1$, $x^{312} + x^{128} + x^{15} + x^5 + 1$, $x^{448} + x^{64} + x^{39} + x^{33} + 1$ and $x^{544} + x^{128} + x^3 + x + 1$. For $d_{\mathtt{ext}} = 184$, we choose $k_3 = 16$ because it is a multiple of 8. This permits to improve the multiplication by x^{16} by using bytewise shifts (PSLLDQ and PALIGNR instructions). Moreover, we save two alignments (since $k_2 < 16$) thanks to shifts by block of 16 bits. For $d_{\mathtt{ext}} = 312$ and $d_{\mathtt{ext}} = 544$, we choose $k_3 = 128$. This improves the multiplication by x^{128}, which does not require shifts when the data are stored on 64-bit or 128-bit registers. For $d_{\mathtt{ext}} = 544$, the input provides $R_{k_1} \cdot x^{32}$, which is used for saving two alignments (since $k_2 < 32$) thanks to shifts by block of 32 bits. For $d_{\mathtt{ext}} = 448$, we choose $k_3 = 64$ because $d_{\mathtt{ext}}$ is a multiple of 64. The multiplication of R_{k_1} by x^{64} is already available in the input. Once that $S_{k_1} + S_{k_2} + S_{k_3}$ is added, we obtain the multiplication by $x^{k_3} = x^{64}$, which saves two alignments.

Results. In Table 9.10, we summarize the results for $d_{\mathtt{ext}} \in \{184, 312, 448, 544\}$, and we compare them to the modular reduction by trinomials for similar sizes (without specific optimizations depending on $d_{\mathtt{ext}}$ or k_1). We also propose implementations of the strategy using $k_3 = k_1 + k_2$. Both methods seem to have the same efficiency, and are approximately 23% slower than by using trinomials. For $d_{\mathtt{ext}} = 184$, the use of $k_3 = 16$ for saving two alignments is as efficient as $k_3 = k_1 + k_2$.

$(d_{\mathtt{ext}}, k_3, k_2, k_1)$	SSSE3	SSSE3	$(d_{\mathtt{ext}}, k_1)$
(184, 16, 9, 7)	13.9	10.5	(186, 11)
(184, 16, 9, 7)	13.9		
(312, 12, 9, 3)	18.6	14.9	(314, 15)
(312, 128, 15, 5)	18.8		
(448, 64, 39, 33)	22.7	18.4	(425, 12)
(544, 128, 3, 1)	27.5	24.0	(574, 13)

Table 9.10: Number of cycles to compute the modular reduction of an element of $\mathbb{F}_2[x]$ of degree $2d_{\mathtt{ext}} - 2$ by $x^{d_{\mathtt{ext}}} + x^{k_3} + x^{k_2} + x^{k_1} + 1$ or $x^{d_{\mathtt{ext}}} + x^{k_1} + 1$, with MQsoft and the SSSE3 instruction set. We use a Skylake processor (LaptopS). The bold values allow a speed-up due to the division by $x^{d_{\mathtt{ext}}}$ or the multiplication by x^{k_3}. We also exploit $(d_{\mathtt{ext}} \bmod 128) \in \{32, 64\}$.

9.2.5 Multi-Squaring in $\mathbb{F}_{2^{d_{\mathtt{ext}}}}$

The multi-squaring [159] is an operation computing successively several squarings. This operation is important to compute the inverse in $\mathbb{F}^{\times}_{2^{d_{\mathtt{ext}}}}$ (in Section 9.2.6), as well as the Frobenius map based on multi-squaring or on modular composition (Section 5.4.5). These algorithms require computing $B^{2^{k'}}$, for $B \in \mathbb{F}_{2^{d_{\mathtt{ext}}}}$ and various values of k'. This is exactly a computation of Frobenius map, where $H = f$ is the degree-$d_{\mathtt{ext}}$ field polynomial defining $\mathbb{F}_{2^{d_{\mathtt{ext}}}}$. For small values of k', the best way is to raise B to the power of two k' times (as in Table 9.7). For larger values of k', the best method is to use Algorithm 22 with $k = k'$ and with precomputed multi-squaring tables to save Step 1. Let $B = \sum_{j=0}^{d_{\mathtt{ext}}-1} b_j \ ^j \in \mathbb{F}_{2^{d_{\mathtt{ext}}}}$ for $\mathbb{F}_{2^{d_{\mathtt{ext}}}} = \mathbb{F}_2[\ ]/(f(\ ))$ (Section 9.1), then $B^{2^{k'}} = \sum_{j=0}^{d_{\mathtt{ext}}-1} b_j \ ^{j 2^{k'}} \bmod f$. The idea of multi-squaring tables is to store $\ ^{j 2^{k'}} \bmod f$ for

$0 \leqslant j \leqslant d_{ext}-1$. Then, multi-squaring is equivalent to the dot product of the vectors $(b_0, \ldots, b_{d_{ext}-1})$ and $(1, {}^{2^{k'}} \bmod f, \ldots, {}^{(d_{ext}-1)2^{k'}} \bmod f)$. The table requires storing $d_{ext}-1$ elements in $\mathbb{F}_{2^{d_{ext}}}$ (1 is not formally stored), and the multi-squaring requires $d_{ext}-1$ multiplications between elements of $\mathbb{F}_2$ and $\mathbb{F}_{2^{d_{ext}}}$, and $d_{ext}-1$ additions in $\mathbb{F}_{2^{d_{ext}}}$. In a variable-time implementation, the multiplication by b_j can be done by a conditional statement. In a constant-time implementation, the value of b_j is duplicated in the mask variable to replace the multiplication by a bitwise AND with this mask.

In a variable-time implementation, the performance can be improved with larger tables [118]. Instead of computing $b_j \, {}^{j2^{k'}} \bmod f$ coefficient by coefficient, the coefficients can be grouped by block of b, and the 2^b possibilities of $\sum_{j=0}^{b-1} b_{ib+j} \, {}^{(ib+j)2^{k'}}$ can be precomputed for $0 \leqslant i \leqslant \frac{d_{ext}}{b} - 1$. This method cannot be used in a constant-time implementation because of the timing attack on the memory latency (Section 4.7). It permits to attack the index of the lookup table. Moreover, the classical countermeasure (Section 6.3.3) makes this strategy exponentially slower in b, since each element of the 2^b-element table has to be used. So, we use a constant-time implementation of multi-squaring without larger tables. Note that this choice is different from setting $b=1$, since the latter requires storing d_{ext} extra null field elements.

9.2.6 Modular Inverse in $\mathbb{F}^{\times}_{2^{d_{ext}}}$

The computation of the inverse in $\mathbb{F}^{\times}_{2^{d_{ext}}}$ is often required for the arithmetic in $\mathbb{F}_{2^{d_{ext}}}[X]$. In our case, it is required to compute the GCD (Algorithm 18) and the split root finding (Section 5.4.7). To compute the modular inverse of $A \in \mathbb{F}^{\times}_{2^{d_{ext}}}$, there are mainly two methods (Section 5.4.2). The first is to use the constant-time extended Euclid–Stevin algorithm [19]. We have coded a first version of this algorithm (optimized for $\mathbb{F}_2$), but the latter seems naturally slow. Maybe an implementation of the fast version could be better. The second is to compute $A^{-1} = A^{2^{d_{ext}}-2}$ by Fermat's little theorem. The exponentiation can be done with the square-and-multiply method (Algorithm 14), costing $d_{ext}-1$ squarings and $d_{ext}-2$ multiplications in $\mathbb{F}_{2^{d_{ext}}}$. The Itoh–Tsujii Multiplicative Inversion Algorithm (ITMIA) [108] permits to modify the way to compute the power with an addition chain. It requires $d_{ext}-1$ squarings and only $O(\log_2(d_{ext}))$ multiplications in $\mathbb{F}_{2^{d_{ext}}}$. The number of multiplications depends on the length of the chosen addition chains.

Algorithm 36 ITMIA for a specific addition chain.

```
 1: function Inverse(A ∈ F^×_{2^d_ext})
 2:     m ← 1                                        ▷ Case i = ⌊log2(d_ext − 1)⌋.
 3:     A_K ← A                                      ▷ A^(2^m − 1) = A.
 4:     for i from ⌊log2(d_ext − 1)⌋ − 1 to 0 by −1 do
 5:         Q ← A_K^(2^m)                            ▷ Multi-squaring to obtain A^(2^(2m) − 2^m).
 6:         A_K ← Q × A_K                            ▷ A^(2^(2m) − 1).
 7:         m ← ⌊(d_ext − 1) / 2^i⌋
 8:         if m mod 2 = 1 then
 9:             A_K ← A_K^2 × A                      ▷ (A^(2^(m−1) − 1))^2 × A = A^(2^m − 1).
10:         end if
11:     end for                                      ▷ m = d_ext − 1 and so A_K = A^(2^(d_ext − 1) − 1).
12:     return A_K^2                                 ▷ A^(2^d_ext − 2) = A^(−1).
13: end function
```

ITMIA is described in Algorithm 36 for a specific addition chain which consists in reading the bits of $d_{ext}-1$ from left to right. It requires computing successive squarings. The multi-squaring can be computed more quickly with lookup tables (Section 9.2.5).

Algorithm 36 is useful because it automatically proposes an addition chain for all values of d_{ext}, but it is not always optimal. The choice of the best addition chain is not easy, because it depends on the performance of multiplication, squaring and multi-squaring. Moreover, there is a large set of addition chains. This problem is studied in [118], which proposes a software generating an efficient C++ inversion code. This software searches the addition chain which maximizes the performance of the generated code. However, the generator does not propose implementations of multi-squaring tables in constant-time. For the moment, MQsoft uses Algorithm 36, but we propose in Appendix C examples of addition chains chosen to minimize the number of field multiplications. We have improved Algorithm 36 with multi-squaring tables to compute the variable Q when i is zero or one. Because multi-squaring tables are huge, we use them only for the parameters of certain schemes as GeMSS, DualModeMS and Gui, and for the values of d_{ext} used to evaluate the performance of MQsoft. The corresponding file in MQsoft requires 4.5 MB. We could decrease the size. As for the FFT (Section 5.4.4), we could compute $B^{2^{k'}}$ as $\sum_{j=0}^{\lfloor \frac{d_{ext}-1}{2} \rfloor} b_{2j}^{\,j2^{k'+1}} + {}^{2^{k'}} \sum_{j=0}^{\lfloor \frac{d_{ext}}{2} \rfloor -1} b_{2j+1}^{\,j2^{k'+1}} \bmod f$. For one extra field multiplication, we only need the half of the table (which is used two times). Obviously, we could also use Horner's rule by block (Sections 5.4.3 and 9.5.3).

9.2.7 Performance of the Arithmetic in $\mathbb{F}_{2^{d_{ext}}}$

Table 9.11 compares the performance of arithmetic operations in our library with respect to several open source libraries (listed in Section 6.2.1). We choose the irreducible trinomial $f = x^{d_{ext}} + x^{k_1} + 1$ with $k_1 \in [\![2, 31]\!]$ to create the field $\mathbb{F}_{2^{d_{ext}}}$. All operations use modular reduction. We have measured the performance of FLINT, but the times are not relevant in our context. It turns out that for $d_{ext} = 252$, NTL is 100 to 200 times faster than FLINT. The main reason is that FLINT does not have a special implementation for binary fields. We have used the type fq_nmod_t which stores each element of $\mathbb{F}_{2^{d_{ext}}}$ as a polynomial in $\mathbb{F}_2[x]$ where each coefficient is stored on one word (instead of one bit). Magma is also taken into account. The results are not significant because Magma is slowed down by its user interface. We remark that the squarings and multiplications of NTL are faster for $d_{ext} = 126$ than for $d_{ext} = 62$. It can probably be explained by the fact that NTL does not use a trinomial for $d_{ext} = 62$. Our implementation is 3.5 to 4.5 times faster than NTL for multiplication and 5 to 6 times faster for squaring. We think that NTL is slowed down by its interface. For the inversion, the measurements are not comparable because NTL is not implemented in constant-time. However, we have a speed-up of two on average.

We now compare MQsoft to the constant-time arithmetic of [30], when trinomials are used to build $\mathbb{F}_{2^{d_{ext}}}$ (Table 9.12). In $\mathbb{F}_{2^{233}}$, they compute the squaring in 18 cycles and the multiplication in 38 cycles. We have approximately the same performance for $d_{ext} = 239$: 13.9–20.7 cycles for squaring and 34.7–39.8 cycles for multiplying (on LaptopS). MQsoft is slower with dependencies but faster with arrays. We note that the processor of [30] is a Haswell, which penalizes the performance, but is 31% faster than this of LaptopS. In $\mathbb{F}_{2^{409}}$, they compute the square in 28 cycles and the multiplication in 97 cycles. MQsoft is slightly faster, with 23.4–24.7 cycles for squaring and 78.1–81.2 cycles for multiplying. For the inversion, [30] is approximately two times slower. Our library takes advantage of the use of multi-squaring tables. We replace $d_{ext}-1$ squarings by approximately $\frac{1}{4}d_{ext}$ squarings and two multi-squarings.

(d_{ext}, k_1)	operation	Magma	NTL	MQsoft (PCLMULQDQ + AVX2)	
				dependencies	array
(62,29)	squaring	416	220	14.1	8.0
	mul	444	231	17.4	9.3
	inverse	13 183	1 868	×	1 154
(126,21)	squaring	440	105	18.5	10.5
	mul	494	124	26.4	15.2
	inverse	27 353	3 457	×	1 509
(186,11)	squaring	437	119	21.1	14.4
	mul	529	144	32.1	24.5
	inverse	40 200	4 918	×	2 281
(252,15)	squaring	455	128	22.3	15.5
	mul	558	169	39.8	34.7
	inverse	51 720	7 809	×	3 166
(314,15)	squaring	480	139	25.9	21.1
	mul	629	211	54.0	50.2
	inverse	66 220	9 515	×	4 345
(366,29)	squaring	490	150	26.9	20.4
	mul	653	238	66.3	63.8
	inverse	76 704	11 813	×	5 352
(425,12)	squaring	500	163	27.0	24.3
	mul	714	286	78.4	79.9
	inverse	94 846	14 974	×	6 562
(506,23)	squaring	510	174	28.1	24.2
	mul	761	320	84.7	87.4
	inverse	115 681	18 601	×	7 778
(574,13)	squaring	521	201	29.8	31.0
	mul	922	579	112.2	118.3
	inverse	129 266	23 185	×	11 467

Table 9.11: Number of cycles by operation in $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS).

d_{ext}	implementation	squaring	multiplication	inversion
233	[30]	18	38	6074
239	MQsoft	13.9–20.7	34.7–39.8	< 3166
409	[30]	28	97	15182
	MQsoft	23.4–24.7	78.1–81.2	6470

Table 9.12: Number of cycles by operation in $\mathbb{F}_{2^{d_{ext}}}$, for [30] (Haswell Core i7-4770 CPU at 3.4 GHz) and MQsoft (LaptopS).

9.3 Efficient Implementation of Root Finding over $\mathbb{F}_{2^{d_{ext}}}$

The most expensive part of the signature generation is to find the roots of a HFE polynomial $F \in \mathbb{F}_{2^{d_{ext}}}[X]$ as defined in Equation (2.6). F is a degree-D monic polynomial which is sparse because it has approximately $\frac{1}{2}\log_2(D)^2$ non-zero coefficients. We have chosen to implement Berlekamp's algorithm [161, Algorithm 14.15] which finds the roots with an asymptotic complexity[3] of $O\left(d_{ext}D^2 + (d_{ext} + \log(s))s^2\log(s)\right)$ operations in $\mathbb{F}_{2^{d_{ext}}}$, where s is the number of roots of F [161, Theorem 14.11 adapted for $r = s$ and $d = 1$]. For HFE polynomials, the factor $O\left(d_{ext}D^2\right)$ can be easily transformed into $O\left(d_{ext}D\log_2(D)^2 + D^2\right)$ operations in $\mathbb{F}_{2^{d_{ext}}}$, by using the sparse structure of F during the classical Frobenius map (Section 5.4.5). We can also transform it into $O\left(\log_2(d_{ext})D^{2.085} + d_{ext}D\right)$ operations in $\mathbb{F}_{2^{d_{ext}}}$ by using the modular composition coupled to Karatsuba's polynomial multiplication algorithm. Moreover, the HFE polynomial does not have many roots, so we can assume that s is negligible, yielding a final complexity of $O\left(d_{ext}D\log_2(D)^2 + D^2\right)$ or $O\left(\log_2(d_{ext})D^{2.085} + d_{ext}D\right)$ operations in $\mathbb{F}_{2^{d_{ext}}}$.

For the general polynomials, the author of [138] proposed the successive resultant algorithm (SRA). It requires $O\left(d_{ext}^3D^2 + d_{ext}^4\right)$ operations in $\mathbb{F}_2$ to find roots. Fast SRA requires $O\left(d_{ext}^2D + d_{ext}^3\right)$ operations in $\mathbb{F}_2$ with fast arithmetic. The step in $O\left(d_{ext}^4\right)$ (or $O\left(d_{ext}^3\right)$) can be precomputed for a fixed finite field. In comparison to Berlekamp, SRA is interesting only when the polynomial has many roots. In our case, HFE polynomials do not have many roots (Table 2.1). In [100] and [61], the root finding is improved for split and separable polynomials, when the cardinality of the multiplicative group ($2^{d_{ext}} - 1$ here) is smooth.

Improving root finding for sparse polynomials is a hard task. In [28], the authors proposed the first sub-linear (in q) algorithm which detects the existence of roots of t-monomials over $\mathbb{F}_q$. Its complexity is $4^{t+o(1)}q^{\frac{t-2}{t-1}+o(1)}$ bit operations. This method is not interesting for HFE polynomial because it is not sparse enough and because in practice d_{ext} is greater than the level of security. It costs approximately $4^{o(1)}2^{d_{ext}+o(d_{ext})}D^{\log_2(D)}$ bit operations in our case.

In this section, we study the performance of our implementation of Berlekamp's algorithm. For each operation required, we study different strategies and compare their practical performance to choose the best, in function of security parameters. Since the version of MQsoft used in [84], we have obtained important speed-ups. The results described in this section have been updated. In particular, we introduce the use of modular composition to speed up the Frobenius map, and the GCD is implemented in constant-time via the Euclid–Stevin algorithm.

9.3.1 Polynomial Squaring over $\mathbb{F}_{2^{d_{ext}}}$

The computations of Frobenius map (Section 5.4.5) and trace (Section 5.4.6) require computing repeated squarings in $\mathbb{F}_{2^{d_{ext}}}[X]$. As explained in Section 5.1.3, squaring is linear. For a D-coefficient polynomial, this operation requires D squarings in $\mathbb{F}_{2^{d_{ext}}}$. This fact is confirmed in practice (Table 9.13). For $d_{ext} = 175$ and $k_1 = 16$, squaring in $\mathbb{F}_{2^{d_{ext}}}$ requires approximately 9.7 cycles (Table 9.9). The cost of squaring in $\mathbb{F}_{2^{d_{ext}}}[X]$ is approximately $9.7 \cdot D$ cycles. For $D = 4097$ the performance is rather $19.4 \cdot D$ cycles. It is due to the size of the input, which is greater than the L1 cache (32 KiB). This generates cache misses (Section 6.1.2).

[3] More generally, the complexity is $O((d_{ext} + \log(D))M(D) + (d_{ext} + \log(s))M(s)\log(s))$ operations in $\mathbb{F}_{2^{d_{ext}}}$.

9.3.2 Polynomial Multiplication over $\mathbb{F}_{2^{d_{ext}}}$

Fast algorithms which are presented in Sections 5.1.4, 5.4.1, 5.4.3 and 5.4.5 require fast polynomial multiplications. As explained in Section 5.1.2, the main fast multiplication in binary fields is Karatsuba's method. In `MQsoft`, we propose e cient implementations of the classical multiplication and Karatsuba's algorithm over $\mathbb{F}_{2^{d_{ext}}}$.

Optimized implementation of Karatsuba's algorithm. Let be the number of coe cients of the input polynomials. Karatsuba's multiplication algorithm is well-known when is a power of two [161, Algorithm 8.1], and can be easily extended for all degrees (Section 5.1.2). We allocate an array before the first call to Karatsuba's algorithm to save a subquadratic number of memory allocations. Here, the order of recursive calls is important to minimize the required memory. For each call to Karatsuba's algorithm, we store the result of the first two recursive calls (P_1 and P_2) in the original output. Then, the third recursive call requires storing its result (P_3), as well as the sum of the lower part and higher part of each input. We use the first half of this array for this. For the three recursive calls, we use the second half as a new array in the recursive call.

When is a power of two, an allocation of 4 coe cients for the array is enough. Else, we need to allocate some extra coe cients. Each recursive call requires one extra coe cient when the current number of coe cients is odd ($\frac{}{2} = \frac{}{2} + 1$). Thus, we assume the worst case. We consider $= 2^{30} - 1$ as a reasonable maximum value, which corresponds to the case where all recursive calls have an odd number of coe cients. Each recursive call requires one extra coe cient to store both sums of half-inputs, as well as for the result. So we need three extra coe cients during a recursive call. For 30 recursive calls, we require allocating 90 coe cients. So, we upper bound the allocation size of the array by $4 + 90$. Of course, this bound can be adapted in function of the Hamming weight of , and in function of the threshold where the classical multiplication is called instead of Karatsuba's algorithm. In our implementation, we use an inlined classical multiplication for polynomials having a degree strictly less than four.

Minimizing the number of modular reductions. For the classical multiplication and Karatsuba's algorithm, we perform multiplications in $\mathbb{F}_{2^{d_{ext}}}$ without the field modular reduction, and we accumulate the results (as a dot product over $\mathbb{F}_{2^{d_{ext}}}$). Finally, each coe cient of the output is reduced, when it is necessary. Indeed, in some contexts, we can also accumulate products in $\mathbb{F}_{2^{d_{ext}}}[X]$, and accumulate their coe cients in $\mathbb{F}_{2^{d_{ext}}}$ without the field modular reduction (as for a dot product over $\mathbb{F}_{2^{d_{ext}}}[X]$). This strategy doubles the cost of additions, as well as this of the memory allocation of coe cients for Karatsuba's algorithm. In return, we avoid a quadratic (or subquadratic) number of modular reductions, which accelerates the polynomial multiplication.

About multiplications of di erent degree operands. Sometimes, fast algorithms require multiplications of two polynomials having di erent degrees (Section 5.4.1). To multiply A, B in $\mathbb{F}_{2^{d_{ext}}}[X]$ respectively of degree d_a, d_b such that $d_a \geqslant d_b$, we just split A into blocks of size $d_b + 1$ in order to apply the multiplication of each block by B. In practice, it is slightly faster to compute the product of the last block with a recursive call, since the latter can have less than $d_b + 1$ coe cients.

Results. We summarize in Table 9.13 the performance of the multiplication of two equal-degree polynomials. We recall that the cost of one multiplication in $\mathbb{F}_2[x]$, for polynomials requiring three 64-bit words, is approximately 15.5 cycles (Table 9.4). Our multiplications are very e cient. The

classical multiplication costs approximately $15.5 \cdot D^2$ cycles, whereas Karatsuba's multiplication algorithm costs approximately $15.5 \cdot 3D^{\log_2(3)}$. Karatsuba's algorithm is always faster than the classical multiplication. We use it to have an efficient modular composition (Section 9.3.4). NTL is on average four times slower than MQsoft. We have also tested the multiplication of Magma. The latter is on average three times slower than NTL.

d_{ext}	D	squaring		classical mul.		Karatsuba's mul.	
		MQsoft	NTL	MQsoft	NTL	MQsoft	NTL
175	17	0.000179	0.00219	0.00564	0.0246	0.00492	0.0209
	129	0.00131	0.0163	0.277	0.410	0.113	1.16
	513	0.00515	0.0691	4.34	19.8	0.989	3.60
	4097	0.0811	0.528	275	1170	25.9	87.2

Table 9.13: Number of megacycles to compute the multiplication of two D-coefficient polynomials over $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS).

9.3.3 Polynomial Euclidean Division over $\mathbb{F}_{q^{d_{ext}}}$ and Sparse Divisors

In this section, we generalize our results for $\mathbb{F}_{q^{d_{ext}}}$ instead of $\mathbb{F}_{2^{d_{ext}}}$. During the root finding algorithm (Algorithm 25), the Frobenius map can be computed with d_{ext} q-exponentiations in $\mathbb{F}_{q^{d_{ext}}}[x]/(H)$ (Section 5.4.5). In characteristic two, this corresponds to $d_{ext}\log_2(q)$ modular squarings. A square has the property that all odd degree terms are null. We propose to exploit this property to speed up the modular reduction by H. Moreover, the HFE-based schemes require finding the roots of a HFE polynomial. In this case, H is a HFE polynomial, which is sparse. Here, we study how to exploit the structure of H, then we propose to modify it to improve the Euclidean division of a square by H. In this section, we assume that we want to perform the Euclidean division of a d_a degree-polynomial A by a degree-d_h polynomial H over $\mathbb{F}_{q^{d_{ext}}}$. When H is a HFE polynomial, we call D its degree. Except in Algorithm 37 which introduces the classical way to perform a sparse modular reduction, the results are specialized in characteristic two. Here, we list the use of the modular reduction by a HFE polynomial during the signing process:

- during the Frobenius map (Section 5.4.5) using the classical repeated squaring algorithm,
- during the Frobenius map using the modular composition by a HFE polynomial (Algorithm 21),
- during the generation of the multi-squaring table in Algorithm 22.

Classical Euclidean division by a sparse polynomial. The classical Euclidean division (Algorithm 16) is naturally faster when the divisor is sparse. Let K be the number of non-zero coefficients of H. For a fixed i, each term of $H - h_{d_h} \cdot x^{d_h}$ is multiplied by $-q_{i-d_h} \cdot x^{i-d_h}$. Then, the result is added to R. When H is monic, one step of Algorithm 16 requires $K-1$ field multiplications and additions. So, the total complexity is $(d_a - d_h + 1)(K-1)$ field multiplications and additions, and the number of field modular reductions can go down to $d_a + 1$ with the accumulator principle. When H is not monic, Algorithm 16 costs $d_a - d_h + 1$ additional field multiplications and one inverse in $\mathbb{F}^{\times}_{q^{d_{ext}}}$. This additional cost can be avoided by precomputing $h_{d_h}^{-1} \cdot H$.

During the signing process of HFEv−-based schemes, we reduce a product or a square modulo H. So, we can consider $d_a = 2d_h - 2$, implying a cost of $(d_h - 1)(K - 1)$ field multiplications and additions for each modular reduction.

Algorithm 37 Polynomial Euclidean division of A by a degree-D HFE polynomial over $\mathbb{F}_{q^{d_{ext}}}$.

1: **function** EuclideanDivRemHFE($A \in \mathbb{F}_{q^{d_{ext}}}[x]$, $H \in \mathbb{F}_{q^{d_{ext}}}[x]^*$)
2: $\quad c \leftarrow \text{LeadingCoefficient}(H)^{-1}$
3: $\quad Q \leftarrow 0$
4: $\quad R \leftarrow A$
5: $\quad$ **for** k from d_a to D by -1 **do**
6: $\quad\quad q_{k-D} \leftarrow r_k \cdot c$
$\quad\quad$ ▷ Sparse computation of $R - q_{k-D} \cdot \left(H - c^{-1} \cdot x^D\right) \cdot x^{k-D}$.
7: $\quad\quad R \leftarrow R - q_{k-D} \cdot C \cdot x^{k-D}$ ▷ Constant term of H.
8: $\quad\quad$ **for** i from 0 to $\lfloor \log_q(D) \rfloor - 1$ **do**
9: $\quad\quad\quad R \leftarrow R - q_{k-D} \cdot B_i \cdot x^{q^i} \cdot x^{k-D}$ ▷ Linear term of H.
10: $\quad\quad\quad$ **for** j from 0 to i **do** ▷ Quadratic terms of H.
11: $\quad\quad\quad\quad R \leftarrow R - q_{k-D} \cdot A_{i,j} \cdot x^{q^i + q^j} \cdot x^{k-D}$
12: $\quad\quad\quad$ **end for**
13: $\quad\quad$ **end for**
14: $\quad\quad i \leftarrow \lfloor \log_q(D) \rfloor$
15: $\quad\quad$ **if** $D \neq q^i$ **then** ▷ In this part, the leading term is avoided.
16: $\quad\quad\quad R \leftarrow R - q_{k-D} \cdot B_i \cdot x^{q^i} \cdot x^{k-D}$ ▷ Linear term of H.
17: $\quad\quad\quad$ **for** j from 0 to $\log_q\left(D - q^i\right) - 1$ **do** ▷ Quadratic terms of H.
18: $\quad\quad\quad\quad R \leftarrow R - q_{k-D} \cdot A_{i,j} \cdot x^{q^i + q^j} \cdot x^{k-D}$
19: $\quad\quad\quad$ **end for**
20: $\quad\quad$ **end if**
21: $\quad\quad r_k \leftarrow 0$ ▷ The new R has a degree at most $k - 1$.
22: $\quad$ **end for**
23: $\quad$ **return** (Q, R)
24: **end function**

When H is a HFE polynomial (Equation (2.6)), we obtain Algorithm 37. Since $K = O\left(\log_q(D)^2\right)$ coefficients, this Euclidean division requires $O\left(D \log_q(D)^2\right)$ field operations (by considering $d_a = 2D - 2$ here). This bound is much better than the $O\left(D^2\right)$ field operations required by the dense divisors. It seems hard to propose better algorithms. Firstly, the fast Euclidean division of dense inputs cannot be better than $O(D)$ field operations. The complexity of the naive approach is already close to this bound, and moreover, the fast multiplications (in particular the FFT) are slower in characteristic two. Secondly, the fast algorithms do not exploit the structure of inputs, in general. Therefore, we propose some methods to improve the number of operations by constant factors.

Improving the Euclidean division of a square by special HFE polynomials. Here, we exploit the fact that the dividend is a square to improve Algorithm 37 (for $d_a = 2D - 2$). All terms of odd degree are null. We show that the complexity can be divided by a maximum factor of two. This factor depends on the largest odd degree such that the corresponding term in the divisor is non-null. We introduce a new notation to define such an integer. Let Q be a univariate polynomial.

We denote by $\mathcal{D}(Q)$ the largest odd integer i such that the degree-i term of Q is not null (i.e. $q_i \neq 0$). If it does not exist, we set $\mathcal{D}(Q) = -\infty$. The following lemma permits to demonstrate the main result (Theorem 10) of this part.

Lemma 8. Let $A \in \mathbb{F}_{q^{d_{ext}}}[x]$ be a polynomial of degree at most $2D-2$, $H \in \mathbb{F}_{q^{d_{ext}}}[x]$ be of degree D, $Q, R \in \mathbb{F}_{q^{d_{ext}}}[x]$ be respectively the quotient and remainder of the Euclidean division of A by H and $d = \mathcal{D}(H)$. If $\mathcal{D}(A) = -\infty$ and if D is even, then $\mathcal{D}(Q) \leqslant d-2$.

Proof. Let $H = \sum_{j=0}^{D} h_j x^j$ and $Q = \sum_{j=0}^{D-2} q_j x^j$. By definition of $\mathcal{D}$, $\mathcal{D}(Q) \leqslant d-2$ is equivalent to $q_i = 0$ for all odd i such that $i > d-2$. By definition of Q, $q_i = 0$ for $i < 0$ and $i > D-2$, so we show the lemma for the values of q_i such that $D-2 \geqslant i > \max(-1, d-2)$.

To do it, we use a proof by induction on an odd j such that $D-1 \geqslant j > \max(-1, d-2)$. The base case, $j = D-1$, is trivial since $q_j = 0$ for $j > D-2$. Now, assume that $q_k = 0$ for all odd k such that $D-1 \geqslant k > j > \max(-1, d-2)$, and consider the $\mathrm{coef}_j(H)$ notation which corresponds to the coefficient of the power j of x in H. To show that $q_j = 0$, firstly we show these two properties:

(1) $\mathrm{coef}_{D+j}(HQ) = 0$,

(2) $\mathrm{coef}_{D+j}(HQ) = q_j h_D$.

Proof of these two properties:

(1) By definition, $A = HQ + R$ and so $A - R = HQ$. Because $\mathcal{D}(A) = -\infty$ by hypothesis, $\mathcal{D}(A-R) = \mathcal{D}(R) \leqslant D-1 < D+j$ and $D+j$ is odd so $\mathrm{coef}_{D+j}(A-R) = 0$.

