Formal methods for the analysis of cache-timing leaks and key generation in cryptographic implementations

Alexander Schaub

▶ To cite this version:

Alexander Schaub. Formal methods for the analysis of cache-timing leaks and key generation in cryptographic implementations. Cryptography and Security [cs.CR]. Institut Polytechnique de Paris, 2020. English. NNT : 2020IPPAT044 . tel-03205242

HAL Id: tel-03205242
https://theses.hal.science/tel-03205242

Submitted on 22 Apr 2021

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.

INSTITUT
POLYTECHNIQUE
DE PARIS

NNT : 2020IPPAT044

Thèse de doctorat

Méthodes formelles pour l'analyse de fuites cache-timing et la génération de clés dans les implémentations cryptographiques

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 7/12/2020, par

Alexander Schaub

Composition du Jury :

David Naccache Professeur, Ecole Nationale Supérieure (Département d'Informatique)	Président
Ingrid Verbauwhede Professeur, KU Leuven (COSIC)	Rapporteur
Avi Mendelson Professeur, Technion (Faculty of Computer Science)	Rapporteur
Joseph Boutros Professeur, Texas A&M University at Qatar (Electrical and Computer Engineering)	Examinateur
Yuval Yarom Assistan Professor, University of Adelaide (School of Computer Science)	Examinateur
Olivier Rioul Professeur, Télécom Paris (ComNum)	Co-directeur de thèse
Sylvain Guilley Associate Professor, Télécom Paris (ComNum)	Directeur de thèse
Pierre Loidreau Ingénieur, Université Rennes 1 (IRMAR)	Invité
Jean-Luc Danger Directeur d'études, Télécom Paris (SSH)	Invité

Formal methods for the analysis of cache-timing leaks and key generation in cryptographic implementations

Alexander Schaub
Télécom Paris

Contents

Résumé en français xiii

Introduction xxi

1 State of the Art in Cache-Timing Attacks 1

1.1 General Computer Architecture Background . . . 2
1.1.1 Virtual Memory . . . 2
1.1.2 Cache Architecture . . . 2
1.1.3 Cache Hierarchy . . . 3
1.1.4 Cache Organization . . . 4
1.2 Security Implications of the Cache . . . 4
1.2.1 Secret-dependent Memory Access . . . 5
1.2.2 Secret-dependent Branching . . . 5
1.2.3 Exploiting Cache-Timing Leakages . . . 7
1.2.4 Avoiding Cache-Timing Attacks . . . 9
1.3 Measurement Methods . . . 10
1.3.1 PRIME+PROBE . . . 10
1.3.2 EVICT+TIME . . . 12
1.3.3 FLUSH+RELOAD . . . 13
1.3.4 CacheBleed and MemJam . . . 13
1.3.5 BranchScope . . . 15
1.4 Attack Improvement Methods . . . 15
1.4.1 Performance Degradation Attacks . . . 16
1.4.2 Methods Based on SGX . . . 17

2 STAnalyzer 19

2.1 Introduction . . . 19
2.1.1 Related Work . . . 20
2.1.2 Background in Dependency Analysis . . . 20
2.2 Definitions and Notations . . . 22
2.2.1 Variables, Pointers and Values . . . 22
2.2.2 Dependency Graph . . . 25
2.2.3 Leakage Analysis . . . 26
2.2.4 Indirect Dependencies . . . 26
2.3 Description of the Algorithm . . . 28

2.3.1 Abstract Syntax Tree (AST) 28
2.3.2 Dependency Graph Interpretation 29
2.3.3 Expression Evaluation 31
2.3.4 Instruction Interpretation 32
2.3.5 Output Presentation 38
2.3.6 Limitations 38
2.4 Results 39
2.4.1 Benchmarks 39
2.4.2 Applications on Other Cryptographic Algorithms 40
2.5 Conclusion 40

3 Applications of the Static Analysis Techniques 43
3.1 Collecting Ground Truth: Rediscovery of Known CVEs 43
3.1.1 CVE-2019-9494 and CVE-2019-9495 44
3.1.2 CVE-2018-124049 45
3.2 Analyzing 1st Round PQC Candidates 46
3.2.1 Analysis Methodology 47
3.2.2 Result Overview 47
3.2.3 Analysis of Vulnerabilities 48
3.3 Analyzing 2nd Round PQC Candidates 50
3.3.1 LUOV 50
3.3.2 Round5 52
3.3.3 qTesla 53
3.3.4 MQDSS 54
3.3.5 LEDAcrypt 55
3.3.6 Picnic 58
3.3.7 NTSKEM 59
3.3.8 LAC 61
3.3.9 SIKE 62
3.3.10 Other Analyzed Candidates 63
3.3.11 Summary 65
3.4 Perspectives 65

4 On the Stochastic Model of PUFs 69
4.1 An Introduction to Physically Unclonable Functions (PUFs) 69
4.1.1 “Weak” and “Strong” PUFs 69
4.1.2 Toy Example: The Ring Oscillator 70
4.1.3 Description of the Analyzed PUF Designs 70
4.2 Stochastic Models for PUFs 73
4.2.1 Delay Distribution 73
4.2.2 Measurement Noise Distribution 74

5 On the Reliability of PUFs **75**
5.1 An Improved Analysis of Reliability and Entropy for Delay PUFs 75
5.2 Delay PUF Model 76
5.3 Delay PUF Reliability and Entropy 78
5.3.1 Reliability Assessment 78
5.3.2 Reliability Enhancement by Delay Knowledge 81
5.3.3 Entropy After Filtering Out Unreliable Bits 84
5.4 The “Two-Metric” Method 85
5.4.1 Motivation and Definition 85
5.4.2 Reliability of the The “Two-Metric” Method 87
5.4.3 Security 88
5.4.4 Entropy 89
5.5 Translation for Various PUF Architectures 89
5.5.1 RO-PUF 89
5.5.2 RO sum PUF 90
5.5.3 Loop PUF 90
5.6 Experiments and Validation with Real Silicon 91
5.6.1 Architecture of the Test Circuit 91
5.6.2 BER and Entropy Measurement 91
5.7 Effect of Environmental Changes: Temperature 93
5.7.1 Assumptions 93
5.7.2 Average BER 94
5.7.3 Effect on Delay PUFs 94
5.7.4 Impact on the “Two-Metric” Method 95
5.8 Conclusion 95
5.A Verifying the Reliability of a PUF 97
5.A.1 Illustrating Example 97
5.A.2 Finding n and t 97

6 Entropy Estimation of PUFs via Chow Parameter **101**
6.1 Introduction 101
6.1.1 Notations and Definitions 102
6.1.2 Motivation 104
6.1.3 State of the Art 106
6.1.4 Our Contributions 107
6.2 Closed-form expressions 107
6.2.1 Preliminaries 107
6.2.2 Case $n = 3$ 109
6.2.3 Case $n = 4$ 109
6.3 The Chow Parameters of PUFs 111
6.3.1 All PUFs are Attainable 111
6.3.2 Chow Parameters Characterize PUFs 112
6.3.3 Consequence on the Max-Entropy 113
6.3.4 Order and Sign Stability of Chow Parameters 114
6.4 Equivalence Classes and Chow Parameters 115

6.5 Monte-Carlo Algorithm . . . 117
6.6 Entropies Estimation . . . 118
6.6.1 Estimating the Max-Entropy H_0 . . . 118
6.6.2 Estimating the Shannon Entropy H_1 . . . 119
6.6.3 Estimating the Collision Entropy H_2 . . . 119
6.6.4 Estimating the Min-Entropy H_∞ . . . 120
6.7 Conclusions and Perspectives . . . 122

List of Figures

1 Organisation du cache (exemple) xiv
2 Évolution du taux d'erreur moyen en fonction du rapport signal sur bruit . xvi
3 Compromis entropie-fiabilité après filtrage par seuil xvii
4 Taux d'erreur binaire en utilisant deux paires de seuils xviii
5 Estimation des entropies des PUFs pour différentes valeurs de n xix

1.1 Cache architecture on a modern hyper-treaded quadcore processor 3
1.2 Cache lines and sets (2-way associativity) 4
1.3 Victim and attacker process access data at different cache lines. 6
1.4 Victim accesses memory mapped to the cache line observed by the attacker. 6
1.5 Attacker incurs a cache miss because its data has been evicted by the victim. 6

2.1 Comparison between source (a) and compiled (b) code (gcc with -02 optimizations) 21
2.2 Example of a control flow graph with influence regions 27
2.3 AST of the program described in Listing 2.5 29
2.4 Visualization of Algorithm 2.3.1 applied to the dependency graph in Table 2.3 30
2.5 Example of C code with tagged variable 32
2.6 Example C code with variable swapping 37
2.7 Output from STAnalyzer on mbedtls' Blowfish implementation 39
2.8 Example code falsely reported as leaking 39

3.1 Total number of potential vulnerabilities found for each analyzed candidate 48

4.1 An oscillating chain of 5 inverters 70
4.2 Ring Oscillator PUF 71
4.3 Arbiter PUF 72
4.4 RO-sum PUF 72
4.5 Loop PUF 73

5.1 pdf of Δ and noise for a given challenge C. 79
5.2 Polar representation of X and Y. 80
5.3 Expected BER as a function of the SNR. 80
5.4 Attacking the helper data by using unreliable bits as pivot 82
5.5 Unreliable area vs distributions of Δ_C and the noise Z. 82
5.6 Remaining average entropy after filtering unreliable bits as a function of the BER to reach. 84

5.7 Metric M0: Bit extraction according to a and the pdf of Δ 85
5.8 Metric M1: Bit extraction according to $-T_1, T_2$ and the pdf of Δ_C 86
5.9 Metric M2: Bit extraction according to $-T_2, T_1$ and the pdf of Δ_C 86
5.10 Choice of metric and extracted bit value. 88
5.11 Average BER with and without two-metric helper data 89
5.12 Experimental validation of the SNR and remaining entropy. 92
5.13 Distribution of temperature dependency coefficients (for 49 distinct oscillators, as well as 64 challenges of the same oscillator, for three different oscillators) 94
5.14 Average BER when Σ is known or unknown at measurement stage 96
5.15 Rejection probability as a function of PUF error rate ($n = 10^7, t = 120$) . . 98
5.16 Rejection probability as a function of PUF error rate ($n = 227168250, t = 2382$) 99

6.1 Distribution of delays obtained via circuit simulation 105
6.2 Entropy estimates for $n \leq 10$. The upper bound of the min-entropy (dashed line) is taken from [1]. 122

ACKNOWLEDGEMENTS.

First and foremost, I would like to thank my PhD advisors, Oliver Rioul and Sylvain Guilley, for their invaluable advice, patience and enthousiasm during my PhD study. Our fruitful collaboration started back in 2013, saw the publication of numerous papers, and allowed me now to earn a PhD. I will always remain grateful for all the time and effort they spent to help me in my studies.

I would also like to thank the reviewers, Ingrid Verbauwhede and Avi Mendelsson, for accepting to review my manuscript, and all the jury members, David Naccache, Yuval Yarom, Joseph Boutros, Pierre Loidreau and Jean-Luc Danger, for attending my defense. I appreciated your thoughtful remarks and suggestions, and found the discussion we had very enriching.

I would like to extend my gratitude to all the faculty members from COMELEC I had the pleasure to interact with during my time at Télécom Paris - especially Jean-Luc Danger, whose technical knowledge of PUFs and their designs has been invaluable for my work.

These three and half last years would have been much more dull without all the great colleagues with whom I shared my joys and doubts. Michael Timbert, Sébastien Carré, Thuy Ngo, and all the others: thank you very much for all the time we spent together, in front of a whiteboard or around a beer.

Finally, I would like to thank my family for their support, and my soon-to-be wife Luce-Marie for being there for me, brightening my days during this exhausting but stimulating journey !

Rennes, 2021

Glossary

BTF Boolean Treshold Function. xvi, 107, 111, 113, 115, 118, 122

CPU Central Processing Unit. 1–3, 16, 17

KiB kibiobyte = 1024 bytes. 2, 3, 11

MiB mebiobyte = 2^{20} bytes. 3, 11

OS Operating System. 2, 13, 16–18

PUF Physically Unclonable Function. iv, vii, xiv–xvi, 69–74, 97, 101–107, 111–121

RAM Random Access Memory. 1, 2, 4

SGX Security Guard Extensions. 17, 18

VM Virtual Machine. 11

Résumé en français

La cryptographie est omniprésente dans notre monde actuel hyperconnecté, protégeant nos communications, sécurisant nos moyens de paiement. Alors que les algorithmes cryptographiques sont en général bien compris, il est bien plus compliqué de les implémenter correctement. Les failles Spectre et Meltdown ont fini de nous convaincre qu'il est extrêmement complexe de gérer la sécurité sur un processeur moderne. Cependant, des attaques par canaux auxiliaires reposant sur des principes similaires ont été utilisés pour casser les implémentations cryptographiques logicielles plus d'une décennie plus tôt et continueront d'être exploités bien après que ces deux failles ne soient plus qu'un lointain souvenir.

Alors que des preuves de sécurité sont devenues un prérequis pour la publication d'algorithmes cryptographiques, leurs implémentations ont été vérifiées avec moins d'insistance. De ce fait, les avancées théoriques dans la cryptanalyse de primitives modernes se font relativement rares: depuis sa standardisation, AES a perdu un total de deux (!) bits de sécurité théoriques, RSA avec un module de 2048 bits restera très probablement sécurisé jusqu'à être rattrapé par la puissance de calcul des ordinateurs. Cependant, les implémentations ont été cassées bien plus souvent. AES, RSA, DSA, ECDSA - des implémentations largement déployées ont été cassées à cause de canaux auxiliaires logiciels. Des erreurs dans la génération de clés RSA ont causé le rappel de millions de cartes à puce produites par Infineon. En bref, la sécurité des implémentations pourrait fortement bénéficier de meilleurs garanties théoriques.

Dans cette thèse, j'ai appliqué ce raisonnement à deux sujets différents, l'un portant sur la sécurité logicielle, l'autre sur la sécurité matérielle.

La première moitié de la thèse explore les canaux auxiliaires logiciels dits "cache-timing". Ce genre de vulnérabilités apparaît lorsque la durée d'une opération cryptographique, ou l'état du cache après cette opération, dépend d'une information sensible et peut être récupérée par un programme tiers. C'est le cas lorsqu'une opération de branchement dépend d'une information secrète comme une clé privée, ou si la mémoire est accédée à une adresse qui dépend de cette information secrète.

En effet, sur des processeurs modernes, le temps d'accès du processeur à la mémoire principale (la RAM) est relativement long. Afin d'améliorer la performance des processeurs, différents niveaux de cache sont utilisés. Ce sont des mémoires plus petites (de l'ordre de la centaine de kilo-octets pour les plus petits à une dizaine de méga-octets pour les plus grands) mais beaucoup plus rapides que la RAM. En général, trois niveaux de cache sont présents: du plus petit et rapide, au plus grand et plus lent, il s'agit du cache L1, du cache L2 (tous deux propres à chaque cœur d'un processeur multicœur) et le cache L3 (en général partagé entre tous les coeurs).

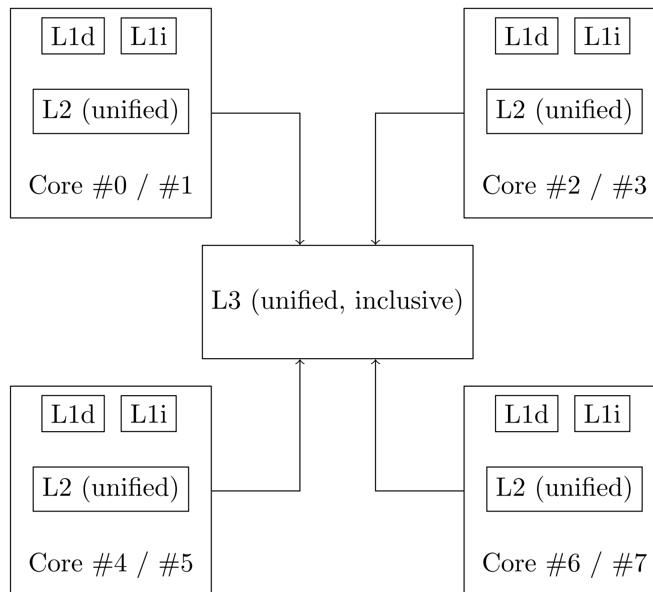


Figure 1: Organisation du cache (exemple)

Bien que les systèmes d'exploitation mettent en œuvre une séparation de la mémoire en fonction des processus qui s'exécutent sous son contrôle, une telle séparation n'est pas implémentée pour le cache. Ainsi, tous les programmes s'exécutant sur le même cœur (pour les caches L1 et L2) voire le même processeur (pour le cache L3) partagent le même cache. Un programme malveillant peut alors manipuler l'état du cache pour déduire des informations sur l'état d'exécution d'un programme ciblé. Plus précisément, des informations concernant les adresses mémoire accédées par le programme cible peuvent être retrouvées. Il peut s'agit des indices utilisés pour accéder à des tableaux, mais aussi des directions des branchements. En effet, les instructions exécutées par un programme sont également mis dans le cache, et en fonction de la direction des branchements, seules les instructions d'une des deux branches (branche `if` ou branche `else`) sont exécutées, et donc chargées.

L'objectif principal de la thèse est de détecter et prévenir ce genre de fuites. A cet égard, j'ai développé un outil qui effectue une analyse de dépendance sur des programmes écrits en C afin de déterminer quelles variables contiennent des valeurs dépendant d'un secret, et lève une alerte lorsqu'une telle variable risque de fuiter (branchement ou accès à une adresse mémoire dépendant de sa valeur). Le fonctionnement de cet outil est le suivant.

Tout d'abord, les données sensibles doivent être identifiées et annotées. Il s'agit principalement des clés privées d'implémentations cryptographiques et de l'aléa secret utilisé par certains algorithmes. Ensuite, l'outil détermine l'ensemble des variables qui dépendent de ces données sensibles. Pour cela, un graphe bipartite est construit, reliant les *variables* définies dans le programme en C à analyser, à un certain nombre de *valeurs*. La première valeur est la valeur spéciale `secret`, qui indique que la variable contient des données sensibles. Ce n'est pas la seule valeur à considérer, cependant. Premièrement,

pour des raisons de performance, il est préférable d'analyser chaque fonction séparément. Or, lors de l'analyse d'une fonction, on ne peut pas encore déterminer si les arguments avec lesquels elle sera appelée dépendent de données sensibles ou non. Ainsi, les *valeurs initiales* (abstraites) des arguments de fonctions sont également considérés comme des valeurs dans cette analyse. Cela permet de composer l'analyse de fonctions en substituant les valeurs initiales par les valeurs des variables lorsque celle-ci est appelée. Finalement, il faut aussi déterminer quelles variables peuvent être référencées par les *pointeurs* définis dans le programme en C. Pour cela, les *adresses mémoires* de toutes les variables sont également des valeurs considérées dans le graphe de dépendance. Bien sûr, la valeur exacte des adresses est inconnue puisque le programme analysé n'est jamais exécuté. À la place, une valeur abstraite est employée pour désigner l'adresse de chaque variable. Ce graphe de dépendance est ensuite mis à jour lors de l'analyse du code source fourni.

En parallèle, les *fuites* sont enregistrées: il s'agit des valeurs entrant dans le calcul des adresses mémoires accédées (comme les indices de tableaux) ainsi que les valeurs utilisées pour calculer le résultat d'un branchement conditionnel. Le programme analysé est considéré comme sûr lorsque la valeur spéciale `sensible` ne fuite pas.

Cet outil est ensuite utilisé pour analyser la plupart des candidats du processus de standardisation de cryptographique post-quantique initié par le NIST. Comme les ordinateurs quantiques seraient capables de casser les algorithmes à clé publique actuellement déployés (comme RSA et ECDSA), il est nécessaire de trouver des primitives cryptographiques alternatives. Puisque ces nouveaux algorithmes sont relativement récents, leurs implémentations ont été moins scrutés jusqu'à présent. Parmi les 21 soumissions qui ont pu être traités, environ un tiers contenaient de potentielles fuites exploitables par des attaques "cache-timing".

La deuxième moitié de la thèse est consacrée aux "physically unclonable functions", ou PUFs. Ce sont des circuits dont on peut extraire des identifiants imprédictibles mais stables. De petites variations incontrôlables dans les propriétés des semi-conducteurs sont amplifiés pour produire un comportement imprédictible. Cela permet, par exemple, de générer des clés cryptographiques. Des garanties théoriques pour deux caractéristiques fondamentales des PUFs sont présentées dans cette thèse, applicables à une large famille de PUFs: la stabilité de l'identifiant, perturbée par des bruits de mesure, et l'entropie disponible, dérivée du modèle mathématique du PUF.

Le principe de fonctionnement des PUFs est le suivant: afin de générer un bit d'identifiant, un *challenge* est envoyé au PUF. Ce *challenge* correspond à une suite de n bits, où n dépend des caractéristiques du PUF. Dans l'idéal, la *réponse* que constitue ce bit d'identifiant est stable (toujours identique pour un même PUF et *challenge*) mais imprédictible (on ne peut pas le "deviner" avec moins de 50% de chances de se tromper, même en connaissant les réponses à d'autres *challenges* pour un même PUF ou ceux à n'importe quel *challenge* pour un PUF différent).

Le défaut de stabilité provient essentiellement des bruits de mesure. En effet, les PUFs que j'ai considérés dans cette thèse sont basés sur des boucles d'inverseurs, permettant de construire des oscillateurs dont la fréquence varie légèrement d'une instance à l'autre. La réponse du PUF est ensuite obtenue en calculant la différence entre deux fréquences d'oscillations dans le PUF. Cela peut provenir de deux oscillateurs distincts dans le circuit, comme pour le RO-sum PUF [93] ou le Ring Oscillator PUF [95], ou de deux configurations

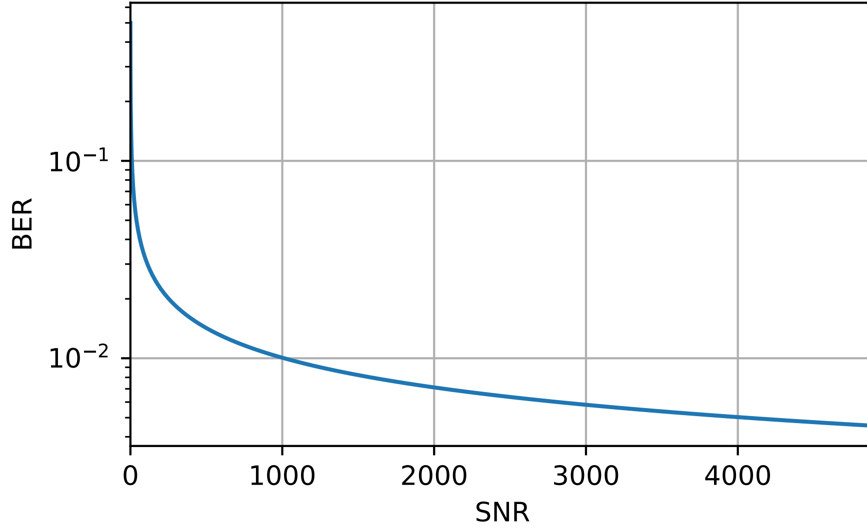


Figure 2: Évolution du taux d'erreur moyen en fonction du rapport signal sur bruit

différentes d'un même oscillateur comme pour le Loop PUF [92]. En modélisant la fréquence d'oscillation et le bruit de mesure par des lois normale, modélisation vérifiée par des mesures et des simulations, ont peut obtenir des expressions explicites du taux d'erreur binaire (ou BER pour *Bit Error Rate*). Si la fréquence des oscillations suit une loi normale centrée de variance Σ^2 et le bruit gaussien additif une loi normale centrée de variance σ^2, alors, en définissant le ratio signal sur bruit (ou SNR pour *Signal to Noise Ratio*) par $\frac{\Sigma^2}{\sigma^2}$, on peut montrer que le taux d'erreur *moyen* du PUF est égal à

$$\widehat{BER} = \frac{1}{\pi} \arctan(\frac{1}{\sqrt{SNR}}).$$

Même pour des valeurs élevées du SNR, cette valeur reste relativement importante (voir Figure 2), de l'ordre de quelques pour cents même pour des valeurs du SNR au-dessus de 4000.

Ce taux d'erreur élevé provient principalement des quelques *challenges* aux réponses instables. En effet, comme la différence de fréquences, utilisée pour calculer le bit de réponse, suit une loi normale, une proportion non négligeable de ces différences ont une valeur autour de 0, résultant en un taux d'erreur très important pour ces réponses, autour de 50%. Une première méthode pour améliorer la fiabilité du PUF consiste alors à filtrer les *challenges* qui résulteraient en une différence de fréquence inférieure, en valeur absolue, à un seuil égal à un multiple de σ, par exemple $W\sigma$. On peut alors montrer que le taux d'erreur moyen des *challenges* restants après le filtrage est égal à

$$\widehat{\mathrm{BER}}_{filt} = \frac{2}{\mathrm{erfc}(\frac{W}{\sqrt{2}\sqrt{\mathrm{SNR}}})} \left(T(W, \frac{1}{\sqrt{\mathrm{SNR}}}) + \frac{1}{4} \mathrm{erf}(\frac{W}{\sqrt{2}\sqrt{\mathrm{SNR}}})(\mathrm{erf}(\frac{W}{\sqrt{2}}) - 1) \right)$$

où erf et erfc représentent respectivement la fonction d'erreur et la fonction d'erreur

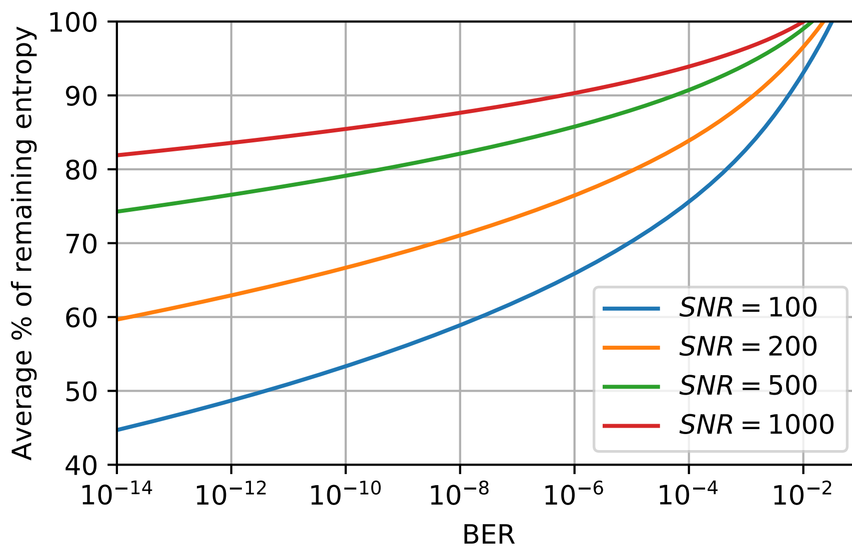


Figure 3: Compromis entropie-fiabilité après filtrage par seuil

complémentaire, et T la fonction T d'Owen:

$$T(h,a) = \frac{1}{2\pi}\int_0^a \frac{e^{-\frac{1}{2}h^2(1+x^2)}}{1+x^2}\,\mathrm{d}x.$$

Ce filtrage, cependant, diminue *l'entropie utile* du PUF: comme certains challenges sont mis de côté, la taille des identifiants générés est plus petite. En moyenne, on montre que, en supposant que le PUF sans filtrage fournit n bits de réponse indépendants, l'entropie restante après filtrage est égale à

$$H = n \cdot \mathrm{erfc}(\frac{W}{\sqrt{2\mathrm{SNR}}}).$$

En combinant ces deux formules, on peut caractériser entièrement le compromis fiabilité-entropie défini par le filtrage par seuil (voir Figure 3). En général, le SNR est fixé pour un modèle de PUF donné. La caractérisation permet alors de prévoir comment dimensionner le PUF pour obtenir une fiabilité et une entropie donnée.