(2) $HQ = \sum_{r=0}^{2D-2} \sum_{\ell=0}^{r} q_\ell h_{r-\ell} x^r$, so $\mathrm{coef}_{D+j}(HQ) = \sum_{\ell=0}^{D+j} q_\ell h_{D+j-\ell}$. But $q_\ell = 0$ for $\ell > D-2$ and $h_{D+j-\ell} = 0$ for $D+j-\ell > D$, so $\mathrm{coef}_{D+j}(HQ) = \sum_{\ell=j}^{D-2} q_\ell h_{D+j-\ell}$. When $\ell > j$ is odd, $q_\ell = 0$ by induction hypothesis. When ℓ is even, $h_{D+j-\ell} = 0$ because $D+j-\ell$ is odd and $\mathcal{D}(H) = d < D+j-\ell$. So $\sum_{\ell=j}^{D-2} q_\ell h_{D+j-\ell} = q_j h_D$.

These two properties imply $\mathrm{coef}_{D+j}(HQ) = q_j h_D = 0$. Because $h_D \neq 0$, this implies that $q_j = 0$. □

We can now demonstrate Theorem 10.

Theorem 10. Let H be a HFE polynomial of degree D in $\mathbb{F}_{q^{d_{ext}}}[x]$ where the s terms of highest odd degree have been removed ($0 \leqslant s \leqslant \lceil \log_q(D) \rceil$), and let $A \in \mathbb{F}_{q^{d_{ext}}}[x]$ be a square of degree at most $2D-2$. If D and q are even, then the computation of the classical Euclidean division (Algorithm 37) of A by H can be accelerated by a factor $(D-1)/\left(\frac{D}{2} + \lfloor q^{\lceil \log_q(D) \rceil - s - 2} \rfloor\right)$.

Proof. During Algorithm 37, A is a square and q is even so $\mathcal{D}(A) = -\infty$, and so Lemma 8 can be applied. Let $d = \mathcal{D}(H)$, the iterations for k odd and strictly greater than $D+d-2$ can be removed because $q_{k-D} = 0$. So, the number of iterations for k odd is $\max\left(\frac{(D+d-2)-(D+1)}{2} + 1, 0\right) = \max\left(\frac{d-1}{2}, 0\right)$, whereas the number of iterations for k even is $\frac{D}{2}$. So, Algorithm 37 can be used with $\max\left(\frac{D+d-1}{2}, \frac{D}{2}\right)$ iterations.
Then, H is a HFE polynomial and q is even, so $d = -\infty$, $d = 1$ or $d = q^i + 1$ for $i > 0$. Assume $s < \lceil \log_q(D) \rceil$, implying $d > 0$. By removing the degree-$(q^i + 1)$ terms for i from $\lceil \log_q(D) \rceil - 1$ to $\lceil \log_q(D) \rceil - s$ by -1, d is equal to 1 or $q^{\lceil \log_q(D) \rceil - s - 1} + 1$. This implies that the number of iterations can be written as $\frac{D}{2} + \lfloor q^{\lceil \log_q(D) \rceil - s - 2} \rfloor$. Note that the floor of $q^{\lceil \log_q(D) \rceil - s - 2}$ is useful only if $d \leqslant 1$, which makes $\frac{D}{2}$ iterations. This number cannot be smaller: all odd degree terms of the quotient

are null. Now, assume that $s = \lceil \log_q(D) \rceil$, implying $d = -\infty$. As previously stated, we cannot do better than $\frac{D}{2}$ iterations. So, the previous formula is still correct.
Finally, Algorithm 37 requires $D-1$ iterations, so the proposed modification accelerates it of a factor $(D-1)/\left(\frac{D}{2} + \lfloor q^{\lceil \log_q(D)\rceil - s - 2} \rfloor\right)$. This factor is at most two. □

Let K be the number of terms of the HFE polynomial (without removed terms), and s be the number of removed terms. For $s = 0$, the modular reduction costs $(D-1)(K-1)$ multiplications and additions in $\mathbb{F}_{q^{d_{ext}}}$, whereas by removing terms (with an even value of D), the cost is $\max\left(\frac{D+d-1}{2}, \frac{D}{2}\right)(K-1-s)$ field multiplications and additions. The main gain is due to the smaller number of loop rounds during Algorithm 37, which is given by Theorem 10. However, there is also a slight speed-up generated by the fact that terms are removed. Algorithm 37 has to be slightly modified to take into account this remark (j can start to one instead of zero when i is greater or equal to $\lceil \log_q(D) \rceil - s$).

Security and performance. When H has no term of odd degree, we obtain that during the computation of $X^{q^{d_{ext}}} \bmod H$, no odd degree term appears because $R = A - HQ$ and A, H and Q do not have odd degree terms. This result allows to perform computations only for even degree terms, dividing by two the cost of the squaring and this of the modular reduction. But in practice, removing all terms of odd degree of the HFE polynomial decreases the security. By using the Frobenius sustainer [165], there exists an equivalent key between $H(x^{\frac{1}{2}})$ and $H(x)$ for $q = 2$. So, the security is equivalent to a degree $\frac{D}{2}$ HFE polynomial. We will show in Table 9.14 that in this case, D has to be multiplied by two to obtain the original security. We note that for $q > 2$, if we remove all odd degree terms as well as X^2, then there exists an equivalent key between $H(x^{\frac{1}{q}})$ and $H(x)$. This fact implies a security corresponding to a degree $\frac{D}{q}$ HFE polynomial.

Table 9.14 studies the impact of $d = \mathcal{D}(H)$ on both the security and the theoretical speed-up over the classical Euclidean division compared to the case $D = 513$. We have done an experimental test to analyze the degree of regularity [82, 11, 14, 22] (Section 4.4.1) in function of the number of removed terms. We measure it during the Gröbner basis attack on HFE for $d_{ext} = m = 30$. We observe that removing a small number of odd degree terms appears not to affect the security. The security decreases as soon as eight terms are removed. The results confirm that the security to attack H of degree D without odd degree terms is the same as attacking a HFE polynomial of degree $\frac{D}{2}$: the degree of regularity increases between $\frac{D}{2} = 512$ and $\frac{D}{2} = 513$. The case $d = 1$ seems to have the same behavior, but in the general case, the degree of regularity does not necessarily decrease (for $D = 130$, the degree of regularity is 4 for $d = -\infty$ but 5 for $d = 1$, cf. Table 9.15).

The column speed-up on k corresponds to the obtained speed-up by decreasing the number of iterations during Algorithm 37. The other speed-up column is the overall speed-up, which uses the fact that removing terms decreases the number of multiplications for one iteration. Removing the highest terms generates the main part of the overall speed-up. In Section 7.8.9, we propose to choose $s = 3$ for $D \geqslant 128$.

Degree of regularity. We have measured the D_{reg}^{Exp} observed in practice for HFE in function of s. The results are summarized in Table 9.15. When s is small, the degree of regularity is not impacted. For the largest values of s, the degree of regularity decrements. As soon as D is multiplied by two, we have observed that the degree of regularity does not decrement anymore.

D	d	removed terms	D_{reg}^{Exp}	nb. of it.	speed-up on k	speed-up
512	257	none	5	384	25%	27%
513	513	none	6	512	reference	ref.
514	257	x^{513}	6	385	25%	25%
	129	x^{513}, x^{257}		321	37%	39%
	65	$x^{513}, x^{257}, x^{129}$		289	44%	46%
	33	$x^{513}, \dots, x^{65}$		273	47%	50%
	17	$x^{513}, \dots, x^{33}$		265	48%	53%
	9	$x^{513}, \dots, x^{17}$		261	49%	54%
	5	$x^{513}, \dots, x^{9}$		259	49%	56%
	3	$x^{513}, \dots, x^{5}$	5	258	50%	57%
	1	$x^{513}, \dots, x^{3}$		257	50%	58%
	$-\infty$	all odds		257	50%	59%
1024	1	$x^{513}, \dots, x^{3}$	5	512	0%	0%
	$-\infty$	all odds		512	0%	2%
1026	1	$x^{1025}, \dots, x^{3}$	6	513	0%	-4%
	$-\infty$	all odds		513	0%	-2%

Table 9.14: Impact of $d = \mathcal{D}(H)$ on both the cost of Algorithm 37 and D_{reg}^{Exp} the degree of regularity of the corresponding HFE algebraic system of 30 equations in 30 variables over $\mathbb{F}_2$.

minimum m	D	s	D_{reg}^{Exp}
$\geqslant 9$	17	0	4
$\geqslant 15$	18	$s \leqslant 3$	4
$160 \geqslant m \geqslant 5$		$4 \leqslant s \leqslant 5$	3
$\geqslant 16$	129	0	5
$\geqslant 16$	130	$s \leqslant 5$	5
$\geqslant 18$		6	5
$\geqslant 23$		7	5
$70 \geqslant m \geqslant 9$		8	4
$\geqslant 24$	513	0	6
$\geqslant 24$	514	$s \leqslant 6$	6
$\geqslant 25$		7	6
$35 \geqslant m \geqslant 16$		$8 \leqslant s \leqslant 10$	5
$\geqslant 32$	4097	0	7
$\geqslant 32$	4098	$s \leqslant 10$	7
$\geqslant 33$		11	7
$35 \geqslant m \geqslant 24$		$12 \leqslant s \leqslant 13$	6

Table 9.15: Degree of regularity in the case of HFE algebraic systems of m equations in m variables over $\mathbb{F}_2$, in function of s. The maximum value of s is $\lceil \log_2(D) \rceil$.

About the MinRank attacks. The security of HFE against the Kipnis–Shamir attack (Section 4.6.3) seems not to be impacted by the parameter s. Compared to a plain HFE polynomial (i.e. $s = 0$), we set to zero the last s coefficients in the first column of F (Figure 9.2). However, the first coefficient of F corresponds to X^2 which has an even degree, so the rank does not decrease. We also remark that the last row of F is not null, since the leading coefficient corresponding to X^{2^4+2} is present.

$$F = \begin{pmatrix} * & 0 & 0 & 0 & 0 \\ * & * & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 \\ 0 & * & * & * & 0 \\ 0 & * & 0 & 0 & 0 \end{pmatrix}$$

Figure 9.2: Example of (truncated) matrix $F \in \mathcal{M}_5\left(\mathbb{F}_{2^{d_{\mathrm{ext}}}}\right)$ for $D = 18$ and $s = 3$. The three removed coefficients are in bold.

Removing even degree terms in characteristic two. We can also remove some even degree terms. Assume $D = q^\ell + 1$ for $\ell > 0$. Thus, by removing the terms of highest even degree, a part of even degree terms in the quotient becomes null. Similarly to the $\mathcal{D}$ notation, we introduce $\mathcal{D}_e$, which corresponds to the largest even integer instead of odd integer in the definition of $\mathcal{D}$.

Lemma 9. *Let $A \in \mathbb{F}_{q^{d_{\mathrm{ext}}}}[X]$ be a polynomial of degree at most $2D - 2$, $H \in \mathbb{F}_{q^{d_{\mathrm{ext}}}}[X]$ be of degree D, $Q, R \in \mathbb{F}_{q^{d_{\mathrm{ext}}}}[X]$ be respectively the quotient and remainder of the Euclidean division of A by H and $d = \mathcal{D}_e(H)$. If $\mathcal{D}(A) = -\infty$ and if D is odd, then $\mathcal{D}_e(Q) \leqslant d - 2$.*

Proof. The proof (Section B.10) is similar to this of Lemma 8. The fundamental idea is to remark that the first odd degree term appearing in the current remainder during the Euclidean division is X^{D-2+d}. To make it vanish, we use the term X^{d-2} in the quotient, which will be the first even degree term appearing. So, terms of odd degree strictly greater than $D - 2 + d$ are null in the current remainder, implying terms of even degree strictly greater than $d - 2$ are null in the quotient. □

With Lemma 9, we obtain that at most $\max\left(\frac{D+d-1}{2}, \frac{D}{2}\right)$ terms of the quotient are not null, as when we removed odd degree terms. However, the default of this method is the d value. Whereas by removing one odd degree term (with D even), $\mathcal{D}(H) = q^{\lceil \log_q(D) \rceil - 2} + 1$, we need removing $\lceil \log_q(D) \rceil - 1$ terms to obtain $\mathcal{D}_e(H) = q^{\lceil \log_q(D) \rceil - 2}$. So, it is faster to remove odd degree terms with an even D. Moreover, removing even degree terms could be dangerous for the security, especially because the term $X^{q^{\lfloor \log_q(D) \rfloor}}$ corresponds to vinegar variables in the HFEv polynomial.

Results. In `MQsoft`, we have implemented in constant-time the classical Euclidean division and its fast version (Section 5.1.4). Then we have implemented Algorithm 37 which exploits the sparse structure of HFE polynomials. In Table 9.16, we compare these methods. The classical Euclidean division requires approximately D^2 multiplications, like the classical multiplication. So, times are similar to these of Table 9.13. For the fast Euclidean division, we do not take into account the time to generate the inverse of the reciprocal polynomial of the divisor. In our practical applications, the latter can be computed one time for all modular reductions. So, the fast Euclidean division requires the lower half of the result of two multiplications. Except for $D = 17$, we use Karatsuba's multiplication algorithm. The obtained timings correspond approximately to two times these of

Table 9.13. Then, when the dividend is a square, one of the two multiplications has a square as operand. We take advantage of this in our implementation of Karatsuba's algorithm, which divides (approximately) by two the cost of the corresponding multiplication. The obtained timings correspond to 1.5 times these of Table 9.13, which is consistent with our improvement.

Finally, we compare our general Euclidean divisions with the division by a sparse HFE polynomial. The latter is very e cient because it requires approximately $\frac{1}{2}D\log_2(D)^2$ multiplications, and a speed-up of approximately 44% is obtained by removing three terms in the HFE polynomial. We do not succeed to obtain a speed-up with the fast Euclidean division. This is explained by the use of Karatsuba's multiplication algorithm. The cost of the latter is not quasi-linear, compared to the $\frac{1}{2}D\log_2(D)^2$ multiplications of the sparse Euclidean division.

d_{ext}	D	Euclidean div.	fast Euclidean div.		HFE Euclidean div.	
			div.	div. sqr.	div.	div. sqr.
175	17	0.00509	0.00575	0.00465	0.00392	×
	129	0.274	0.214	0.171	0.0700	0.0383
	513	4.28	1.92	1.54	0.436	0.237
	4097	274	51.1	41.0	5.81	3.30

Table 9.16: Number of megacycles to compute the remainder of the Euclidean division of a degree-$(2D-2)$ polynomial by a degree-D monic polynomial over $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS). In the div. sqr. column, we consider the division of a square. For the division by a HFE polynomial, we increment D and consider $s=3$.

9.3.4 Modular Composition of Polynomials over $\mathbb{F}_{2^{d_{ext}}}$

We have implemented our adaptation (Algorithm 20) of the modular composition of polynomials of Brent and Kung [42] over $\mathbb{F}_{2^{d_{ext}}}$, without using matrix product over $\mathbb{F}_{2^{d_{ext}}}$. Coupled to our e cient implementation of Karatsuba's multiplication algorithm (Section 9.3.2), we obtain the results described in Table 9.17. Our modular composition is on average seven or eight times faster than the `CompMod` function from `NTL`. As for the polynomial multiplication over $\mathbb{F}_{2^{d_{ext}}}$, `Magma` is three times slower than `NTL` (we used the `ModularComposition` function). Finally, we have studied the performance for $s=3$, meaning we remove the terms of degree 33, 65 and 129 (respectively 129, 257 and 513) in the `HFEv` polynomial for $D=130$ (respectively $D=514$). We obtain a very small speed-up, due to the improvement of the modular reduction of polynomials over $\mathbb{F}_{2^{d_{ext}}}$ when operands are squares (Section 9.3.3).

9.3.5 Frobenius Map in $\mathbb{F}_{2^{d_{ext}}}[X]$

The core of Algorithm 25 (Section 5.4.8) is to compute $X^{2^{d_{ext}}} \bmod H$. In Section 5.4.5, we show that this computation can be done with the classical repeated squaring algorithm, using or not multi-squaring tables as in Section 9.2.5. The main di erence with Section 9.2.5 is that the coe cients are in $\mathbb{F}_{2^{d_{ext}}}$ instead of $\mathbb{F}_2$. So, the tables are too large to be precomputed. However, they can be computed more quickly when H is a `HFE` polynomial. Then, these two strategies can be embedded with the use of modular composition (Algorithm 21). We can replace the last $\frac{d_{ext}}{2}$ squarings by a modular composition, coupled to a $2^{\frac{d_{ext}}{2}}$-exponentiation of the coe cients. For the first $\frac{d_{ext}}{2}$

squarings, we can repeat this process recursively by replacing the half of squarings by a modular composition, as done previously. So, an efficient modular composition allows interesting speed-ups on the computation of the Frobenius map. For $d_{\texttt{ext}} \leqslant 576$ and $D \leqslant 514$, we use up to three modular compositions to speed up the repeated squaring algorithm.

$d_{\texttt{ext}}$	D	s	modular composition			
			`Magma`	`NTL`	`MQsoft`	(b)
175	17	0	1.07	0.304	0.0474	(4)
	129	0	64.0	26.4	3.49	(8)
	130	3	62.8	30.8	3.27	(11)
	513	0	510	444	56.5	(16)
	514	3	499	501	53.0	(20)
	4097	0	94300	30400	3590	(50)
	4098	3	93900	30600	3400	(50)

Table 9.17: Number of megacycles to compute the modular composition of two D-coefficient polynomials modulo a monic `HFE` polynomial of degree D over $\mathbb{F}_{2^{d_{\texttt{ext}}}}$. We use a Skylake processor (LaptopS). (b) corresponds to the chosen number of blocks for Algorithm 20. For $D = 33$, we recommend $b = 4$.

Let $\mathcal{C}_F$ be the cost of the Frobenius map in $\mathbb{F}_{q^{d_{\texttt{ext}}}}[X]$. When the input is an `HFE` polynomial, we have three possible complexities, which correspond respectively to the classical Frobenius map, this based on the modular composition, and finally the Frobenius map using multi-squaring tables:

$$\mathcal{C}_F = \begin{cases} O\left(\left(d_{\texttt{ext}} - \lceil \log_q(D) \rceil\right) \cdot \log_2(q) \cdot D \log_q(D)^2\right) & \text{operations in } \mathbb{F}_{q^{d_{\texttt{ext}}}}, \\ O\left(\log_2(d_{\texttt{ext}}) \cdot D^{2.085} + d_{\texttt{ext}} \cdot \log_2(q) \cdot D\right) & \text{operations in } \mathbb{F}_{q^{d_{\texttt{ext}}}}, \\ O\left(q^{k'} \cdot D \log_q(D)^2 + \frac{d_{\texttt{ext}}}{k'} \cdot D \cdot \left(D + k' \log_2(q)\right)\right) & \text{operations in } \mathbb{F}_{q^{d_{\texttt{ext}}}}. \end{cases}$$

These complexities are obtained from Section 5.4.5, by replacing the cost of the modular reduction by this of Algorithm 37. The factor $D^{2.085}$ corresponds to $O(D^{0.5})$ uses of Karatsuba's polynomial multiplication algorithm over $\mathbb{F}_{q^{d_{\texttt{ext}}}}$.

Table 9.18 summarizes the performance of both strategies for the Frobenius map, and compares our implementation to `NTL` and `Magma`. We use the `Modexp` function from `Magma` and the `PlainFrobeniusMap` and `ComposeFrobeniusMap` functions from `NTL`, both computing $X^{2^{d_{\texttt{ext}}}}$ mod H. We have also studied the strategy from Section 9.3.3 which permits to improve the Frobenius map by removing odd degree terms in the `HFE` polynomial. We remove three terms when $D = 130$ or $D = 514$. The results confirm the theoretical speed-ups: when the classical Frobenius map is used without the modular composition, `MQsoft` saves approximately 44% of computations by removing three terms. `Magma` is also improved by this trick, probably because it also uses the classical Euclidean division, and does not compute multiplication by zero. It is not the case for `NTL`, because it uses the fast Euclidean division (Algorithm 17).

The multi-squaring strategy is the fastest when D is small compared to $d_{\texttt{ext}}$, and is improved thanks to modular composition. When D is larger, we find a threshold where the classical Frobenius map with modular composition is faster than the use of multi-squaring tables. Finally, we observe

a last threshold where the use of modular composition is clearly inefficient because D is too large. Removing odd degree terms is interesting for HFE-based NIST submissions which use D equals 129 or more. This implies increasing by one the original parameters ($D = 129$ and $D = 513$). Without taking into account this change, our best Frobenius map is 6 to 14 times faster than NTL. Magma is faster than NTL for fairly large degrees such as 513.

d_{ext}	D	s	Magma	NTL		MQsoft (rep. sqr)		MQsoft (multi-sqr)	
			Modexp	normal	comp.	normal	comp.	normal	comp.
175	17	0	11.3	4.46	2.12	0.696	0.280	0.274	0.204
	129	0	159	130	123	12.0	9.30	11.2	9.88
	130	3	101	146	134	6.73	6.65	11.2	9.67
	513	0	957	1050	1870	74.9	92.6	151	139
	514	3	606	1150	2050	41.6	73.4	150	135
358	17	0	36.5	15.0	4.31	3.95	0.892	1.43	0.694
	129	0	539	466	257	72.0	35.5	61.7	37.9
	130	3	329	524	280	39.4	26.8	61.2	36.5
	513	0	3260	3460	3950	447	364	853	614
	514	3	1980	3860	4260	245	256	845	600

Table 9.18: Number of megacycles to compute the Frobenius map modulo a HFE polynomial over $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS).

9.3.6 Greatest Common Divisor of Polynomials over $\mathbb{F}_{2^{d_{ext}}}$

The second most important step of the root finding algorithm (Algorithm 25) is the computation of the GCD of two degree-D polynomials over $\mathbb{F}_{2^{d_{ext}}}$ (Sections 5.4.1, 7.4.11 and B.8). We have implemented variable-time and constant-time GCD algorithms which require $O(D^2)$ field multiplications. We have also implemented the half-GCD algorithm ([161, Algorithm 11.8], [36, Algorithm 6.8]), which uses $O(M(D)\log(D))$ field multiplications. Our implementation is based on Karatsuba's polynomial multiplication algorithm (Section 9.3.2) and the fast Euclidean division. The half-GCD algorithm is variable-time, and is not interesting for a degree 514 or less. So, we do not consider it here. All algorithms used are implemented with a constant-time arithmetic, but the variable-time GCD algorithm is variable because of the variable number of successive remainders. We use Methods 1 and 3 from Section 5.4.1 in Algorithm 18. Its constant-time implementation follows the principle of Algorithm 53, with optimizations due to the characteristic two.

d_{ext}	D	Magma	NTL	MQsoft (var.)	MQsoft (cst)
175	17	0.716	0.236	0.0137	0.0213
	129	10.0	3.94	0.649	1.17
	513	125	40.6	7.42	15.6
358	17	1.71	0.597	0.0389	0.0528
	129	21.5	7.94	1.76	3.14
	513	228	75.0	20.7	39.5

Table 9.19: Number of megacycles to compute the GCD of two polynomials of degree D and $D-1$ over $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS).

Table 9.19 compares the performance of GCD algorithms. Our classical variable-time GCD is 3.5 to 4.5 times better than NTL. This results of the difference of performance between our operations in $\mathbb{F}_{2^{d_{ext}}}$, except for small degrees such as 17 where we obtain a factor 15 or 17. In this case, the use of the Euclid–Stevin strategy which does not compute inverse in $\mathbb{F}^{\times}_{2^{d_{ext}}}$ is really efficient. This strategy is rather recent (2019), which can explain that NTL does not use it for the moment.

9.3.7 Performance of the Root Finding Algorithm over $\mathbb{F}_{2^{d_{ext}}}$

Table 9.20 compares the best implementation of root finding algorithm of each library for the parameters of GeMSS and Gui. The results are similar to the performance of Frobenius map, which is the critical part of the root finding algorithm. MQsoft is six to thirteen times faster than NTL.

d_{ext}	D	s	Magma	NTL	MQsoft
177	17	0	13.8	2.49	0.241
266	17	0	27.3	4.03	0.604
358	17	0	40.7	5.27	0.835
184	33	0	40.6	9.11	0.901
175	129	0	174	136	10.5
	130	3	114	145	7.89
265	129	0	380	216	27.9
	130	3	241	226	22.4
312	129	0	437	235	28.4
	130	3	275	236	22.5
358	129	0	571	273	39.3
	130	3	356	274	30.5
174	513	0	1140	1120	91.1
	514	3	769	1140	58.7
265	513	0	2410	2360	281
	514	3	1580	2380	183
354	513	0	3690	4520	399
	514	3	2340	4550	285
448	513	0	4700	5560	580
	514	3	2850	5430	457

Table 9.20: Number of megacycles to find the roots of a HFE polynomial of degree D over $\mathbb{F}_{2^{d_{ext}}}$. We use a Skylake processor (LaptopS).

We have presented in this section the main algorithms that we have implemented in MQsoft to find the roots of HFE polynomials efficiently. However, our library proposes extra functions. The half-GCD requires implementing Karatsuba's polynomial multiplication algorithm and the fast Euclidean division over $\mathbb{F}_{2^{d_{ext}}}$. So, we propose a fast version of the root finding algorithm, based on a Frobenius map using the fast Euclidean division. This permits to have an efficient implementation of root finding for general applications, using a constant-time arithmetic in the base field.

9.4 Generation and Evaluation of MQ Systems

In this section, we study how to implement efficiently important steps of the keypair generation (Algorithm 29) and verifying process (Algorithm 32) of `GeMSS`, as well as for the verifying process (Algorithm 11) of `DualModeMS`. These steps are based on multivariate quadratic (MQ) systems that we represent as quadratic forms.

In Section 9.4.1, we compute the multivariate representation (in n_{var} variables) of `HFEv` polynomials. Then, we study efficient implementations about multivariate quadratic systems. The evaluation can be implemented in variable-time to evaluate the public-key, unlike the `HFEv` polynomial where the evaluation of the vinegar part has to be achieved in constant-time. We also note that for multivariate-based encryption schemes, we can re-use our constant-time implementation to evaluate the public-key. In addition, the performance of an evaluation depends mainly on the storage format. It is often faster to use padding to align data, but this increases the required size to store the multivariate quadratic system. To obtain the smallest size of public-key, we can pack data, then unpack it before performing the evaluation. However, the extra cost of unpacking can significantly impact the performance. So, we propose two kinds of algorithms. The first allows to directly evaluate the system, whereas the second minimizes the time to unpack and evaluate the system. Especially during the verifying process, we take into account that a system is sometimes unpacked one time for several evaluations.

Finally, the inner verifying process (Algorithm 4) cannot be re-used during the dual verifying process (Algorithm 11). It is due to the number of equations, which falls to $k/2$ equations instead of m. Moreover, this small number of equations is evaluated a large number of times. In Section 9.4.5, we introduce the idea of multipoint evaluation via a monomial representation of points. This method turns to be very efficient over small fields in characteristic two, such as $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$, but inefficient over $\mathbb{F}_2$. Over $\mathbb{F}_2$, we evaluate each equation in each point with the technique described in Section 9.4.2.

9.4.1 Generating the Components of a **HFEv** Polynomial

Here, we explain how to obtain the multivariate polynomials $f_1, \ldots, f_{d_{ext}} \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]$ of f from a `HFEv` polynomial $F \in \mathbb{F}_{q^{d_{ext}}}[X, v_1, \ldots, v_v]$ (Section 7.1.2). The principle is to symbolically compute $F\left(\sum_{k=1}^{d_{ext}} x_k \cdot \theta_k, v_1, \ldots, v_v\right) \in \mathbb{F}_{q^{d_{ext}}}[x_1, \ldots, x_{n_{var}}]$ in the form $\sum_{k=1}^{d_{ext}} f_k \cdot \theta_k$ (Equation (7.2)). To do it, we start by introducing the matrix representation of F over $\mathbb{F}_{q^{d_{ext}}}$ based on quadratic forms. Then, we replace X by $\sum_{k=1}^{d_{ext}} x_k \cdot \theta_k$ to obtain the multivariate representation of F. This method is described in [84].

Matrix representation of **HFEv** polynomials as quadratic forms. `HFE` polynomials (Equation (2.6)) can be naturally represented by using the matrix representation of quadratic forms [111] (Section 4.6.3), as for multivariate quadratic systems (Section 7.4.1). Let $R = \lfloor \log_q(D) \rfloor + 1$ and

$$\mathbf{X} = \left(X, X^q, X^{q^2}, X^{q^3}, \ldots, X^{q^{R-1}}\right).$$

Then, we have:

$$F(\mathbf{X}) = \mathbf{X} \cdot Q \cdot \mathbf{X}^T + L \cdot \mathbf{X}^T + C, \tag{9.2}$$

with $L = (B_0, \ldots, B_{R-1}) \in \mathcal{M}_{1,R}\ \mathbb{F}_{q^{d_{ext}}}$ representing the linear terms of F, and where the matrix $Q \in \mathcal{M}_R\ \mathbb{F}_{q^{d_{ext}}}$ is upper triangular such that:

$$Q_{i,j} = \begin{matrix} A_{i,i} & \text{if } i = j \text{ and } q \neq 2, \\ A_{j,i} & \text{if } i < j, \\ 0 & \text{otherwise.} \end{matrix}$$

Now, we extend this definition to include vinegar variables. Let $v = (v_1, \ldots, v_v)$ be the vector of vinegar variables. The $(v_1, \ldots, v_v)$ map (Equation (2.9)) is quadratic in the vinegar variables. So, we can write it as

$$(v) = v \cdot W \cdot v^T + V \cdot v^T + C,$$

where $W \in \mathcal{M}_v\ \mathbb{F}_{q^{d_{ext}}}$, $V \in \mathcal{M}_{1,v}\ \mathbb{F}_{q^{d_{ext}}}$ and $C \in \mathbb{F}_{q^{d_{ext}}}$ correspond respectively to the quadratic, linear and constant terms of , and W is upper triangular. Similarly, the $_i(v)$ are linear in the vinegar variables, and can be written as

$$_i(v) = V_X^{(i)} \cdot v^T + B_i, \text{ for } 0 \leqslant i < R,$$

where $V_X^{(i)} \in \mathcal{M}_{1,v}\ \mathbb{F}_{q^{d_{ext}}}$ and $B_i \in \mathbb{F}_{q^{d_{ext}}}$ correspond respectively to the linear and constant terms of $_i$. Let $V_X \in \mathcal{M}_{R,v}\ \mathbb{F}_{q^{d_{ext}}}$ be the matrix generated from the row vectors $V_X^{(i)}$. Thus, Equation (9.2) becomes:

$$F(X, v) = X\ \ v \cdot \begin{matrix} Q & V_X \\ 0_{v,R} & W \end{matrix} \cdot X\ \ v^T + L\ \ V \cdot X\ \ v^T + C. \quad (9.3)$$

Note that the matrix of quadratic terms of F is upper triangular.

Matrix representation of $\mathcal{F}$ from F. Now, we know the matrix representation of F, and we want to deduce this of $\mathcal{F}$. We start by replacing X by $^{-1}(x)$ in F. By using Equation (2.7), it is easy to check that $X = x \cdot$ where $\in \mathcal{M}_{d_{ext},R}\ \mathbb{F}_{q^{d_{ext}}}$ is such that $_{i,j} = {}^{q^j}_{i+1}$. So, we deduce the multivariate quadratic form of F from Equation (9.3):

$$F\ \ ^{-1}(x), v = x\ \ v \cdot \begin{matrix} \cdot Q \cdot {}^T & \cdot V_X \\ 0_{v,d_{ext}} & W \end{matrix} \cdot x\ \ v^T + L \cdot {}^T\ \ V \cdot x\ \ v^T + C. \quad (9.4)$$

Note that the matrix of quadratic terms of $F\ \ ^{-1}(x), v$ can be chosen upper triangular. To do it, we take the matrix generated by the lower triangular terms (without the diagonal) of $\cdot Q \cdot {}^T$, and we add its transposed to $\cdot Q \cdot {}^T$.

Finally, we obtain $\mathcal{F}$ as $F\ \ ^{-1}(x), v$, which is equivalent to applying on each of the N field elements of $F\ \ ^{-1}(x), v$.

Complexity. We start by evaluating the cost of computing $F\ \ ^{-1}(x)$ via Equation (9.4), when no vinegar variable is considered.

- Firstly, the computation of requires $O\ d_{ext} \log_q(D)$ field q-exponentiations. This matrix does not depend on F and so can be precomputed. The memory cost of storing is $d_{ext}\ \lfloor \log_q(D) \rfloor + 1$ elements of $\mathbb{F}_{q^{d_{ext}}}$.

- Secondly, the computation of L^T requires $O\left(d_{\mathtt{ext}}\log_q(D)\right)$ operations in $\mathbb{F}_q[x]$ and $O\left(d_{\mathtt{ext}}\right)$ modular reductions (which is better than $O\left(d_{\mathtt{ext}}\log_q(D)\right)$ operations in $\mathbb{F}_{q^{d_{\mathtt{ext}}}}$).

- Then, we compute $\cdot Q$ with a classical matrix product, requiring $O\left(d_{\mathtt{ext}}\log_q(D)^2\right)$ operations in $\mathbb{F}_q[x]$ and $O\left(d_{\mathtt{ext}}\log_q(D)\right)$ modular reductions.

- Finally, we multiply the previous result by ^{T}, requiring $O\left(d_{\mathtt{ext}}^2\log_q(D)\right)$ operations in $\mathbb{F}_q[x]$ and $O\left(d_{\mathtt{ext}}^2\right)$ modular reductions.

Now, we consider vinegar variables. We add the following step.

- The computation of $\cdot V_X$ requires $O\left(d_{\mathtt{ext}} v \log_q(D)\right)$ operations in $\mathbb{F}_q[x]$ and $O\left(d_{\mathtt{ext}} v\right)$ modular reductions.

By removing the precomputation step which requires $O\left(d_{\mathtt{ext}}\log_q(D)\log_2(q)\right)$ operations in $\mathbb{F}_{q^{d_{\mathtt{ext}}}}$, we obtain an overall cost of $O\left(d_{\mathtt{ext}}\log_q(D)\cdot(n_{\mathtt{var}}+\log_q(D))\right)$ operations in $\mathbb{F}_q[x]$ and $O\left(d_{\mathtt{ext}}\cdot(n_{\mathtt{var}}+\log_q(D))\right)$ modular reductions.

Remark 17. For certain choices of , the computation of , as well as the computations using it, could be less expensive. In the implementation of GeMSS, is the canonical basis of $\mathbb{F}_{2^{d_{\mathtt{ext}}}}$, and the field polynomial f of $\mathbb{F}_{2^{d_{\mathtt{ext}}}}$ is a trinomial. This implies that the elements of are sparse, which simplifies the multiplication. When f is a field AOP (Section 5.3.4), the (field) multiplication by $x^{i\cdot q^j} = x^{i\cdot q^j \bmod d_{\mathtt{ext}}+1} \bmod f$ is roughly a left (circular) shift by $iq^j \bmod (d_{\mathtt{ext}}+1)$ positions.

9.4.2 Evaluation of a MQ Polynomial over $\mathbb{F}_2$

Here, we describe how to evaluate efficiently a multivariate quadratic polynomial p, such that

$$p = \sum_{i=1}^{n_{\mathtt{var}}}\sum_{j=1}^{i} p_{i,j}\, x_i x_j + \sum_{i=1}^{n_{\mathtt{var}}} p_i x_i + c \in \mathbb{F}_q[x_1,\ldots,x_{n_{\mathtt{var}}}].$$

This operation is used during the verifying process of MI-based schemes (Section 2.3.3), in particular when we use the hybrid representation of the public-key (Section 7.4.8). We also use it during the dual verifying process, when we verify the validity of inner signatures with a derived public-key of $\leqslant 2$ equations (Section 3.4).

By using the matrix representation of quadratic forms, we can write p as

$$p(x) = x\cdot Q\cdot x^T + x\cdot L^T + c, \tag{9.5}$$

where $c\in\mathbb{F}_q$, $L\in\mathcal{M}_{1,n_{\mathtt{var}}}(\mathbb{F}_q)$ and $Q\in\mathcal{M}_{n_{\mathtt{var}}}(\mathbb{F}_q)$ is a lower triangular matrix. We emphasize that Q is not upper triangular here. This choice allows to improve the performance when the intuitive strategy is used. Thus, we store the i-th row of Q over $\mathbb{F}_2$ as an $n_{\mathtt{var}}$-bit integer for $1\leqslant i\leqslant n_{\mathtt{var}}$.

From now, we consider $q=2$, and we store L on the diagonal of Q. An intuitive constant-time strategy is to directly compute Equation (9.5). In order to compute efficiently $Q\cdot x^T$, we store Q in the row-major order such that each row is a vector of bits. Thus, each dot product can be computed with a bitwise logical AND, followed by the computation of the parity bit of the Hamming weight. However, the latter is rather expensive compared to a simple logical AND. So, we do not compute it

fully. For each row of Q, we xor the products (which use AND) in a -bit register, then we directly multiply this non-reduced result by the corresponding value x from. We accumulate vectors of bits instead of bits, and at the end of the evaluation, we compute only one time the parity bit of the Hamming weight. The latter can be computed by repeating the process of aligning the lower and higher parts, then xoring them, until a 64-bit (or 32-bit) vector is obtained. For these sizes, we could continue to repeat the previous dichotomic process, but the use of the POPCNT instruction (which computes the Hamming weight) is faster. When this instruction is not available, we can use the variable-precision SWAR (SIMD Within A Register) algorithm, our variant computing the parity of the Hamming weight, or the PCLMULQDQ instruction to multiply the 64-bit vector (as a binary polynomial) by the degree-63 binary AOP, followed by a logical right shift by 63 positions.

Algorithm 38 Classical 64-bit implementation in C programming language of the SWAR algorithm.

```
1: /* n is the input, a 64-bit register. */
2: /* n is the output. n contains the Hamming weight of the input. */
3: #define COUNTBITS64_SWAR(n) \
4:   n-= (n >> 1) & ((uint64_t)0x5555555555555555); /* HW of each 2-bit block */\
5:   n= ( n       & ((uint64_t)0x3333333333333333))\
6:     +((n >> 2) & ((uint64_t)0x3333333333333333));/* HW of each 4-bit block */\
7:   n=(((n + (n >> 4)) & ((uint64_t)0xF0F0F0F0F0F0F0F)) /* HW of each byte */\
8:      * ((uint64_t)0x101010101010101)) >> 56;          /* sum of the 8 HWs */

9: #define PARITY_COUNTBITS64_SWAR(n) /* our variant computing the parity bit */\
10:   n^=n >> 1; /* parity of the HW of each 2-bit block */\
11:   n^=n >> 2; /* parity of the HW of each 4-bit block */\
12:   n=((n&((uint64_t)0x1111111111111111))\
13:      *  ((uint64_t)0x8888888888888888)) >> 63;/* sum of the 16 HWs modulo 2 */
```

The evaluation of p can be optimized with an unpacked representation of Q, where each row of Q is aligned to the used register size, with a zero padding. Compared to a packed representation, this avoids having to align x^T for each dot product and turns to be more efficient. However, a packed representation minimizes the required memory. So, the unpacked representation implies a memory penalty or adding an unpacking step, which can be expensive. We study this point in Section 7.4.8.

We note that [53] proposed a method which balances the size of the rows of Q. We denote this idea the *diagonal strategy*, since a *diagonal* representation of Q is used. The idea is to generate a matrix Q' whose i-th row is the diagonal of size $n_{var} - i$ of Q (from up to down) followed by this of size i, for $0 \leqslant i \leqslant \frac{n_{var}}{2}$. The obtained matrix has $\frac{n_{var}}{2} + 1$ plain rows. This representation can seem better, since the number of rows is divided by two, and the balanced rows make easier the memory management. However, this representation requires shifting $\frac{n_{var}}{2}$ times the vector of variables during the evaluation. The implementation of this operation is rather slow over $\mathbb{F}_2$. That is why we use the intuitive strategy.

In our variable-time implementation over $\mathbb{F}_2$, we improve the intuitive strategy by using the fact that a random value of x_i is null with probability 0.5. We know that each row of $Q \cdot x^T$ will be multiplied by x_i. So, when $x_i = 0$, we do not compute the corresponding row. When the rows of Q have to be unpacked, we can avoid unpacking the i-th row if $x_i = 0$, also improving the unpacking step.

9.4.3 Variable-Time Evaluation of MQ Systems over $\mathbb{F}_q$

Now, we study how to take advantage of the structure of a multivariate quadratic system to evaluate it efficiently. The evaluation of MQ systems in variable-time is important for the verifying process of MI-based schemes (Section 2.3.3). Similarly to the previous section, let $p \in \mathbb{F}_{q^m}[x_1, \ldots, x_{n_{var}}]$ be the monomial representation of a MQ system of m equations in n_{var} variables over $\mathbb{F}_q$, such that

$$p(x) = x \cdot Q \cdot x^T + L \cdot x^T + c,$$

where $c \in \mathbb{F}_{q^m}$, $L \in \mathcal{M}_{1,n_{var}}(\mathbb{F}_{q^m})$ and $Q \in \mathcal{M}_{n_{var}}(\mathbb{F}_{q^m})$ is upper triangular. We can write p as:

$$p(x) = x \cdot (Q \cdot x^T + L^T) + c.$$

In variable-time, and when q is small enough, the classical trick [15] is to look the value of $x_i \in \mathbb{F}_q$ before computing the i-th row of $Q \cdot x^T + L^T$. If x_i is null, then we do not compute the i-th row since the latter will be multiplied by zero. This happens with probability $\frac{1}{q}$ for a random value of x_i. Else, the computation of the i-th row requires multiplying the i-th row of Q by x^T. Here again, we can optimize the multiplication by $x_j \in \mathbb{F}_q$ for $i \leqslant j \leqslant n_{var}$. If x_j is null, then the multiplication can be avoided. We also note that when x_i or x_j is equal to one (or minus one), the multiplication becomes trivial. This strategy is very efficient on $\mathbb{F}_2$ (Algorithm 39), because approximately a quarter of the monomials are non-null. The average cost of this strategy is approximately $\frac{n_{var}(n_{var}+1)}{2} \cdot \frac{(q-1)^2}{q^2}$ multiplications in $\mathbb{F}_q$. In our implementation, this strategy seems to give apparent speed-ups for $q \leqslant 16$. We also observe that the verification of x_i alone can be more efficient.

Algorithm 39 Variable-time evaluation of a MQ system of m equations in n_{var} variables over $\mathbb{F}_2$.

```
1:  function Evaluation_MQsystem(p ∈ F_{2^m}[x_1,...,x_{n_var}], y = (y_1,...,y_{n_var}) ∈ F_2^{n_var})
2:      acc ← c                                        ▷ Constant term of p.
3:      for i from 1 to n_var do
4:          if y_i = 1 then
5:              acc ← acc + p_i                         ▷ Linear term of p.
6:              for j from i + 1 to n_var do
7:                  if y_j = 1 then
8:                      acc ← acc + p_{i,j}             ▷ Quadratic term of p.
9:                  end if
10:             end for
11:         end if
12:     end for
13:     return acc
14: end function
```

The previous method exploits the potential low Hamming weight of x. However, this method is expensive for large Hamming weights. The authors of [15] proposed a differential trick over $\mathbb{F}_2$. Let $L_1(x)$ be the following multivariate linear polynomial over $\mathbb{F}_q$:

$$L_1(x) = c - p(1_{n_{var}}) - \sum_{i=1}^{n_{var}} \left(2 \cdot p_{i,i} + \sum_{j=1}^{i-1} p_{j,i} + \sum_{j=i+1}^{n_{var}} p_{i,j} \right) x_i .$$

Thus, we have the following differential property:

$$p(x) - p\left(x + 1_{n_{\text{var}}}\right) = L_1(x).$$

So, the difference between $p(x)$ and $p(x + 1_{n_{\text{var}}})$ is linear. This property is well-known for MQ polynomials [58], by replacing $1_{n_{\text{var}}}$ by any vector over $\mathbb{F}_q$. However, the choice of $1_{n_{\text{var}}}$ allows the following strategy [15] to evaluate p over $\mathbb{F}_2$. If the Hamming weight of x is smaller than a certain threshold t (e.g. $\frac{n_{\text{var}}}{2}$), then $p(x)$ is directly computed. Else, $p(x + 1_{n_{\text{var}}}) + L_1(x)$ is computed. We summarize this strategy in Algorithm 40. The interest of Algorithm 40 is that Step 0 can be precomputed for a fixed public-key p. This requires storing $m \cdot (n_{\text{var}} + 1)$ bits.

Algorithm 40 Variable-time evaluation of a MQ system over $\mathbb{F}_2$ using the differential trick.

Input: $p \in \mathbb{F}_{2^m}[x_1, \ldots, x_{n_{\text{var}}}]$, $y = (y_1, \ldots, y_{n_{\text{var}}}) \in \mathbb{F}_2^{n_{\text{var}}}$, $t \in \mathbb{N}$.
Output: $p(y) \in \mathbb{F}_{2^m}$.

0. Precompute $L_1(x) = c - p\left(1_{n_{\text{var}}}\right) - \sum_{i=1}^{n_{\text{var}}} \left(\sum_{j=1}^{i-1} p_{j,i} + \sum_{j=i+1}^{n_{\text{var}}} p_{i,j} \right) x_i$. The latter depends only on p.
1. Compute h the Hamming weight of y, e.g. with several calls to the `POPCNT` instruction or the SWAR algorithm 38.
2. If $h \leqslant t$, then return $p(y)$ by using Algorithm 39.
3. Else, return $p\left(y + 1_{n_{\text{var}}}\right) + L_1(y)$ by using Algorithm 39. Note that the computation of $L_1(y)$ can be trivially included in the evaluation of p.

However, Algorithm 40 does not speed up the evaluation of the public-key in `GeMSS`. We think this is due to the Hamming weight, which is close to $\frac{n_{\text{var}}}{2}$ for random vectors. Finally, we emphasize that we cannot use specific values of x to speed up a constant-time implementation.

9.4.4 Implementing an Efficient Evaluation of MQ Systems over $\mathbb{F}_2$

Here, we study how to implement efficiently the evaluation of multivariate quadratic systems over $\mathbb{F}_2$. The evaluation of the public-key is the main part of the verifying process of `GeMSS` (Section 7.1.4). It is iterated nb_ite times. Since the verification is a public process, it does not need to be protected against timing attacks. As in Section 9.4.3, we can exploit the fact that for a random input, the evaluation of a monomial $x_i x_j$ in $\mathbb{F}_2$ is null with probability 0.75, and so avoid 75% of computations. However, the evaluation in constant-time is required during the signature generation (Section 7.1.3) and keypair generation by evaluation-interpolation (Section 7.4.7) to evaluate the map of the `HFEv` polynomial (Equation (7.1)), which is quadratic in the vinegar variables. It is also used in other contexts, for example to encrypt a message for the `HFE`-based encryption schemes. In this section, we study both variable-time evaluation and constant-time evaluation.

To evaluate the public-key p, we can use different representations. The representation by equation consists in storing the m equations of p separately ($p \in \mathbb{F}_2[x_1, \ldots, x_{n_{\text{var}}}]^m$), whereas the representation by monomial consists in storing the monomials of p separately ($p \in \mathbb{F}_{2^m}[x_1, \ldots, x_{n_{\text{var}}}]$).

In the previous section, we studied the fast evaluation of [53], which applies naturally on the representation by equation (with a diagonal representation of each equation). In [56], the authors presented a faster evaluation. To do so, they used a monomial representation of the public-key. Both, [53] and [56], used the AVX2 instruction set. We have chosen the monomial representation as in [56], because it naturally exploits the fact that on average, 75% of monomials are null.

Our variable-time evaluation only uses the classical method [15], as presented in Algorithm 39. We initialize an accumulator acc to the constant term $c \in \mathbb{F}_{2^m}$ of p, and for each term $p_{i,j}\, x_i x_j$ with $p_{i,j} \in \mathbb{F}_{2^m}$ for $1 \leqslant i < j \leqslant n_{\texttt{var}}$, we add $p_{i,j}$ to acc if and only if $x_i = x_j = 1$.

Then, we have vectorized Algorithm 39. To do it, we just store acc with 256-bit registers, and we use 256-bit `load`, `store` and bitwise `XOR` instructions to perform vectorial computations. When $\frac{m}{64}$ is not a multiple of four, we sometimes add the use of 64-bit or 128-bit registers to speed up the implementation. Algorithm 39 is vulnerable to timing attacks (Section 4.7), since the bitwise `XOR` is used if and only if $x_i = x_j = 1$. The traditional way to avoid this attack (Section 6.3.2) is to replace the conditional statement by a multiplication by x_i (respectively x_j). But x_i and x_j are in $\mathbb{F}_2$, so the multiplication can be accelerated: it is equivalent to applying a mask which is the duplication of x_i (respectively x_j) m times. With this strategy, we obtain Algorithm 41.

Algorithm 41 Constant-time evaluation of a MQ system of m equations in $n_{\texttt{var}}$ variables over $\mathbb{F}_2$.

1: **function** Cst-time_evaluation_MQsystem($p \in \mathbb{F}_{2^m}[x_1, \ldots, x_{n_{\texttt{var}}}]$, $y = (y_1, \ldots, y_{n_{\texttt{var}}}) \in \mathbb{F}_2^{n_{\texttt{var}}}$)
2:     **for** i **from** 1 **to** $n_{\texttt{var}}$ **do**
3:         $T_{mask}[i] \leftarrow -y_i$ ▷ Duplicate the bit y_i to create a mask.
4:     **end for**
5:     acc $\leftarrow c$ ▷ Constant term of p.
6:     **for** i **from** 1 **to** $n_{\texttt{var}}$ **do**
7:         $L \leftarrow p_i$
8:         **for** j **from** $i+1$ **to** $n_{\texttt{var}}$ **do**
9:             $L \leftarrow L + (p_{i,j}$ `AND` $T_{mask}[j])$ ▷ Apply the mask on $p_{i,j}$ (compute $p_{i,j} \cdot y_j$).
10:         **end for**
11:         acc $\leftarrow$ acc $+ (L$ `AND` $T_{mask}[i])$ ▷ Apply the mask on L (compute $L \cdot y_i$).
12:     **end for**
13:     **return** acc
14: **end function**

To vectorize Algorithm 41, acc and L are stored in 256-bit registers. However, the optimal way to put each mask in a 256-bit register is not trivial. On the one hand, we can store 256-bit masks in the array T_{mask}. In this way, the `load` instruction permits to create the 256-bit register. On the other hand, we can store 64-bit masks in the array T_{mask}. The creation of the 256-bit register is done by one call to `VPBROADCASTQ`, which duplicates a 64-bit mask in a 256-bit register. This idea is described in Algorithm 42. We propose a new idea, described in Algorithm 43. Firstly, we unroll with a depth four the loop in j. Secondly, we store 64-bit masks in the array T_{mask}, but we load four 64-bit masks in one 256-bit register. Then, to create a 256-bit mask from one of the four 64-bit masks, we use the `VPERMQ` instruction. It permits to create a 256-bit register where each 64-bit part is one of the four 64-bit part of the input. In particular, we use it to duplicate one 64-bit part of the input (which is a mask) in a 256-bit register. This method is the best: it requires only one load for four masks, unlike the two previous methods which require four loads (four 256-bit loads for the first method and four 64-bit loads for the second method).

Then, to apply the mask on $p_{i,j}$, we remark that the VPMASKMOVQ instruction permits to load data and apply the mask in only one instruction. It permits to accelerate the evaluation.

More generally, our new method using VPERMQ permits to improve the constant-time vector-matrix product over $\mathbb{F}_2$, but is interesting only when the variable T_{mask} is computed one time for several products sharing the same vector. We remark that this method is faster on Skylake processors, but on Haswell processors, the use of VPBROADCASTQ (Algorithm 42) remains faster.

Algorithm 42 Improvement of Algorithm 41 with AVX2, VPMASKMOVQ and VPBROADCASTQ.

```
 8: for j from i + 1 to n_var do
 9:     L ← L + VPMASKMOVQ(p_{i,j}, VPBROADCASTQ(T_mask[j]))          ▷ Compute p_{i,j} y_j.
10: end for
```

Algorithm 43 Improvement of Algorithm 41 with AVX2, VPMASKMOVQ and VPERMQ.

```
 8: for j from i + 1 to n_var − 3 by 4 do
 9:     y64x4 ← VMOVDQU(T_mask + j)    ▷ Load T_mask[j], T_mask[j + 1], T_mask[j + 2] and T_mask[j + 3].
10:     y0 ← VPERMQ(y64x4, 0x00)                                  ▷ Duplicate T_mask[j].
11:     y1 ← VPERMQ(y64x4, 0x55)                                  ▷ Duplicate T_mask[j + 1].
12:     y2 ← VPERMQ(y64x4, 0xAA)                                  ▷ Duplicate T_mask[j + 2].
13:     y3 ← VPERMQ(y64x4, 0xFF)                                  ▷ Duplicate T_mask[j + 3].
14:     L ← L + VPMASKMOVQ(p_{i,j}, y0)      ▷ Load p_{i,j} and apply the mask (compute p_{i,j} y_j).
15:     L ← L + VPMASKMOVQ(p_{i,j+1}, y1)                         ▷ Compute p_{i,j+1} y_{j+1}.
16:     L ← L + VPMASKMOVQ(p_{i,j+2}, y2)                         ▷ Compute p_{i,j+2} y_{j+2}.
17:     L ← L + VPMASKMOVQ(p_{i,j+3}, y3)                         ▷ Compute p_{i,j+3} y_{j+3}.
18: end for
19: for j from j to n_var do
20:     L ← L + VPMASKMOVQ(p_{i,j}, VPBROADCASTQ(T_mask[j]))          ▷ Compute p_{i,j} y_j.
21: end for
```

Table 9.21 shows the performance of the evaluation that uses the AVX2 instruction set. To improve the performance, we use the -funroll-loops option of GCC which unrolls loops to improve the use of the pipeline. The factor of performance between variable-time and constant-time implementation depends on m: the factor is two for small values of m and four for larger values. The performance is affected by cache penalties when the public-key is too large. For $m = n_{var} = 256$, we compare our code with the efficient implementation of [56], by using a similar processor (ServerH). We have similar times for the constant-time evaluation, and a speed-up of 1.38 for the variable-time evaluation. This speed-up is mainly due to unrolled loops. Moreover, we have split the i loop (respectively the loop j) into two loops with an Euclidean division by 64: the first is a loop for q from 0 to $\frac{i}{64}$, and the second is a loop for r from 0 to 63. In this way, for extracting x_i, which is the i-th bit from a vector of 64-bit words, we take the r-th bit of the q-th word. It permits to simplify the extraction of bits from 64-bit registers.

For the constant-time evaluation, we have obtained our best times on Haswell by using Algorithm 42. However, on Skylake, Algorithm 43 is faster. For a MQ system of 256 equations in 256 variables over $\mathbb{F}_2$ (cf. Table 9.22), we obtain 61.4 kc with Algorithm 42 against 55.5 kc with Algorithm 43. Since Algorithm 42 is state-of-the-art on Haswell, we have obtained a new speed record on Skylake, by a factor 1.1. For comparison, we obtain 23.2 kc for the variable-time evaluation.

For m requiring one 64-bit word (respectively two 64-bit words), we use the 256-bit registers to perform computations in $\mathbb{F}_{2^m}$ by pack of four elements (respectively two elements). This method implies the use of masks to compute $p_{i,j}x_j$ for four (respectively two) successive values of j. Optimizing the cases m requiring one or two 64-bit words permits to use a new strategy of parallelization: with k cores, the public-key can be split in k packets of 64 equations (respectively 128 equations), and each core can apply one time the evaluation for its part of the public-key. In a general way, m can be split in the way to use evaluation algorithms for smaller number of equations.

m	64	128	192	256	320	384	448	512
constant-time	2.01	14.1	44.7	89.1	196	318	478	853
variable-time	1.15	6.46	17.1	37.3	74.5	120	191	205

Table 9.21: Number of kilocycles to evaluate a MQ system of m equations in m variables over $\mathbb{F}_2$. We use a Haswell processor (ServerH) with the AVX2 instruction set. Turbo Boost is not used.

m	64	128	192	256	320	384	448	512
constant-time	1.49	7.04	30.1	55.5	142	202	341	610
variable-time	0.841	3.87	12.3	23.2	51.0	75.5	133	144

Table 9.22: Number of kilocycles to evaluate a MQ system of m equations in m variables over $\mathbb{F}_2$. We use a Skylake processor (DesktopS) with the AVX2 instruction set. Turbo Boost is used.

9.4.5 Multipoint Evaluation of a MQ Polynomial

Here, we study how to perform efficiently the evaluation of a multivariate quadratic polynomial

$$p = \sum_{i=1}^{n_{var}} \sum_{j=i}^{n_{var}} p_{i,j}\, x_i x_j + \sum_{i=1}^{n_{var}} p_i x_i + c \in \mathbb{F}_q[x_1, \ldots, x_{n_{var}}]$$

in a set of evaluation points $z_1, \ldots, z \in \mathbb{F}_q^{n_{var}}$. This operation is used when we verify the validity of inner signatures with a derived public-key of $\leqslant 2$ equations (Section 3.4). For the sake of simplicity, we only consider the evaluation of the quadratic terms. As in Section 9.4.3, the quadratic terms of p can be written as a quadratic form, i.e., $x \cdot Q \cdot x^T$, where Q is an upper triangular matrix in $\mathcal{M}_{n_{var}}(\mathbb{F}_q)$ such that for $1 \leqslant i \leqslant j \leqslant n_{var}$, the coefficient (i, j) of Q corresponds to the monomial $x_i x_j$ of p. We have:

$$Q_{i,j} = \begin{cases} p_{i,j} & \text{if } i \leqslant j, \\ 0 & \text{otherwise.} \end{cases}$$

It is well-known that in characteristic two, the monomial representation of a MQ system is optimal for its evaluation in a point [15, 64]. It is due to the vectorization of the product of each monomial $x_i x_j$ by the corresponding coefficient from each equation. In our case, this representation is not optimal since we only have $\leqslant 2$ equations to evaluate (Section 8.6). So, we cannot re-use the code of the evaluation available in the inner mode to verify inner signatures during the dual verifying

process. However, we have a fairly large set of evaluation points ($16 \leqslant \quad \leqslant 256$). We propose the dual idea of the previous evaluation. We store each equation one by one, but we use a monomial representation of the evaluation points. Then, we will be able to multiply efficiently each coefficient of p by the corresponding variables from each evaluation point.

Let $z_k = \begin{pmatrix} z_1^{(k)} & \cdots & z_{n_{var}}^{(k)} \end{pmatrix} \in \mathbb{F}_q^{n_{var}}$ for $1 \leqslant k \leqslant \quad$, and $z \in \mathcal{M}_{n_{var}, \quad}(\mathbb{F}_q)$ be the monomial representation of $z_1, \ldots, z \quad$, i.e. z is the following matrix stored in the row-major order:

$$z = \begin{pmatrix} z_1^{(1)} & z_1^{(2)} & \cdots & z_1^{(\;)} \\ z_2^{(1)} & z_2^{(2)} & \cdots & z_2^{(\;)} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n_{var}}^{(1)} & z_{n_{var}}^{(2)} & \cdots & z_{n_{var}}^{(\;)} \end{pmatrix}.$$

Then, we demonstrate that:

$$\begin{pmatrix} p(z_1) & p(z_2) & \cdots & p(z \;) \end{pmatrix} = \begin{pmatrix} 1 & \cdots & 1 \end{pmatrix} \cdot \left(z \odot (Q \cdot z) \right),$$

where $\odot$ stands for pointwise product.

Proof. Let k be an integer such that $1 \leqslant k \leqslant \quad$. By definition, we have:

$$p(z_k) = \sum_{i=1}^{n_{var}} \sum_{j=i}^{n_{var}} p_{i,j} \, z_i^{(k)} z_j^{(k)}.$$

The coefficient (i, k) of $Q \cdot z$ is $\sum_{j=i}^{n_{var}} p_{i,j} \, z_j^{(k)}$, and its pointwise product with z gives $z_i^{(k)} \left(\sum_{j=i}^{n_{var}} p_{i,j} \, z_j^{(k)} \right)$. Finally, the coefficient $(1, k)$ of $\begin{pmatrix} 1 & \cdots & 1 \end{pmatrix} \cdot \left(z \odot (Q \cdot z) \right)$ is

$$\sum_{i=1}^{n_{var}} z_i^{(k)} \left(\sum_{j=i}^{n_{var}} p_{i,j} \, z_j^{(k)} \right),$$

which is exactly $p(z_k)$. □

The matrix product $Q \cdot z$ is the core of this evaluation. By multiplying each coefficient of Q by the corresponding row of z, we can re-use the efficient parallel multiplication by a scalar proposed in Section 7.4.9. Finally, the pointwise product can be performed with the parallel general multiplication also described in Section 7.4.9. This process is summarized in Algorithm 44.

Remark 18. The matrix product $Q \cdot z$ could be performed with a fast matrix product, but this would be inefficient for our practical sizes of parameters.

We summarize the results obtained with Algorithm 44 in Table 9.23. The chosen parameters correspond to the 128-bit and 256-bit security levels of `Rainbow` and `DualModeMS` (Section 8.6). The multipoint evaluation is efficient over $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$: for each point, in one cycle, approximately 10 monomials of p are evaluated for $q = 16$, and 5 monomials of p are evaluated for $q = 256$. The

elements of $\mathbb{F}_{256}$ are two times larger than in $\mathbb{F}_{16}$, which explains the previous factor two. Over $\mathbb{F}_2$, the evaluation seems not to be e cient: only 26 monomials of p are evaluated in one cycle, whereas 256 elements of $\mathbb{F}_2$ can be multiplied in parallel. The monomial representation of points is not optimal for $\mathbb{F}_2$, so our final implementation uses another approach (Section 9.4.2). However, it allows to easily vectorize the evaluation of a compressed equation, since the coe cients of p are extracted one by one. Then, for $n_{\mathtt{var}}$ equals 88 and 96, we propose two values of . The first requires storing monomials on 128 bits, whereas the second requires 256 bits. On 256 bits, we completely exploit the AVX2 instruction set, whereas on 128 bits, the half of the register is not used. So, we have proposed specific implementations to fully exploit these registers. It is very e cient over $\mathbb{F}_{256}$: the number of cycles by point is similar between $= 16$ and $= 32$. The implementation is less e cient over $\mathbb{F}_{16}$, with a loss of a factor of 29% for $= 32$.

Algorithm 44 Multipoint evaluation of one MQ polynomial (with linear part and constant).

1: **function** Multipoint_evaluation_MQ($p \in \mathbb{F}_q[x_1, \dots, x_{n_{\mathtt{var}}}], z_1, \dots, z \in \mathbb{F}_q^{n_{\mathtt{var}}}$)
2: acc $\leftarrow$ $c \cdots c \in \mathbb{F}_q$ ▷ Duplication of the constant term of p.
3: **for** i from 1 to $n_{\mathtt{var}}$ **do**
4: $L \leftarrow p_i \cdots p_i \in \mathbb{F}_q$ ▷ Duplication of a linear term of p.
5: **for** j from i to $n_{\mathtt{var}}$ **do**
6: **if** $p_{i,j} \neq 0$ **then** ▷ Optional improvement.
7: **if** $p_{i,j} = 1$ **then** ▷ Optional improvement.
8: $L \leftarrow L + z_j^{(1)} \cdots z_j^{( )}$
9: **else**
10: $L \leftarrow L + z_j^{(1)} \cdots z_j^{( )} \cdot p_{i,j}$ ▷ Multiplication by a scalar.
11: **end if**
12: **end if**
13: **end for**
14: acc $\leftarrow$ acc $+ L \odot z_i^{(1)} \cdots z_i^{( )}$ ▷ Pointwise product of vectors.
15: **end for**
16: **return** acc ▷ Return $p(z_1) \cdots p(z)$ the vector of the evaluations of p.
17: **end function**

---

q	$n_{\mathtt{var}}$	N		nb. kilocycles	nb. cycles by point
2	566	160462	256	1570	6140
16	96	4656	32	18.0	563
			64	28.0	437
256	88	3916	16	12.4	774
			32	22.5	720
	188	17766	32	90.6	2830

Table 9.23: Performance of the evaluation of 1 MQ equation in $\mathbb{F}_q[x_1, \dots, x_{n_{\mathtt{var}}}]$ in points. We use a Skylake processor (LaptopS), with the AVX2 instruction set. Turbo Boost is not used. The cost of computing the monomial representation of the points is included in the measurements.

9.5 Multipoint Evaluation of Univariate Polynomial Systems

For each cryptographic operation of the dual mode (Chapter 3), we have to evaluate one or several univariate polynomials (the `MAC` polynomials), whose coefficients live in a small degree extension of $\mathbb{F}_q$. Typically, we have $q = 2$ for `DualModeMS` and $q = 16$ or $q = 256$ for `Dual Rainbow` (Section 8.6). Moreover, the extension field contains between 2^{18} and 2^{24} elements, and the degree of the `MAC` polynomials is between 2^9 and 2^{13}. The number of equations and evaluation points depend on the cryptographic operations. So, we adapt our choice of algorithms accordingly.