Une deuxième méthode pour améliorer la fiabilité de ces PUFs consiste à changer de métrique pour obtenir un bit de réponse: au lieu de prendre le signe de la différence de fréquences, on peut définir deux seuils et considérer que la réponse du PUF est de 0 si la différence de fréquence est comprise entre ces deux seuils, et de 1 sinon. En choisissant bien ces seuils, qui dépendent de la variance des fréquences σ^2, l'entropie par réponse reste de 1 bit. En choisissant judicieusement deux paires de deux seuils, et en appliquant la paire de seuils en fonction de la différence de fréquences de façon à minimiser le taux d'erreur, on peut améliorer la fiabilité du PUF sans diminuer son entropie (voir Figure 4). Les désavantages de cette méthode sont la complexité supplémentaire introduite par les comparaisons aux seuils, et surtout la nécessité de déterminer précisément la variance des

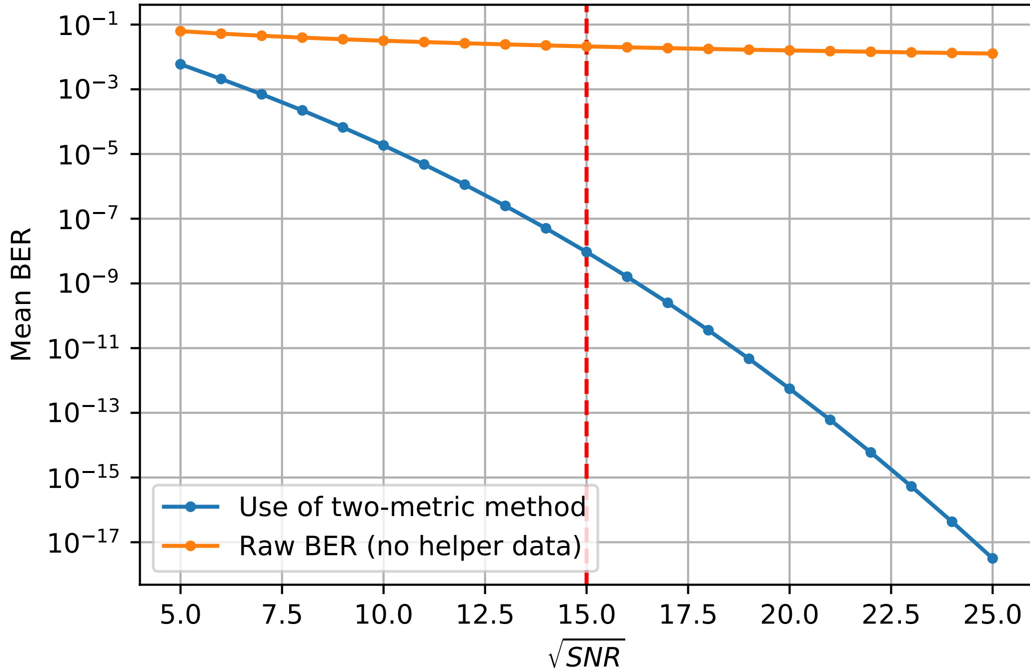


Figure 4: Taux d'erreur binaire en utilisant deux paires de seuils

différences de fréquence. En effet, une mauvaise estimation de cette grandeur biaiserait la réponse des PUFs.

Enfin, la dernière partie de la thèse concerne l'estimation de l'entropie des PUFs. Dans la précédente partie, nous avons supposé que toutes les réponses des PUFs étaient indépendantes. Or, pour une large famille de PUFs, comprenant les Arbiter PUF [91], les RO-sum PUFs [93] et les Loop PUFs [92], il n'est pas possible d'extraire plus de n réponses indépendantes pour un PUF de taille n, alors qu'il existe 2^n *challenges* possibles. En autorisant l'utilisation de challenges pour lesquels les réponses sont liées, l'entropie que l'on peut extraire d'un tel PUF est alors augmentée. Cependant, calculer cette entropie n'est pas évidente. En effet, chaque réponse à un challenge pouvant prendre deux valeurs différentes, il pourrait existe au total jusqu'à 2^{2^n} PUFs différents. Cependant, tous ces PUFs ne sont pas tous réalisables. En effet, la réponse d'un PUF à un challenge donné suit, pour les PUFs considérés, la formule suivante:

$$b = \text{sign}(\sum_{i=1}^{n} c_i x_i) = \text{sign}(c \cdot x)$$

où c_i représente la i-ème bit du challenge (-1 ou $+1$) et x_i représente une différence de fréquences "élémentaire", distribuée selon une loi normale. Dans ma thèse, je rapproche cette formulation des fonctions booléennes à seuil (ou BTF pour *Boolean Threshold Function*). Ceci permet de démontrer, d'une part, que l'entropie de ces PUFs, même en considérant tous les 2^n challenges possibles, ne peux dépasser n^2. D'autre part, en utilisant les propriétés de symétrie établies pour les BTFs, j'ai pu développer un algorithme

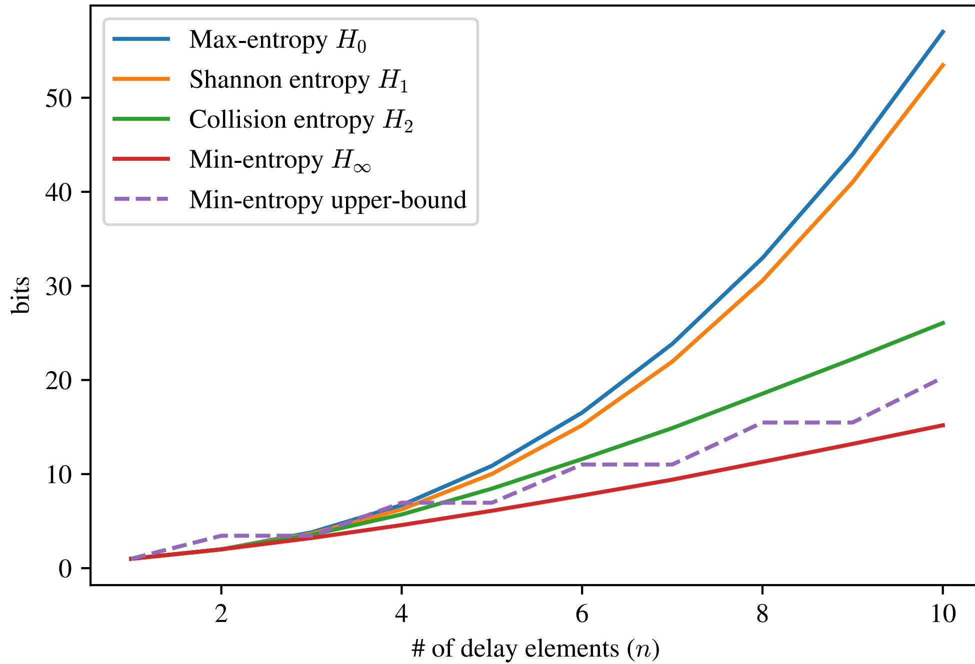


Figure 5: Estimation des entropies des PUFs pour différentes valeurs de n

efficace d'estimation de l'entropie des PUFs par simulation. Ceci a permis d'obtenir des estimations fiables des différentes entropies (entropie de Shannon, entropie de collision, min-entropie) pour des PUFs jusqu'à la taille $n = 10$ (voire Figure 5).

Cependant, les PUFs utilisés habituellement ont des tailles bien plus élevés, de l'ordre de $n \approx 100$ ou plus. La simulation de l'entropie n'est alors plus possible. Cependant, les résultats obtenus semblent indiquer que l'entropie de Shannon suit approximativement la max-entropy, asymptotiquement équivalente à n^2. Si cette tendance est confirmée, cela montrerait que l'on peut théoriquement extraire bien plus de n bits d'entropie d'un PUF de taille n. Pour appliquer ces résultats en pratique, il faudrait également déterminer quels sont les challenges optimaux à choisir. En effet, pour un PUF de taille par exemple $n = 32$, il n'est pas envisageable de produire un identifiant en utilisant tous les $2^{32} \approx 4 \cdot 10^9$ challenges possibles. La question est alors de savoir si c'est possible de choisir, par exemple, $k_1 \cdot 32^2$ challenges pour obtenir une entropie de $k_2 \cdot 32^2$ bits, avec k_1 et k_2 des constantes qui restent à déterminer.

Introduction

Buying products on the Internet, chatting online with friends and family, or working remotely from anywhere on earth: these activities are part of our everyday life. They would not be possible without modern cryptography. It prevents our credit card information from being stolen, our private conversations to be shared with strangers, and industrial secrets from being revealed.

Such accidents still happen, because getting the security of *any* system right is a challenging problem. The news is filled with reports of data breaches, credentials thieves and stolen credit card numbers. However, the underlying cryptography is almost never at fault. For instance, 20 years after is has been standardized, the best-known attacks against the popular *Advanced Encryption Standard* (AES) reduce its theoretical security by a mere *two* bits. While the key sizes for RSA have had to be increased in order to account for algorithmic advances and increases in computing power, large key sizes (2048 bits and above) are still considered out of reach for even the most powerful attackers. As of today, the confidence in the currently deployed cryptographic algorithms remains high.

But the same cannot be said for the *implementations* of these cryptographic algorithms. The first practical *side-channel* attacks against a specific implementation of AES were published in 2005 and improved ever since. A weak method of generating private RSA keys, revealed in 2017 under the name *ROCA vulnerability*, compromised the security of millions of smartcards and other sensitive devices. Cache-timing side-channels have been used to break all kinds of cryptographic algorithms: RSA, ECDSA, AES, BLISS, ...

Therefore, securing the concrete implementation of cryptographic algorithms constitutes today a major challenge. While there are proofs of security for numerous cryptographic algorithms, less effort has been put into at least *verifying* that the *implementations* are also safe.

A significant number of causes can compromise the security of an implementation: timing leaks, data leaks through the cache, numerous ”classical“ programming errors, bad handling of edge cases and invalid values, . . . Whether a given cause leads to a cryptographic break depends on numerous external factors: the hardware on which the program runs, the presence or absence of an operating system, the type of operating system, the general mitigations that have been implemented, and more.

Furthermore, the emergence of quantum computers poses a new threat to traditional cryptographic algorithms. Public-key cryptography needs to be redesigned in order to resist against attacks using these new machines when they will become available. Algorithms that can resist against quantum computers are known as *post-quantum cryptography* algorithms, and designing such algorithms is currently an active field of research. *Implementing* these

algorithms securely is another challenge. Indeed, the implementations are very recent and have therefore been analyzed much less than the implementations of traditional cryptographic algorithms. The risk of remaining implementation errors or side-channel leaks is quite high.

Verifying that an implementation cannot be compromised therefore amounts to a daunting task, given the variety of existing implementations and the compounding complexity of the causes and contexts that need to be taken into account. While it is possible to verify that an implementation is safe on a specific context—processor model and version, operating system version and configuration—the effort becomes almost infeasible when multiplied by the number of different contexts that can be encountered in the real world.

A more feasible approach consists in first *abstracting* the whole complexity of the context in order to create a simpler *model*, and then show interesting properties on this model. The drawback incurred by this approach is a loss of precision in the analysis. However, the results obtained by such an analysis are much more broadly applicable–on every system that fits the description of that model.

A similar approach can be applied to another crucial component of implementation security: the generation of keys. It ultimately requires a physical source of randomness, but assessing the entropy of that source with high confidence is not trivial. Statistical test suites can help to find some biases but offer no theoretical guarantees for the quality of the random generator. However, one can *abstract* the architecture of the generator in order to obtain a *model* which these guarantees can be derived from.

In this thesis, we will apply this line of reasoning to two independent problems in security, one in software security, the other in hardware security. The first half of this thesis will explore the automatic detection of cache-timing attacks. It will feature a tool written during my thesis that performs abstract execution of programs written in the C programming language in order to detect potential cache-timing leaks, as well as extensive case reports highlighting its performance on real-world code and in particular on the implementations of post-quantum cryptography algorithms.

The second half of this thesis deals with physically unclonable functions, or PUFs. These are hardware security primitives that ideally exhibit unpredictable physical properties (propagation delays, bias in memory cells, . . .) that can be exploited to obtain a consistent identifier (a bitstring) from a given PUF. Two fundamental characteristics of a PUF are therefore the measure of *unpredictability* of the properties and the *consistency* of its response. These two characteristics are the subjects of two chapters in the second half of this thesis. The distribution of both the PUF characteristics and the measurement noise are modeled for a certain class of PUFs, which allows us to compute their *entropy*—a measure of the unpredictability—and the bit error rate (BER) of the identifier generated using different methods—a measure of the response consistency. For the first time, this provides theoretical guaranties on the PUF behavior, allowing for more efficient designs and higher confidence in the security of the PUFs.