9.5.1 Multiplication in $\mathbb{F}_q$ and Accumulators

The core of the multipoint evaluation of the `MAC` polynomials is the multiplication in $\mathbb{F}_{q^{\ell}}$. Here, we propose efficient multiplications when $q = 2$. When $q = 16$ or $q = 256$, we use an isomorphism between $\mathbb{F}_{q^{\ell}}$ and $\mathbb{F}_{2^{\ell'}}$ with $\ell' = \ell\log_2(q)$, permitting to exploit these multiplications. For $\ell' \leqslant 8$, the multiplication can be performed in parallel with the field representation used in Section 7.4.9. For larger extension degrees of $\mathbb{F}_2$, it seems hard to have an efficient parallel multiplication. The optimal way seems be to compute sequentially each multiplication with the `PCLMULQDQ` instruction, which computes the product of two degree-63 polynomials in $\mathbb{F}_2[x]$. So, we use the polynomial representation (Section 5.2.1). We define $\mathbb{F}_{2^{\ell'}}$ as $\mathbb{F}_2$ quotiented by an irreducible trinomial of degree ℓ', if the latter exists (cf. Section B.9), or by an irreducible pentanomial of degree ℓ' otherwise. This choice permits to compute the multiplication with the `PCLMULQDQ` instruction, followed by the field modular reduction based on algorithms dedicated to sparse polynomials (Section 5.3.3). We can also use an irreducible AOP or ESP (Section 5.3.4) for certain extension degrees.

Then, we propose a trick to perform two multiplications in $\mathbb{F}_{2^{\ell'}}$ when $\ell' \leqslant 21$, with only one call to `PCLMULQDQ`. Let $a_0, a_1, b_0, b_1 \in \mathbb{F}_{2^{\ell'}}$. The computation of $a_0 \times b_0$ and $a_1 \times b_1$ can be performed as following. We put the ℓ' bits of a_0 in the lower bits of a 64-bit register, and the ℓ' bits of a_1 in the higher bits. We repeat this process for b_0, b_1. Then, we call `PCLMULQDQ` which returns the product in a 128-bit register R. Finally, the $2\ell' - 1$ lower bits of R are $a_0 \times b_0$, whereas the $2\ell'$ higher bits of R are $a_1 \times b_1$ (with the last bit which is necessarily null and so is useless). We have also computed $a_0 \times b_1 + a_1 \times b_0$ in the $2\ell' - 1$ middle bits of R, and the remaining bits are null.

Example 5 (Double product via `PCLMULQDQ`). Let $a_0, a_1, b_0, b_1 \in \mathbb{F}_{2^{21}}$. We set $A = a_0 + a_1 X^{43}$ and $B = b_0 + b_1 X^{43}$. The multiplication of A by B gives $a_0 b_0 + (a_0 b_1 + a_1 b_0) X^{43} + a_1 b_1 X^{86}$. Since the products are 41-coefficient polynomials over $\mathbb{F}_2$, the three parts of $A \times B$ are disjoint.

The structure of the result is inefficient if the products have to be reduced directly after the multiplication, because it requires extracting them before performing the modular reduction. However, this structure is adapted to the concept of accumulator. Coupled to Algorithms 45 and 46 from Section 9.5.3, this trick permits to speed up the implementation by a factor two when $\ell' \leqslant 21$. This factor can be improved with the `VPCLMULQDQ` instruction, available since September 2019 on the Ice Lake processors (Section 6.1.3). The `VPCLMULQDQ` instruction performs four times in parallel the `PCLMULQDQ` instruction, multiplying by four the performance of the Horner strategy.

Remark 19. Our trick can also be used to accumulate products of degree-1 polynomials over $\mathbb{F}_{2^{\ell'}}$ when $\ell' \leqslant 21$. The terms of degree zero, one and two from the result are respectively at the position $0, 64 - \ell'$ and $128 - 2\ell'$.

Finally, we perform the modular reduction with the shift-and-add strategy (Section 9.2.3). When the products are 64-bit aligned, this permits to compute two modular reductions in parallel with the SSE2 instruction set. For $' = 19$ and $' = 24$, we cannot define $\mathbb{F}_{2'}$ with an irreducible trinomial. So, we take an irreducible pentanomial, which slows down the performance of the modular reduction. We use the irreducible pentanomials $x^{19} + x^6 + x^5 + x + 1$ and $x^{24} + x^4 + x^3 + x + 1$. The fact that x^1 appears allows a small optimization. Moreover, the modular reduction can be accelerated thanks to the specific relation $x^{k_3} + x^{k_2} + x + 1 = (1 + x) \cdot (1 + x^{k_2})$ (Section 9.2.4).

9.5.2 Structured Evaluation Point Set

During the keypair generation of the dual mode, the number of equations is fairly large (between 48 and 512) and the number of evaluation points is very large (between 2^{18} and 2^{24}). However, we can choose these points. We use the very efficient additive FFT over $\mathbb{F}_2$ presented in Section 5.4.4. This method allows to choose a subset of points corresponding to a basis of $\log_2(\ )$ elements in $\mathbb{F}_2$. When $q = 16$ or $q = 256$, we use an isomorphism between $\mathbb{F}_q$ and $\mathbb{F}_{2'}$, with $' = \log_2(q)$, then we use the previous method to solve the multipoint evaluation problem. Here, the multiplication in $\mathbb{F}_{2'}$ is the core of the FFT. More particularly, the results have to be reduced after each multiplication. Therefore, we cannot vectorize them with the trick from Section 9.5.1.

In Table 9.24, we summarize the results obtained with the additive FFT. The chosen parameters correspond to the 128-bit and 256-bit security levels of `DualModeMS` and `Dual Rainbow`. The additive FFT is very efficient: the cost of each point is between 20 and 50 cycles, whereas the univariate polynomial has between 1000 and 8000 coefficients. The results are slower in $\mathbb{F}_{2^{24}}$ because we cannot define this field with an irreducible trinomial over $\mathbb{F}_2$.

$'$	D_{MAC}		nb. megacycles	nb. cycles by point
21	1833	2^{18}	7.00	26.7
20	931	2^{18}	6.14	23.4
	8023	2^{18}	11.1	42.3
24	1305	2^{18}	8.27	31.5
	5921	2^{18}	12.0	45.9

Table 9.24: Performance of the evaluation of a univariate polynomial of degree D_{MAC} over $\mathbb{F}_{2'}$ in points. We use a Skylake processor (LaptopS), with `PCLMULQDQ` and the AVX2 instruction set.

9.5.3 Random Evaluation Point Set

During the signing and verifying processes of the dual mode, the univariate polynomial system has to be evaluated in a small number of random points (between 16 and 539) in $\mathbb{F}_q$. In this case, the additive FFT cannot be used because the evaluation point set has to be generated from a sub-basis of $\mathbb{F}_q$. So, we use Algorithm 20 as a variant of Horner's rule (Section 5.4.3). Let a be an evaluation point from a ring. The original rule consists in repeating the process of multiplying a degree-0 polynomial by a^1 and adding it another degree-0 polynomial. In our variant (Algorithm 46), we multiply a degree-$(s-1)$ polynomial by a^s, with s a tuned parameter in function of the number of equations. This method requires a precomputation step (Algorithm 45): for each point, we have to compute the vector of its powers until s. Then, we use them to evaluate each degree-$(s-1)$ polynomial via a dot product of the coefficient vector by this of the powers of the current point.

Algorithm 45 Horner precomputation (with a step of 4).

1: **function** powers($a \in \mathcal{R}, s \in \mathbb{N}^*$)
2: $a_0 \leftarrow 1, a_1 \leftarrow a, a_2 \leftarrow a \times a, a_3 \leftarrow a_2 \times a, a_4 \leftarrow a_3 \times a$ ▷ Initialization step, a_i is $a^i \in \mathcal{R}$.
3: **for** i **from** 1 **to** $\frac{s+1}{4} - 1$ **do** ▷ The products are independent and reduced in $\mathcal{R}$.
4: $a_{4i} \leftarrow a_{4i-4} \times a_4$
5: $a_{4i+1} \leftarrow a_{4i-3} \times a_4$
6: $a_{4i+2} \leftarrow a_{4i-2} \times a_4$
7: $a_{4i+3} \leftarrow a_{4i-1} \times a_4$
8: **end for**
9: **for** j **from** 0 **to** $s + 1 \bmod 4$ **do**
10: $a_{4(i+1)+j} \leftarrow a_{4i+j} \times a_4$
11: **end for**
12: **return** $(a_0, a_1, \ldots, a_s)$
13: **end function**

Algorithm 46 Horner by block for the evaluation of a degree-d univariate polynomial G in a.

1: **function** Horner_by_block($G \in \mathbb{F}_q[X], a \in \mathbb{F}_q, s \in \mathbb{N}^*$)
2: $(a_0, a_1, \ldots, a_s) \leftarrow$ powers(a, s) ▷ Precomputation step for a fixed a.
3: $b \leftarrow \frac{d+1}{s}$, $r \leftarrow d + 1 \bmod s$
4: acc $\leftarrow 0$
5: **for** i **from** 0 **to** $b - 1$ **do**
6: **for** j **from** 0 **to** $s - 1$ **do** ▷ This loop can be unrolled (as in Algorithm 45).
7: acc $\leftarrow$ acc + $g_{i \cdot s + j} \times a^j$ ▷ Multiplication without the modular reduction.
8: **end for**
9: acc $\in \mathbb{F}_q$ ▷ Reduction in $\mathbb{F}_q$.
10: acc $\leftarrow$ acc $\times a^s$ ▷ Multiplication without the modular reduction.
11: **end for**
12: **if** $r \neq 0$ **then**
13: **for** j **from** 0 **to** $r - 1$ **do** ▷ This loop can be unrolled.
14: acc $\leftarrow$ acc + $g_{i \cdot s + j} \times a^j$ ▷ Multiplication without the modular reduction.
15: **end for**
16: acc $\in \mathbb{F}_q$ ▷ Reduction in $\mathbb{F}_q$.
17: acc $\leftarrow$ acc $\times a^s$ ▷ Multiplication without the modular reduction.
18: **end if**
19: **return** acc $\in \mathbb{F}_q$ ▷ Reduction in $\mathbb{F}_q$.
20: **end function**

In Algorithms 45 and 46, we propose to unroll loops to remove dependencies between the multiplications. In this way, the implementation can be improved with vectorized multiplications in $\mathbb{F}_{2^\ell}$ (Section 9.5.1). We summarize the results obtained in Table 9.25. The chosen parameters correspond to the 128-bit security level of `DualModeMS` and `Dual Rainbow`. The evaluations are e cient. The number of cycles to evaluate one polynomial is approximately its number of coe cients, so the evaluation of one coe cient costs just one cycle. We obtain a better performance for $\ell \leqslant 21$ by using the parallel multiplication presented in Section 9.5.1. Finally, when the number of equations

is 256 or more, the use of Algorithm 46 is similar to computing the dot product of the coefficient vector of G by the vector of the powers of a until D_{MAC}.

ℓ	D_{MAC}	nb. equations	s	nb. kilocycles	nb. cycles by equation
21	1833	256	1834	282	1102
		2	108	2.70	1348
		1	72	1.50	1504
20	931	64	156	37.6	587
		1	72	0.964	964
24	1305	48	264	77.4	1612
		1	44	2.12	2121

Table 9.25: Performance of the evaluation of univariate polynomial systems of degree D_{MAC} over $\mathbb{F}_{2^\ell}$. We use a Skylake processor (LaptopS), with PCLMULQDQ and the AVX2 instruction set.

9.6 Performance of MQsoft (Final Version)

9.6.1 Detailed Performance of HFE-Based Keypair Generation

Table 9.26 summarizes the time of most important steps of the keypair generation (Section 7.1.2). These steps are achieved in constant-time. The generation of f is computed as explained in Section 9.4.1. We have vectorized the multiplication by S and T, which is based on vector-matrix products over $\mathbb{F}_2$ implemented with the AVX2 instruction set. When we use f, the multiplication by S is the crucial part of the keypair generation (if PCLMULQDQ is available). For the evaluation-interpolation strategy (Section 7.4.7), the computation of $\mathcal{F} \circ \mathcal{S}$ via our multipoint evaluation is very efficient.

scheme	(d_{ext}, D, Δ, v)	gen. f	$f(x \cdot S)$		$\mathcal{F} \circ \mathcal{S}$	apply T
GeMSS128	(174, 513, 12, 12)	4.87	25.7	25.8	11.0	7.02
BlueGeMSS128	(175, 129, 13, 14)	3.80	26.8	26.9	9.42	7.30
RedGeMSS128	(177, 17, 15, 15)	2.28	28.1	28.6	6.89	7.63
GeMSS192	(265, 513, 22, 20)	25.3	121	100	41.5	24.8
BlueGeMSS192	(265, 129, 22, 23)	20.1	126	97.2	37.4	25.8
RedGeMSS192	(266, 17, 23, 25)	12.1	129	106	28.5	26.5
GeMSS256	(354, 513, 30, 33)	59.7	380	384	75.4	74.7
BlueGeMSS256	(358, 129, 34, 32)	47.8	393	394	67.5	76.6
RedGeMSS256	(358, 17, 34, 35)	28.6	405	410	57.4	78.1
Gui-184	(184, 33, 16, 16)	3.29	32.0	32.0	8.58	8.56
Gui-312	(312, 129, 24, 20)	27.3	197	199	47.2	48.0
Gui-448	(448, 513, 32, 28)	114	995	994	163	146

Table 9.26: Number of megacycles for main steps of the keypair generation with our library. We use a Skylake processor (LaptopS). $\mathcal{F} \circ \mathcal{S}$ is computed from $\mathcal{F}$ with the evaluation-interpolation strategy. The column $f(x \cdot S)$ is split in two. The first column corresponds to the composition of $\mathcal{F}$ with $\mathcal{S}$, whereas the second column corresponds to the composition of $\mathcal{T} \circ \mathcal{F}$ with $\mathcal{S}$.

9.6.2 Performance of HFE-Based Schemes

In this part (Table 9.27), we give the last results of performance of the HFE-based signature schemes GeMSS and Gui. The performance for the variants of GeMSS is available in Section 7.5.6, whereas the performance for a large set of parameters is given in Section 7.8.10. We have obtained interesting speed-ups compared to [84]. For Gui, we obtain big speed-ups on the keypair generation. The keypair generation of MQsoft is between 30 and 90 times faster than the round 1 implementation. This is due to our efficient evaluation-interpolation algorithm, but also to the Gui implementation which does not consider important optimizations in their evaluation-interpolation algorithm. For the signing and verifying processes, we obtain respectively factors 2.5 and 1.8. The signing process of Gui costs approximately $\sum_{i=1}^{nb_ite} \exp(i)$ times the cost of root finding (Table 9.20), which is consistent. The authors of Gui [62] claimed to achieve the EUF-CMA property (Section 7.6.2). Their method implies aborting as soon as one of the nb_ite calls to the root finding algorithm fails to find a unique root.

scheme	$(d_{\mathtt{ext}}, D, \ , v, nb_ite)$	key gen.	sign	verify
GeMSS128	(174,513,12,12,4)	19.6 × 6.03	608 × 2.09	0.106 × 1.57
GeMSS192	(265,513,22,20,4)	69.4 × 7.9	1760 × 1.83	0.304 × 1.47
GeMSS256	(354,513,30,33,4)	158 × 9.32	2490 × 2.16	0.665 × 1.76
FGeMSS(266)	(266,129,10,11,1)	53.7 × 8.22	44 × 2.9	0.0365 × 2.64
Gui-184	(184,33,16,16,3)	23.5 × 31.7	28.5 × 2.6	0.0712 × 1.89
Gui-312	(312,129,24,20,2)	116 × 41.9	308 × 2.53	0.161 × 1.85
Gui-448	(448,513,32,28,2)	356 × 91.7	5710 × 3.44	0.562 × 1.62

Table 9.27: Number of megacycles (Mc) for each cryptographic operation with our library for a Skylake processor (LaptopS), followed by the speed-up between the best implementation provided for the NIST submissions (Table 7.37) versus our implementation. For example, 19.6 × 6.03 means a performance of 19.6 Mc with MQsoft, and a performance of 19.6 × 6.03 = 118 Mc for the NIST implementations.

9.6.3 Performance of the Dual Mode

In Section 8.4.3, we presented large speed-ups obtained with MQsoft on DualModeMS. In Tables 9.28 and 9.29, we summarize the performance of the inner and dual modes. In Table 9.28, we provide two measurements for the signing process of HFE-based schemes. The first corresponds to the signing process with the time to decompress the secret-key, whereas the second assumes that the secret-key is already decompressed. This second value is useful to measure the impact of the inner mode on the dual mode: the secret-key is decompressed one time for inner signatures.

Impact of the inner mode. The keypair generation of the dual mode requires generating the keypair of the inner mode. This has a slight impact on the performance (at most 6% of the dual keypair generation). Then, the dual signing process requires generating inner signatures. This is the core of the signing process, taking between 41% and 68% for Dual Rainbow, between 79% and 92% for RedDualModeMS, and at least 94% for DualModeMS which has a long signing process.

Finally, for `Dual Rainbow`, the multipoint evaluation of multivariate quadratic equations (Section 9.4.5) is faster than the inner verifying process. Its impact is very low on the dual verifying process (7%, cf. Table 9.23). For the `HFE`-based schemes, the field is $\mathbb{F}_2$ so the multipoint evaluation is slower, but its impact on the dual mode is limited.

scheme	$(, q, n_{var}, m)$	key gen.	sign	verify
`FGeMSS(266)` / `Inner.DualModeMS128`	(128, 2, 277, 256)	53.1	44.0 (/ 43.8)	0.0368
`Inner.DualModeMS192`	(192, 2, 420, 384)	203	89.2 (/ 88.3)	0.124
`Inner.DualModeMS256`	(256, 2, 566, 512)	609	153 (/ 145)	0.377
`Inner.RedDualModeMS128`	(128, 2, 280, 256)	45.7	2.24 (/ 0.973)	0.0378
`Inner.RedDualModeMS192`	(192, 2, 423, 384)	179	5.51 (/ 1.76)	0.125
`Inner.RedDualModeMS256`	(256, 2, 569, 512)	543	10.0 (/ 3.11)	0.375
`Rainbow-Ia`	(128, 16, 96, 64)	10.5	0.0692	0.0193
`Rainbow-Ic`	(128, 256, 88, 48)	23.1	0.231	0.0591
`Rainbow-IIIc`	(192, 256, 140, 72)	97.3	0.637	0.114
`Rainbow-Vc`	(256, 256, 188, 96)	137	0.856	0.210

Table 9.28: Performance of the inner mode in megacycles. We use a Skylake processor (LaptopS), with `PCLMULQDQ` and the AVX2 instruction set. Turbo Boost is not used. The second value of the signing process corresponds to the time to sign without decompressing the secret-key.

Impact of the `MAC` polynomials. During the keypair generation, m univariate polynomials are evaluated in points of $\mathbb{F}_{2^{\log_2(q)}}$. For $= 2^{18}$, this takes between 39% and 48% of the dual keypair generation (cf. Table 9.24). For the signing process, the same polynomials are evaluated in points of $\mathbb{F}_{2^{\log_2(q)}}$. This takes a part of the signing process of `Dual Rainbow` (18% for `Dual Rainbow-Ia` and 24% for `Dual Rainbow-Ic`, cf. Table 9.25), and is negligible compared to the generation of the inner signatures in `DualModeMS` (less than 3% of the dual signing process). Finally, the evaluation of `MAC` polynomials in points is very negligible for the dual verifying process. Since ≤ 2, the evaluations are fast (between 6% and 12% of the dual verifying process for the 128-bit security level, cf. Table 9.25).

Impact of the Merkle tree. During the keypair generation, the creation of the Merkle tree requires computing digests from $(m \log_2(q))$-bit sequences and $- 2$ digests from 4 -bit sequences. For large values of , the number of computations is large and implies a slow keypair generation. The generation of the Merkle tree takes between 31% and 39% of the dual keypair generation for `Dual Rainbow`, whereas it takes between 21% and 24% for the `HFE`-based dual modes. Then, the verifying process requires computing digests from $(m \log_2(q))$-bit sequences and $\cdot (\log_2( ) -)$ digests from 4 -bit sequences. That makes between 200 and 700 digests for our parameters. The impact of the hash function on the dual verifying process is between 37% and 48% for `Dual Rainbow`, and between 21% and 32% for the `HFE`-based dual modes. The performance of the hash function gives a lower bound on the practical efficiency of the keypair generation and verifying process.

Remark 20. Here, we do not use the technique from Section 8.5.5 which minimizes the size public-key plus signature. The latter requires computing at most $2^{\lceil \log_2( )\rceil} - 2 + \cdot \log_2( ) - \lceil \log_2( )\rceil$ digests from 4 -bit sequences during the verifying process, which does not significantly change the performance compared to the parameter sets from Section 8.6 (where $= 0$).

scheme	$(, , , , \log_2( ), ,)$	key gen.	sign	verify
`DualModeMS128`	(128,64,2,21,18,18,4)	3710	2800	0.643
`DualModeMS192`	(192,96,2,20,18,33,5)	6770	8470	1.73
`DualModeMS256`	(256,256,1,20,18,51,5)	12700	38000	3.95
`RedDualModeMS128`	(128,128,1,18,18,19,5)	3800	136	0.743
`RedDualModeMS192`	(192,192,1,18,18,34,6)	6870	409	1.85
`RedDualModeMS256`	(256,256,1,18,18,53,6)	13000	1040	3.95
`Dual Rainbow-Ia`	(128,32,1,5,18,16,4)	999	3.21	0.265
`Dual Rainbow-Ic`	(128,16,1,3,18,17,4)	969	5.49	0.299
`Dual Rainbow-IIIc`	(192,24,1,3,18,31,5)	1610	26.2	0.752
`Dual Rainbow-Vc`	(256,32,1,3,18,47,6)	2540	66.3	1.63

Table 9.29: Performance of the dual mode in megacycles. We use a Skylake processor (LaptopS), with `PCLMULQDQ` and the AVX2 instruction set. Turbo Boost is not used.

Performance on Haswell. In Table 9.30, we show the impact of the processor on the dual mode. The performance of the `PCLMULQDQ` instruction impacts the use of `MAC` polynomials in all cryptographic operations. Moreover, the arithmetic in $\mathbb{F}_q$ of `Rainbow` is impacted by the performance of the AVX2 instruction set, which is slower on Haswell.

scheme	$(, , , , \log_2( ), ,)$	key gen.	sign	verify
`DualModeMS128`	(128,64,2,21,18,18,4)	3750	3650	0.635
`DualModeMS192`	(192,96,2,20,18,33,5)	6720	10800	1.77
`DualModeMS256`	(256,256,1,20,18,51,5)	11900	45200	3.84
`RedDualModeMS128`	(128,128,1,18,18,19,5)	3840	169	0.694
`RedDualModeMS192`	(192,192,1,18,18,34,6)	6820	498	1.76
`RedDualModeMS256`	(256,256,1,18,18,53,6)	12100	1260	3.88
`Dual Rainbow-Ia`	(128,32,1,5,18,16,4)	938	3.78	0.268
`Dual Rainbow-Ic`	(128,16,1,3,18,17,4)	992	7.74	0.329
`Dual Rainbow-IIIc`	(192,24,1,3,18,31,5)	1910	41.8	0.913
`Dual Rainbow-Vc`	(256,32,1,3,18,47,6)	3340	120	2.04

Table 9.30: Performance of the dual mode in megacycles. We use a Haswell processor (ServerH), with `PCLMULQDQ` and the AVX2 instruction set. Turbo Boost is not used.

Chapter 10

Approximate PoSSo

In Chapter 8, we proposed `DualModeMS` and `Dual Rainbow`, the dual modes respectively based on `HFEv-` and `Rainbow`. Their security relies on the hardness of `APoSSo` (Problem 4), which is an open question (Section 4.5). We close the gap by proving that `APoSSo` is NP-complete (Definition 1). In Section 10.1, we study `APoSSo` by using a reduction to the `Generalized MinRank` problem [80], which provides a method to solve `APoSSo`. Then, we compute the dimension of `APoSSo`. The obtained result is identical to the dimension of `Generalized MinRank`. In Section 10.2, we study the relation between `APoSSo` and `PoSSo`. We obtain that `APoSSo` is similar to solving an instance of `PoSSo` whose considered system is the minus variant of the one considered in `APoSSo`. The number of minus equations is the target rank of `APoSSo`. In particular, we introduce a reduction in polynomial time from `APoSSo` to `PoSSo`, which leads to demonstrate the NP-completeness of `APoSSo`. Finally, we confirm our results with practical experiments highlighting the behavior of `APoSSo` (Section 10.3).

10.1 Reduction from **`APoSSo`** to **`Generalized MinRank`**

We propose to consider the matrix representation of the `APoSSo` problem (Problem 4).

Problem 6. Approximate `PoSSo` (`APoSSo`$(q,\ ,m,n_{var},D,r)$), matrix version. Let q,m,n_{var},D, and r be non-negative integers such that $r < \min(\ ,m)$. Given p a degree-D multivariate polynomial system in $\mathbb{F}_q[x_1,\dots,x_{n_{var}}]^m$ and $y_1,\dots,y\ $ in $\mathbb{F}_q^m$, the problem is to find vectors $x_1,\dots,x\ $ in $\mathbb{F}_q^{n_{var}}$ such that the rank of the matrix

$$M = \begin{pmatrix} p(x_1) - y_1 \\ \vdots \\ p(x\ ) - y\ \end{pmatrix} \in \mathcal{M}_{\ ,m}\left(\mathbb{F}_q\right) \quad (10.1)$$

is less or equal to r.

This problem is similar to the `Generalized MinRank` problem [80] (Problem 7). Here, we introduce a reduction from `APoSSo` to `Generalized MinRank`.

Problem 7. `Generalized MinRank` (`GMR`$(q, \ , m, k, D, r)$). Let $q, m, k, D,$ and r be non-negative integers such that $r < \min(\ , m)$, and $\mathbb{K}$ be a field. Given M a matrix in $\mathcal{M}_{\ ,m}$ whose entries are degree-D polynomials in $\mathbb{K}[x_1, \ldots, x_k]$, the problem is to find the set of points at which the evaluation of M has rank at most r. In this thesis, we consider $\mathbb{K} = \mathbb{F}_q$.

Theorem 11. Let $q, m, n_{\mathtt{var}}, D,$ and $r < \min(\ , m)$ be integers.
Then, `APoSSo`$(q, \ , m, n_{\mathtt{var}}, D, r) \leqslant$ `GMR`$(q, \ , m, \ \cdot n_{\mathtt{var}}, D, r)$.

Proof. Let M be the matrix defined by Equation (10.1). By definition of the `APoSSo` problem, we search $x_1, \ldots, x$ in $\mathbb{F}_q^{n_{\mathtt{var}}}$ such that $\mathrm{rank}(M) \leqslant r$. Here, we do not have an instance of `Generalized MinRank`: the variables of p are shared by the columns of M, but the solution vector changes between each row. Therefore, we just extend the number of variables to $n_{\mathtt{var}}$, then we create the matrix M' in $\mathcal{M}_{\ ,m} \ \mathbb{F}_q[x_{1,1}, \ldots, x_{\ ,n_{\mathtt{var}}}]$ such that its i-th row only contains the i-th block of $n_{\mathtt{var}}$ variables. M' is as follows:

$$M' = \begin{pmatrix} p(x_{1,1}, \ldots, x_{1,n_{\mathtt{var}}}) - y_1 \\ \vdots \\ p(x_{i,1}, \ldots, x_{i,n_{\mathtt{var}}}) - y_i \\ \vdots \\ p(x_{\ ,1}, \ldots, x_{\ ,n_{\mathtt{var}}}) - y \end{pmatrix}. \quad (10.2)$$

We obtain an instance of `GMR`$(q, \ , m, \ \cdot n_{\mathtt{var}}, D, r)$, which once solved, returns a solution vector $(x_1, \ldots, x \ ) \in \mathbb{F}_q^{\ \cdot n_{\mathtt{var}}}$. The evaluation of M' in $(x_1, \ldots, x \ )$ is equal to M. So, its rank is at most r and $(x_1, \ldots, x \ )$ solves the instance of the `APoSSo` problem. □

This reduction allows to use the known attacks against `Generalized MinRank` to study the hardness of `APoSSo` and its dimension. In Section 10.3, we propose experiments to evaluate the practical hardness of solving the `APoSSo` problem. To do it, we modelize the `APoSSo` problem as a non-linear multivariate system whose equations live in $\mathbb{F}_q[x_{1,1}, \ldots, x_{\ ,n_{\mathtt{var}}}]$, then we solve it by computing its Gröbner basis (as in Section 4.4). The complexity of the Gröbner basis algorithms is exponential in the number of solutions. In `DualModeMS` (Chapter 8), we only need one solution to forge a signature. Thus, we fix variables to solve a zero dimensional system. Since each fixed variable decrements the dimension of `APoSSo`, the number of variables that we fix is the dimension of `APoSSo`. We compute it in Theorem 12 under the assumption that the top left block of size $r \times r$ of M' (Equation (10.2)) is invertible in the fraction field, then we remove this assumption in Corollary 2. The result coincides with the dimension of `Generalized MinRank` [80, Theorem 10].

Theorem 12. Let $q, m, n_{\mathtt{var}}, D,$ and $r < \min(\ , m)$ be integers, $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$ be a system of m degree-D polynomials in $n_{\mathtt{var}}$ variables, $p^* \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$ be p without its constant terms, and $y_1, \ldots, y \ \in \mathbb{F}_q^m$. If the top left $r \times r$ block of M' (Equation (10.2)) is invertible in the fraction field, if the polynomials of p^* are $\mathbb{F}_q$-linearly independent, and if $\ \cdot n_{\mathtt{var}} \geqslant (\ - r)(m - r)$, then the dimension of `APoSSo`$(q, \ , m, n_{\mathtt{var}}, D, r)$ instantiated with p and $y_1, \ldots, y$ is $\ \cdot n_{\mathtt{var}} - (\ - r)(m - r)$.

Proof. In the `APoSSo` problem, we search $(x_1, \ldots, x \ )$ such that $\mathrm{rank}(M) \leqslant r$ (Equation (10.1)). We propose to use a reduction to the `PoSSo` problem. To do it, we use linear combinations of the first r rows of M' (Equation (10.2)) to make the first r columns of its last $\ - r$ rows vanish, then we build an instance of `PoSSo` with the last $\ - r$ rows of M'.

Let

$$M'_r = \quad M'_{r,r} \quad M'_{r,m-r} \quad \in \mathcal{M}_{r,m} \quad \mathbb{F}_q[x_{1,1}, \ldots, x_{r,n_{\text{var}}}]$$

be the submatrix of M' created from its first r rows. We want to make the first r columns of the last $\;-r$ rows of M' vanish, i.e. for $r < i \leqslant \;$, we search a vector $z_i = (z_{i,1}, \ldots, z_{i,r})$ such that:

$$M'' = \begin{matrix} I_r & 0_{r,\;-r} \\ z_{r+1} & \\ \vdots & I_{\;-r} \\ z & \end{matrix} \cdot M' = \begin{matrix} M'_{r,r} & M'_{r,m-r} \\ 0_r & * \\ \vdots & \vdots \\ 0_r & * \end{matrix} \in \mathcal{M}_{\;,m} \quad \mathbb{F}_q(x_{1,1}, \ldots, x_{\;,n_{\text{var}}}) \; . \tag{10.3}$$

Let $y_k = (y_{k,1}, \ldots, y_{k,m}) \in \mathbb{F}_q^m$ for $1 \leqslant k \leqslant \;$. For a fixed i, we search z_i such that:

$$z_i \cdot M'_{r,r} = - \quad p_1(x_{i,1}, \ldots, x_{i,n_{\text{var}}}) - y_{i,1}, \ldots, p_r(x_{i,1}, \ldots, x_{i,n_{\text{var}}}) - y_{i,r} \quad .$$

This is equivalent to solving a classical linear algebra system. Under the assumption that $M'_{r,r}$ is invertible in the fraction field, i.e. in $\mathbb{F}_q(x_{1,1}, \ldots, x_{r,n_{\text{var}}})$, we solve this system by using $M'^{-1}_{r,r}$ in $\mathcal{M}_r \quad \mathbb{F}_q(x_{1,1}, \ldots, x_{r,n_{\text{var}}})$. We only use the fraction field to allow the use of Gaussian elimination (represented by $M'^{-1}_{r,r}$). We obtain:

$$z_i = - \quad p_1(x_{i,1}, \ldots, x_{i,n_{\text{var}}}) - y_{i,1}, \ldots, p_r(x_{i,1}, \ldots, x_{i,n_{\text{var}}}) - y_{i,r} \quad .M'^{-1}_{r,r}.$$

Now that $z_i \in \mathbb{F}_q(x_{1,1}, \ldots, x_{r,n_{\text{var}}}, x_{i,1}, \ldots, x_{i,n_{\text{var}}})^r$ is known, we conclude by computing Equation (10.3). For $r < i \leqslant \;$, the i-th row of M'' is given by:

$$\begin{aligned} z_i \cdot M'_r + p(x_{i,1}, \ldots, x_{i,n_{\text{var}}}) - y_i &= \quad p(x_{i,1}, \ldots, x_{i,n_{\text{var}}}) - y_i \quad \cdot I_m \\ - \quad p_1(x_{i,1}, \ldots, x_{i,n_{\text{var}}}) - y_{i,1}, \ldots, p_r(x_{i,1}, \ldots, x_{i,n_{\text{var}}}) - y_{i,r} &\quad \cdot M'^{-1}_{r,r} \cdot \quad M'_{r,r} \quad M'_{r,m-r} \quad . \end{aligned} \tag{10.4}$$

We remark that this transformation is equivalent to the computation of $p(x_{i,1}, \ldots, x_{i,n_{\text{var}}}) - y_i \quad \cdot T'$, for $T' \in \mathcal{M}_m \quad \mathbb{F}_q(x_{1,1}, \ldots, x_{r,n_{\text{var}}})$ such that:

$$T' = I_m - \begin{matrix} M'^{-1}_{r,r} \cdot M'_r \\ 0_{m-r,m} \end{matrix} = \begin{matrix} 0_{r,r} & -M'^{-1}_{r,r} \cdot M'_{r,m-r} \\ 0_{m-r,r} & I_{m-r} \end{matrix} \quad .$$

We deduce that M'' is as follows:

$$M'' = \begin{matrix} p(x_{1,1}, \ldots, x_{1,n_{\text{var}}}) - y_1 \\ \vdots \\ p(x_{r,1}, \ldots, x_{r,n_{\text{var}}}) - y_r \\ p(x_{r+1,1}, \ldots, x_{r+1,n_{\text{var}}}) - y_{r+1} \quad .T' \\ \vdots \\ p(x_{\;,1}, \ldots, x_{\;,n_{\text{var}}}) - y \quad .T' \end{matrix} \in \mathcal{M}_{\;,m} \quad \mathbb{F}_q(x_{1,1}, \ldots, x_{\;,n_{\text{var}}}) \; .$$

We have built M'' from M' by using operations on rows of M', implying $\text{rank}(M') = \text{rank}(M'')$. Now, we study when $\text{rank}(M''(\mathbf{x}_1, \ldots, \mathbf{x})) \leqslant r$. On the one hand, $M''(\mathbf{x}_1, \ldots, \mathbf{x})$ is defined only if we choose $\mathbf{x}_1, \ldots, \mathbf{x}_r$ in $\mathbb{F}_q^{n_{\text{var}}}$ such that $\det(M'_{r,r}(\mathbf{x}_1, \ldots, \mathbf{x}_r)) \neq 0$. So, $\text{rank}(M'_r(\mathbf{x}_1, \ldots, \mathbf{x}_r)) = r$. On the other hand, the last $\;-r$ rows of $M''(\mathbf{x}_1, \ldots, \mathbf{x})$ are independent of the first r rows because

$\det(M'_{r,r}(x_1,\ldots,x_r)) \neq 0$ and the first r columns of the last $\ell - r$ rows are null (Equation (10.3)). Thus, we obtain $\operatorname{rank}(M''(x_1,\ldots,x_\ell)) \leqslant r$ if and only if the last $\ell - r$ rows of $M''(x_1,\ldots,x_\ell)$ are null. The corresponding rows of M'' generate an instance of PoSSo of at most $(\ell - r)(m-r)$ non-zero equations in $\ell \cdot n_{\mathtt{var}}$ variables. To obtain the dimension of this system, we have to demonstrate that these equations are $\mathbb{F}_q$-linearly independent.

We start by demonstrating under what assumptions these $(\ell - r)(m-r)$ equations are non-zero. Let $r < i \leqslant \ell$ and $r < j \leqslant m$. From Equation (10.4), the coefficient (i,j) of M'' is zero if and only if $p_j(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,j}$ is equal to the $(j-r)$-th component of:

$$\left(p_1(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,1},\ldots,p_r(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,r}\right) \cdot M'^{-1}_{r,r} \cdot M'_{r,m-r}.$$

Since the set of variables of $p_j(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,j}$ is disjoint from these of $M'^{-1}_{r,r} \cdot M'_{r,m-r}$, this equality is possible if and only if $p_1 - y_{i,1} = \cdots = p_r - y_{i,r} = p_j - y_{i,j} = 0$ or if the $(j-r)$-th column of $M'^{-1}_{r,r} \cdot M'_{r,m-r}$ lifts in $\mathbb{F}_q$. So, we assume the negation of both previous cases. Here, we note that the second case implies that $p_j(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,j}$ is the $\mathbb{F}_q$-linear combination of the $p_k(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,k}$ for $1 \leqslant k \leqslant r$, defined by the $(j-r)$-th column of $M'^{-1}_{r,r} \cdot M'_{r,m-r}$.

Then, we demonstrate under what new assumptions the $(\ell - r)(m-r)$ equations are $\mathbb{F}_q$-linearly independent. Since the set of variables of $p_j(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,j}$ depends on $i > r$, we just verify that for a fixed i, the $p_j(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,j} - \left(p_1(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,1},\ldots,p_r(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,r}\right).M'^{-1}_{r,r}.M'_{r,m-r}$ for $j > r$ are $\mathbb{F}_q$-linearly independent. Previously, we assumed that the columns of $M'^{-1}_{r,r} \cdot M'_{r,m-r}$ do not lift in $\mathbb{F}_q$. So, we can just consider the part $p_j(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,j}$ of each equation. We obtain that the $p_j(x_{i,1},\ldots,x_{i,n_{\mathtt{var}}}) - y_{i,j}$ for $j > r$ have to be $\mathbb{F}_q$-linearly independent.

Finally, these assumptions are equivalent to considering that for all $i > r$, the vectors of $p(x_1,\ldots,x_{n_{\mathtt{var}}}) - y_i$ have to be $\mathbb{F}_q$-linearly independent. We satisfy the latter by assuming that the polynomials of p^* are $\mathbb{F}_q$-linearly independent.

By construction, if we find $(x_1,\ldots,x_\ell)$ such that $\operatorname{rank}(M''(x_1,\ldots,x_\ell)) \leqslant r$, we obtain that $\operatorname{rank}(M(x_1,\ldots,x_\ell)) \leqslant r$, and so $(x_1,\ldots,x_\ell)$ solves the instance of APoSSo. If $\ell \cdot n_{\mathtt{var}} \geqslant (\ell - r)(m-r)$, the dimension of the studied instance of PoSSo is $\ell \cdot n_{\mathtt{var}} - (\ell - r)(m-r)$, and this dimension is the dimension of APoSSo. □

In Theorem 12, we give the dimension of APoSSo by assuming that the top left block of size $r \times r$ of M' is invertible. Without loss of generality, this assumption allows to simplify the proof by considering this block for the Gaussian elimination. In the proof of Corollary 2, we demonstrate that the existence of an invertible block of size $r \times r$ in M' is enough.

Corollary 2. Let $q, m, n_{\mathtt{var}}, D, \ell$ and $r < \min(\ell, m)$ be integers, $p \in \mathbb{F}_q[x_1,\ldots,x_{n_{\mathtt{var}}}]^m$ be a system of m degree-D polynomials in $n_{\mathtt{var}}$ variables, and $p^* \in \mathbb{F}_q[x_1,\ldots,x_{n_{\mathtt{var}}}]^m$ be p without its constant terms. If the polynomials of p^* are $\mathbb{F}_q$-linearly independent, and if $\ell \cdot n_{\mathtt{var}} \geqslant (\ell - r)(m-r)$, then the dimension of APoSSo$(q,\ell,m,n_{\mathtt{var}},D,r)$ instantiated with p is $\ell \cdot n_{\mathtt{var}} - (\ell - r)(m-r)$.

Proof. Let $y_1,\ldots,y_\ell \in \mathbb{F}_q^m$. We start by permuting rows and columns of M (Equation (10.2)) in order to make invertible in the fraction field the top left block of size $r \times r$ of the transformed M'. This permutation exists because if the polynomials of p^* are $\mathbb{F}_q$-linearly independent, then we can find r columns which are $\mathbb{F}_q$-linearly independent, and so $\operatorname{rank}(M') \geqslant r$.

- Firstly, assume that $_1, \ldots, _r$ are indices such that the corresponding rows of M' are $\mathbb{F}_q$-linearly independent. In this case, there exists a permutation matrix $P_1 \in \mathcal{M} \quad \mathbb{F}_q$ such that $P_1 \cdot M'$ is the matrix M' where the rows i and $_i$ have been permuted for $1 \leqslant i \leqslant r$. The new APoSSo problem corresponds to search $(x'_1, \ldots, x')^T = P_1 \cdot (x_1, \ldots, x)^T$ in $\mathbb{F}_q^{\cdot n_{\mathtt{var}}}$ such that the dimension of the vector space generated by $p(x') - y'_1, \ldots, p(x') - y'$ is less or equal to r, for $(y'_1, \ldots, y')^T = P_1 \cdot (y_1, \ldots, y)^T$ in $\mathbb{F}_q^{\cdot m}$.
- Secondly, assume that $'_1, \ldots, '_r$ are indices such that the corresponding columns of the first r rows of $P_1 \cdot M'$ are $\mathbb{F}_q$-linearly independent. Once again, there exists a permutation matrix $P_2 \in \mathcal{M}_m \quad \mathbb{F}_q$ such that $P_1 \cdot M' \cdot P_2$ is the matrix $P_1 \cdot M'$ where the columns i and $'_i$ have been permuted for $1 \leqslant i \leqslant r$. The new APoSSo problem corresponds to search $x'_1, \ldots, x'$ in $\mathbb{F}_q^{n_{\mathtt{var}}}$ such that the dimension of the vector space generated by $p'(x') - y''_1, \ldots, p'(x') - y''$ is less or equal to r, for $p' = p \cdot P_2$ and $(y''_1, \ldots, y'')^T = (y'_1, \ldots, y')^T \cdot P_2$ in $\mathbb{F}_q^{\cdot m}$.

So, the instances of APoSSo associated to M' and $P_1 \cdot M' \cdot P_2$ are equivalent. We conclude by calling Theorem 12 on the latter instance to obtain the expected dimension. □

We confirm Corollary 2 by experiments in Magma (Section 6.2.1). We have measured the dimension of random instances of APoSSo verifying the assumptions of Corollary 2. To do it, we use the reduction to Generalized MinRank described in Theorem 11. Then, we compute the Gröbner basis of the obtained system via the F4 algorithm (Section 4.4). Thereby, we obtain the dimension. However, the degree of regularity increases quickly for small parameters, whereas the F4 algorithm is exponential in the degree of regularity. This reduces the number of parameter sets in our experiments.

In Table 10.1, we instantiate APoSSo with square systems. The practical dimension coincides with the theory. When m, $n_{\mathtt{var}}$ and r are fixed, we increase . Thus, the dimension grows like r .

r	$m = n_{\mathtt{var}}$	dimension										
1	2	3	4	5	6	7	8	9	10	11	12	13
1	3	4	5									
2	3	×	8	10	12	14	16	18				
$\longrightarrow$		2	3	4	5	6	7	8	9	10	11	12

Table 10.1: Practical dimension of APoSSo over $\mathbb{F}_{65521}$, for $D = 2$.

In Table 10.2, we take smaller values of , but we propose a larger variation of other parameters. Once again, the practical dimension coincides with the theory.

	2								3											4			
m	2			3			4		2			3						4		2	3		4
$n_{\mathtt{var}}$	2	3	4	2	3	4	3	4	2	3	4	2	3	4	2	3	4	3	4	2	3	3	4
r	1								1						2					1		2	3
dim	3	5	7	2	4	6	3	5	4	7	10	2	5	8	5	8	11	7	10	5	6	10	15

Table 10.2: Practical dimension of APoSSo over $\mathbb{F}_{65521}$, for $D = 2$.

10.2 Double Reduction between APoSSo and PoSSo

In [157], the authors quickly gave a reduction from APoSSo of m equations to PoSSo (Problem 1) of $m - r$ equations. This reduction is explained in Section 4.5. Here, we further study this reduction and how the structure of p in APoSSo impacts that of PoSSo. Then, we introduce the reduction in the opposite direction, allowing to have the equivalence. We recall that the question of the hardness of APoSSo was an open question. We start by demonstrating the reduction from APoSSo to PoSSo.

Theorem 13. Let $q, m, n_{\mathtt{var}}, D,$ and $r < \min(, m)$ be integers, $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$ be a system of m degree-D polynomials in $n_{\mathtt{var}}$ variables, $c \in \mathbb{F}_q^m$ be the vector of constant terms of p, $y_1, \ldots, y \in \mathbb{F}_q^m$, and T' be a matrix in $\mathcal{M}_m\ \mathbb{F}_q$ such that r columns are null and $\mathrm{rank}(T') = m - r$. If $c, y_1, \ldots, y_r$ are linearly independent, then we have APoSSo$(q, , m, n_{\mathtt{var}}, D, r) \leqslant$ PoSSo$(q, m - r, n_{\mathtt{var}}, D)$. Moreover, an instance of APoSSo instantiated with p and $y_1, \ldots, y$ is reduced to instances of PoSSo instantiated with $p_i' = (p - y_i) \cdot T'$ for $r < i \leqslant$.

Proof. We search $(x_1, \ldots, x\ )$ such that $\mathrm{rank}(M) \leqslant r$ (Equation (10.1)). The idea of this reduction is to fix r rows of M to reach a rank r, then use linear combinations of these rows to make r elements in the other rows of M vanish. We finish by solving instances of PoSSo to make the remaining elements vanish.

We start by randomly fixing $x_1, \ldots, x_r$. Let R be the matrix in $\mathcal{M}_{r,m}\ \mathbb{F}_q$ generated by the first r rows of M, and assume $\mathrm{rank}(R) = r$. Let R_1 and R_2 be the matrices respectively in $\mathcal{M}_r\ \mathbb{F}_q$ and $\mathcal{M}_{r,m-r}\ \mathbb{F}_q$ such that $R = \ R_1\ \ R_2\ $. Without loss of generality, we can assume that R_1 is invertible. Else, we can swap columns of M in the initial problem without changing the set of solutions.

Now, we want to make the first r columns of the last $- r$ rows of M vanish. For $r < i \leqslant$, we search a vector $z_i = (z_{i,1}, \ldots, z_{i,r})$ such that for $1 \leqslant j \leqslant r$,

$$\sum_{k=1}^{r} z_{i,k}\ \ p_j(x_k) - y_{k,j} \ = -\ p_j(x_{i,1}, \ldots, x_{i,n_{\mathtt{var}}}) - y_{i,j} \ ,$$

i.e. the linear combination of the rows of R makes $M_{i,j}$ vanish. This is equivalent to solving classical linear algebra systems. R_1 is invertible so we can find z_i such that

$$z_i \cdot R_1 = -\ p_1(x_{i,1}, \ldots, x_{i,n_{\mathtt{var}}}) - y_{i,1}, \ldots, p_r(x_{i,1}, \ldots, x_{i,n_{\mathtt{var}}}) - y_{i,r} \ .$$

We conclude by adding this linear combination to the i-th row of M. We obtain:

$$p - y_i + z_i \cdot R = (p - y_i) \cdot I_m -\ p_1(x_{i,1}, \ldots, x_{i,n_{\mathtt{var}}}) - y_{i,1}, \ldots, p_r(x_{i,1}, \ldots, x_{i,n_{\mathtt{var}}}) - y_{i,r} \ \cdot R_1^{-1} \cdot R.$$

So, this transformation is equivalent to the computation of $(p - y_i) \cdot T'$, with T' such that:

$$T' = \begin{matrix} 0_{r,r} & -R_1^{-1} \cdot R_2 \\ 0_{m-r,r} & I_{m-r} \end{matrix} \in \mathcal{M}_m\ \mathbb{F}_q\ .$$

The new instances of PoSSo have at most $m - r$ non-zero components. The transformation by T' keeps the original degree of p. So, we obtain $(- r)$ instances of PoSSo$(q, m - r, n_{\mathtt{var}}, D)$, which once solved, return $x_{r+1}, \ldots, x$. By construction, the last $- r$ rows of M are null, and so $(x_1, \ldots, x\ )$ solves the instance of APoSSo.

However, we assumed that $\mathrm{rank}(R) = r$. If $\mathrm{rank}(R) < r$, we can change the choice of $x_1, \ldots, x_r$. Since we assume that $c, y_1, \ldots, y_r$ are linearly independent, we propose to set $x_1, \ldots, x_r$ to zero. In this way, the rows of R are composed of $c - y_1, \ldots, c - y_r$, and so $\mathrm{rank}(R) = r$ by hypothesis. □

Remark 21. In a realistic attack, the adversary does not set $x_1, \ldots, x_r$ to zero, but uses the fact that $\mathrm{rank}(R) < r$ to speed up the attack. Indeed, the adversary can randomly fix $x_1, \ldots, x_r$, such that the rank of the first r rows of M is exactly r. If he finds $x_1, \ldots, x$ such that $\mathrm{rank}(M) \leqslant r$, then the instance of APoSSo is solved. Else, he attacks the last $- r'$ rows of M, generating $- r'$ instances of $\texttt{PoSSo}(q, m - r, n_{\texttt{var}}, D)$.

So, the obtained instances of PoSSo are linear combinations of the original system. We use this property to study the structure of the MI-based systems in the instances of PoSSo. In particular, Corollary 3 shows that Theorem 13 reduces HFE-based instance of APoSSo in instances of HFE− with r minus equations.

Corollary 3. Let $q, m, n_{\texttt{var}}, D,$ and $r < \min(, m)$ be integers, $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{\texttt{var}}}]^m$ be a MI-based system (like HFE and Rainbow) of m degree-D polynomials in $n_{\texttt{var}}$ variables, i.e. $p = T \circ \mathcal{F} \circ S$, $c \in \mathbb{F}_q^m$ be the vector of constant terms of p, and $y_1, \ldots, y \in \mathbb{F}_q^m$. If $\texttt{APoSSo}(q, , m, n_{\texttt{var}}, D, r)$ is instantiated with p and $y_1, \ldots, y$, and if $c, y_1, \ldots, y_r$ are linearly independent, then the reduction to $\texttt{PoSSo}(q, m - r, n_{\texttt{var}}, D)$ corresponds to solve the minus variant of a new system keeping the structure of p.

Proof. Let $p = T \circ \mathcal{F} \circ S$ be the MI-based system which instantiates $\texttt{APoSSo}(q, , m, n_{\texttt{var}}, D, r)$. According to Theorem 13, for $r < i \leqslant$, the instance of $\texttt{PoSSo}(q, m - r, n_{\texttt{var}}, D)$ is instantiated with $p_i'' = T_i'' \circ \mathcal{F} \circ S$ for T_i'' a map such that:

$$T_i'' : \mathbb{F}_q^{m'} \to \mathbb{F}_q^m$$
$$v \mapsto v \cdot T'' + t_i'',$$

where $T'' = TT'$ and $t_i'' = (t - y_i)T'$. The first r columns of T' are null, implying the first r columns of T'' and t_i'' are null. So, p_i'' has exactly $m - r$ non-zero equations, and the underlying structure corresponds to MI-based system with r minus equations.

We note that when $m' > m$, the initial MI-based system uses the minus variant. In this case, we have to only take the first m polynomials of p (Section 2.2). To be consistent, we set T the matrix generated by the first m columns of T, and set $T'' = TT'$. □

Remark 22. Theorem 13 can be used with a reduction to $\texttt{PoSSo}(q, m - r', n_{\texttt{var}}, D)$, where $r' \leqslant r$. This allows to choose the value of r' which minimizes the complexity of attacks against HFE-based instances of APoSSo.

Then, we introduce the reduction from PoSSo to APoSSo.

Theorem 14. Let $q, m, n_{\texttt{var}}, D,$ and $r < \min(, m)$ be integers. If $0 \leqslant r < m$, then $\texttt{PoSSo}(q, m - r, n_{\texttt{var}}, D) \leqslant \texttt{APoSSo}(q, r + 1, m, n_{\texttt{var}}, D, r)$. Moreover, an instance of PoSSo instantiated with $p = (p_1, \ldots, p_{m-r}) \in \mathbb{F}_q[x_1, \ldots, x_{n_{\texttt{var}}}]^{m-r}$ can be reduced to an instance of APoSSo instantiated with $p' = (0, \ldots, 0, p_1, \ldots, p_{m-r}) \in \mathbb{F}_q[x_1, \ldots, x_{n_{\texttt{var}}}]^m$.

Proof. The idea of this proof is to create an instance of APoSSo such that solving it is equivalent to finding the zeros of p. To do it, we have to take p as input of APoSSo. Then, we would like to have $\mathrm{rank}(M) \leqslant r$ if and only if (for example) its last row is null. So, we propose to choose $y_1, \ldots, y_r$ to force the first r rows of M to be independent, for any choice of $x_1, \ldots, x_r$. This choice does not seem trivial. So, we propose to expand M with the r columns of the identity matrix.

Let $p' = (0, \ldots, 0, p_1, \ldots, p_{m-r}) \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$ and $y_1, \ldots, y_{r+1} \in \mathbb{F}_q^m$ such that y_{r+1} is the null vector, and for $1 \leqslant i \leqslant r$, y_i is the vector whose ith component is minus one, and the other components are null. Assume we can solve an instance of APoSSo$(q, r+1, m, n_{\mathtt{var}}, D, r)$ instantiated with p' and $y_1, \ldots, y_{r+1}$. We obtain $x_1, \ldots, x_{r+1} \in \mathbb{F}_q^{n_{\mathtt{var}}}$ such that $\mathrm{rank}(M) \leqslant r$, where $M \in \mathcal{M}_{r+1,m}(\mathbb{F}_q)$ from Equation (10.1) becomes:

$$M = \begin{pmatrix} 1 & 0 & \cdots & 0 & p(x_1) \\ 0 & 1 & \ddots & \vdots & p(x_2) \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 1 & p(x_r) \\ 0 & \cdots & 0 & 0 & p(x_{r+1}) \end{pmatrix}.$$

The first r rows of M are independent, so $\mathrm{rank}(M) = r$. Moreover, M is in row echelon form, so $p(x_{r+1}) = 0_{m-r}$ and x_{r+1} is a solution of the instance of PoSSo$(q, m-r, n_{\mathtt{var}}, D)$. Here, we assume the existence of solutions. By construction, the instance of APoSSo does not have solutions if and only if the instance of PoSSo does not have solutions (since $\mathrm{rank}(M) = r+1$ for any choice of $x_1, \ldots, x_{r+1}$). □

Since PoSSo is NP-complete when solved over a finite field with $D \geqslant 2$ [135] (Section 2.2), we obtain by Theorem 14 that APoSSo is NP-hard. In fact, APoSSo is NP-complete, because we also have that a candidate solution of APoSSo can be verified in polynomial time (Lemma 10).

Lemma 10. APoSSo is in NP.

Proof. For demonstrating this, we provide the following verifier. In input, we consider an instance of APoSSo$(q, \ell, m, n_{\mathtt{var}}, D, r)$, and a certificate $(x_1, \ldots, x_\ell) \in \mathbb{F}_q^{\ell \cdot n_{\mathtt{var}}}$ (note that its size is polynomial in the size of the instance). We evaluate the polynomials of p in each point of this certificate, in polynomial time. Thus, we obtain the M matrix (Equation (10.1)) whose coefficients live in $\mathbb{F}_q$. Then, we compute in polynomial time its row echelon form with a classical Gaussian elimination in $\mathcal{M}_{\ell,m}(\mathbb{F}_q)$, which directly gives its rank. Finally, we check if the latter is less or equal to r. If the verification succeeds, then we return accept. Else, we return reject. □

We conclude by the reciprocal reduction between PoSSo and APoSSo (Theorem 15).

Theorem 15. Let $q, m, n_{\mathtt{var}}, D, \ell$ and $r < \min(\ell, m)$ be integers, $p \in \mathbb{F}_q[x_1, \ldots, x_{n_{\mathtt{var}}}]^m$ be a system of m degree-D polynomials in $n_{\mathtt{var}}$ variables, $c \in \mathbb{F}_q^m$ be the vector of constant terms of p, and $y_1, \ldots, y_\ell \in \mathbb{F}_q^m$. If $c, y_1, \ldots, y_r$ are linearly independent, then APoSSo$(q, \ell, m, n_{\mathtt{var}}, D, r) \equiv$ PoSSo$(q, m-r, n_{\mathtt{var}}, D)$.

Proof. By Theorems 13 and 14, and by using the trivial reduction APoSSo$(q, r+1, m, n_{\mathtt{var}}, D, r) \leqslant$ APoSSo$(q, \ell, m, n_{\mathtt{var}}, D, r)$, we have the double implication. □

10.3 Experimental Attacks on **APoSSo**

Finally, we propose some experiments to confirm the behavior of APoSSo after reduction to PoSSo. To do it, we study the hardness of solving:

- random instances of APoSSo compared to random systems (PoSSo),
- HFE-based instances of APoSSo compared to instances of HFE- (Section 2.4.1),
- Rainbow-based instances of APoSSo compared to instances of Rainbow- (Section 2.4.3).

We compute the Gröbner basis of these instances and compare the reached degree of regularity (Section 4.4.1). We optimize the attacks by randomly fixing variables until a square system is obtained, i.e. a zero dimensional system. We also fix $n = r + 1$ to reduce the APoSSo problem to just one instance of PoSSo. We note that Rainbow- is introduced only to observe the behavior of the degree of regularity.

HFE-based instances of APoSSo over $\mathbb{F}_2$. We compute the Gröbner basis over $\mathbb{F}_2$ by adding the field equations to the PoSSo system (Sections 4.3 and 4.4.1). In this case, the degree of regularity of a random system of m quadratic equations in n_{var} variables is the smallest index i such that the term z^i of the Hilbert series S (Equation (4.8)) is non-positive.

In Figure 10.1, we show the evolution of the degree of regularity in function of $m - r$, for instances of APoSSo instantiated with random square systems and HFE square systems. The instances of APoSSo instantiated with random systems of m equations have exactly the same degree of regularity as a random system of $m - r$ equations, confirming Theorem 13. Then, for HFE degree 17, the degree of regularity of the HFE-based instances of APoSSo is bounded by 4 for $r \in \{1, 2\}$, by 5 for $r \in \{3, 4\}$, and by 6 for $r = 5$. It is exactly the behavior of a HFE- system of $m - r$ equations in m variables (once r variables are randomly fixed), as studied in Table 7.17, and confirms Corollary 3.

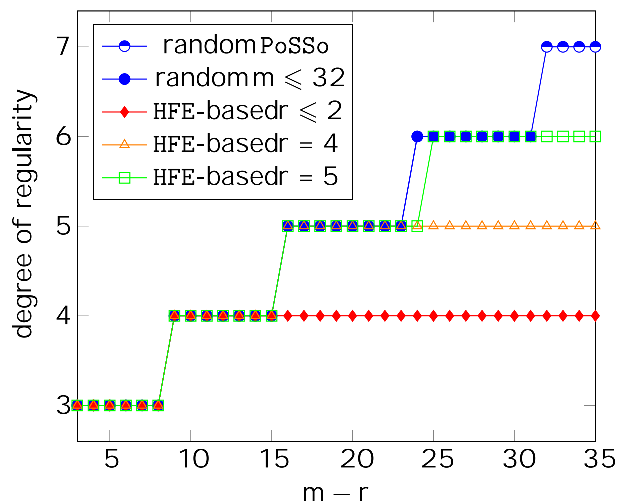

Figure 10.1: Degree of regularity of APoSSo$(2, r+1, m, m, 2, r)$ in function of $m - r$. The HFE-based instances of APoSSo have a HFE degree 17.

Rainbow-based instances of APoSSo over $\mathbb{F}_{256}$ We compute the Gröbner basis over $\mathbb{F}_{256}$ without adding the field equations to the PoSSo system. In this case, the degree of regularity of a random system of m degree d equations in $n_{\mathtt{var}}$ variables is the smallest index i such that the term z^i of the Hilbert series H (Equation (10.5)) is non-positive [14, Proposition 6].

$$H(z) = \frac{\left(1 - z^d\right)^m}{(1 - z)^{n_{\mathtt{var}}}}. \tag{10.5}$$

When $d = 2$ and $n_{\mathtt{var}} = m$, it is well-known that the degree of regularity is $m + 1$. Then, we propose to use the hybrid approach [20, 21]. We randomly fix $n_{\mathtt{var}} - m +$ variables to decrease the degree of regularity. For a random system, Equation (10.5) with $n_{\mathtt{var}} = m -$ gives the degree of regularity. In a realistic attack, an exhaustive search is performed on variables, requiring to solve q instances of PoSSo. In our experiments, we do not try to find a solution. We just compare the degree of regularity between instances of PoSSo and APoSSo to confirm the behavior of APoSSo.

In Figure 10.2, we show the evolution of the degree of regularity in function of $m - r$, for instances of APoSSo instantiated with random systems and Rainbow systems. Each curve corresponds to the behavior of a Rainbow-based APoSSo system by using the hybrid approach (we remove variables). These curves also correspond to the behavior of an instance of Rainbow with r minus equations, since we obtain exactly the same results for these systems. Our experiments are consistent with Corollary 3. We also obtain that these curves coincide with the behavior of a random instance of APoSSo, as well as a random system of $m - r$ equations. It is due to the secure choice of o_1, o_2 and v_1, which is similar to the Rainbow schemes [64] of the NIST PQC standardization process.

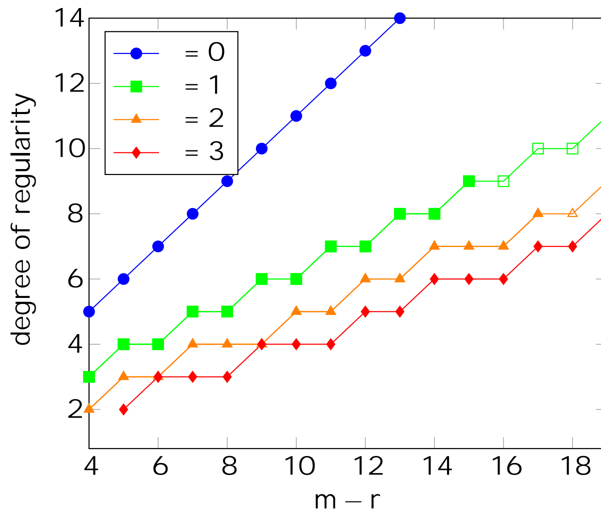

Figure 10.2: Degree of regularity of APoSSo$(256, r + 1, 2m, m, 2, r)$ in function of $m - r$, by using the hybrid approach. In the Rainbow-based instances of APoSSo, $o_1 = \frac{m}{2}$, $o_2 = \frac{m}{2}$ and $v_1 = 2m$. Here, we complete some missing experimental results, represented by empty geometric forms, by using the Hilbert series (Equation (10.5)). The other empty geometric forms are not visible because of the perfect superposition with their respective full geometric forms.

In Figure 10.3, we fix $m = 16$, and we study the degree of regularity in function of r. Here, we have chosen $v_1 = 1$ to have low instances of Rainbow. The results obtained for Rainbow-based instances of AFPoSSo of $m-r$ equations are identical to instances of Rainbow− with r minus equations, confirming Corollary 3. As for the HFE−-based schemes, the minus variant increases the security (when r is reasonable). For $r \geqslant 7$, the degrees of regularity coincide with the values of $m-r$ in Figure 10.2, which are also the degrees of regularity of a random system of $m-r$ equations.

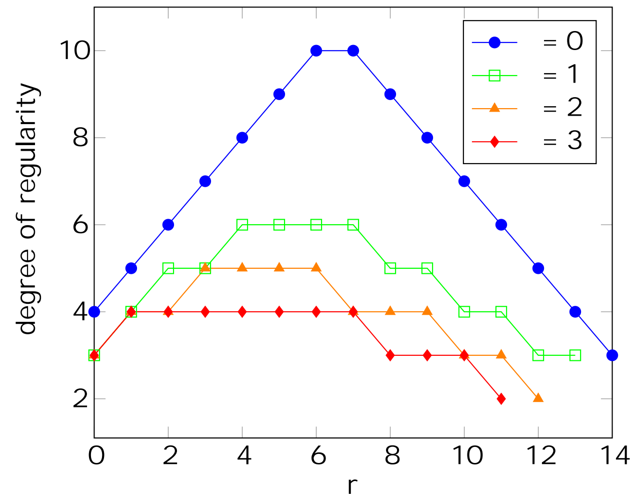

Figure 10.3: Degree of regularity of AFPoSSo$(256, r+1, 17, 16, 2, r)$ in function of r, by using the hybrid approach. Rainbow is used with $o_1 = o_2 = 8$ and $v_1 = 1$.

Conclusion

In this PhD thesis, we improve the efficiency of HFE-based schemes and the SBP transformation. HFE is a 25-year-old MI-based scheme, which provides a large public-key but very short signatures. Its signing process is equivalent to finding the roots of a sparse polynomial, which is rather slow, whereas its verifying process is fast since this corresponds to the evaluation of a multivariate quadratic system. Introduced in July 2017, the SBP technique is a secure transformation which provides dual sizes. Its security is based on the AroSSo problem, which is assumed hard to solve. Here, we summarize how this PhD thesis contributes to the history of HFE and the SBP transformation.