This manuscript is divided into six chapters, of which the first three cover cache-timing side-channel attacks and the next three cover PUFs.

Chapter 1 consists of an introduction to cache-timing side-channel attacks. These attacks threaten the security of cryptographic implementations executed on any processor

that possesses a cache. Because the cache is often shared between programs executing on the same processor, sometimes even on different processor cores, they constitute a shared state between an attacker and its victim process. If the state of the cache depends on sensitive information handled by the victim process, such as cryptographic keys or sensitive randomness, then the attack can break the security of that implementation by strategically probing the cache. This is the case when the victim process branches on a sensible value, or accesses memory at an address that depends on a sensitive value (such as an array access at a sensitive offset). Several ways of probing the cache have been published during the last few years, and this chapter summarizes a few of them. They differ in both scope (vulnerable cache type, applicable processor families) and performance (period between two probing events, false positives and negatives).

Chapter 2 presents a methodology for automatically detecting potential cache-timing leakages in programs written in the C programming language. It consists in a static analysis in two parts, which are performed in parallel. The first is a *dependency analysis*, which determines which variables might contain sensitive values. It has to take into account pointer aliasing, dependency tracking through function calls, and indirect dependency flow. The second part consist in recording which variables govern the program control flow and which ones are used to compute addresses for memory accesses. If such a variable depends on a sensitive value, a leakage is reported. This methodology is implemented in the tool STAnalyzer which is available from `https://gitlab.telecom-paris.fr/sen/stanalysis`.

Chapter 3 summarizes the results obtained with this tool on several real-world examples: known cache-timing vulnerabilities on the NSS and wpa_supplicant libraries, as well as most candidates of the first and second-round candidates for the NIST post-quantum cryptography standardization process. All known vulnerabilities were caught by this tool, although a larger number of unrelated warnings were also triggered by the NSS library. Some may be false positives, while others are potential vulnerabilities that cannot be ruled out by static analysis. A large number of the first round post-quantum candidates found to have potential vulnerabilities, while the updated implementations of the second round candidates were generally more robust to cache-timing attacks. The tool proved to be useful on these real-world examples, uncovering real issues without being overwhelmed by false positives.

Chapter 4 introduces PUFs, and explains what they are used for and how they can be build. Several kinds of PUFs are presented, with a special emphasis on so-called *delay PUFs*. These exploit delay differences in oscillating circuits, so that each manufactured circuit exhibits a different oscillation frequency. Numerous PUF designs are based on this basic building block: arbiter PUF, ring-oscillator (RO) PUF, RO-Sum PUF, loop-PUF among others. These designs are described in more detail in this chapter.

Chapter 5 presents results about the reliability of PUFs, especially delay PUFs. Because of inherent fluctuations of the oscillating frequency, different PUF measurements might lead to different identifiers, which defeats the purpose of a PUF. The authentication of the PUF might fail, and if the error rate is too high, it might never succeed at all. Different methods to improve the PUF reliability are presented in this chapter. For instance, one can filter out “unreliable” elements from the PUF response. After proposing a model for

the PUF noise and delay distribution, we present a closed-form formula to determine the new error rates after filtering. This provides a more efficient way to predict the reliability of a PUF and therefore choose the right design parameters for a given application. Another method that consists in changing the way delay measurements are performed in order to obtain the PUF identifier is also presented, and we compute the reliability for this method. The theoretical results are then compared with measurements from real PUF circuits.

Chapter 6 presents results about the entropy of delay PUFs. The abstract description of these PUFs reveals a link between them and a class of boolean functions called boolean threshold functions (BTF). Using theoretical results from the study of BTFs, we derive an efficient algorithm that allows us to estimate the entropy of the PUF distribution up to PUFs of size 10. These results suggest a quadratic increase in entropy with the PUF size. While we show that the entropy is indeed *bounded* by a quadratic function of the PUF size, the asymptotic behavior for the Shannon entropy remains an open question.

Chapter 1

State of the Art in Cache-Timing Attacks

Know yourself and know your enemy, and you will never be defeated.

Sun Tzu, The Art of War.

Modern CPUs utilize numerous optimizations in order to increase their performance. The instructions that need to be executed and the data that these instructions are working with need to be loaded from memory. Usually, these reside in the main memory or RAM. However, accessing the RAM is slow, taking about 100 to 150 cycles before the first bytes can be read [2]. In order to overcome this bottleneck, CPUs include several layers of much faster cache, with access times as low as 4 cycles [2]. Programs that make use of them can see a significant improvement in performance. However, the lack of isolation between the cache lines accessed by different processes (see Section 1.1.2) can lead to security risks, as described in the next sections.

The security threat created by the caches is exacerbated by other performance optimizations. Speculative execution is a mechanism by which the CPU executes instructions before it can be determined whether they should be executed or not. For instance, instructions that try to access data for which the process has no access rights should cause the process to be aborted. Because checking for the access rights takes time, and access rights violations are fairly rare in most programs, it makes sense, from a performance point of view, to continue executing instructions while simultaneously checking for the access rights. In case of a violation, the executed instructions are then rolled back before the process is stopped. Speculative execution can also happen when the CPU executes a branching operation, which is used to implement conditional statements and loops. Because determining which branch should be executed (the `if` or the `else` branch, loop continuation or loop exit) might also be slow in some situations, the CPU can choose to execute the instructions in one branch, and roll back the state if it chose the wrong

branch. However, the cache state is often *not* rolled back, and it might reveal sensitive informations. This is the main cause of the Spectre and Meltdown attacks, major security vulnerabilities that affected a wide variety of computer systems around the world [3, 4], which exploit speculative execution respectively after a branch misprediction and an access right violation.

The remainder of this chapter is organized as follows. First, I will explain the cache architecture of modern CPUs and the security implications of their organization. I will then present different methods that allow an attacker to infer information about the cache state, before presenting a brief overview of published attacks that use these methods.

1.1 General Computer Architecture Background

1.1.1 Virtual Memory

The OS has to deal with two memory spaces: physical and virtual memory.

Physical memory represents the memory available to the operating system, and depends on the hardware it runs on. It corresponds to the amount of RAM available to the processor (plus, optionally, some reserved hard disk space which is not relevant to the discussion).

Processes running on the OS, however, do not have directly access to the physical memory. Instead, memory accesses are done via *virtual memory*, and the translation between physical and virtual memory is performed by the OS. There are two main reasons for this design. Firstly, it reduces the complexity of the memory allocation for processes, as they would otherwise need to know what memory is used by all other processes. Second, it allows for isolating the memory between processes, so that one rogue process cannot read or modify sensitive data of another process.

The operating system presents to each process a *virtual* address space of size 2^{64} bits, which might or might not be addressable in its entirety[1] . When a process accesses a valid virtual address for the first time, the operating system will provision some physical memory, and record the mapping between the virtual address, the identifier of that process, and the address of the provisioned physical memory. Since provisioning and recording the mapping causes a delay, it is not done at the individual byte level. Rather, a whole *page* is provisioned at once. Their size is usually 4 KiB but depending on the system configuration, may be bigger. The mapping between virtual and physical addresses is encoded in so-called *page tables*, and cached in the *Translation Lookaside Buffer* (TLB).

1.1.2 Cache Architecture

A cache is a reserved set of fast memory, used to improve the performance of a CPU. Contrary to the RAM, programs cannot directly access the content of a cache. Instead, every time the CPU needs to fetch data from RAM, it first checks whether it is already present in one cache. If so, the program experiences a lower latency for accessing the data - this is called a cache *hit*. If not, the latency is higher, but the data is saved into one

[1]For 64-bit operating systems. In practice, only 2^{48} bits are addressable on most common processors.

or more caches in order to speed up subsequent memory accesses, and this constitutes a cache *miss*.

Albeit some details of the cache implementation are hidden by the CPU manufacturers, the general behavior is well documented and of particular importance for programmers wishing to optimize their implementation, as well as those, as we will see, that care about their security.

1.1.3 Cache Hierarchy

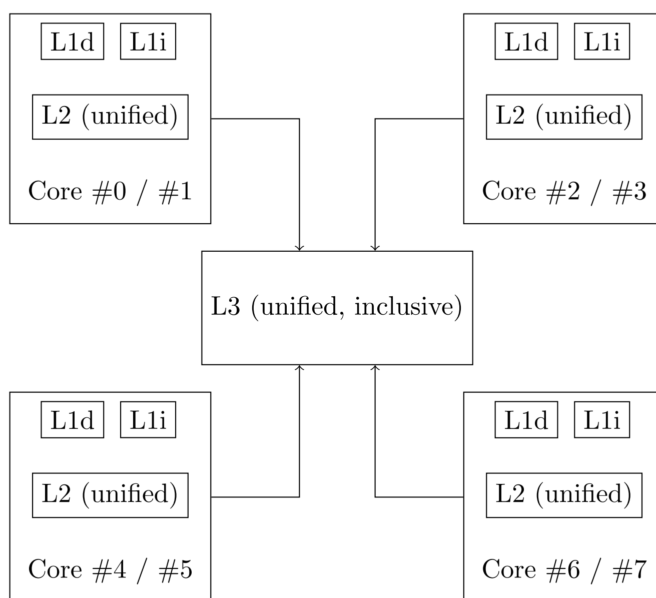


Figure 1.1: Cache architecture on a modern hyper-treaded quadcore processor

Modern processors make use of three distinct levels of cache, named L1, L2 and L3.

The L1 cache is the fastest, but also smallest, of the three levels. Every processor core possesses two L1 caches, one for caching data (L1D cache), the other for caching instructions (L1I cache). Their size is in general around 32KiB each, and the latency can be as low as 4 cycles [5, §2.1.3]. This cache being private to each physical processor core, processes executing on distinct cores do not have access to each other's L1 cache. However, two processes executing on different *virtual* cores, but on the same physical core on hyper-threaded processors, do share the L1 cache.

The L2 cache is slightly larger than the L1 cache, at around 256KiB. It has also a higher access time of 12 cycles [5, §2.1.3], and is private to each physical processor core. Only one cache for both data and instructions is present on modern processors.

Finally, the L3 cache (or LLC, for Last Level Cache) is shared among all processor cores. Its size varies depending on the CPU model, but it is generally comprised between 8MiB and several dozen MiB [6] for modern consumer-oriented CPUs. Also, the L3 cache is inclusive, that is, any data stored in the L1 or L2 cache of any core must be also present in the L3 cache.

Both the L2 and L3 caches are *unified*, meaning they can hold indifferently instruction or data.

1.1.4 Cache Organization

Caches are organized into so-called *cache lines* (sometimes called *cache blocks*). The cache line is the smallest unit of memory that is replaced in the cache, and their size is usually 64 bytes. Whenever data is loaded from the RAM into the cache, or from one cache to a smaller cache, the data corresponding to a whole cache line is transferred. Similarly, when a cache line is evicted from the cache, due to a special assembly instruction (`clflush`) or to make room for new data, the whole cache line is removed at once. This mechanism serves to reduce the number of transfers between the main memory and the caches.

Cache lines themselves are organized into cache *sets*. The 64 bytes starting at a given 64-byte aligned (physical or virtual) address can only be saved to a limited number of cache lines. These cache lines form the cache set of that address. For a given cache, the number of those cache lines is fixed and the same for every address, but it might vary between the different caches of the same processor. If there are n possible cache lines for a given address, the cache is said to have a n-way associativity (see Figure 1.2). If this set is determined from the physical address of the concerned memory region, then it is called *physically indexed*, and virtually indexed otherwise.

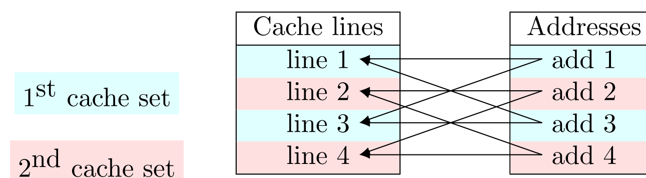


Figure 1.2: Cache lines and sets (2-way associativity)

Under certain conditions, data can be loaded into the cache even though no data from the cache line has been accessed. This is known as data prefetching [5, §2.3.5.4]. For instance, if the memory is accessed sequentially, two cache lines might be loaded into the cache on every cache miss - the line corresponding to the accessed data, as well as the next 64 bytes of memory (or the previous ones, if the memory is accessed sequentially but in reverse order).

1.2 Security Implications of the Cache

The cache constitutes a shared resource between processes. By default, no isolation is enforced between cache memory belonging to different processes. This opens up the possibility of side-channel attacks exploiting the cache.

If an attacker is able to determine the state of the cache after or during the execution of a program (see Section 1.3 for details), and if the state of the cache depends on some sensitive values handled by the program, then the attacker is able to gain knowledge about

these sensitive values. We speak of *leakages* if the cache state depends on sensitive values. Leakages arise for mainly two reasons.

1.2.1 Secret-dependent Memory Access

One reason for leakages to happen are secret-depend memory accesses. These occur if a program accesses memory at an address that depends on secret data, either via pointer arithmetic, or by accessing an array at a secret-dependent index. For instance, one round of the popular AES [7] encryption algorithm consists in a bytewise substitution, which is most efficiently implemented using a lookup table, and the index of the lookup operation depends on the secret key.

Secret-dependent memory accesses can lead to two kinds of vulnerabilities caused by the cache. First, the total execution time of the victim process might vary depending on sensitive data. For instance, this behavior has been demonstrated by Bernstein on a (now outdated) AES implementation found in OpenSSL [8] and lead to a full key recovery attack. It is due to the fact that the number of cache misses during the execution of an AES encryption varies depending on the key and plaintext. Therefore, although the number of instructions executed during AES encryption remains constant, the encryption time varies.