- We proposed two digital signature schemes to the NIST PQC standardization process. On the one hand, DualModeMS is based on a new technique, implying that it was not worked enough to be competitive with other candidates. Therefore, DualModeMS was not selected to the second round, but we greatly improve its performance in this PhD thesis. On the other hand, GeMSS is currently an alternate candidate of the third round. This scheme proposes the smallest signature size of the NIST candidates. We propose an important state-of-the-art of cryptanalysis techniques, as well as a method to select security parameters. At each round of the competition, we proposed new parameters to improve the performance of GeMSS. Unfortunately, a recent attack (November 2020 [158]) seriously impacts the security of GeMSS, as well as HFEv- in general. For the future, we should study this attack in order to repair HFE, if it is possible.

- We propose a large range of algorithms to compute efficiently arithmetic operations in finite fields, leading to a root finding algorithm in an extension field. Some of them are specialized in characteristic two. In particular, we exploit the structure of squares. We also minimize the number of modular reductions to improve the core of polynomial arithmetic. Our algorithms are dedicated to sparse polynomials, including optimizations for HFE polynomials. When several strategies are possible to solve a problem, we select the best by comparing their performance in function of the parameters.

- MQsoft is a new library which implements efficiently these algorithms. MQsoft outperforms the implementation of arithmetic in binary fields from NTL or Magma, as well as the existing implementations of HFEv--based schemes. We use the AVX2 instruction set and the PCLMULQDQ instruction to accelerate them. We are careful about the use of secret data, which is always used in constant-time. An exception is made about the degree of $\mathrm{GCD}(H, x^{q^{d_{ext}}} - x \bmod H)$ during the root finding algorithm, but we propose constant-time split root finding algorithms when the degree is strictly less than four.

- MQsoft supports the HFEv--based schemes for a large number of parameter sets. This feature allows a better understanding of the impact of the parameters on the performance.
- We propose the first efficient implementation of the SBP transformation. We succeed to obtain interesting performance, making Dual GeMSS and Dual Rainbow competitive with the other NIST candidates. These results are obtained from algorithms adapted to the parameters, an efficient use of vector instructions, as well as proposals to optimize parameters without loss of security.
- The APoSSo problem is the core of the security of the SBP transformation. We study the dimension of APoSSo, then we make the link between the APoSSo problem of a polynomial system, and the APoSSo problem of its minus variant. Both results are supported by experiments.
- We introduce the proof that APoSSo is hard in the worst case. Now, we should study if APoSSo is hard on average.

MQsoft being a new library, it can be improved. We propose some ideas to improve it.

- Our implementation of matrix operations over $\mathbb{F}_2$, such as the matrix product, determinant, Gaussian elimination and matrix inversion, does not exploit vector instructions. This could slightly improve the keypair generation of HFE-based schemes, as well as the signing process when the secret-key is generated from a seed.
- MQsoft does not exploit the AVX-512 instruction set. Our implementation could be easily extended to support it, which would significantly improve the performance.
- MQsoft does not exploit the VPCLMULQDQ instruction. The latter allows to significantly improve the polynomial multiplication over $\mathbb{F}_2$, which significantly impacts the keypair generation and signing process. However, finding the best implementation of the polynomial multiplication requires a serious and careful study.
- When the PCLMULQDQ instruction is not available, the multiplication in $\mathbb{F}_{2^{d_{ext}}}$ of MQsoft is slow. We have proposed a SSE2 implementation to decrease the slow-down, but we think the best solution is to use isomorphisms between $\mathbb{F}_{2^{d_{ext}}}$ and $\mathbb{F}_{16^{d_{ext}/4}}$ or $\mathbb{F}_{256^{d_{ext}/8}}$, as suggested for the third round. This implies implementing efficient multiplications in $\mathbb{F}_{2^{d_{ext}}}$, based on parallel multiplications in $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$
- Our polynomial operations over $\mathbb{F}_{2^{d_{ext}}}$ are based on the classical multiplication and Karatsuba's algorithm. It could be interesting to try faster multiplications such as the Toom–Cook multiplication or the FFT convolution algorithm.
- The keypair generation of the dual mode is based on an additive FFT. We use the PCLMULQDQ instruction to perform multiplications, but we should compare this method to the use of bitslicing.
- Rainbow, as submitted to the second round [64], is broken [10]. We have to update MQsoft according to the third round submission of Rainbow [65].

- We propose Dual Rainbow, the dual mode of Rainbow, but we do not consider the dual mode of cyclic Rainbow, which allows to generate a large part of the public-key from a seed. This process allows to decrease the size of the signature of its dual mode. Our implementation could be updated to support this technique.
- Finally, we could propose an implementation of Rainbow in MQsoft. For the moment, we use this of Rainbow [64] for an experimental use of its dual mode.

The author of this PhD thesis entirely implemented MQsoft, which can be improved. However, the end of this PhD thesis sounds like the end of MQsoft. We have studied many algorithms to obtain an efficient arithmetic. Proposing a new library in this direction seems to be a very interesting perspective of research. Moreover, we could improve the parallelism by coupling parallel algorithms to the use of several cores, as well as SIMD instruction sets, on the core of arithmetic. In this sense, the new HPFFA (High-Performance Finite Field Arithmetic) library could fully exploit the power of computers.

Bibliography

[1] Gorjan Alagic, Jacob Alperin-Sheri , Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the second round of the NIST post-quantum cryptography standardization process. National Institute of Standards and Technology Interagency or Internal Report 8309, July 2020. https://doi.org/10.6028/NIST.IR.8309.

[2] Gorjan Alagic, Jacob Alperin-Sheri , Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the first round of the NIST post-quantum cryptography standardization process. National Institute of Standards and Technology Internal Report 8240, January 2019. https://doi.org/10.6028/NIST.IR.8240.

[3] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):213–242, 2019.

[4] Diego F. Aranha, Armando Faz-Hernández, Julio López Hernandez, and Francisco Rodríguez-Henríquez. Faster implementation of scalar multiplication on Koblitz curves. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 177–193. Springer, 2012.

[5] Diego F. Aranha, Julio López Hernandez, and Darrel Hankerson. E cient software implementation of binary field arithmetic using vector instruction sets. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *Progress in Cryptology - LATINCRYPT 2010, First International Conference on Cryptology and Information Security in Latin America, Puebla, Mexico, August 8-11, 2010, Proceedings*, volume 6212 of *Lecture Notes in Computer Science*, pages 144–161. Springer, 2010.

[6] Diego F. Aranha, Julio López Hernandez, and Darrel Hankerson. High-speed parallel software implementation of the T pairing. In Josef Pieprzyk, editor, *Topics in Cryptology - CT-RSA 2010, The Cryptographers' Track at the RSA Conference 2010, San Francisco, CA, USA, March 1-5, 2010. Proceedings*, volume 5985 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2010.

[7] Gwénolé Ars, Jean-Charles Faugère, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. Comparison between XL and Gröbner basis algorithms. In Pil Joong Lee, editor, *Advances in*

Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings, volume 3329 of Lecture Notes in Computer Science, pages 338–353. Springer, 2004.

[8] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE, 94(2):370–382, 2006.

[9] Magali Bardet. Étude des systèmes algébriques surdéterminés. Applications aux codes correcteurs et à la cryptographie. PhD thesis, Université Pierre et Marie Curie - Paris VI, December 2004.

[10] Magali Bardet, Maxime Bros, Daniel Cabarcas, Philippe Gaborit, Ray A. Perlner, Daniel Smith-Tone, Jean-Pierre Tillich, and Javier A. Verbel. Algebraic attacks for solving the rank decoding and MinRank problems without Gröbner basis. CoRR, abs/2002.08322, 2020.

[11] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In International Conference on Polynomial System Solving – ICPSS, pages 71–75, 2004.

[12] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of the F5 Gröbner basis algorithm. J. Symb. Comput., 70:49–70, 2015.

[13] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Pierre-Jean Spaenlehauer. On the complexity of solving quadratic boolean systems. J. Complex., 29(1):53–75, 2013.

[14] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Bo-Yin Yang. Asymptotic behaviour of the degree of regularity of semi-regular polynomial systems. In The Effective Methods in Algebraic Geometry Conference – MEGA 2005, pages 1–14, 2005.

[15] Côme Berbain, Olivier Billet, and Henri Gilbert. Efficient implementations of multivariate quadratic systems. In Eli Biham and Amr M. Youssef, editors, Selected Areas in Cryptography, 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers, volume 4356 of Lecture Notes in Computer Science, pages 174–187. Springer, 2006.

[16] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[17] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings, volume 8086 of Lecture Notes in Computer Science, pages 250–272. Springer, 2013.

[18] Daniel J. Bernstein and Bo-Yin Yang. Asymptotically faster quantum algorithms to solve multivariate quadratic equations. In Lange and Steinwandt [114], pages 487–506.

[19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019(3):340–398, 2019.

[20] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Hybrid approach for solving multivariate systems over finite fields. *Mathematical Cryptology*, 3(3):177–197, 2009.

[21] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Solving polynomial systems over finite fields: improved analysis of the hybrid approach. In Joris van der Hoeven and Mark van Hoeij, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC'12, Grenoble, France - July 22 - 25, 2012*, pages 67–74. ACM, 2012.

[22] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Cryptanalysis of HFE, multi-HFE and variants for odd and even characteristic. *Des. Codes Cryptogr.*, 69(1):1–52, 2013.

[23] Ward Beullens, Jean-Charles Faugère, Eliane Koussa, Gilles Macario-Rat, Jacques Patarin, and Ludovic Perret. PKP-based signature scheme. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings*, volume 11898 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2019.

[24] Ward Beullens and Bart Preneel. Field lifting for smaller UOV public keys. In Arpita Patra and Nigel P. Smart, editors, *Progress in Cryptology - INDOCRYPT 2017 - 18th International Conference on Cryptology in India, Chennai, India, December 10-13, 2017, Proceedings*, volume 10698 of *Lecture Notes in Computer Science*, pages 227–246. Springer, 2017.

[25] Ward Beullens, Bart Preneel, and Alan Szepieniec. Public key compression for constrained linear signature schemes. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 300–321. Springer, 2018.

[26] Ward Beullens, Bart Preneel, Alan Szepieniec, and Frederik Vercauteren. LUOV. Submission to the NIST Post-Quantum Cryptography Standardization Process, December 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[27] Ward Beullens, Bart Preneel, Alan Szepieniec, and Frederik Vercauteren. LUOV. Submission to the Second Round of the NIST Post-Quantum Cryptography Standardization Process, April 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[28] Jingguo Bi, Qi Cheng, and J. Maurice Rojas. Sub-linear root detection, and new hardness results, for sparse polynomials over finite fields. In Manuel Kauers, editor, *International Symposium on Symbolic and Algebraic Computation, ISSAC'13, Boston, MA, USA, June 26-29, 2013*, pages 61–68. ACM, 2013.

[29] Olivier Billet and Jintai Ding. Overview of cryptanalysis techniques in multivariate public key cryptography. In Massimiliano Sala, Shojiro Sakata, Teo Mora, Carlo Traverso, and Ludovic Perret, editors, *Gröbner Bases, Coding, and Cryptography*, pages 263–283. Springer, 2009.

[30] Manuel Bluhm and Shay Gueron. Fast software implementation of binary elliptic curve cryptography. *J. Cryptogr. Eng.*, 5(3):215–226, 2015.

[31] Andrey Bogdanov, Thomas Eisenbarth, Andy Rupp, and Christopher Wolf. Time-area optimized public-key engines: MQ-cryptosystems as replacement for elliptic curves? *IACR Cryptol. ePrint Arch.*, 2008:349, 2008.

[32] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[33] Joppe W. Bos, Thorsten Kleinjung, Ruben Niederhagen, and Peter Schwabe. ECC2K-130 on Cell CPUs. In Daniel J. Bernstein and Tanja Lange, editors, *Progress in Cryptology - AFRICACRYPT 2010, Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3-6, 2010. Proceedings*, volume 6055 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2010.

[34] Wieb Bosma, John J. Cannon, and Graham Matthews. Programming with algebraic structures: Design of the MAGMA language. In Malcolm A. H. MacCallum, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC '94, Oxford, UK, July 20-22, 1994*, pages 52–57. ACM, 1994.

[35] Wieb Bosma, John J. Cannon, and Allan K. Steel. Lattices of compatibly embedded finite fields. *J. Symb. Comput.*, 24(3/4):351–369, 1997.

[36] Alin Bostan, Frédéric Chyzak, Marc Giusti, Romain Lebreton, Grégoire Lecerf, Bruno Salvy, and Éric Schost. Algorithmes efficaces en calcul formel, August 2017. 686 pages. Édition 1.0.

[37] Alin Bostan and Éric Schost. Polynomial evaluation and interpolation on special sets of points. *J. Complex.*, 21(4):420–446, 2005.

[38] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in F_2. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2010.

[39] Charles Bouillaguet, Pierre-Alain Fouque, and Gilles Macario-Rat. Practical key-recovery for all possible parameters of SFLASH. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 667–685. Springer, 2011.

[40] Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[41] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in GF(2)[x]. In Alfred J. van der Poorten and Andreas Stein, editors, *Algorithmic*

Number Theory, 8th International Symposium, ANTS-VIII, Banff, Canada, May 17-22, 2008, Proceedings, volume 5011 of Lecture Notes in Computer Science, pages 153–166. Springer, 2008.

[42] Richard P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. J. ACM, 25(4):581–595, 1978.

[43] Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Gröbner-basis computations with boolean polynomials. J. Symb. Comput, 44(9):1326–1345, 2009.

[44] David Brumley and Dan Boneh. Remote timing attacks are practical. Comput. Networks, 48(5):701–716, 2005.

[45] Bruno Buchberger. Bruno Buchberger's PhD thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. J. Symb. Comput, 41(3-4):475–511, 2006.

[46] Bruno Buchberger, George E. Collins, Rüdiger Loos, and R. Albrecht. Computer algebra symbolic and algebraic computation. SIGSAM Bull., 16(4):5, 1982.

[47] Jonathan F. Buss, Gudmund Skovbjerg Frandsen, and Jeffrey O. Shallit. The computational complexity of some problems of linear algebra. J. Comput. Syst. Sci., 58(3):572–596, 1999.

[48] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. GeMSS: A Great Multivariate Short Signature. Submission to the NIST Post-Quantum Cryptography Standardization Process, December 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[49] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. GeMSS: A Great Multivariate Short Signature. Submission to the Second Round of the NIST Post-Quantum Cryptography Standardization Process, April 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[50] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. GeMSS: A Great Multivariate Short Signature. Submission to the Third Round of the NIST Post-Quantum Cryptography Standardization Process, October 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[51] Anna Inn-Tung Chen, Ming-Shing Chen, Tien-Ren Chen, Chen-Mou Cheng, Jintai Ding, Eric Li-Hsiang Kuo, Frost Yu-Shuang Lee, and Bo-Yin Yang. SSE implementation of multivariate PKCs on modern x86 CPUs. In Christophe Clavier and Kris Gaj, editors, Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings, volume 5747 of Lecture Notes in Computer Science, pages 33–48. Springer, 2009.

[52] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Faster multiplication for long binary polynomials. CoRR, abs/1708.09746, 2017.

[53] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. From 5-pass MQ-based identification to MQ-based signatures. In Jung Hee Cheon and Tsuyoshi Takagi, editors, Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part, volume 10032 of Lecture Notes in Computer Science, pages 135–165, 2016.

[54] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. MQDSS. Submission to the NIST Post-Quantum Cryptography Standardization Process, December 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[55] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. MQDSS. Submission to the Second Round of the NIST Post-Quantum Cryptography Standardization Process, April 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[56] Ming-Shing Chen, Wen-Ding Li, Bo-Yuan Peng, Bo-Yin Yang, and Chen-Mou Cheng. Implementing 128-bit secure MPKC signatures. IEICE Trans. Fundam. Electron. Commun. Comput. Sci, 101-A(3):553–569, 2018.

[57] Nicolas T. Courtois. Efficient zero-knowledge authentication based on a linear algebra problem minrank. In Colin Boyd, editor, Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings, volume 2248 of Lecture Notes in Computer Science, pages 402–421. Springer, 2001.

[58] Nicolas T. Courtois. Generic attacks and the security of Quartz. In Yvo Desmedt, editor, Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings, volume 2567 of Lecture Notes in Computer Science, pages 351–364. Springer, 2003.

[59] Nicolas T. Courtois. Short signatures, provable security, generic attacks and computational security of multivariate polynomial schemes such as HFE, Quartz and Sflash. IACR Cryptol. ePrint Arch., 2004:143, 2004.

[60] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding, volume 1807 of Lecture Notes in Computer Science, pages 392–407. Springer, 2000.

[61] James H. Davenport, Christophe Petit, and Benjamin Pring. A generalised successive resultants algorithm. In Sylvain Duquesne and Svetla Petkova-Nikova, editors, Arithmetic of Finite Fields - 6th International Workshop, WAIFI 2016, Ghent, Belgium, July 13-15, 2016, Revised Selected Papers, volume 10064 of Lecture Notes in Computer Science, pages 105–124, 2016.

[62] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Gui. Submission to the NIST Post-Quantum Cryptography Standardization Process, December

2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[63] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow. Submission to the NIST Post-Quantum Cryptography Standardization Process, December 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[64] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow. Submission to the Second Round of the NIST Post-Quantum Cryptography Standardization Process, April 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[65] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang, Matthias Kannwischer, and Jacques Patarin. Rainbow. Submission to the Third Round of the NIST Post-Quantum Cryptography Standardization Process, October 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[66] Jintai Ding, Joshua Deaton, Kurt Schmidt, Vishakha, and Zheng Zhang. Cryptanalysis of the lifted unbalanced oil vinegar signature scheme. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 279–298. Springer, 2020.

[67] Jintai Ding and Thorsten Kleinjung. Degree of regularity for HFE-. *IACR Cryptol. ePrint Arch.*, 2011:570, 2011.

[68] Jintai Ding, Ray A. Perlner, Albrecht Petzoldt, and Daniel Smith-Tone. Improved cryptanalysis of HFEv- via projection. In Lange and Steinwandt [114], pages 375–395.

[69] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175, 2005.

[70] Jintai Ding and Bo-Yin Yang. Multivariate public key cryptography. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, *Post-Quantum Cryptography*, pages 193–241. Springer, Berlin, Heidelberg, 2009.

[71] Jintai Ding and Bo-Yin Yang. Degree of regularity for HFEv and HFEv-. In Philippe Gaborit, editor, *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, volume 7932 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2013.

[72] Nir Drucker and Shay Gueron. A toolbox for software optimization of QC-MDPC code-based cryptosystems. *J. Cryptogr. Eng.*, 9(4):341–357, 2019.

[73] Vivien Dubois, Pierre-Alain Fouque, Adi Shamir, and Jacques Stern. Practical cryptanalysis of SFLASH. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th*

Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings, volume 4622 of Lecture Notes in Computer Science, pages 1–12. Springer, 2007.

[74] Vivien Dubois and Nicolas Gama. The degree of regularity of HFE systems. In Masayuki Abe, editor, Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings, volume 6477 of Lecture Notes in Computer Science, pages 557–576. Springer, 2010.

[75] Christian Eder and Jean-Charles Faugère. A survey on signature-based algorithms for computing Gröbner bases. J. Symb. Comput., 80:719–784, 2017.

[76] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F4). Journal of Pure and Applied Algebra, 139(1-3):61–88, 1999.

[77] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero: F5. In ISSAC'02, pages 75–83. ACM press, 2002.

[78] Jean-Charles Faugère. Algebraic cryptanalysis of HFE using Gröbner bases. Research Report RR-4738, INRIA, 2003.

[79] Jean-Charles Faugère. FGb: A library for computing Gröbner bases. In Fukuda et al. [87], pages 84–87.

[80] Jean-Charles Faugère, Mohab Safey El Din, and Pierre-Jean Spaenlehauer. On the complexity of the generalized MinRank problem. J. Symb. Comput., 55:30–58, 2013.

[81] Jean-Charles Faugère, Kelsey Horan, Delaram Kahrobaei, Marc Kaplan, Elham Kashefi, and Ludovic Perret. Fast quantum algorithm for solving multivariate quadratic equations. CoRR, abs/1712.07211, 2017.

[82] Jean-Charles Faugère and Antoine Joux. Algebraic cryptanalysis of hidden field equation (HFE) cryptosystems using Gröbner bases. In Dan Boneh, editor, Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings, volume 2729 of Lecture Notes in Computer Science, pages 44–60. Springer, 2003.

[83] Jean-Charles Faugère, Ludovic Perret, and Jocelyn Ryckeghem. DualModeMSA Dual Mode for Multivariate-based Signature. Submission to the NIST Post-Quantum Cryptography Standardization Process, December 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[84] Jean-Charles Faugère, Ludovic Perret, and Jocelyn Ryckeghem. Software Toolkit for HFE-based Multivariate Schemes. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019(3):257–304, 2019.

[85] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings, volume 263 of Lecture Notes in Computer Science, pages 186–194. Springer, 1986.

[86] Pierre-Alain Fouque, Gilles Macario-Rat, Ludovic Perret, and Jacques Stern. Total break of the l-IC signature scheme. In Ronald Cramer, editor, *Public Key Cryptography - PKC 2008, 11th International Workshop on Practice and Theory in Public-Key Cryptography, Barcelona, Spain, March 9-12, 2008. Proceedings*, volume 4939 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2008.

[87] Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors. *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*, volume 6327 of *Lecture Notes in Computer Science*. Springer, 2010.

[88] Giordano Fusco and Eric Bach. Phase transition of multivariate polynomial systems. In Jin-yi Cai, S. Barry Cooper, and Hong Zhu, editors, *Theory and Applications of Models of Computation, 4th International Conference, TAMC 2007, Shanghai, China, May 22-25, 2007, Proceedings*, volume 4484 of *Lecture Notes in Computer Science*, pages 632–645. Springer, 2007.

[89] M. Peeters G. Bertoni, J. Daemen and G. Van Assche. A software interface for Keccak. 2013.

[90] François Le Gall. Algebraic complexity theory and matrix multiplication. In Katsusuke Nabeshima, Kosaku Nagasaka, Franz Winkler, and Ágnes Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, page 23. ACM, 2014.

[91] Shuhong Gao, Yinhua Guan, and Frank Volny. A new incremental algorithm for computing Groebner bases. In Wolfram Koepf, editor, *Symbolic and Algebraic Computation, International Symposium, ISSAC 2010, Munich, Germany, July 25-28, 2010, Proceedings*, pages 13–19. ACM, 2010.

[92] Shuhong Gao, Frank Volny IV, and Mingsheng Wang. A new framework for computing Gröbner bases. *Math. Comput.*, 85(297):449–465, 2016.

[93] Shuhong Gao and Todd D. Mateer. Additive fast Fourier transforms over finite fields. *IEEE Trans. Inf. Theory*, 56(12):6265–6272, 2010.

[94] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[95] Louis Goubin and Nicolas T. Courtois. Cryptanalysis of the TTM cryptosystem. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2000.

[96] Louis Granboulan, Antoine Joux, and Jacques Stern. Inverting HFE is quasipolynomial. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 345–356. Springer, 2006.

[97] Torbjörn Granlund and al. GNU multiple precision arithmetic library 6.1.2, December 2002. https://gmplib.org/.

[98] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying Grover's algorithm to AES: quantum resource estimates. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings*, volume 9606 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2016.

[99] Joseph F. Grcar. How ordinary elimination became gaussian elimination. *Historia Mathematica*, 38(2):163–218, 2011.

[100] Bruno Grenet, Joris van der Hoeven, and Grégoire Lecerf. Randomized root finding over finite FFT-fields using tangent Graeffe transforms. In Kazuhiro Yokoyama, Steve Linton, and Daniel Robertz, editors, *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2015, Bath, United Kingdom, July 06 - 09, 2015*, pages 197–204. ACM, 2015.

[101] Johann Großschädl and Guy-Armand Kamendje. Instruction set extension for fast elliptic curve cryptography over binary finite fields $GF(2^m)$. In *14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2003), 24-26 June 2003, The Hague, The Netherlands*, page 455. IEEE Computer Society, 2003.

[102] Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219. ACM, 1996.

[103] William B. Hart. Fast library for number theory: An introduction. In Fukuda et al. [87], pages 88–91.

[104] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[105] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Jean-Philippe Aumasson. SPHINX+. Submission to the Second Round of the NIST Post-Quantum Cryptography Standardization Process, April 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[106] IEEE standard specifications for public-key cryptography. *IEEE Std 1363-2000*, pages 1–228, 2000.

[107] Intel architecture instruction set extensions and future features programming reference, March 2020. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[108] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Inf. Comput.*, 78(3):171–177, 1988.

[109] Toshiya Itoh and Shigeo Tsujii. Structure of parallel multipliers for a class of fields $GF(2^m)$. *Inf. Comput.*, 83(1):21–40, 1989.

[110] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding, volume 1592 of Lecture Notes in Computer Science, pages 206–222. Springer, 1999.

[111] Aviad Kipnis and Adi Shamir. Cryptanalysis of the HFE public key cryptosystem by relinearization. In Michael J. Wiener, editor, Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings, volume 1666 of Lecture Notes in Computer Science, pages 19–30. Springer, 1999.

[112] Donald Ervin Knuth. The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition. Addison-Wesley, 1998.

[113] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings, volume 1109 of Lecture Notes in Computer Science, pages 104–113. Springer, 1996.

[114] Tanja Lange and Rainer Steinwandt, editors. Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings, volume 10786 of Lecture Notes in Computer Science. Springer, 2018.

[115] Tanja Lange and Tsuyoshi Takagi, editors. Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings, volume 10346 of Lecture Notes in Computer Science. Springer, 2017.

[116] A. A. Levitskaya. Systems of random equations over finite algebraic structures. Cybernetics and Systems Analysis, 41:67–93, January 2005.

[117] Daniel Lokshtanov, Ramamohan Paturi, Suguru Tamaki, R. Ryan Williams, and Huacheng Yu. Beating brute force for systems of polynomial equations over finite fields. In Philip N. Klein, editor, Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, pages 2190–2202. SIAM, 2017.

[118] Jeremy Maitin-Shepard. Optimal software-implemented Itoh-Tsujii inversion for GF(2^m). IACR Cryptol. ePrint Arch., 2015:28, 2015.

[119] Tsutomu Matsumoto and Hideki Imai. Public quadratic polynomial-tuples for efficient signature-verification and message-encryption. In Christoph G. Günther, editor, Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of of Cryptographic Techniques, Davos, Switzerland, May 25-27, 1988, Proceedings, volume 330 of Lecture Notes in Computer Science, pages 419–453. Springer, 1988.

[120] Alexander Maximov. On roots factorization for PQC algorithms. IACR Cryptol. ePrint Arch., 2020:27, 2020.

[121] Alexander Maximov and Helena Sjoberg. On fast multiplication in binary finite fields and optimal primitive polynomials over GF(2). IACR Cryptol. ePrint Arch., 2017:889, 2017.

[122] Ralph C. Merkle. A certified digital signature. In Brassard [40], pages 218–238.

[123] Mohamed Saied Emam Mohamed and Albrecht Petzoldt. The shortest signatures ever. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *Progress in Cryptology - INDOCRYPT 2016 - 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings*, volume 10095 of *Lecture Notes in Computer Science*, pages 61–77, 2016.

[124] Peter L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Trans. Computers*, 54(3):362–369, 2005.

[125] Ronald C. Mullin, I. M. Onyszchuk, Scott A. Vanstone, and R. M. Wilson. Optimal normal bases in GF(p^n). *Discret. Appl. Math.*, 22(2):149–161, 1989.

[126] National Institute of Standards and Technology. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards (FIPS) Publication 202, August 2015. http://dx.doi.org/10.6028/NIST.FIPS.202.

[127] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, December 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[128] National Institute of Standards and Technology. Round 1 submissions - post-quantum cryptography | CSRC, December 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[129] National Institute of Standards and Technology. Round 2 submissions - post-quantum cryptography | CSRC, January 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[130] National Institute of Standards and Technology. Round 3 submissions - post-quantum cryptography | CSRC, July 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[131] Thomaz Oliveira, Julio López Hernandez, Diego F. Aranha, and Francisco Rodríguez-Henríquez. Two is the fastest prime: lambda coordinates for binary elliptic curves. *J. Cryptogr. Eng.*, 4(1):3–17, 2014.

[132] Jacques Patarin. Cryptanalysis of the Matsumoto and Imai public key scheme of Eurocrypt'88. In Don Coppersmith, editor, *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 1995.

[133] Jacques Patarin. Hidden fields equations (HFE) and isomorphisms of polynomials (IP): two new families of asymmetric algorithms. In Ueli M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 1996.

[134] Jacques Patarin, Nicolas T. Courtois, and Louis Goubin. QUARTZ, 128-bit long digital signatures. In David Naccache, editor, *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2001.

[135] Jacques Patarin and Louis Goubin. Trapdoor one-way permutations and multivariate polynomials. In Yongfei Han, Tatsuaki Okamoto, and Sihan Qing, editors, *Information and Communication Security, First International Conference, ICICS'97, Beijing, China, November 11-14, 1997, Proceedings*, volume 1334 of *Lecture Notes in Computer Science*, pages 356–368. Springer, 1997.

[136] Mike Paterson and Larry J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2(1):60–66, 1973.

[137] Ludovic Perret. *Bases de Gröbner en Cryptographie Post-Quantique. (Gröbner bases techniques in Quantum-Safe Cryptography)*. 2016.

[138] Christophe Petit. Finding roots in GF(p^n) with the successive resultant algorithm. *LMS Journal of Computation and Mathematics (special issue for ANTS-XI)*, 17-A:203–217, 2014.

[139] Albrecht Petzoldt. *Selecting and reducing key sizes for multivariate cryptography*. PhD thesis, Darmstadt University of Technology, Germany, 2013.

[140] Albrecht Petzoldt. On the complexity of the hybrid approach on HFEv-. *IACR Cryptol. ePrint Arch.*, 2017:1135, 2017.

[141] Albrecht Petzoldt and Johannes A. Buchmann. A multivariate signature scheme with an almost cyclic public key. *IACR Cryptol. ePrint Arch.*, 2009:440, 2009.

[142] Albrecht Petzoldt, Stanislav Bulygin, and Johannes Buchmann. A multivariate signature scheme with a partially cyclic public key. In Carlos Cid and Jean-Charles Faugère, editors, *Proceedings of the 2nd International Conference on Symbolic Computation and Cryptography (SCC 2010)*, pages 229–235, June 2010.

[143] Albrecht Petzoldt, Stanislav Bulygin, and Johannes A. Buchmann. CyclicRainbow - A multivariate signature scheme with a partially cyclic public key. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010 - 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15, 2010. Proceedings*, volume 6498 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2010.

[144] Albrecht Petzoldt, Ming-Shing Chen, Bo-Yin Yang, Chengdong Tao, and Jintai Ding. Design principles for HFEv- based multivariate signature schemes. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 311–334. Springer, 2015.

[145] Bart Preneel. New european schemes for signature, integrity and encryption (NESSIE): A status report. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002,*

Paris, France, February 12-14, 2002, Proceedings, volume 2274 of Lecture Notes in Computer Science, pages 297–309. Springer, 2002.

[146] OpenSSL Project. OpenSSL version 1.1.1h. https://www.openssl.org/.

[147] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM, 21(2):120–126, 1978.

[148] Koichi Sakumoto, Taizo Shirai, and Harunaga Hiwatari. On provable security of UOV and HFE signature schemes against chosen-message attack. In Bo-Yin Yang, editor, Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings, volume 7071 of Lecture Notes in Computer Science, pages 68–82. Springer, 2011.

[149] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Brassard [40], pages 239–252.

[150] Peter Schwabe and Bas Westerbaan. Solving binary MQ with Grover's algorithm. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings, volume 10076 of Lecture Notes in Computer Science, pages 303–322. Springer, 2016.

[151] Adi Shamir. Efficient signature schemes based on birational permutations. In Douglas R. Stinson, editor, Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings, volume 773 of Lecture Notes in Computer Science, pages 1–12. Springer, 1993.

[152] Peter W. Shor. Polynominal time algorithms for discrete logarithms and factoring on a quantum computer. In Leonard M. Adleman and Ming-Deh A. Huang, editors, Algorithmic Number Theory, First International Symposium, ANTS-I, Ithaca, NY, USA, May 6-9, 1994, Proceedings, volume 877 of Lecture Notes in Computer Science, page 289. Springer, 1994.

[153] Victor Shoup. NTL: A library for doing number theory. January 2003. http://www.shoup.net/ntl/.

[154] Joseph H. Silverman. Fast multiplication in finite fields $GF(2^n)$. In Çetin Kaya Koç and Christof Paar, editors, Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings, volume 1717 of Lecture Notes in Computer Science, pages 122–134. Springer, 1999.

[155] Simon Stevin. L'arithmétique. Imprimerie de Christophe Plantin, 1585.

[156] Richard G. Swan. Factorization of polynomials over finite fields. Pacific J. Math., 12(3):1099–1106, 1962.