The second kind of vulnerability can only be exploited by an attacker running unprivileged code on the same processor as the victim. If the victim and attacker processes share the same cache, then the cache-access pattern of the victim can influence the behavior of the attacker process, and vice-versa.

For instance, the attacker process could identify a memory address that might be used by the victim, depending on certain properties of sensitive data. They then identify an other memory address that maps to the same cache line as the memory of the victim process[2] (Figure 1.3). When the victim process indeed accesses that specific address, it occurs a cache-miss and evicts the cache line owned by the attacker process (Figure 1.4). This has two effects: first, the attacker process will observe that loading the memory will now take longer, as the requested data is not in the cache anymore and has to be fetched from slower main memory (Figure 1.4). Second, the victim process incurs a cache miss that it would not have incurred without the attacker process. Therefore, the execution time of the victim might depend on which address is repetitively accessed by the attacker. If they are able to correlate this behavior with sensitive data manipulated by the victim, then they might be able to recover it.

1.2.2 Secret-dependent Branching

The second reason for leakages to occur are secret-depending branching operations. These are caused by programs executing certain instructions conditionally on some sensitive data. For instance, a loop can be executed a variable number of times, or the condition of an `if-else` branch could depend on that sensitive data. In some cases, this can cause the execution time of a program to vary depending on sensitive information, and such behavior

[2]or a set of n memory addresses that map to the n possible cache lines of the victim memory address in case of a cache with n-way associativity

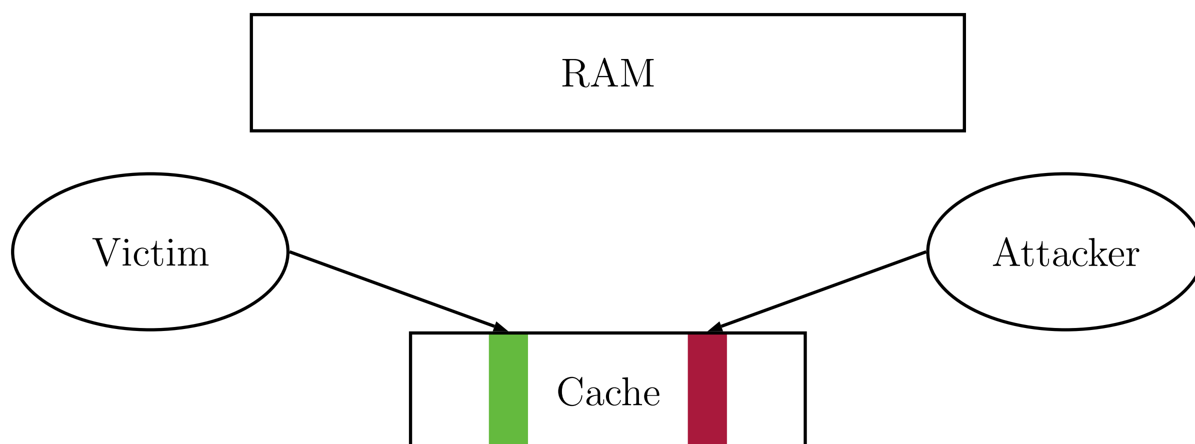


Figure 1.3: Victim and attacker process access data at different cache lines.

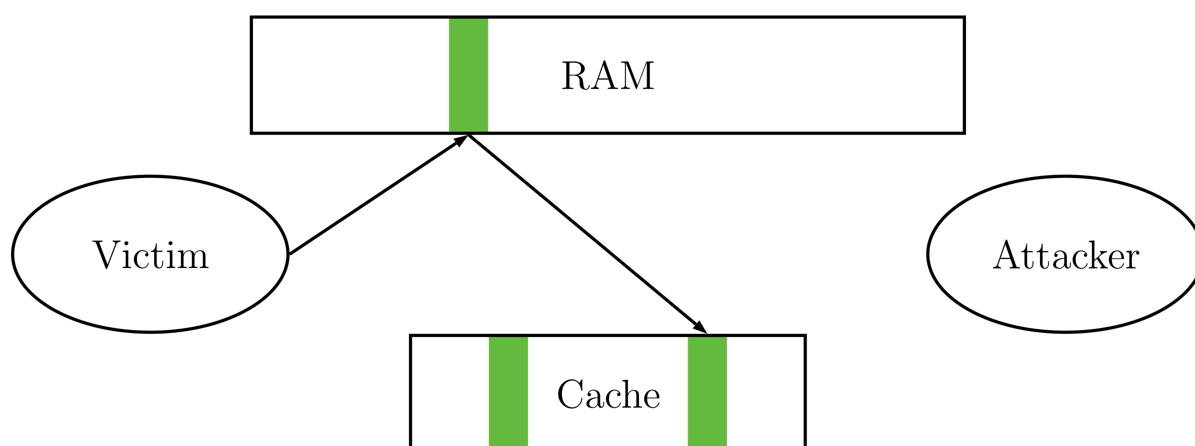


Figure 1.4: Victim accesses memory mapped to the cache line observed by the attacker.

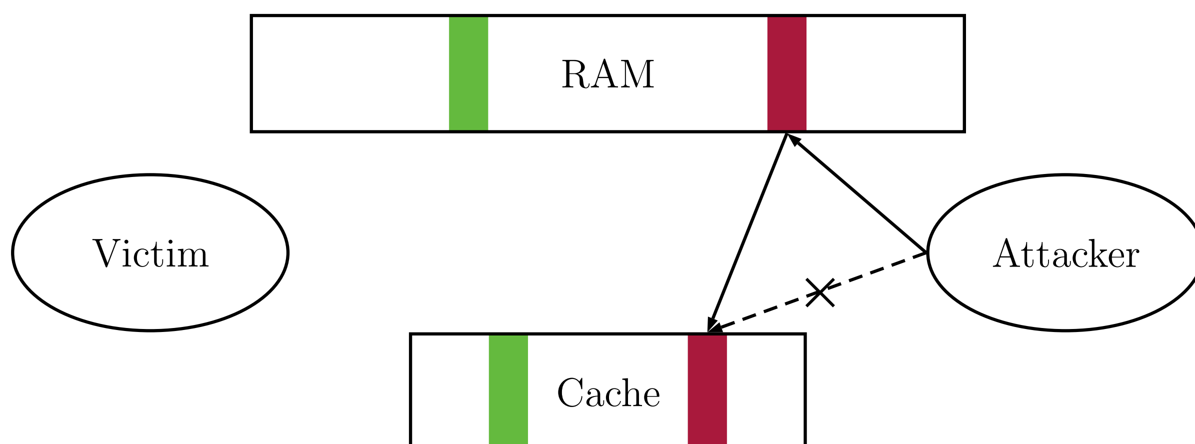


Figure 1.5: Attacker incurs a cache miss because its data has been evicted by the victim.

has been exploited in numerous attacks, to construct decryption oracles for TLS [9, 10] or

to attack various cryptographic primitives [11, 12].

Cache-timing attacks are even more powerful, and can reveal information about sensitive data even when the two branches of an `if-else` branch execute the same functions [13]. This is due to the fact that any instruction executed by a program is saved into the cache, so that when the instruction is executed again, it does not need to be loaded from main memory. However, this implies that the cache state depends on the instructions that have been executed. For instance, two branches can call the same functions, but the instructions preparing the function arguments and executing the function calls are considered different for caching purposes. Therefore, an attacker that can determine the cache state of the victim might be able to determine which branch was taken.

1.2.3 Exploiting Cache-Timing Leakages

In many cases, a successful cache-timing attack can allow the attacker to fully recover the victim key. How this is done, however, depends on the algorithm executed by the victim and the sensitive information leaked to the attacker.

Leakage of the key On some implementations, the state of the cache directly depends on the key. This is for instance the case for unprotected RSA implementations. In RSA, the decryption step consists in a modular exponentiation with the secret key. The most straightforward implementation consist in implementing this operation via modular fast exponentiation, also called “square-and-multiply”.

Algorithm 1.2.1 RSA “square-and-multiply”

Input: n: public key, d: private exponent, c: ciphertext
Output: $c^d \mod n$
  $x \leftarrow 1$
  for $i = \lfloor \log_2(d) \rfloor$ down to 0 **do**
    $x \leftarrow x * x \mod n$
    if $d \,\&\, (1 \ll i) \neq 0$ **then**
      $x \leftarrow c * x \mod n$
    end if
  end for
  return x

The total execution time depends on the secret exponent d, but it only depends on the number of bits set to 1. In contrast, a cache-timing attack with high enough resolution would be able to determine for each round of the loop whether the extra modular multiplication by c is executed or not. However, the timing variations were actually enough to retrieve the secret key in older reference implementations of RSA [11].

Because of these vulnerabilities, “square-and-multiply” has not been used to implement RSA decryption in a long time, and other fast exponentiation algorithms such as Montgomery multiplication [14] are preferred. These resisted simple timing attacks but were ultimately broken using modern cache-timing attacks [15, 16].

Leakage of the nonce In some cryptographic implementations, protecting the random nonce generated during the signature is of paramount importance. This is mainly the case for ECDSA signatures [17] using non-deterministic nonces. Nonce misuse was used in order to break digital signatures on the Sony PlayStation 3 [18] or to steal Bitcoins from insecure Android wallets [19]. Similarly, recovering even partial or approximate information about the nonce allows to recover the secret key [13, 20, 21]. In [13], a cache-timing attack is performed on an ECDSA implementation using Montgomery ladder for point multiplication. The pseudocode for this procedure is given in Algorithm 1.2.2. The algorithm works in a similar fashion as traditional fast multiplication, but does not exhibit timing variations as a point addition and a point doubling is executing for every bit of the input parameter k, regardless of the value of that bit. However, the code still branches on the value of the bits of k. When the instructions on the two branches map to distinct cache lines, as is the case in the OpenSSL implementation attacked by Yarom et al., then a cache-timing attack can reveal the value of k. Once most bits of the nonce are known, the private key can be recovered.

Algorithm 1.2.2 Elliptic curve point multiplication using Montgomery ladder

Input: k: nonce $\in \mathbb{N}$, $\mathcal{P}$: curve point
Output: $k\mathcal{P}$

```
R_0 ← O                                  ▷ Additive zero
R_1 ← P
for i = ⌊log_2(k)⌋ down to 0 do
    if n & (1 ≪ i) = 0 then
        R_1 ← R_0 + R_1
        R_0 ← 2R_0
    else
        R_0 ← R_0 + R_1
        R_1 ← 2R_1
    end if
end for
return R_0
```

Leakage of the randomness More generally, the randomness used to perform cryptographic operations must be kept secret when it is not part of the output. When side-channel leaks reveal this private randomness, then the key might be recovered. Such a vulnerability was exploited in order to break implementations of the post-quantum signature scheme BLISS [22] (as well as the variant BLISS-B [23]) using cache-timing attacks [24, 25]. We will here quickly describe the attack against the strongSwan [26] implementation of BLISS.

In BLISS, public and private keys are elements of the ring $R_{2q} = \mathbb{Z}_{2q}[x]/(x^n + 1)$. Elements of this ring we will be denoted in **bold** for the remainder of this section. In particular, the public key is an element $\mathbf{a} \in R_{2q}$ while the secret key is a pair of sparse elements $(\mathbf{s}_1, \mathbf{s}_2) \in R_{2q}^2$. Crucially, the signature contains an element of R_{2q} equal to

$$\mathbf{z}_1 = \mathbf{y}_1 + (-1)^b \mathbf{s}_1 \cdot \mathbf{c} \mod 2q$$

where $\mathbf{y}_1$ is generated according to a n-dimensional discrete Gaussian distribution, b is chosen uniformly at random in $\{\pm 1\}$, and $\mathbf{c} \in \{0,1\}^n \subset R_{2q}$ corresponds to a hash of the message and other values (and is also included in the signature). Now, because the elements of $\mathbf{c}$ are equal to 0 or 1, the ring multiplication $\mathbf{s}_1 \cdot \mathbf{c}$ actually correspond to a vector-matrix multiplication $\mathbf{s}_1 \cdot C$ where C is a matrix with entries in $\{-1, 0, 1\}$. In particular, the i-th coordinate of $\mathbf{s}_1 \cdot \mathbf{c}$ is equal to the scalar product $\langle \mathbf{s}, \mathbf{c}_i \rangle$ where $\mathbf{c}_i$ is the i-th column of C.

The cache-timing attack now works as following. The generation of $\mathbf{y}_1$ is done component-wise using a precomputed cumulative density table which contains the probabilities that an integer sampled from a discrete half-Gaussian is lower than k for values up to a fixed cutoff (the sign of the Gaussian is sampled separately). The strongSwan implementation of BLISS has the distinctive property that this table is *not accessed* when the sampled value is equal to 0, and this event can therefore easily be detected using a cache-timing attack.

Now suppose that for a given i, we know thanks to the cache-timing attack that $\mathbf{y}_{1,i} = 0$ and we also have that $\mathbf{z}_{1,i} = 0$ by inspecting the signature. We then deduce that $\langle \mathbf{s}_1, \mathbf{c}_i \rangle = 0 \mod 2q$. Furthermore, due to the sparseness constraint in $\mathbf{s}_1$ and the choice of q in the actual implementations, the absolute value of $\langle \mathbf{s}_1, \mathbf{c}_i \rangle$ is much smaller than q and therefore, this equality must hold in $\mathbb{Z}$, not merely in $\mathbb{Z}_{2q}$. Because $\mathbf{c}_i$ can also be recovered from the signature, we have learned one vector of the nullspace of $\mathbf{s}_1$. Once enough such vectors have been collected, $\mathbf{s}_1$ can be computed, either by solving a system of linear equations, or by applying the LLL algorithm [27] in order to cope with the measurement errors that might occur. Because $\mathbf{s}_2$ can actually be recovered from $\mathbf{s}_1$, the key is then fully recovered.

1.2.4 Avoiding Cache-Timing Attacks

Several countermeasures have been proposed in order to thwart cache-timing attacks. Most of them target specific methods to read the cache state. For instance, disabling simultaneous multithreading (SMT) makes reading the L1 cache more difficult (see Section 1.3.1), and disabling memory sharing between processes can avoid attacks based on the `clflush` instruction (see Section 1.3.3). However, these countermeasures reduce the performance of the whole system, and do not prevent all attacks. Other suggestions include performing cache flushes after any process finishes, which would only work in conjunction with the aforementioned techniques and would further slow down the system, or degrade the performance of available timer functions, which has been tried and was circumvented [28].

Adding more randomness to the cryptographic operations is another possibility. One can either resort to masking, as is done to prevent physical side-channel attacks [29], or add random slowdowns in order to minimize timing differences. When using masking, one has to be careful to avoid *higher-order* attacks [30] which could reveal the mask and render the countermeasure useless. Random slowdowns have to be implemented with care: the maximum slowdown and the granularity of the waiting periods need to be sufficiently high [10]. Also, they mainly work against timing attacks, and it is not clear how they can prevent cache-timing attacks from succeeding.

An other direction consists in detecting cache-attacks rather than preventing them.

Most measurement methods induce a higher than usual number of cache misses, and these can be detected by using performance counters [31]. However, it is not clear what to do once a cache-timing attack is detected - if the system shuts down, then the false positive rate of such methods needs to be *extremely low* for them to be useful. Furthermore, new measurement methods that circumvent these detectors could be designed, rendering them useless.

Finally, the most promising solution is to implement cryptographic algorithms without secret-dependent branching and memory accesses. This solution provably protects from cache-timing leakages and can be implemented without any other hardware or software modifications. Verification tools that assess the absence of leakages can provide a much-needed help for implementing constant-time algorithms. We designed and implemented such a tool. It is described in Chapter 2 and various results obtained thanks to the tool are provided in Chapter 3.

1.3 Measurement Methods

Cache-timing attacks exploit the dependency between the state of the cache and the value of sensitive data of a program, such as keys or security-relevant randomness. However, an attacker cannot directly access the state of the cache. In order to determine which memory addresses were accessed by the victim program, indirect methods have to be employed.

1.3.1 PRIME+PROBE

The first method that was described to perform cache-timing attacks is presumably PRIME+PROBE [32]. The first versions targeted the L1 cache, and therefore, the processes of the victim and the attacker need to run on the same physical core. This method is comprised of two phases:

- PRIME phase: the attacker accesses memory from its own memory space in order to completely fill the L1 cache with data under its control. They first need to determine how memory addresses are translated into cache sets (see Figure 1.2), and then accesses one memory address per cache set.
- PROBE phase: after the victim process has executed, the attacker records the access times for all the memory addresses accessed during the PRIME phase. If the access time is short, the data is still present in the cache. Therefore, the victim process has *not* accessed a memory address that maps to the same cache set. On the contrary, if the access time is slow, the data was evinced from the cache, and thus the victim process *did* access a memory address that maps to the same cache set. After all addresses have been accessed and the access time recorded, the attacker process waits for a while for the victim to execute before performing a new PROBE phase.

The order of operations is therefore:
PRIME $\mapsto$ wait $\mapsto$ PROBE $\mapsto$ wait $\mapsto$ PROBE ...

Because the L1 data and instruction caches are separated on most Intel and AMD processors, an attacker targeting the L1 instruction cache must execute instructions at

addresses that will map to all the addresses in the monitored cache sets, instead of just accessing the memory [33].

The main advantage of the PRIME+PROBE attack is that it does not need shared memory between the attacker and the victim process. Only the cache needs to be shared, which requires the two process to run on the same physical core. On the other hand, PRIME+PROBE is fairly slow compared with newer methods, and is also noisier. Indeed, the attacker cannot make the difference between two memory accesses (or instruction executions) of different virtual addresses that map to the same cache set. This method therefore does not achieve a granularity of one cache line. Also, background tasks unrelated to the victim process may also access the same cache sets as the victim, adding more noise to the measurements.

PRIME+PROBE on L3 cache The PRIME+PROBE attack has also been extended to the L3 cache [34, 35, 36]. The main advantage is that this attack works across cores and even VMs [36] with no simple mitigation. However, two complications arise when applying PRIME+PROBE to the L3 cache:

- The L3 cache is much larger than the L1, at a few MiB compared to about 32 KiB for the L1 caches. Therefore, priming and probing the whole L3 is much slower. This would reduce the time-resolution of this method by several orders of magnitude. Therefore, this attack requires a profiling phase, in which the attacker determines which cache sets are relevant for their attack.
- The L3 cache of modern Intel processors is partitioned into so-called *slices*. These slices span several cache sets, and the slice which will contain a given data will depend on its physical address. However, the function that maps physical addresses into slice indices is not public and depends on the processor family. However, this mapping can be reverse-engineered [37, 38, 35, 39], or the attack can be build in such a way to be independent of this mapping [34].

PRIME+ABORT A variant of PRIME+PROBE is called PRIME+ABORT [40]. It uses Intel's Transactional Synchronization Extension (TSX) to improve the timing resolution of the PRIME+PROBE side-channel, and targets either the L1 or the L3 cache.

TSX allows a program to execute a series of instructions inside a *transaction*, such that either all instructions are executed, or none of them. Because any operation that writes to memory during a transaction might be reverted during an abort, the processor needs to buffer all write operations that occur during the transaction. These operations are actually buffered in the L1 data cache.

Therefore, an issue occurs if any cache line that is written during the transaction is evinced from the L1 cache. This causes the transaction to abort, and causes a callback to a user-defined function. The PRIME+ABORT measurement techniques exploits this design in order to improve on the PRIME+PROBE techniques. It works as follows for the L1 cache:

- PRIME: the attacker starts a transaction, and *writes* to addresses spanning a whole *cache set* of L1 memory. Then, they wait in a loop. This phase is similar to

the PRIME phase from PRIME+PROBE, except that it is done inside a TSX transaction, and the attacker writes to the memory addresses, instead of reading from them. Also, only one cache set will be monitored.

- ABORT: When the transaction aborts, the attacker knows that a cache line from the monitored cache set was evicted. The attacker records this event, and then executes the PRIME code again.

There are mainly two advantages for this method. The first one is that no profiling phase is necessary to determine thresholds for “slow” and “fast” memory accesses. Second, there is a performance improvement over PRIME+PROBE. Indeed, the overhead of the attack is lower, as there is no probing phase, and the callback on transaction abortion is very fast. As a consequence, PRIME+ABORT is less likely to miss memory accesses from the victim. Furthermore, according to their authors [40], the number of false positives is also lower.

The L3 variant is similar, and shares the weaknesses with the PRIME+PROBE on the L3 cache. Notably, caches slices need to be determined before the attack can be launched. The other difference with the L1 variant is that a transactional abort happens also when data that was *read* during a transaction is evinced from the L3 cache, as opposed to only data that is *written* to the L1 cache. During the PROBE phase, the attacker therefore only needs to read data, in the same way as in the PRIME+PROBE technique.

1.3.2 EVICT+TIME

EVICT+TIME [32] infers the cache access pattern of a victim process by slowing it down selectively. The attack works as follows:

- Setup: the attacker executes the victim process, ensuring that the cache is filled with data used by that program.

- EVICT: using the same technique as PRIME+PROBE, the attacker pollutes a specific cache set, replacing data cached by the victim program with its own.

- TIME: the attacker runs the victim process again and measures the time it takes to complete. A fast execution time reveals that the evinced data would not have been accessed during the execution of the victim.

This attack works best on the L1 cache, because determining all addresses belonging to a given cache set is straightforward. While this attack is hard do detect and mitigate against, there are several shortcomings. Only the first access to the evinced data is slowed down (if it is accessed), so the attacker cannot distinguish between one or more accesses to the targeted cache set. Also, only one measurement is possible per execution of the victim process. More modern techniques, such as CacheBleed and MemJam (see Section 1.3.4) slow down the victim program proportionally to the number of memory accesses to the targeted addresses. Furthermore, they allow to target specific addresses even *inside* a cache *line*, instead of targeting all addresses belong to the cache set.

1.3.3 FLUSH+RELOAD

The FLUSH+RELOAD [41] technique is a more modern measurement technique, that targets the L3 cache. The main advantage is that this method is able to monitor individual cache lines, as opposed to whole cache sets as in the different PRIME+PROBE variants. It is also faster, easier to implement, and subject to a low rate of false positives. Furthermore, several cache lines can be monitored at the same time. It works as follows:

- FLUSH: the attacker select a series of virtual addresses corresponding to data shared with the victim. This can be code shared with the victim, if they make use of dynamically-linked libraries, or read-only data if the OS uses page sharing. They then execute the `clflush` instruction on these addresses, causing them to be evinced from the L3 cache, and thus also from the L1 and L2 caches due to the inclusiveness of the L3 cache.
- RELOAD: After the victim process has executed for a while, the attacker accesses the addresses selected in the first phase. If the access is fast, then the victim has accessed data corresponding to the same address. Else, because the data had previously been evinced from all caches, the access is slow. Once all addresses have been accessed, the attacker executes the FLUSH phase again.

The order of operations is therefore as follows:
FLUSH $\mapsto$ wait $\mapsto$ RELOAD $\mapsto$ FLUSH $\mapsto$ wait $\mapsto$ RELOAD $\mapsto$ FLUSH ...

Because this attack targets the L3 cache, it works in a cross-core setting: victim and attacker do not need to execute on the same physical core. The only weakness of this method is that victim and attacker need to share code. If page sharing is disabled by the OS, this can only happen when the victim uses dynamically-linked libraries.

FLUSH+FLUSH FLUSH+FLUSH [42] is a variant of the FLUSH+RELOAD attack. They mostly share the same weaknesses and advantages, with FLUSH+FLUSH having a slightly lower precision but also a lower latency. Furthermore, it causes far less cache misses than FLUSH+RELOAD, making an attack harder to detect.

In this attack, the RELOAD phase is replaced by a second, modified FLUSH phase:

- FLUSH^2: After the victim process has executed for a while, the attacker times the `clflush` operation on the selected addresses. If the operation is fast, then nothing needed to be done: the data was not present in the cache, and the victim did not access it. If it was slower, then the data was present in the cache, because it was accessed by the victim process.

The order of operations is therefore as follows:
FLUSH $\mapsto$ wait $\mapsto$ FLUSH^2 $\mapsto$ wait $\mapsto$ FLUSH^2 ...

1.3.4 CacheBleed and MemJam

All previously listed measurement techniques have a resolution of either a cache line, or a cache set. CacheBleed [43] and MemJam [44] are two techniques that provide a resolution finer than a cache line - up to 4 bytes in the case of MemJam.

CacheBleed CacheBleed uses cache bank contention to slow down a victim program that accesses memory located at a specific offset of a cache line. In order to speed up cache access, older Intel architectures (up to Ivy Bridge) make use of so-called *cache banks*. Every cache line is divided into several cache banks, which can be accessed concurrently. Therefore, if one program accesses data at different offsets of a cache line, the data can be fetched from the cache simultaneously. However, concurrent accesses to the same cache bank causes one of the two accesses to stall, increasing the latency. The authors use this observation to break the RSA encryption routine from OpenSSL 1.0.2f, which accesses the same cache lines, but not cache banks, regardless of the secret key.

MemJam Newer Intel processors (Haswell architecture and newer) do not use cache banks anymore. However, the MemJam technique is able to achieve a similar functionality as CacheBleed using so-called *false dependencies*.

True dependencies arise on a write and a subsequent read on the same memory address. These two operations cannot be reordered, because the result of the read operation depends on that of the write operations. Therefore, there is a dependency between those operations - the read *must* be executed *after* the write - and those two operations cannot be executed simultaneously. Therefore, executing those two operations is slower than a write and a read to two *different* memory locations.

False dependencies can arise when two independent operations are wrongly considered as being dependent by the CPU. One such type of false dependencies is called *4K aliasing* [5, §11.8], where a write and read to two distinct memory addresses that are a multiple 4096 bytes apart are falsely considered by the CPU as being dependent.

Because the translation between virtual and physical memory is done at the *page* level (see Section 1.1.1) which usually have a size of 4096 bytes, the 12[3] last bits of the physical and virtual memory addresses are identical. Therefore, addresses that differ in the last 12 bits cannot refer to the same physical address. However, if the last 12 bits are identical, aliasing is possible, and verifying whether the two *virtual* addresses refer to the same *physical* address requires to perform a virtual-to-physical-address translation first. Because of the additional delay introduced by the translation, the CPU considers that a read and a write operation to these addresses are dependent, and starts executing the first one instead of waiting for the result of the translation before deciding whether both can be executed simultaneously. The consequence is that a write and a read these two addresses, even if they do not represent the same data, are slower than operations to addresses that are not a multiple of 4096 apart.

MemJam exploits this additional latency by selectively slowing down a victim process, depending on the last bits of the addresses being accessed. It works by repetitively writing data to a given address, which slows down memory accesses to addresses being a multiple for the victim process. In order to be effective, the attacker and victim process must share the same physical core, and should thus execute on the two virtual cores of some physical hyper-threaded core.