[157] Alan Szepieniec, Ward Beullens, and Bart Preneel. MQ signatures for PKI. In Lange and Takagi [115], pages 224–240.

[158] Chengdong Tao, Albrecht Petzoldt, and Jintai Ding. Improved key recovery of the HFEv- signature scheme. IACR Cryptol. ePrint Arch., 2020:1424, 2020.

[159] Jonathan Taverne, Armando Faz-Hernández, Diego F. Aranha, Francisco Rodríguez-Henríquez, Darrel Hankerson, and Julio López Hernandez. Software implementation of binary elliptic curves: Impact of the carry-less multiplier on scalar multiplication. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2011.

[160] Jeremy Vates and Daniel Smith-Tone. Key recovery attack for all parameters of HFE-. In Lange and Takagi [115], pages 272–288.

[161] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (3. ed.)*. Cambridge University Press, 2013.

[162] Christopher Wolf. "Hidden Field Equations" (HFE) - variations and attacks. Master's thesis, Universität Ulm, December 2002. http://www.christopher-wolf.de/dpl.

[163] Christopher Wolf. Efficient public key generation for HFE and variations. In Ed Dawson and Wolfgang Klemm, editors, *Cryptographic Algorithms and their Uses - 2004, International Workshop, Gold Coast, Australia, July 5-6, 2004, Proceedings*, pages 78–93. Queensland University of Technology, 2004.

[164] Christopher Wolf. *Multivariate quadratic polynomials in public key cryptography*. **Univ.** Leuven Heverlee, 2005.

[165] Christopher Wolf and Bart Preneel. Equivalent keys in multivariate quadratic public key systems. *IACR Cryptol. ePrint Arch.*, 2005:464, 2005.

Appendices

Appendix A

Size of MI-Based NIST Candidates

The secret-key of MI-based schemes (Section 2.4) is composed of two affine (or linear) maps, and a trapdoor function. In this part, we compute exactly the size of the trapdoor for HFEv, UOV and Rainbow. Then, we give the exact sizes of the corresponding NIST candidates (Tables A.1 and A.2), by taking into account their specificity.

HFEv trapdoor. We compute the number of coefficients of HFEv polynomial, which permits to derive the secret-key size. Let $r = \lfloor \log_q(D) \rfloor$. From Equation (2.9), we observe that:

- The number of coefficients of $(v_1, \ldots, v_v)$ is given by Equation (2.3) specialized for v variables and d equals two. We obtain $\binom{v+2}{2}$ coefficients in $\mathbb{F}_{q^{d_{\text{ext}}}}$ if $q \neq 2$, and $\binom{v+1}{2} + 1$ coefficients otherwise.
- The number of coefficients of each $_i(v_1, \ldots, v_v)$ is given by Equation (2.3) specialized for v variables and d equals one. We obtain $\binom{v+1}{1} = v + 1$ coefficients in $\mathbb{F}_{q^{d_{\text{ext}}}}$. Then, $_i(v_1, \ldots, v_v)$ is defined for $0 \leqslant i \leqslant r$, implying a total of $(r+1)(v+1)$ coefficients in $\mathbb{F}_{q^{d_{\text{ext}}}}$.
- The number of coefficients corresponding to $A_{i,j}$ is $\binom{r+1}{2} + \lfloor \log_q(D + 1 - q^r) \rfloor$ if $q \neq 2$, and $\binom{r}{2} + \lfloor \log_q(D + 1 - q^r) \rfloor$ otherwise. The part with the binomial corresponds to exponents strictly less than q^r, whereas the part with the logarithm counts the number of exponents strictly greater than q^r.

Therefore, the total number of coefficients of HFEv polynomial in $\mathbb{F}_{q^{d_{\text{ext}}}}[X, v_1, \ldots, v_v]$ is:

$$N_{D,v,q} = \begin{cases} \dfrac{(v+r+2)^2 + v - r}{2} + \log_q(D + 1 - q^r) & \text{if } q \neq 2, \\[2ex] \dfrac{3 + (v+r+1)^2 + v - r}{2} + \log_q(D + 1 - q^r) & \text{otherwise.} \end{cases} \tag{A.1}$$

Observe that HFE polynomials are sparse: only $N_{D,0,q} = O\left(\log_q(D)^2\right)$ coefficients are non-zero. This property is important to accelerate the root finding algorithm (Section 5.4.8) used during the signing process (Section 2.3.2).

UOV trapdoor. For one equation of $\mathcal{F}$ (Equation (2.10)), its number of coefficients in $\mathbb{F}_q$ is:

$$N_{\mathcal{F}} = \begin{cases} \frac{(v+1)(v+2)}{2} + (v+1)m' & \text{if } q \neq 2, \\ \frac{v(v+1)}{2} + 1 + (v+1)m' & \text{otherwise.} \end{cases} \tag{A.2}$$

Thus, the size of $\mathcal{F}$ is $m' N_{\mathcal{F}} \log_2(q)$ bits.

Rainbow trapdoor. The size of $\mathcal{F}$ is the sum of u trapdoors of UOV. Via Equation (A.2), we obtain that $\mathcal{F}$ contains:

$$\sum_{k=1}^{u} o_k \cdot \left(\frac{(v_k+1)(v_k+2)}{2} + (v_k+1)o_k \right) \quad \text{elements of } \mathbb{F}_q \text{ if } q \neq 2,$$

$$\sum_{k=1}^{u} o_k \cdot \left(\frac{v_k(v_k+1)}{2} + 1 + (v_k+1)o_k \right) \quad \text{elements of } \mathbb{F}_q \text{ otherwise.}$$

scheme	(level, q, d_{ext}, D, Δ, v, nb_ite)	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
QUARTZ	(80[(1)], 2, 103, 129, 3, 4, 4)	72.2375[(2)]	3.733375[(3)]	16	16
Gui-184	(II[(4)], 2, 184, 33, 16, 16, 2)	422.121[(5)]	14.984[(5)]	no seed	29[(6)]
Gui-184 (updated)	(II, 2, 184, 33, 16, 16, 3)				33[(6)]
Gui-312	(IV, 2, 312, 129, 24, 20, 2)	1990.044[(5)]	41.754[(5)]	no seed	47[(6)]
Gui-448	(VI, 2, 448, 513, 32, 28, 2)	5903.404[(5)]	94.756[(5)]	no seed	67[(6)]
GeMSS128	(I, 2, 174, 513, 12, 12, 4)	352.188	13.43775	16	32.25
BlueGeMSS128	(I, 2, 175, 129, 13, 14, 4)	363.609	13.696375		33.75
RedGeMSS128	(I, 2, 177, 17, 15, 15, 4)	375.21225	13.104		35.25
WhiteGeMSS128	(I, 2, 175, 513, 12, 12, 3)	358.172125	13.558625		29.375
CyanGeMSS128	(I, 2, 177, 129, 14, 13, 3)	369.72475	13.40675		30.5
MagentaGeMSS128	(I, 2, 178, 17, 15, 15, 3)	381.46075	13.222375		31.625
GeMSS192	(III, 2, 265, 513, 22, 20, 4)	1237.9635	34.069375	24	51.375
BlueGeMSS192	(III, 2, 265, 129, 22, 23, 4)	1264.116375	35.377375		52.875
RedGeMSS192	(III, 2, 266, 17, 23, 25, 4)	1290.542625	34.791125		54.375
WhiteGeMSS192	(III, 2, 268, 513, 21, 21, 3)	1293.84775	35.766125		46.625
CyanGeMSS192	(III, 2, 270, 129, 23, 22, 3)	1320.801625	35.26175		47.75
MagentaGeMSS192	(III, 2, 271, 17, 24, 24, 3)	1348.033375	34.69225		48.875
GeMSS256	(V, 2, 354, 513, 30, 33, 4)	3040.6995	75.892125	32	72
BlueGeMSS256	(V, 2, 358, 129, 34, 32, 4)	3087.963	71.4595		73.5
RedGeMSS256	(V, 2, 358, 17, 34, 35, 4)	3135.591	71.887375		75
WhiteGeMSS256	(V, 2, 364, 513, 31, 29, 3)	3222.69075	70.994125		64.125
CyanGeMSS256	(V, 2, 364, 129, 31, 32, 3)	3272.016375	73.201		65.25
MagentaGeMSS256	(V, 2, 366, 17, 33, 33, 3)	3321.716625	70.408125		66.375
FGeMSS(266)	(I, 2, 266, 129, 10, 11, 1)	1232.128	24.553625	16	34.625
Inner.DualModeMS128					
FGeMSS(402)	(III, 2, 402, 640, 18, 18, 1)	4243.728	62.60175	24	52.5
Inner.DualModeMS192	(III, 2, 402, 129, 18, 18, 1)		59.586750		
FGeMSS(537)	(V, 2, 537, 1152, 25, 26, 1)	10161.088	122.721875	32	70.375
Inner.DualModeMS256	(V, 2, 540, 129, 28, 26, 1)[(7)]	10269.568	116.252		70.75

(1) Original security level in bits, but QUARTZ was broken.
(2) The authors did not provide the exact size, and seemed to consider the linear terms of p. We compute the public-key size without these terms to be consistent with other schemes.
(3) The authors did not provide the exact size. However, they claimed to use 30497 random bits to generate the secret-key from the seed, when the LU decomposition is used. They considered $\mathcal{T}$ in $A^{-1}_{n_{var}}(\mathbb{F}_q)$, but in reality $\mathcal{T}$ is in $A^{-1}_{d_{ext}}(\mathbb{F}_q)$. So, the correct number is 29657 bits. We deduce the secret-key size by adding 210 bits, corresponding to the fact that the LU decomposition in $\mathbb{F}_2$ allows to save the random generation of the diagonal.
(4) The original security level was 128 bits, but the generic attack from Section 7.7.2 breaks it in 2^{112} evaluations.
(5) The sizes of [62, Table 2] are wrong. In particular, the public-key size contains the size of quadratic terms x_i^2. The sizes that we compute here are consistent with the practical implementation provided by [62] (by removing an extra useless byte from this implementation).
(6) The signature size is given without the 128-bit salt.
(7) The original scheme used $d_{ext} = 544$ and $\Delta = v = 32$. However, the correction of an error in [83, Table 13] implies $\Delta + v = 54$ instead of 64. So, we update parameters according the philosophy of GeMSS.

Table A.1: Exact size of the keys and signature of the HFEv--based schemes submitted to the NIST PQC standardization process, as well as QUARTZ from the NESSIE project [145].

round	scheme	(q, v_1, o_1, o_2)	|pk| (kB)	|sk| (kB)	|seed| (B)	|sign| (B)
1	Rainbow-Ia	(16, 32, 32, 32)	152.096	100.208	no seed	64
	Rainbow-Ib	(31, 36, 28, 28)	151.583625	106.180875		78
	Rainbow-Ic	(256, 40, 24, 24)	192.24	143.384		104
	Rainbow-IIIb	(31, 64, 32, 48)	524.40175	380.3535		112
	Rainbow-IIIc	(256, 68, 36, 36)	720.792	537.78		156
	Rainbow-IVa	(16, 56, 48, 48)	565.488	376.14		92
	Rainbow-Vc	(256, 92, 48, 48)	1723.68	1274.316		204
	Rainbow-VIa	(16, 76, 64, 64)	1351.36	892.078		118
	Rainbow-VIb	(31, 84, 56, 56)	1352.70375	944.5795		147
2	Rainbow-Ia	(16, 32, 32, 32)	148.992	92.992	no seed	64
	cyclic Rainbow-Ia		58.144		64[2]	
	Rainbow-Ic[1]	(256, 40, 24, 24)	187.968	132.64	no seed	104
	cyclic Rainbow-Ic[1]		58.496		64[2]	
	Rainbow-IIIc	(256, 68, 36, 36)	710.64	511.480	no seed	156
	cyclic Rainbow-IIIc		206.744		64[2]	
	Rainbow-Vc	(256, 92, 48, 48)	1705.536	1227.136	no seed	204
	cyclic Rainbow-Vc		491.936		64[2]	
3	Rainbow-Ia	(16, 36, 32, 32)	161.6	103.648	no seed	66
	CZ-Rainbow-Ia[3]		60.192		64[2]	
	Rainbow-IIIc	(256, 68, 32, 48)	882.08	626.048	no seed	164
	CZ-Rainbow-IIIc[3]		264.608		64[2]	
	Rainbow-Vc	(256, 96, 36, 64)	1930.6	1408.736	no seed	212
	CZ-Rainbow-Vc[3]		536.136		64[2]	

[1] `Rainbow-Ic` is not submitted to the second round of the NIST PQC standardization process. However, this implementation is available in the submitted implementation. We use this scheme in Section 8.6.3.
[2] The secret-key contains the 32-byte public seed. The latter is also stored in the public-key.
[3] `circumzenithal Rainbow`, new name of `cyclic Rainbow`.

Table A.2: Exact size of the keys and signature of `Rainbow` submitted to the NIST PQC standardization process. We consider the evolution of these schemes during the three rounds.

Appendix B

More Algorithms in $\mathbb{F}_q[x]$

In this appendix, we give more details about methods used for arithmetic in $\mathbb{F}_q[x]$ (or $\mathbb{F}_{q^{d_{\mathrm{ext}}}}[x]$ when relevant). In particular, we present some algorithms mentioned in Chapter 5.

B.1 Karatsuba-Like Formulae

In Section 5.1.2, we mentioned the three-term Karatsuba-like formulae [124], which allows to multiply two degree-two polynomials with only six multiplications (instead of seven with the standard method). Here, we explain one of these formulae [124, Equation (3) with $C = 0$].

Let A, B be polynomials in $\mathbb{F}_q[x]$ such that

$$A = A_0 + A_1 x + A_2 x^2, B = B_0 + B_1 x + B_2 x^2.$$

One way to compute $A \times B$ is to compute the six following products:

- $P_0 = A_0 \times B_0$,
- $P_1 = A_1 \times B_1$,
- $P_2 = A_2 \times B_2$,
- $Q_0 = (A_0 + A_1) \times (B_0 + B_1)$,
- $Q_1 = (A_0 + A_2) \times (B_0 + B_2)$,
- $Q_2 = (A_1 + A_2) \times (B_1 + B_2)$.

Then, we write $A \times B$ with these six products:

$$A \times B = P_0 + (Q_0 - P_0 - P_1)x + (Q_1 - P_0 + P_1 - P_2)x^2 + (Q_2 - P_1 - P_2)x^3 + P_2 x^4.$$

In characteristic two, we can save an addition by computing only one time $P_0 + P_1$ or $P_1 + P_2$.

B.2 Euclidean Division without Computing the Remainder

In Section 5.1.4, we explained that the computation of the quotient can be separated from the computation of the remainder, because the quotient depends only on the higher degree terms of A. We present this idea in Algorithm 47. We compute the quotient without updating the d_b smallest degree terms of A. Thus, the remainder could be computed as $(A - B \times Q) \bmod x^{d_b}$.

Algorithm 47 Polynomial Euclidean division without computing the remainder.

1: function EuclideanDiv($A \in \mathbb{F}_q[x], B \in \mathbb{F}_q[x]^*$)
2: $c \leftarrow b_{d_b}^{-1}$
3: $Q \leftarrow A/x^{d_b}$ Quotient of the Euclidean division of A by x^{d_b}.
4: for i from $d_a - d_b$ to $\max(1, d_b)$ by -1 do
5: $q_i \leftarrow q_i \times c$
6: $Q \leftarrow Q - q_i \times (B - b_{d_b} x^{d_b}) \times x^{i-d_b}$
7: end for
8: for i from $\max(1, d_b) - 1$ to 1 by -1 do
9: $q_i \leftarrow q_i \times c$
10: $Q \leftarrow Q - q_i \times (B - b_{d_b} x^{d_b}) / x^{d_b - i}$
11: end for
12: $q_0 \leftarrow q_0 \times c$
13: return Q
14: end function

B.3 Newton Iteration

The fast Euclidean division (Section 5.1.4) requires computing the modular inverse of F mod H, for $H = x$ with $\in \mathbb{N}^*$. This can be performed e ciently with Newton iteration [161, Algorithm 9.3]. The latter is presented in Algorithm 48, which requires $f_0 = 1$. When $f_0 = 0$, F is not invertible since x divides F. Else, when $f_0 \neq 1$, we can call Algorithm 48 with $F' = f_0^{-1} \cdot F$. The latter will return G' such that $F' \cdot G' = 1 \bmod x$, and so $f_0^{-1} \cdot G'$ is the inverse of F modulo x.

Algorithm 48 Newton iteration.

1: function NewtonIte($F \in \mathbb{F}_q[x]$ such that $f_0 = 1$, $\in \mathbb{N}^*$)
2: $r \leftarrow \lceil \log_2( ) \rceil$
3: $G_0 \leftarrow 1$ $F \cdot G_0 = 1 \bmod x$.
4: for i from 1 to r do
5: $G_{i,0} \leftarrow G_{i-1}$
6: $G_{i,1} \leftarrow (-F \cdot G_{i-1}^2 \bmod x^{\lceil \frac{}{2^{r-i}} \rceil}) / x^{\lceil \frac{}{2^{r-i+1}} \rceil}$ The quotient of the Euclidean division is the higher half of the result.
7: $G_i \leftarrow G_{i,0} + G_{i,1} \cdot x^{\lceil \frac{}{2^{r-i+1}} \rceil}$ $F \cdot G_i = 1 \bmod x^{\lceil \frac{}{2^{r-i}} \rceil}$.
8: end for
9: return G_r $F \cdot G_r = 1 \bmod x$.
10: end function

Newton iteration is an iterative process which computes F^{-1} with a precision of coefficients (i.e. $F \cdot F^{-1} = 1 \bmod x$). It starts with a precision of one coefficient, then doubles the precision at each iteration until it reaches a precision of coefficients. So, at each step, the lower half of G_i is G_{i-1}, whereas its higher half is computed with one squaring and one multiplication in $\mathbb{F}_q[x]$. The size of the polynomials increases during the algorithm, so the cost of Algorithm 48 is much less than $2\lceil \log_2( )\rceil$ operations in $\mathbb{F}_q[x]$. Its cost is bounded by $3M() + O()$ field operations [161, Exercise 9.6].

Remark 23. In characteristic two, squaring is linear and so the cost of Algorithm 48 is bounded by $2M() + O()$ field operations.

B.4 FFT Variant of the Polynomial Evaluation

In Section 5.4.3, we proposed a Baby-Step Giant-Step approach to compute the evaluation of $G \in \mathbb{F}_q[x]$ in an element $a \in \mathcal{R}$, for $\mathcal{R}$ a ring. Here, we note that G can also be split by using the FFT decomposition (Section 5.4.4). In Algorithm 49, the b blocks are generated from the coefficients whose indices are equal modulo b. This strategy is very similar to the Baby-Step Giant-Step strategy. Roughly, we just swap a and c compared to Algorithm 20, and the matrix from Step 2 is transposed.

Algorithm 49 Polynomial evaluation using the FFT decomposition.

Input: $G \in \mathbb{F}_q[x], a \in \mathcal{R}, b \in \mathbb{N}^*$ such that $b \leqslant d_g + 1$.
Output: $G(a)$.

0. Let $s = \frac{d_g+1}{b}$ and $G(x) = \sum_{i=0}^{b-1} G_i\left(x^b\right) \cdot x^i$, with $G_i \in \mathbb{F}_q[x]$, $\deg(G_i) < s$.
1. Let $c = a^b$ and compute $c, c^2, c^3, \ldots, c^{s-1}$ (e.g. with Algorithm 45). These are the baby steps.
2. Re-use them to evaluate $G_0, \ldots, G_{b-1}$ in c. These evaluations can be performed with the following matrix-vector product:
$$\begin{pmatrix} g_0 & \cdots & g_{b(s-2)} & g_{b(s-1)} \\ g_1 & \cdots & g_{b(s-2)+1} & g_{b(s-1)+1} \\ \vdots & \ddots & \vdots & \vdots \\ g_{b-1} & \cdots & g_{b(s-1)-1} & g_{bs-1} \end{pmatrix} \cdot \begin{pmatrix} 1_{\mathcal{R}} \\ c \\ \vdots \\ c^{s-1} \end{pmatrix} = \begin{pmatrix} G_0(c) \\ G_1(c) \\ \vdots \\ G_{b-1}(c) \end{pmatrix}.$$
3. Compute $G(a)$ as:
$$\sum_{i=0}^{b-1} G_i(c) \times a^i. \tag{B.1}$$
Equation (B.1) can be performed with Horner's rule, which allows to save the computation of the powers of a. These are the giant steps.

The cost of Algorithm 49 is similar to Algorithm 20, with an additional penalty to compute a^b during Step 1.

B.5 Frobenius Map, Right-to-Left Version

Here, we present the right-to-left square-and-multiply algorithm of the Frobenius map (Algorithm 50) described in Section 5.4.5. Algorithm 50 requires $\lfloor\log_2(k)\rfloor + \mathtt{HW}(k) - 1$ modular compositions in $\mathbb{F}_{q^{d_{ext}}}[x]$ and $(k-1)\cdot d_h$ q-exponentiations in $\mathbb{F}_{q^{d_{ext}}}$. However, the latter can be computed with $(\lfloor\log_2(k)\rfloor + \mathtt{HW}(k) - 1)\cdot d_h$ calls to Algorithm 22, using $\lfloor\log_2(k)\rfloor + \mathtt{HW}(k) - 1$ multi-squaring tables.

Algorithm 50 Frobenius map using the right-to-left square-and-multiply algorithm on q^k.

```
1:  function FrobMapRL(H ∈ F_{q^{d_ext}}[x], k ∈ N*)
2:      r ← ⌊log_2(k)⌋
3:      X_0 ← x^q mod H                                    ▷ X_0 = x^{q^{2^0}} mod H.
4:      for i from 1 to r do
5:          X_i ← X_{i-1}^{(2^{i-1})} ∘ X_{i-1} mod H      ▷ Square: X_i = x^{q^{2^i}} mod H.
6:      end for      ▷ We can compute X_i^{(k mod 2^i)} then X_i^{(2^i)} for ⌊k/2^i⌋ mod 2 = 1.
7:      i_0 ← 0
8:      while ⌊k/2^{i_0}⌋ mod 2 = 0 do
9:          i_0 ← i_0 + 1
10:     end while            ▷ i_0 is the greatest integer such that 2^{i_0} divides k.
11:     Y_{i_0} ← X_{i_0}          ▷ Y_{i_0} = x^{q^{2^{i_0}}} = x^{q^{k mod 2^{i_0+1}}} mod H.
12:     for i from i_0 + 1 to r do
13:         if ⌊k/2^i⌋ mod 2 = 1 then          ▷ All steps of multiply are computed here.
14:             Y_i ← X_i^{(k mod 2^i)} ∘ Y_{i-1} mod H       ▷ Y_i = x^{q^{k mod 2^{i+1}}} mod H.
15:         else
16:             Y_i ← Y_{i-1}
17:         end if
18:     end for
19:     return Y_r                                 ▷ i = r and Y_r = x^{q^k} mod H.
20: end function
```

B.6 Structured Exponentiation and Frobenius Norm

In Section 5.4.5, we studied how to optimize the exponentiation to the power q^k. The Hamming weight of q^k in base q is one and so can be performed with k exponentiations to the power q or by using multi-squaring tables. Here, we study how to optimize the exponentiation to the power $q^k - 1$ for $k \in \mathbb{N}$, which is useful to compute the inverse by Fermat's little theorem (Equation (5.11)). The inversion in $\mathbb{F}^{\times}_{q^{d_{ext}}}$ requires the exponent $q^{d_{ext}} - 2$ that we can write as $(q^k - 1)\times q + q - 2$, for $k = d_{ext} - 1$ (cf. Algorithm 52). We show that the exponentiation algorithm used in $\mathtt{ITMIA}$ [108] can be generalized when the power is the sum of the terms of a geometric sequence with common ratio q. In particular, this concerns the Frobenius norm, defined by:

$$N_k(x) = \prod_{i=0}^{k-1} x^{q^i} = x^{\sum_{i=0}^{k-1} q^i}. \tag{B.2}$$

Raising an element A to the power of $q^k - 1$ with the square-and-multiply algorithm (where square is a q-exponentiation) is expensive. The Hamming weight of $q^k - 1$ in base q is k, implying $k-1$ steps of multiplication. In 1988, T. Itoh and S. Tsujii introduced the Itoh–Tsujii Multiplicative Inversion Algorithm (ITMIA) [108]. This algorithm allows to compute the power $q^k - 1$ with only $O(\log_2(k))$ steps of multiplication. Firstly introduced for $q = 2$ (Algorithm 36), we remark that the strategy used can be naturally generalized for any power having the form

$$K = c \sum_{i=0}^{\frac{k}{\ell}-1} q^{\ell \cdot i} = c \times \frac{q^k - 1}{q^\ell - 1}, \quad \ell \in \mathbb{N}^*, c < q^\ell .$$

In fact, K is the $(k - \ell)$-degree -ESP [109] evaluated in q^ℓ and multiplied by the constant c (Section 5.3.4). This form allows to perform operations directly on the power of q (from the numerator) instead of the power of A. Similarly to Equation (5.14), the following properties are used to perform the multiplication by two and the addition on the power of q:

$$\begin{aligned} A^{c\frac{q^{2k'}-1}{q^\ell-1}} &= \left(A^{c\frac{q^{k'}-1}{q^\ell-1}}\right)^{q^{k'}} \times A^{c\frac{q^{k'}-1}{q^\ell-1}} \\ A^{c\frac{q^{k'+k''}-1}{q^\ell-1}} &= \left(A^{c\frac{q^{k'}-1}{q^\ell-1}}\right)^{q^{k''}} \times A^{c\frac{q^{k''}-1}{q^\ell-1}}. \end{aligned} \tag{B.3}$$

So, K can be computed by decomposing $\frac{k}{\ell}$ in base two, then by applying the square-and-multiply algorithm, where squaring corresponds to multiply the exponent by two, whereas multiply corresponds to add ℓ to the exponent (i.e. $k'' = \ell$ in Equation (B.3)). We obtain Algorithm 51, which requires only $O\left(\log_2\left(\frac{k}{\ell}\right)\right)$ steps of multiplication.

Algorithm 51 Itoh–Tsujii exponentiation for a specific (left-to-right) addition chain. $\mathcal{R}$ is a ring.

1: **function** ITexp($A \in \mathcal{R}, q \geqslant 2, k \in \mathbb{N}^*, \ell \in \mathbb{N}^*, c < q^\ell$)
2: $\quad n \leftarrow \frac{k}{\ell}$
3: $\quad m \leftarrow 1$ ▷ Case $i = \lfloor \log_2(n) \rfloor$.
4: $\quad A_K \leftarrow A^c$ ▷ $A^{c \sum_{i=0}^{m-1} q^{\ell i}} = A^c$.
5: $\quad$ **for** i from $\lfloor \log_2(n) \rfloor - 1$ to 0 by -1 **do**
6: $\quad\quad Q \leftarrow A_K^{q^{\ell \cdot m}}$ ▷ Multi-squaring to obtain $A^{c \sum_{i=m}^{2m-1} q^{\ell i}}$.
7: $\quad\quad A_K \leftarrow Q \times A_K$ ▷ $A^{c \sum_{i=0}^{2m-1} q^{\ell i}}$.
8: $\quad\quad m \leftarrow \left\lfloor \frac{n}{2^i} \right\rfloor$
9: $\quad\quad$ **if** $m \bmod 2 = 1$ **then**
10: $\quad\quad\quad A_K \leftarrow A_K^{q^\ell} \times A^c$ ▷ $\left(A^{c \sum_{i=0}^{m-2} q^{\ell i}}\right)^{q^\ell} \times A^c = A^{c \sum_{i=0}^{m-1} q^{\ell i}}$.
11: $\quad\quad$ **end if**
12: $\quad$ **end for** ▷ $m = n$ and so $A_K = A^{c \sum_{i=0}^{\frac{k}{\ell}-1} q^{\ell i}}$.
13: $\quad$ **return** A_K
14: **end function**

Link with the Frobenius trace. Consider the modified Algorithm 51 where the multiplications are replaced by additions (i.e. $\times$ becomes $+$). Thus, we obtain Algorithm 23 where y is initialized to A and the modular composition is replaced by an exponentiation (the Frobenius map is removed). With this change, we can compute $\mathrm{Tr}_k(r_0 x) \in \mathcal{R} = \mathbb{F}_{q^{d_{ext}}}[x]/(H)$ with $\mathrm{ITexp}(r_0 x, q, k, 1, 1)$ and $\mathrm{HTr}_k(A) \in \mathcal{R} = \mathbb{F}_{2^{d_{ext}}}$ with $\mathrm{ITexp}(A, 2, k+1, 2, 1)$, both with only $O(\log_2(k))$ additions.

Use of the modular composition. As in Section 5.4.6, the modular composition can be used when the exponentiation is computed modulo a univariate polynomial. A first version of this strategy is proposed in [161, Algorithm 14.55] to compute the Frobenius norm (Equation (B.2)). This algorithm can be improved and generalized by coupling Algorithms 50 and 51, or by modifying Algorithm 23.

Applications. The inverse in $\mathbb{F}_{q^{d_{ext}}}^{\times}$ can be computed by using Algorithm 52, which is the so-called Itoh–Tsujii Multiplicative Inversion Algorithm when $q = 2$ (Algorithm 36).

Algorithm 52 `ITMIA` for a specific (left-to-right) addition chain.

1: **function** Inverse($A \in \mathbb{F}_{q^{d_{ext}}}^{\times}$)
2: $\quad A_K \leftarrow \mathrm{ITexp}(A, q, d_{ext}-1, 1, q-1)$ $\qquad \triangleright\ A_K = A^{\sum_{i=0}^{d_{ext}-2}(q-1)q^i} = A^{q^{d_{ext}-1}-1}$.
3: $\quad$ **return** $A_K^q \times A^{q-2}$ $\qquad \triangleright\ A^{q^{d_{ext}}-2} = A^{-1}$.
4: **end function**

The Itoh–Tsujii exponentiation can be used for other structured powers. In Example 6, we propose a simple idea to compute the cubic root. This method can be used to compute r-th roots in $\mathbb{F}_{2^{d_{ext}}}$ for certain values of r and d_{ext}.

Example 6 (Cubic root in $\mathbb{F}_{2^{d_{ext}}}$). Let d_{ext} be an odd positive integer, $K = \frac{1}{3}\left(2^{d_{ext}+1} - 1\right) \in \mathbb{N}$ and $A \in \mathbb{F}_{2^{d_{ext}}}$. We have $K = 1 \times \frac{2^{d_{ext}+1}-1}{2^2-1} = \sum_{i=0}^{\frac{d_{ext}+1}{2}-1} 2^{2i}$, i.e. $K = 111\ldots111$ in base four. So, we can compute A^K with $\mathrm{ITexp}(A, 2, d_{ext}+1, 2, 1)$. Note that A^K is the cubic root of A, since $A^{3K} = A^{2^{d_{ext}+1}-1} = A^{2^{d_{ext}}} A^{2^{d_{ext}}-1} = A$.

B.7 Degree-Two Split Root Finding in Characteristic Two

In Section 5.4.7, we presented classical methods to solve $X^2 + X + A = 0$ in $\mathbb{F}_{2^{d_{ext}}}[X]$. Here, we show that the computation of the half-trace is equivalent to computing a vector-matrix product over $\mathbb{F}_2$, whose matrix depends only on the field polynomial of $\mathbb{F}_{2^{d_{ext}}}$. Then we study the behavior of M_2 (Equation (5.16)) for irreducible trinomials and AOPs when we solve $X^2 + X + A = 0$ as a linear system over $\mathbb{F}_2$. We also study how to solve efficiently this equation in normal basis.

When d_{ext} is odd, we have:

$$R = \mathrm{HTr}_{d_{ext}}(A) = \sum_{i=0}^{\frac{d_{ext}-1}{2}} A^{4^i} = \sum_{j=0}^{d_{ext}-1} a_j \,\mathrm{HTr}_{d_{ext}}(\theta^j).$$

Thus, we have:

$$\begin{pmatrix} r_0 & r_1 & \cdots & r_{d_{ext}-1} \end{pmatrix} = \begin{pmatrix} a_0 & a_1 & \cdots & a_{d_{ext}-1} \end{pmatrix} \cdot M_H$$

where

$$M_H = \begin{pmatrix} (\mathrm{HTr}_{d_{ext}}(\ _0)) \\ (\mathrm{HTr}_{d_{ext}}(\ _1)) \\ \vdots \\ (\mathrm{HTr}_{d_{ext}}(\ _{d_{ext}-1})) \end{pmatrix} \in \mathcal{M}_{d_{ext}}\ \mathbb{F}_2 .$$

This method is roughly equivalent to proposing the inverse of a specific $M_2 \in GL_{d_{ext}}\ \mathbb{F}_2$ for solving the following classical linear algebra problem:

$$r_0\ \ r_1\ \ \cdots\ \ r_{d_{ext}-1}\ \cdot M_2 = a_0\ \ a_1\ \ \cdots\ \ a_{d_{ext}-1}$$

where

$$M_2 = \begin{pmatrix} {}_0^2 + \ _0 \\ {}_1^2 + \ _1 \\ \vdots \\ {}_{d_{ext}-1}^2 + \ _{d_{ext}-1} \end{pmatrix} = I_{d_{ext}} + \begin{pmatrix} {}_0^2 \\ {}_1^2 \\ \vdots \\ {}_{d_{ext}-1}^2 \end{pmatrix} \in \mathcal{M}_{d_{ext}}\ \mathbb{F}_2 .$$

Solving $X^2 + X + A = 0$ in the canonical basis for a field trinomial. Now, we consider $\mathcal{B}$ the canonical basis of $\mathbb{F}_{2^{d_{ext}}}$. We start by making the sparse structure of M_2 explicit for field trinomials. By using results from Section 5.3.3, we obtain:

$$M_2 = I_{d_{ext}} + \begin{cases} (\ _{2i}) & 0 \leqslant 2i < d_{ext} \\ (\ _{2j-d_{ext}} + \ _{2j-d_{ext}+k_1}) & d_{ext} \leqslant 2j < 2d_{ext} - k_1 \\ (\ _{2k-d_{ext}} + \ _{2k-2d_{ext}+k_1} + \ _{2k-2d_{ext}+2k_1}) & 2d_{ext} - k_1 \leqslant 2k \leqslant 2d_{ext} - 2 \end{cases} \quad \text{(B.4)}$$

Then, we study how to make M_2 invertible. To do it, we introduce the following lemma.

Lemma 11. Let $f_3 = x^{d_{ext}} + x^{k_1} + 1 \in \mathbb{F}_2[x]$ for $0 < k_1 \leqslant \frac{d_{ext}}{2}$, and $f \in \mathbb{F}_2[x]$ be an irreducible polynomial of degree d_{ext}. We define $\mathbb{F}_{2^{d_{ext}}}$ as $\mathbb{F}_2[\ ]/(f(\ ))$. We have the following properties:

1. If d_{ext} is odd and if $\mathcal{D}_e(f) \leqslant 0$, then the zero-th column of M_2 is null.

2. If d_{ext} is even, if k_1 is odd and if $f = f_3$, then the $(d_{ext} - k_1)$-th column of M_2 is null.

3. If d_{ext} is odd, if k_1 is even and if $f = f_3$, then the zero-th and $(d_{ext} - k_1)$-th columns of M_2 are equal. Moreover, only the $(d_{ext} - \frac{k_1}{2})$-th row is set.

Proof. Let $d = d_{ext} - k_1$. We prove each property.

1. Let Q_i and R_i in $\mathbb{F}_2[x]$ be respectively the quotient and the remainder of the Euclidean division of x^{2i} by f for $0 \leqslant i < d_{ext}$. From Lemma 9, we know that $\mathcal{D}_e(Q_i) = -\infty$, i.e. Q_i does not have even degree terms. In particular, x divides Q_i. For $0 < i < d_{ext}$, we obtain that x divides $x^{2i} - f \cdot Q_i = R_i$, and so x divides the reduced form of $x^{2i} + x^i$. For $i = 0$, x divides $x^{2i} + x^i = 0$. All in all, the zero-th column of M_2 is null.

2. We check that the d-th column of each row of M_2 (Equation (B.4)) is null.

   - For $0 \leqslant 2i < d_{ext}$, d is odd so $\ _d \neq \ _{2i}$.

- For $d_{\mathrm{ext}} \leqslant 2j < 2d_{\mathrm{ext}} - k_1$, we have $j = 2j - d_{\mathrm{ext}} + k_1$ and so ${}_{2j-d_{\mathrm{ext}}+k_1} + {}_{j} = 0$ for $j = d$. Otherwise, ${}_{d} \neq {}_{2j-d_{\mathrm{ext}}+k_1}$. Note that ${}_{d} \neq {}_{2j-d_{\mathrm{ext}}}$ since $2j - d_{\mathrm{ext}} < d$.
- For $2d_{\mathrm{ext}} - k_1 \leqslant 2k \leqslant 2d_{\mathrm{ext}} - 2$, d is odd so ${}_{d} \neq {}_{2k-2d_{\mathrm{ext}}+2k_1}$. Moreover, d_{ext} is even implies ${}_{d} \neq {}_{2k-d_{\mathrm{ext}}}$ and $k_1 \leqslant \frac{d_{\mathrm{ext}}}{2}$ implies ${}_{d} \neq {}_{2k-2d_{\mathrm{ext}}+k_1}$.

3. We check that the zero-th and d-th columns of each row of M_2 (Equation (B.4)) are null except for the $(d_{\mathrm{ext}} - \frac{k_1}{2})$-th row.

   - For $0 \leqslant 2i < d_{\mathrm{ext}}$, d is odd so ${}_{d} \neq {}_{2i}$, whereas ${}_{2\times 0} + {}_{0} = 0$ and ${}_{0} \neq {}_{2i}$ otherwise.
   - For $d_{\mathrm{ext}} \leqslant 2j < 2d_{\mathrm{ext}} - k_1$, we have $j = 2j - d_{\mathrm{ext}} + k_1$ and so ${}_{2j-d_{\mathrm{ext}}+k_1} + {}_{j} = 0$ for $j = d$. Otherwise, ${}_{d} \neq {}_{2j-d_{\mathrm{ext}}+k_1}$. Then, d_{ext} is odd so ${}_{0} \neq {}_{2j-d_{\mathrm{ext}}}$. Note that ${}_{d} \neq {}_{2j-d_{\mathrm{ext}}}$ and ${}_{0} \neq {}_{2j-d_{\mathrm{ext}}+k_1}$ since $2j - d_{\mathrm{ext}} < d$ and $2j - d_{\mathrm{ext}} + k_1 \geqslant k_1 > 0$.
   - For $2d_{\mathrm{ext}} - k_1 \leqslant 2k \leqslant 2d_{\mathrm{ext}} - 2$, d is odd so ${}_{d} \neq {}_{2k-2d_{\mathrm{ext}}+2k_1}$. Moreover, $k_1 \leqslant \frac{d_{\mathrm{ext}}}{2}$ implies ${}_{d} \neq {}_{2k-2d_{\mathrm{ext}}+k_1}$. Finally, k_1 is even implies ${}_{2k-2d_{\mathrm{ext}}+k_1} = {}_{0}$ and ${}_{2k-d_{\mathrm{ext}}} = {}_{d}$ when $k = d_{\mathrm{ext}} - \frac{k_1}{2}$. Otherwise, ${}_{d} \neq {}_{2k-d_{\mathrm{ext}}}$ and ${}_{0} \neq {}_{2k-2d_{\mathrm{ext}}+k_1}$. Note that ${}_{0} \neq {}_{2k-d_{\mathrm{ext}}}$ and ${}_{0} \neq {}_{2k-2d_{\mathrm{ext}}+2k_1}$ since $2k - d_{\mathrm{ext}} \geqslant d_{\mathrm{ext}} - k_1$ and $2k - 2d_{\mathrm{ext}} + 2k_1 \geqslant k_1$. □

From Lemma 11, coupled to the fact that the zero-th row of M_2 is null, we deduce:

- When $k_1, \ldots, k = d_{\mathrm{ext}}$ are odd, including irreducible trinomials (= 2) and pentanomials (= 4), the zero-th column of M_2 is null and this implies that $a_0 = 0$. We can make M_2 invertible by setting to one the coefficient $(0, 0)$ of M_2.

- For trinomials, when d_{ext} is even and k_1 is odd, the $(d_{\mathrm{ext}} - k_1)$-th column of M_2 is null and this implies that $a_{d_{\mathrm{ext}}-k_1} = 0$. We can make M_2 invertible by setting to one the coefficient $(0, d_{\mathrm{ext}} - k_1)$ of M_2.

- For trinomials, when d_{ext} is odd and k_1 is even, the zero-th and $(d_{\mathrm{ext}} - k_1)$-th columns of M_2 are equal and this implies that $a_0 = a_{d_{\mathrm{ext}}-k_1}$ (we also know that $r_{d_{\mathrm{ext}}-\frac{k_1}{2}} = a_0$). We can make M_2 invertible by setting to one the coefficient $(0, 0)$ or the coefficient $(0, d_{\mathrm{ext}} - k_1)$ of M_2.

Lemma 11 allows to precompute more easily the inverse of a chosen M_2 for irreducible trinomials such that $k_1 \leqslant \frac{d_{\mathrm{ext}}}{2}$, as well as for certain pentanomials. We can also use M_{H} when d_{ext} is odd.

Solving $X^2 + X + A = 0$ in the normal basis. When we consider $\mathcal{B}$ the normal basis of $\mathbb{F}_{2^{d_{\mathrm{ext}}}}$ (Section 5.2.2), $X^2 + X + A = 0$ can be solved very efficiently. This is due to the relationship $(r_i \cdot {}_{i})^2 = r_i \cdot {}_{i+1 \bmod d_{\mathrm{ext}}}$, which implies that:

$$r_0 = r_{d_{\mathrm{ext}}-1} + a_0,$$
$$r_{i+1} = r_i + a_{i+1} \text{ for } 0 \leqslant i < d_{\mathrm{ext}} - 1.$$

Since R and $R + 1$ are solutions of $X^2 + X + A = 0$, with $(1) = (1, \ldots, 1)$, we can set one variable to zero or one, e.g. $r_{d_{\mathrm{ext}}-1} = 0$. Thus, we obtain an iterative process to compute a solution of $X^2 + X + A = 0$, requiring only $O(d_{\mathrm{ext}})$ operations in $\mathbb{F}_2$. In fact, we have:

$$r_{d_{\mathrm{ext}}-1} = 0,$$
$$r_i = \sum_{j=0}^{i} a_j = \sum_{j=i+1}^{d_{\mathrm{ext}}-1} a_j \text{ for } 0 \leqslant i < d_{\mathrm{ext}}.$$

This process is iterative but we can compute successive r_i in parallel. To do so, duplicate the current r_i in a -bit word acc. Then, compute $r_{i+1}, \ldots, r_{i+}$ with parallel additions in $\mathbb{F}_2$ as follows: for $1 \leqslant j \leqslant$, duplicate a_{i+j} in a -bit word, use a precomputed mask to mask its first $j-1$ elements, and finally xor it to acc. Store the obtained result and repeat this process with $i+$ instead of i as times as necessary.

Additionally, we make M_2 explicit. Note that the previous method is a sophisticated multiplication of (A) by the inverse of M_2, and so is necessarily faster than the latter.

$$M_2 = \begin{pmatrix} 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & 1 \\ 1 & 0 & \cdots & 0 & 1 \end{pmatrix}.$$

Solving $X^2 + X + A = 0$ in the canonical basis for a field AOP. When the field polynomial f is an AOP (Section 5.3.4), d_{ext} is even and M_2 has the following form:

$$M_2 = I_{d_{ext}} + \begin{pmatrix} (\;_{2i}) & 0 \leqslant i < \frac{d_{ext}}{2} \\ (f(\;_1) - \;_1^{d_{ext}}) & \\ (\;_{2j-1-d_{ext}}) & \frac{d_{ext}}{2} < j < d_{ext}. \end{pmatrix}$$

The columns of M_2 generate the following system:

$$\begin{aligned} r_{\frac{d_{ext}}{2}} &= a_0, \\ r_i + a_i + r_{\frac{d_{ext}}{2}} &= \begin{cases} r_{\frac{i}{2}} & \text{if } i \text{ is even, for } 0 \leqslant i < d_{ext} - 1, \\ r_{\frac{d_{ext}}{2} + \frac{i+1}{2}} & \text{if } i \text{ is odd, for } 0 \leqslant i < d_{ext} - 1, \end{cases} \\ r_{d_{ext}-1} &= r_{\frac{d_{ext}}{2}} + a_{d_{ext}-1}. \end{aligned}$$

Once again, we can solve $X^2 + X + A = 0$ with an iterative process [154]. From $a_{\frac{d_{ext}}{2}}$, we deduce r_I for a certain $I \in \mathbb{N}^*$. Then, we compute $r_I + a_I + a_0$ and obtain r_J for a certain $J \in \mathbb{N}^*$. By repeating this process, we obtain all values of r_i for $0 < i < d_{ext}$. Indeed, from Lemma 1, 2 is a generator of $\mathbb{F}^{\times}_{d_{ext}+1}$, and $J = I \cdot 2^{-1} = I \cdot \left(\frac{d_{ext}}{2} + 1\right) \bmod (d_{ext}+1)$. Moreover, we stop this process as soon as $I = d_{ext} - 1$, avoiding the indices d_{ext} and $\frac{d_{ext}}{2}$ of r_J. Thus, we avoid d_{ext} and set $r_{\frac{d_{ext}}{2}} = a_0$.

Let $g_j = \frac{d_{ext}}{2} \cdot \left(\frac{d_{ext}}{2} + 1\right)^j \in \mathbb{F}^{\times}_{d_{ext}+1}$ for $j \in \mathbb{N}$. In fact, we have:

$$\begin{aligned} \sum_{j=0}^{d_{ext}-2} a_{g_j} &= \sum_{j=1}^{d_{ext}-1} a_j = 0, \\ r_{g_0} &= a_0, \\ r_{g_i} + (1 - i \bmod 2) \cdot a_0 &= \sum_{j=0}^{i-1} a_{g_j} = \sum_{j=i}^{d_{ext}-2} a_{g_j} \quad \text{for } 0 \leqslant i < d_{ext} - 1. \end{aligned}$$

Once g_j is known (or precomputed) for $0 \leqslant j < d_{\text{ext}} - 1$, solving this system requires only $O(d_{\text{ext}})$ operations in $\mathbb{F}_2$. We remark that $r_{g_i} = r_{g_{i-1}} + a_{g_{i-1}} + a_0$ for $0 < i < d_{\text{ext}} - 1$, and the division by two is cheap since $n \cdot 2^{-1} \bmod (d_{\text{ext}} + 1)$ is exactly $\lfloor \frac{n}{2} \rfloor + (n \bmod 2) \cdot \frac{d_{\text{ext}}}{2} + 1$ for $0 \leqslant n \leqslant d_{\text{ext}}$.

B.8 Constant-Time GCD for Berlekamp's Algorithm

In this section, we present an adaptation of the constant-time GCD algorithm of [19] (Section 7.4.11) for the root finding algorithm (Algorithm 25). This adaptation works in $\mathbb{F}_q[x]$ for any q. Its extended version can be used for computing the inverse in $\mathbb{F}_q^\times$ (Section 5.4.2).

During the root finding algorithm, $\mathrm{GCD}(R_2, H)$ has to be computed. We could compute it in constant-time via [19], but the strategy used requires the degree of inputs is public and reached. The polynomial H has exactly a degree d_h, but R_2 is the result of the Frobenius map modulo H. Its degree is secret, and less or equal to $d_h - 1$.

However, we observe that the constant-time GCD algorithm of [19] can be used with only one (non-zero) operand whose degree is public and reached. A public upper bound on the degree of the other operand is enough. The algorithm takes this upper bound, and while the current (assumed) leading term is null, the degree will decrement. If a non-zero term is found, then the exact degree is found, and the algorithm runs with the original requirements.

In Algorithm 53, we present a constant-time implementation of this idea, accepting an upper bound on the degree of the left operand. Our description is a bit different from the original [19, Figure 5.1], for several reasons.

- We do not take the reciprocal polynomial of inputs. This implies the use of multiplications by x instead of divisions by x. We think it simplifies the implementation.
- We swap the operands. This multiplies δ by minus one compared to the original algorithm. This choice is consistent with the Euclidean division algorithm. The (assumed) higher degree is naturally the left operand in a Euclidean division (since it is common to write the dividend at the left of the divisor).
- The original algorithm deals with power series field. For our practical use, we restrict it to the polynomial ring. We make this change carefully, in particular about the precision of computations in power series field. To do that, we do not consider the monomials $x^0, \dots, x^{d_{\max} - \mathrm{nb_step}}$ during Steps 16, 17 and 18 if $\mathrm{nb_step} \leqslant d_{\max}$. In fact, the new F from Step 18 is multiple of $x^{\max(0, d_{\max}+1-\mathrm{nb_step})}$ because the old F is zero, or H and the old F are multiple of $x^{\max(0, d_{\max}+1-\mathrm{nb_step})}$. We know that $f_0, \dots, f_{d_{\max}-\mathrm{nb_step}}$ are null.

With Algorithm 53, we can now compute $\mathrm{GCD}(R_2, H)$ as $\mathrm{GCD_EuclidStevin}(R_2, H, d_h - 1)$. This method requires $3 \cdot (d_h^2 - d_h)$ field multiplications.

Remark 24. As explained in Section 5.4.1, the Euclid–Stevin relationship (Step 18) can be replaced by a classical Euclidean relationship, i.e. $F \leftarrow F - f_{d_{\max}} \cdot h_{d_{\max}}^{-1} \cdot H$. Thus, we obtain a constant-time version of the classical Euclidean algorithm. We know that $h_{d_{\max}}$ is always invertible. The computation of $f_{d_{\max}} \cdot h_{d_{\max}}^{-1}$ requires one inversion field but divides by two the number of field multiplications.

Algorithm 53 Constant-time GCD of F and H, where d_h is public and d_f is a public upper bound on $\deg(F)$.

1: **function** GCD_EuclidStevin($F \in \mathbb{F}_q[x], H \in \mathbb{F}_q[x]^*, d_f \geqslant \deg(F)$)
2: $d_h \leftarrow \deg(H), d_{max} \leftarrow \max(d_f, d_h)$
3: $\leftarrow d_f - d_h$ ▷ Gap between d_f and d_h.
4: **if** < 0 **then** ▷ Alignment of the leading terms of F and H.
5: $F \leftarrow F \cdot x^{-}$
6: **else**
7: $H \leftarrow H \cdot x$
8: **end if**
9: **for** nb_step **from** $d_f + d_h$ **to** 1 **by** -1 **do**
10: **if** $f_{d_{max}} \neq 0$ **and** < 0 **then** ▷ Use of constant-time comparison.
11: $b_s \leftarrow 1$
12: **else**
13: $b_s \leftarrow 0$
14: **end if**
15: $\leftarrow$ XOR (XOR $-$) $\cdot b_s$ ▷ When F and H are swapped, $\leftarrow -$.
16: $G \leftarrow (F$ XOR $H) \cdot b_s$
17: $F \leftarrow F$ XOR $G, H \leftarrow H$ XOR G ▷ Conditional swap.
18: $F \leftarrow h_{d_{max}} \cdot F - f_{d_{max}} \cdot H$ ▷ Note that the new $f_{d_{max}}$ is necessarily null. Moreover, if nb_step $\leqslant d_{max}$, then the new $f_0, \ldots, f_{d_{max}-\text{nb_step}}$ are necessarily null.
19: $F \leftarrow F \cdot x, \leftarrow -1$
20: **end for**
21: $d \leftarrow \frac{}{2}$ ▷ Integer division (is even).
22: **return** $H/x^{d_{max}+d}, -d$ ▷ Return the GCD and its degree. The division (a simple shift of coefficients) has to be performed in constant-time.
23: **end function**

Remark 25. In characteristic two, the conditional swap of F and H does not modify the result of $h_{d_{max}} \cdot F - f_{d_{max}} \cdot H$. It is due to the symmetry that appears in this relationship. Instead of the conditional swap, H should be updated independently of F once the relationship is computed.

About the constant-time computation of $H/x^{d_{max}+d}$. At the end of Algorithm 53, we have to compute $H/x^{d_{max}+d}$. This operation is equivalent to shifting by $d_{max}+d$ positions the coefficients of H. Since $-d \leqslant d_h$, we can perform in variable-time $H' = H/x^{d_{max}-d_h}$, then compute H'/x^{d_h+d} in constant-time. Our approach is to perform secretly d_h shifts by one position. The first $d_h + d$ are true shifts of coefficients, unlike the last $-d$ shifts which are dummy shifts. This approach requires $\frac{1}{2} \cdot d_h \cdot (d_h + 1)$ uses of masks and xor on elements of $\mathbb{F}_q$. In our practical use, when we reject polynomials having more than x roots (Section 7.4.12), we can apply the previous strategy with x instead of d_h. This turns to be cheap for small values of x.

Remark 26. We propose to keep secret the degree of the GCD during the root finding algorithm. However, we do not have constant-time split root finding algorithm, except for small degrees. So, when we use a variable-time split root finding algorithm, the division by $x^{d_{max}+d}$ is performed in variable-time.

B.9 List of Irreducible Polynomials over $\mathbb{F}_2$

In Section 5.3, we propose several kinds of irreducible polynomials over $\mathbb{F}_2$ to create a degree-$d_{\mathtt{ext}}$ extension field of $\mathbb{F}_2$, for $d_{\mathtt{ext}} \geqslant 2$. Here, we enumerate the possibilities for small enough extension degrees. For $d_{\mathtt{ext}} \geqslant 4$, it is conjectured that an irreducible pentanomial of degree $d_{\mathtt{ext}}$ exists.

For $d_{\mathtt{ext}} < 593$, there exist irreducible trinomials of degree $d_{\mathtt{ext}}$ over $\mathbb{F}_2$ if and only if $d_{\mathtt{ext}} \in \{2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15, 17, 18, 20, 21, 22, 23, 25, 28, 29, 30, 31, 33, 34, 35, 36, 39, 41, 42, 44, 46, 47, 49, 52, 54, 55, 57, 58, 60, 62, 63, 65, 66, 68, 71, 73, 74, 76, 79, 81, 84, 86, 87, 89, 90, 92, 93, 94, 95, 97, 98, 100, 102, 103, 105, 106, 108, 110, 111, 113, 118, 119, 121, 123, 124, 126, 127, 129, 130, 132, 134, 135, 137, 140, 142, 145, 146, 147, 148, 150, 151, 153, 154, 155, 156, 159, 161, 162, 166, 167, 169, 170, 172, 174, 175, 177, 178, 180, 182, 183, 185, 186, 191, 193, 194, 196, 198, 199, 201, 202, 204, 207, 209, 210, 212, 214, 215, 217, 218, 220, 223, 225, 228, 231, 233, 234, 236, 238, 239, 241, 242, 244, 247, 249, 250, 252, 253, 255, 257, 258, 260, 263, 265, 266, 268, 270, 271, 273, 274, 276, 278, 279, 281, 282, 284, 286, 287, 289, 292, 294, 295, 297, 300, 302, 303, 305, 308, 310, 313, 314, 316, 318, 319, 321, 322, 324, 327, 329, 330, 332, 333, 337, 340, 342, 343, 345, 346, 348, 350, 351, 353, 354, 358, 359, 362, 364, 366, 367, 369, 370, 372, 375, 377, 378, 380, 382, 383, 385, 386, 388, 390, 391, 393, 394, 396, 399, 401, 402, 404, 406, 407, 409, 412, 414, 415, 417, 418, 420, 422, 423, 425, 426, 428, 431, 433, 436, 438, 439, 441, 444, 446, 447, 449, 450, 455, 457, 458, 460, 462, 463, 465, 468, 470, 471, 473, 474, 476, 478, 479, 481, 484, 486, 487, 489, 490, 492, 494, 495, 497, 498, 500, 503, 505, 506, 508, 510, 511, 513, 514, 516, 518, 519, 521, 522, 524, 526, 527, 529, 532, 534, 537, 538, 540, 543, 545, 550, 551, 553, 556, 558, 559, 561, 564, 566, 567, 569, 570, 574, 575, 577, 580, 582, 583, 585, 588, 590\}$.

For $d_{\mathtt{ext}} < 1116$, the degree-$d_{\mathtt{ext}}$ AOP over $\mathbb{F}_2$ is irreducible if and only if $d_{\mathtt{ext}} \in \{1, 2, 4, 10, 12, 18, 28, 36, 52, 58, 60, 66, 82, 100, 106, 130, 138, 148, 162, 172, 178, 180, 196, 210, 226, 268, 292, 316, 346, 348, 372, 378, 388, 418, 420, 442, 460, 466, 490, 508, 522, 540, 546, 556, 562, 586, 612, 618, 652, 658, 660, 676, 700, 708, 756, 772, 786, 796, 820, 826, 828, 852, 858, 876, 882, 906, 940, 946, 1018, 1060, 1090, 1108\}$.

For $d_{\mathtt{ext}} < 3660$ and $s > 1$, the degree-$d_{\mathtt{ext}}$ s-ESP over $\mathbb{F}_2$ is irreducible if and only if $(d_{\mathtt{ext}}, s) \in \{(6, 3), (18, 9), (20, 5), (54, 27), (100, 25), (110, 11), (156, 13), (162, 81), (342, 19), (486, 243), (500, 125), (812, 29), (1210, 121), (1332, 37), (1458, 729), (2028, 169), (2500, 625), (2756, 53), (3422, 59)\}$.

B.10 Proof of Lemma 9

Proof. Let $H = \sum_{j=0}^{D} h_j x^j$ and $Q = \sum_{j=0}^{D-2} q_j x^j$. By definition of $\mathcal{D}_e$, $\mathcal{D}_e(Q) \leqslant d-2$ is equivalent to $q_i = 0$ for all even i such that $i > d-2$. By definition of Q, $q_i = 0$ for $i < 0$ and $i > D-2$, so we show the lemma for the values of q_i such that $D-2 \geqslant i > \max(-1, d-2)$.

To do it, we use a proof by induction on an even j such that $D-1 \geqslant j > \max(-1, d-2)$. The base case $j = D-1$, is trivial since $q_j = 0$ for $j > D-2$. Now, assume that $q_k = 0$ for all even k such that $D-1 \geqslant k > j > \max(-1, d-2)$. On the one hand, $\mathcal{D}(A-R) = \mathcal{D}(R)$ because $\mathcal{D}(A) = -\infty$ by hypothesis, then $\mathcal{D}(R) \leqslant D-1 < D+j$ and $D+j$ is odd. We obtain $\mathrm{coef}_{D+j}(HQ) = \mathrm{coef}_{D+j}(A-R) = 0$. On the other hand, $HQ = \sum_{r=0}^{2D-2} \sum_{\ell=0}^{r} q_\ell h_{r-\ell} x^r$, so $\mathrm{coef}_{D+j}(HQ) = \sum_{\ell=0}^{D+j} q_\ell h_{D+j-\ell}$. But $q_\ell = 0$ for $\ell > D-2$ and $h_{D+j-\ell} = 0$ for $D+j-\ell > D$, so $\mathrm{coef}_{D+j}(HQ) = \sum_{\ell=j}^{D-2} q_\ell h_{D+j-\ell}$. When $\ell > j$ is even, $q_\ell = 0$ by induction hypothesis. When ℓ is odd, $h_{D+j-\ell} = 0$ because $D+j-\ell$ is even and $\mathcal{D}_e(H) = d < D+j-\ell$. So $\sum_{\ell=j}^{D-2} q_\ell h_{D+j-\ell} = q_j h_D$. Finally, we have that $\mathrm{coef}_{D+j}(HQ) = q_j h_D = 0$. Because $h_D \neq 0$, this implies that $q_j = 0$. □

Appendix C

Addition Chains for the ITMIA

In Table C.1, we propose addition chains for the Itoh–Tsujii Multiplicative Inversion Algorithm in $\mathbb{F}_{2^{d_{\mathtt{ext}}}}$ (Sections 9.2.6 and B.6). The values of $d_{\mathtt{ext}}$ are used in multivariate cryptography (Table 7.38). In Algorithm 36, we read the binary decomposition of $d_{\mathtt{ext}} - 1$ from left to right for generating the addition chain. This implies restricting k' to $1 \cdot$ (and $k' \cdot$) in Equation (B.3). This choice seems minimized the number of multiplications for certain values of $d_{\mathtt{ext}}$ (corresponding to yes in the last column of Table C.1). For other values of $d_{\mathtt{ext}}$, we decrease this number by allowing other values of k'' (represented in bold in the addition chain). For example, we compute 51 as $48 + 3$ in the first addition chain. We take small bold values for improving the use of multi-squaring tables.

$d_{\mathtt{ext}}$	addition chain of $d_{\mathtt{ext}} - 1$	nb. of mul	used in Algorithm 36?
103	1, 2, 3, 6, 12, 24, 48, 51, 102	8	no – 9 multiplications
174	1, 2, 4, 5, 10, 20, 21, 42, 84, 168, 173	10	no – 11 multiplications
175	1, 2, 4, 5, 10, 20, 40, 41, 82, 87, 174	10	no – 11 multiplications
177	1, 2, 4, 5, 10, 11, 22, 44, 88, 176	9	yes
184	1, 2, 3, 4, 7, 11, 22, 44, 88, 176, 183	10	no – 12 multiplications
185	1, 2, 4, 5, 10, 11, 22, 23, 46, 92, 184	10	yes
265	1, 2, 4, 8, 16, 32, 33, 66, 132, 264	9	yes
266	1, 2, 4, 8, 16, 32, 33, 66, 132, 264, 265	10	yes
312	1, 2, 4, 5, 7, 14, 19, 38, 76, 152, 304, 311	11	no – 13 multiplications
313	1, 2, 3, 6, 9, 18, 36, 39, 78, 156, 312	10	no – 11 multiplications
354	1, 2, 4, 5, 10, 11, 22, 44, 88, 176, 352, 353	11	yes
358	1, 2, 4, 5, 10, 11, 22, 44, 88, 176, 352, 357	11	no – 12 multiplications
364	1, 2, 4, 5, 10, 11, 22, 44, 88, 176, 352, 363	11	no – 13 multiplications
366	1, 2, 4, 5, 10, 20, 40, 45, 90, 180, 360, 365	11	no – 13 multiplications
402	1, 2, 3, 6, 12, 24, 25, 50, 100, 200, 400, 401	11	yes
448	1, 2, 3, 6, 12, 24, 27, 54, 108, 111, 222, 444, 447	12	no – 15 multiplications
540	1, 2, 3, 4, 8, 16, 32, 64, 67, 134, 268, 536, 539	12	no – 13 multiplications
544	1, 2, 3, 6, 12, 15, 30, 33, 66, 132, 264, 528, 543	12	no – 14 multiplications

Table C.1: Proposed addition chains to minimize the number of multiplications in $\mathbb{F}_{2^{d_{\mathtt{ext}}}}$. The bold numbers are used to create the underlined numbers of the chain.

Résumé

Dans cette thèse, nous étudions la conception de cryptosystèmes multivariés qui sont résistants contre les ordinateurs classiques et quantiques. En particulier, nous proposons deux schémas de signature digitale que j'ai soumis au processus de standardisation de cryptographie post-quantique du NIST : GeMSS et DualModeMS. Ces schémas sont basés sur la famille HFE. Nous proposons des paramètres de sécurité basés sur un état de l'art de vingt ans de cryptanalyse intensive. Puis, nous sélectionnons des paramètres qui maximisent l'efficacité. Nous la mesurons avec une nouvelle bibliothèque : MQsoft. MQsoft est une bibliothèque efficace en C qui supporte un large ensemble de paramètres pour les schémas basés sur HFE. Sa performance surpasse toutes les bibliothèques existantes. Nous expliquons dans cette thèse comment nous obtenons une telle performance. Tandis que GeMSS est un schéma qui a une grande clé publique, mais une signature très courte, DualModeMS est basé sur une transformation qui inverse ce comportement. Cependant, sa sécurité est basée sur l'hypothèse que le problème AMQ est difficile. Nous démontrons que cette hypothèse est vérifiée, et nous confirmons nos résultats avec des expériences utilisant les bases de Gröbner. Finalement, nous proposons la première implémentation de DualModeMS. Nous étudions comment l'implémenter efficacement, et comment optimiser le choix des paramètres de sécurité. Nous étendons aussi DualModeMS à l'utilisation du cryptosystème Rainbow à la place de HFE. Ceci permet d'obtenir des tailles de clés et signature intéressantes.

Abstract

In this thesis, we study the design of multivariate cryptosystems, which are resistant against classical and quantum computers. In particular, we study two digital signature schemes that I submitted to the NIST Post-Quantum Cryptography standardization process: GeMSS and DualModeMS. These schemes are based on the HFE family. We propose security parameters based on a state-of-the-art of twenty years of intensive cryptanalysis. Then, we select secure parameters which maximize the practical efficiency. We measure this with a new library: MQsoft. MQsoft is a fast library in C which supports a large set of parameters for HFE-based schemes. Its performance outperforms all existing libraries. We explain in this thesis how we obtain this result. Whereas GeMSS is a scheme which has a large public-key but a very short signature, DualModeMS is based on a transformation inverting this trade-off. However, its security is based on the assumption that the AMQ problem is hard. We demonstrate that this assumption is verified, and we confirm our results with experiences using Gröbner basis. Finally, we propose the first implementation of DualModeMS. We study how to implement it efficiently, as well as how to optimize the choice of security parameters. We also extend DualModeMS to the Rainbow cryptosystem instead of HFE. This allows to have interesting key sizes and signature sizes.