The granularity of the attack is thus finer than a cache line, at 4 bytes instead of 64 bytes. Indeed, all memory accesses fetch a *word*-sized chunk of memory, totaling

[3] $12 = \log_2(4096)$

4 bytes, thus memory accesses that are part of the same word-sized chunk cannot be distinguished. This method allows the authors to break various block ciphers from Intel's IPP cryptographic library [45], which accesses the same cache lines regardless of the encryption key. The attack compares the encryption times for a program using an unknown key with those of a program using a chosen key and plaintext, and retrieves the encryption key after 2 million recorded encryption durations.

1.3.5 BranchScope

In order to avoid the performance penalty caused by branch mispredictions, modern CPUs implement mechanisms to predict the branch to be taken based on previous executions of the branches. The rationale is that if one branch has been taken during the most recent executions, then that branch is very likely to be taken again during the next one. To implement this mechanism, for every branching instruction, the CPU records how often the branch was taken or not among the last few executions. Depending on this value, during the next execution, either the taken or the not-taken path will be speculatively taken, and upon success or failure, the counter will be updated. In practice, this counter is a 2-bit saturating counter, meaning that it can be in 4 different states: Strongly Non Taken (SN), Weakly Non Taken (WN), Weakly Taken (WT) and Strongly Taken (ST). When the branch is taken during some execution, the counter transitions from SN to WN, from WN to WT or from WT to ST. Conversely, if the branch is not taken, it transitions from ST to WT, from WT to WN or from WN to SN. Meanwhile, the branch predictor will always predict that during the next execution, the branch will be taken if the counter is in the ST state, and will not be taken if it is in the SN state. The decision for the two intermediary states depends on the architecture.

The counter is tied to the virtual address of the branching instruction. Therefore, two branches in two distinct processes may refer to the same counter if the code is loaded at the same virtual address. The BranchScope [46] side-channel exploits this observation. It works as follows. First, the attacker saturates the counter by repetitively taking a branch loaded at the same virtual address than the target branch of the victim process. Then, it waits for the victim process to execute this branch once. Finally, the attacker executes the branch again, recording the execution time of these instructions. A slower execution time indicates a branch miss, which allows to infer the direction of the branch taken by the victim process.

1.4 Attack Improvement Methods

No measurement technique is able to perfectly capture all cache accesses. Cache line prefetching [5, §2.4.4.2] can cause false positives, long execution times of the attacker code can lead to false negatives, and system activity can cause either false positives or false negatives, depending on the measurement method. In order to improve their precision, several methods have been developed, targeting either the victim or the attacker process.

1.4.1 Performance Degradation Attacks

In addition to the methods enabling an attacker to determine the state of the processor cache, micro-architectural side-channel attacks generally make use of techniques known as performance degradation attacks. Determining the cache state takes at least several hundred cycles (FLUSH+FLUSH), but it might be much slower (PRIME+PROBE). If the victim process accesses the cache several times in a short time, some of these accesses might be missed by the attacker. The attacker can try to slow down the execution of the victim process, increasing the time between two cache accesses and reducing the number of missed ones. Two mechanisms to achieve such a slowdown have been proposed in order to mount micro-architectural side-channel attacks.

Attacking the scheduler One technique to slow down the victim process is to cause it to run for only a short time period before it is preempted by the OS [47]. It specifically targets the Linux OS and the scheduler known as the Completely Fair Scheduler.

One crucial task performed by the OS is that of *scheduling*. In general, at a given point in time, more processes have been launched and need to run to completion than there are cores to execute them. In this situation, a given number of processes are *running*, ideally as many as there are CPU cores, while other are either *blocked* and are awaiting some external event (such as the release of some lock), or *ready* and could run if some processor core becomes available. The role of the scheduler is to decide when processes should execute on an available core, and when they should wait so that other processes can make progress towards completion.

To decide in which order the processes should be run, the CFS scheduler associates a *virtual time* τ to every process. When a process is executed, its virtual time increases. When the difference in virtual time between the running process and the process with the lowest virtual time attains a given threshold $\Delta\tau_{max}$, the process is stopped and one of the processes with the *lowest* virtual time is scheduled instead. This ensures that unfairness, defined by the maximum difference in the amount of CPU time allocated to any two processes, is upper-bounded.

Blocked processes are special cases. Because they cannot be executed unless an external event happens, the difference in virtual time between the currently executing process, τ_{run}, and any blocked process can exceed $\Delta\tau_{max}$. However, as soon as a blocked process becomes ready to run, its virtual time is updated and set to $\tau_{run} - \Delta\tau_{max}$ if its virtual time was lower than this value. Thus, a process that was blocked for a long time will be immediately scheduled to run when it is no longer blocked.

This behavior can be abused if an attacker is able to schedule multiple threads to run on the same core as the victim process. Suppose that the attacker launches $N \simeq 100$ threads, numbered from 1 to N, which are all blocked for a long amount of time: each thread $i + 1$ waits on a signal that will eventually be send by the thread i, and the first thread busy-waits for a long period of time before launching the attack. While the attack is active, thread i:

- Measures the cache state as specified by the attack (for instance using FLUSH+RELOAD), and

- Sets a POSIX timer to send a signal to unblock thread $i + 1$ after a set number of cycles, and
- Returns into a blocked state.

The number of cycles for setting the timer is chosen in such a way that the victim process (ideally) executes a single memory access, before the next thread becomes unblocked. Because of its low virtual time, it then preempts the currently running victim thread. This ensures that the attacker will not miss any memory access.

While very powerful if correctly executed, this attack is rater complex to implement. First, it requires to precisely determine the time when the next thread is awoken. This time depends on the duration of the necessary context switch, which might be variable. Also, the attacker cannot ensure that the victim process will be scheduled next, as other processes might be simultaneously runnable on the same CPU. Finally, it is not entirely clear how CPUs with multiple cores should be handled.

Slowing down the execution A second technique consists in making the victim's instructions take more time to execute. If the attacker and victim process share resources on the same processor core, then an intensive utilization of these resources by the attacker processes can slow down the execution of the victim process. Possible target resources could be the cache or the execution ports. Note that this observation can be used to design microarchitectural side-channel attacks: cache contention is used for the EVICT+TIME attack [32], while execution port contention forms the basis of the PORTSMASH attack [48].

In order to improve the precision of microarchitectural side-channel attacks, cache contention has been shown to be fairly effective [49]. This method consists in having a helper process repeatedly evincing cache line corresponding to code executed by the victim process. It works best if the victim spends time executing a small number of instructions in a loop with a large number of iterations. In this case, the helper process can evince the instructions making up the loop body on every iteration. This can slow down the execution time of that loop by a factor up to 40, which corresponds to the ratio between the latency of a memory access from main memory and that of an access from L1 cache [49]. Because the victim process is slowed down, the attacker is less likely to miss memory accesses from the victim process.

The method is easier to implement (the helper process only consists in a loop of `clflush` instructions) and does not depend on the scheduler and OS used. It requires a preparation phase in order to determine which instructions from the victim process to flush, but these do not need to be adjusted during an attack. There are is mainly one major drawback: the factor by which the victim code is slowed down largely depends on the structure of this code. When no tight loop is present, the slow-down factor might be low and the precision of the attack might not be improved enough.

1.4.2 Methods Based on SGX

SGX is a technology created by Intel that allows programs to create secure enclaves. Code executed inside an enclave writes to encrypted memory zones, so that programs executed

outside the enclave, even the OS, cannot spy on programs executing *inside* the enclave. The adversarial model SGX consider thus also comprises processes running with higher privilege level, such as programs run as the system administrator or super-user. While SGX should prevent the OS from recovering sensitive information of programs running in a secure enclave, this protection can be breached in practice using cache-timing attacks. These attacks mainly use the measurement methods methods previously described, but their efficiency can be improved by using methods that are specific to victim code running on a SGX enclave.

CacheZoom [50] The CacheZoom attack makes use of two techniques that improve the precision of the cache measurements. First, it uses a mechanism called "process isolation" to isolate a physical processor core from processes other than the victim and attacker process. This reduces the measurement noise caused by other processes accessing the cache while an attack is carried out. While this technique is not specific to SGX, it requires the attacker to run with administrator rights, which is possible in the threat model SGX targets. The second technique consists in setting a very short timer interrupt delay on the enclave running the victim process. This causes it to be interrupted often, ideally once between every two memory accesses, and allows the attack to measure the cache state without missing a victim memory access. Thus, this technique is somewhat similar to an attack on the scheduler, described in Section 1.4.1. CacheZoom is however easier to set up, while also requiring a more powerful attacker.

The main drawback of CacheZoom is that the victim process may detect the frequent interruptions, and thus abort its execution. Countermeasures based on this property have already been proposed [51, 52].

Software Grand Exposure [53] A second set of techniques, dubbed Software Grand Exposure [53], exploits the additional attacker capabilities under the SGX threat model while minimizing the risk for the attacker to be detected. In addition to isolating the victim and attacker thread from other processes, as in [50], Software Grand Exposure proposes mainly two new techniques to improve the precision of cache measurements. First, it disables OS interrupts for both the attacker and victim processes, and increases the delay between timer interrupts. Thus, the victim and attacker processes are very likely to run uninterrupted while the attack takes place. The advantage is twofold: avoiding interruptions of the victim process reduces the noise caused by restarts of the enclave, and running uninterrupted reduces the number of missed cache accesses by the attacker process. The second method consists in using hardware counters, instead of timestamp counters, to detect whether a cache-hit or cache-miss took place. These allow the attacker to count the number of cache misses, which is more precise than using timing-based methods to determine if a cache miss took place. These counters, however, cannot be used by an unprivileged attacker.

Chapter 2

STAnalyzer: Static Analysis for Cache-Timing Vulnerability Detection in C Programs

> *But life is short and information endless: nobody has time to everything. In practice we are generally forced to choose between an unduly brief exposition and no exposition at all. Abbreviation is a necessary evil and the abbreviator's business is to make the best of a job which, although intrinsically bad, is still better than nothing. He must learn to simplify, but not to the point of falsification. He must learn to concentrate upon the essentials of a situation, but without ignoring too many of reality's qualifying side issues.*
>
> Aldous Huxley

2.1 Introduction

When implementing cryptographic algorithms in software, timing attacks represent a great risk to the security of the algorithm. For instance, the total execution time can depend on the key, which can be revealed by multiple measurements [12]. More subtle attacks exploit the latency difference between the fast processor cache and the slower main memory (RAM). These attacks are collectively known as *cache-timing* side-channel attacks. They can determine, on a branching operation, which branch is being executed even if the branches are balanced and execute the same instructions [13]. They can also reveal the address of memory accesses [24]. These attacks can be mitigated if some simple guidelines are followed: no memory access and no branching operation should depend on sensitive values. A code that follows these guidelines is said to be *constant-time* and immune to cache-timing attacks. In this chapter, we propose an algorithm that is able to determine

whether a program written in the C programming language follows the constant-time guideline. This algorithm is implemented in our tool STAnalyzer.

2.1.1 Related Work

There are several ways to verify that a program follows the constant-time guidelines. It can be checked during program execution, this is called a *dynamic* analysis. One popular tool is *ctgrind*, an extension of the *valgrind*[1] profiler tool. The program executable can be annotated, and the program can be executed in a virtualized environment, where memory accesses and branching operations are monitored. While this approach has the advantage of analyzing real program behavior, and thus minimizing false positives, it is in general not possible to explore all program paths. This might actually lead to false negatives, i.e. missed constant-time policy violations.

Another approach consists in analyzing the program code. One can either inspect directly the C source code [54], the compiled assembler instructions [55], or work with an intermediate representation of the program code [56, 57, 58]. Analyzing the assembler instructions has the advantage of taking into account compiler optimizations that might lead to either vulnerability removal or insertion, but has not been fully implemented for Intel x86 or x86-64 assembly, probably due to the complexity of the Intel assembly language [59]. A more common approach is to analyze intermediate representations, either in the LLVM intermediate language [60] or the Single Statement Assignment (SSA) form [56]. Both of these representations are emitted by the compiler after some optimization has been applied. Therefore, it accounts for some removals or insertions of timing leakages by the compiler, while being easier to analyze than a program written in low-level assembly. However, the link to the original source code is somewhat lost, and it is not necessarily easy from such an analysis to understand where the leaks originate from. Finally, working directly on the source code is also possible. However, the only existing tool [54] does not allow to specify sensitive variables and requires a manual inspection of the leaking variables. We propose an alternative that allows for a fine-grained tracking of sensitive values. At the same time, we leverage the information in the C source code files to provide a meaningful explanation for any source of leakage.

Because the algorithm works directly on the source code, compiler transformations cannot be taken into account. While some leakages might be missed due to the compiler adding branches to the code, this does not seem to occur often in practice; more often, the compiler seems to actually *remove* some leakages. For example, the C code shown in Figure 2.1 uses a conditional branch, but the compiler replaces it with a conditional set instruction (**setg**), a constant-time operation. In the rest of this chapter, compiler-specific behavior is considered out-of-scope.

2.1.2 Background in Dependency Analysis

A program can be defined as a series of operations acting on the program *state*. For programs written in higher-level languages such as C, the state is defined by a set of *variables*. These variables can take different *values*, and the execution of a program consists

[1] https://github.com/agl/ctgrind/

```
1 int sign(int value) {
2   int ret;
3   if (value > 0) ret = 1;
4   else ret= −1;
5   return ret;
6 }
```

(a)

```
1 xor %eax,%eax
2 test %edi,%edi
3 setg %al
4 lea −0x1(%rax,%rax,1),%eax
5 retq
6 nopl 0x0(%rax)
```

(b)

Figure 2.1: Comparison between source (a) and compiled (b) code (gcc with -02 optimizations)

in manipulating these values and assigning them to different variables. For the sake of our analysis, we will ignore other effects of program execution, notably I/O operations (screen printing, network and file operations, etc.).

Static analysis tools infer properties of the program execution from source code only, without actually executing it. Usually, the actual values manipulated by the program are not known. Instead, special abstract values are assigned to the variables, and the result of operations manipulating these values have to be defined. For instance, we can define two values, H and L, where H represents high security values, and L low security ones. The result from any binary operation $\circ$ can then be defined as such: $L \circ L = L$ and $H \circ L = L \circ H = H \circ H = H$.

Information flow analysis consists in determining the values that might influence the variables in a program. For example, variables and values could be assigned security classes. This idea has been adapted by Rodrigues et al. [61] to create FlowTracker, a tool which performs information flow analysis in order to determine whether cache-timing leakages occur in a given program. Both these methods operate on the Static Single Assignment (SSA) format of a program. Rodrigues et al. consider two security classes, *high* and *low*, and their program determines whether any value of the *high* security class leaks due to branching instructions or memory access. As already noted by Denning and Denning [62], in this model, function calls need to be treated with care. Indeed, they might create new dependencies between variables which should be accounted for. Also, since they might be called with arguments of high or low security class, it is not clear which leakages should be reported.

We propose a different idea: instead of tracking only two security classes, we define more abstract values: one for the initial value of any function arguments, for global variables and sensitive values. This makes it possible to track leakages and dependencies across multiple function calls, and allows to present more information about the flow of information inside a program.

The rest of the chapter is organized as follows.Section 2.2 provides the definitions and notations used in the remainder of this chapter. Section 2.3 provides a high-level description of the dependency analysis performed by our tool. Section 2.4 presents some of the results we obtained using it. Finally, Section 2.5 concludes.

2.2 Definitions and Notations

In this section, we will introduce the objects manipulated during the static leakage analysis. Table 2.1 explains the different notations styles used in the remainder of this paper. They serve to distinguish between actual C code, generic elements of the C language, and the objects abstracted from the C code which are used by the leakage analysis.

Table 2.1: Notation styles used in the remainder of this paper

Notation	Description
int x = y;	Concrete C code example
expr	Abstract element of the C language
`var`	Symbolic object manipulated by the static analysis

2.2.1 Variables, Pointers and Values

In order to determine whether sensitive values might leak during the execution of a program, one needs to determine which variables of a C program might actually hold sensitive values. Performing a so-called *dependency analysis* solves this problem. This consists in determining, for any variable of a given program, what other variables might influence its value. In this section and the next, we will introduce the different notions that are required to describe how to perform this dependency analysis. It consists in defining a set of special symbolic *values*, and determining for any *memory location*, labeled by a *variable*, which values its content might depend on.

Definition 1 (Variable)**.** A *variable* is an identifier in a C program. It references a *memory location.* A variable will be denoted by its name, such as `varname`. In addition to all variables declared in the analyzed function, we also refer to the special `\RET` variable, which represents the return value of a function.

Definition 2 (Memory location)**.** A *memory location* is an independent region of memory that can store one or more *values*, as well as the addresses of other memory locations. A memory location will be denoted by the variable that references it, and we will use the same notation for both the variable and the memory location it refers to.

Listing 2.1: C code with variables

```
1 int foo(int a, int b) {
2   int c = a + b;
3   return c;
4 }
```

For instance, the code of Listing 2.1 defines three 4 variables: `a`, `b`, `c` and `\RET`. Each variable also defines a memory location with the same name.

Definition 3 (Value). A *value* is a specific symbol which will be tracked during the dependency analysis. The set of values depends on the function being analyzed. These values are:

- the special `\SECRET` value, which should not be leaked by the program execution and represents some sensitive value,
- all global variables, and
- the initial values of all function arguments (in order to compose the analysis of several functions). The initial value corresponding to the variable `varname` is denoted by $\texttt{varname}^0$ to distinguish it from the variable it originally refers to.

Listing 2.2: C code example

```
1 int foo(int a, int b) {
2         int j = a;
3         int i[2];
4         i[0] = b; i[1] = a;
5         return i[0]+i[1]+j;
6 }
```

As an example, consider the C code in Listing 2.2. The state of the memory locations, at the end of the function execution, can be represented as in Table 2.2.

Variable (memory location)	Values
a	a^0
b	b^0
i	a^0, b^0
j	a^0
\RET	a^0, b^0

Table 2.2: State of the memory locations for the code in Listing 2.2

Definition 4 (Pointer). A *pointer* is a variable referencing one or more other memory locations. If the C code defines a pointer named `varname`, we will denote the variable by `&varname` if it is a pointer on an integral type, by `&&varname` if it is a pointer on a pointer on an integral type, etc. The referenced value will in any case be denoted by `varname`.

Definition 5 (Pointer indirection). The *pointer indirection* level of a variable is the length of the pointer chain until the referenced value is attained. As such, indir(`varname`) = 0, indir(`&varname`) = 1, indir(`&&varname`) = 2, etc.

Pointer *dereferencing* and the *address of* operation change the indirection level of a variable. Therefore,

- &(varname) = &varname
- *(&varname) = varname

During our analysis, we consider that every variable declaration of any pointer simultaneously initializes a corresponding *memory location* of sufficient size. In a real program, this has to be done via a call to malloc or a similar function. Issues related to uninitialized memory are therefore out of scope of this analysis.

Note that according to that definition, a pointer may point to more than one memory location. There are two reasons for this definition. The first is to accommodate arrays of pointers. Such a variable can contain pointers to several memory locations. Our analysis will not be able to determine which one will be accessed when the variable will be dereferenced, and we will have to assume that all of them will be accessed or modified.

The second reason is that, depending on the program execution, a pointer might point to one among several memory locations. Our static analysis must consider all of these possibilities. In order to simplify the analysis, we will again consider that all memory locations will be accessed or modified when the pointer is dereferenced.

An example of C code containing pointers is provided in Listing 2.3, and the state of all memory locations at the end of the function execution is given in Table 2.3.

Listing 2.3: C code example with pointers

```
1 int foo(int a, int b) {
2         int j = a;
3         int i[2];
4         i[0] = b; i[1] = a;
5         int **k = malloc(2*sizeof(int *));
6         k[0] = i; k[1] = &j;
7         return i[0]+i[1]+j;
8 }
```

Note the fact that k is listed among the possible memory locations for the pointer &k is an artifact of our hypothesis that any pointer initializes sufficient memory - we suppose that &k points to some allocated memory location before any other operation is performed. This is not the case in this example, and there is no actual memory location bound to k.

Structures Structures are special types of C variables. A structure defines one or more fields, which will be bound to a single variable. We will consider the different fields as belonging to distinct memory locations. While it is possible to read or modify data of one field by accessing another field, through a buffer overflow for instance, the result of such an operation is arguably undefined behavior. Such techniques should not be used in cryptographic code.

Therefore, in order to handle structures, our analysis will simply consider different fields as corresponding to different variables, and track the dependencies of all the fields independently. One issue, however, arises with self-referential structures, i.e. structures that contain a pointer to a field of the same type as themselves. Such structures are

Variable (memory location)	**Values**	**Pointed memory locations**
a	a^0	–
b	b^0	–
i	a^0, b^0	–
&i	–	i
j	a^0	–
&j	–	j
k	–	–
&k	–	i, j, k
&&k	–	&k
\RET	a^0, b^0	–

Table 2.3: State of the memory locations for code in Listing 2.3

common when implementing linked lists for example. A static analysis cannot know how many distinct memory locations such a structure references. We solve this problem by detecting such types of structures and stop considering new memory locations past a certain recursion depth.

2.2.2 Dependency Graph

The dependency graph encodes the mapping between memory locations (variables), abstract values (see Definition 3) and referenced memory locations (see Definition 4). Table 2.3 provides an example of such a mapping that can be directly encoded into a dependency graph.

Definition 6 (Dependency graph)**.** The *dependency graph* G is a directed graph, where the *vertices* are memory locations (variables) and abstract values. *Edges* in G are only possible from memory locations to values and to other memory locations (or equivalently, there are no outgoing edges from values, only from memory locations). An edge between a memory location and an abstract value signifies that the data stored at that memory location might depend on the abstract value, and an edge from a memory location `A` to `B` means that `A` might contain the address of the memory location `B`.

Definition 7 (Dependency graph notations)**.** For a variable `varname`, we denote by $G(\texttt{varname})$ the neighbor vertices (memory locations and values) of `varname` in G. $G_M(\texttt{varname})$ (resp. $G_V(\texttt{varname})$) represents the set of memory locations (resp. the set of values) among all the neighbors of `varname` in G. These notations are naturally extended for a set of variables S,

$$G(S) = \bigcup_{\texttt{varnanme} \in S} G(\texttt{varname}),$$

and a similar extension holds for G_M and G_V.

Definition 8 (Dependency graph operations)**.** Given two dependency graphs G_1 and G_2, the *union* of these two graphs is denoted by $G_1 \cup G_2$. The vertex set (resp. edge set) of this graph is equal to the union of the vertex sets (resp. edge sets) of G_1 and G_2.

Given a variable `varname` and a set of variables and values E, we denote by $\{\texttt{varname} \to E\}$ the graph with vertices $E \cup \{\texttt{varname}\}$ and edges $\{(\texttt{varname}, e) | e \in E\}$. Furthermore, given a graph G, we denote by $[\texttt{varname} \to E]\, G$ the graph obtained from G after removing all the edges starting from `varname` and adding the edges $\{(\texttt{varname}, e) | e \in E\}$.

2.2.3 Leakage Analysis

If the cache access pattern of a program changes depending on their input, then information about that input might be recovered by a spy process. Conversely, if the cache access pattern is constant, then no information might be retrieved by such methods. The cache access pattern depends on:

- the program control flow, because it changes the access pattern to the *instruction* cache, and
- the addresses of dereferenced pointers, because they change the access pattern to the *data* cache.

The cache access pattern need not be entirely constant for a program to be secure. Rather, it should not depend on *sensitive* values. For a signature algorithm for instance, the access pattern may depend on the data being signed, but not on the secret key that is used. Furthermore, some cryptographic algorithms rely on the output of an RNG, and an attacker able to recover enough information about that RNG might retrieve the secret key [24].

The objective of the leakage analysis is therefore to detect cache access patterns, which we will simply call *leakages*, that depend on sensitive values. However, because our analysis will be performed function by function, the leakage of other values need also to be tracked, namely the values of the function parameters, as well as global variables. These correspond exactly to the symbolic values tracked by the dependency analysis, as defined in Definition 3. Determining the leakages of these values is necessary in order to determine the leakages caused by function calls.

Definition 9 (Leakages)**.** The *leakages* of a function is the set of *values* that might change the control flow and the addresses of dereferenced pointers during the execution of the function. This includes the leakages caused by all the called functions.

2.2.4 Indirect Dependencies

Indirect dependencies are a particular type of *implicit dependency* [61]. Take for example the C code in Listing 2.4. The value returned by this function should depend on the value $\texttt{a}^0$, but this cannot be inferred by considering only the assignments made to the variable `b`. Instead, one has to note that the control flow is controlled by the value of the variable `a` and can lead to an assignment to `b`.

Listing 2.4: Indirect dependency flow example

```
int foo (int a) {
    int b = 1;
    if (a > 0)
        b = −1;
    return b;
}
```

One way to resolve the indirect dependencies is to keep track of all the values that might influence the control flow leading to a given region of the control flow graph. Once that region has been analyzed, one should consider that all variables modified during the analysis of that region depend on the values that influence the control flow graph directly leading to that region.

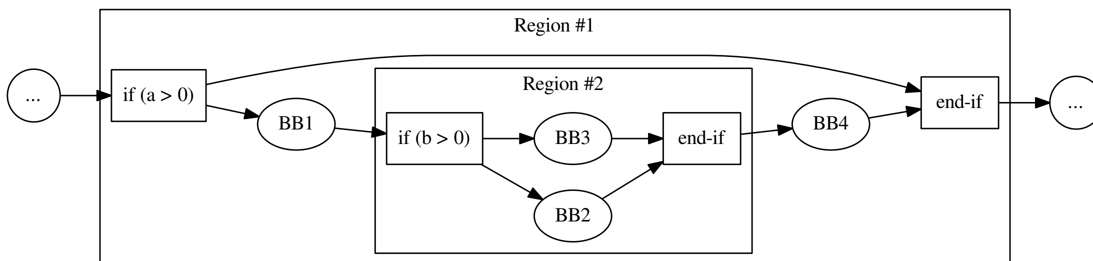


Figure 2.2: Example of a control flow graph with influence regions

In the more complex example represented by Figure 2.2, two regions are highlighted. For the region 2, any variable modified during the execution of either BB2 or BB3 depends on the value of `b` at the beginning of BB4. Similarly, for region 1, any variable modified during the execution of either BB1, BB2, BB3 or BB4 depends on the value of `a` before the next basic block is executed.

It is not immediately obvious why tracking indirect dependencies in order to perform leakage analysis is necessary. Indeed, keeping track of non-sensitive indirect dependencies is not useful, and when indirect dependencies are sensitive, the analyzed program cannot be constant-time. This would imply that the control flow depends on sensitive values, which contradicts the constant-time requirements. However, there are two reasons why it still might be be useful to consider indirect dependencies.

First, the aim of our program is to determine *all* constant-time violations. When indirect dependencies are not considered, then some variables might falsely be considered non-sensitive, and leakages caused by them are missed. Second, some branching operations are compiled into constant-time code by modern compilers, and thus do not necessarily result in a constant-time violation (see Figure 2.1). Our tool allows the user to flag branches as being *safe*, i.e. do not result in a constant-time violation. In this case,

indirect dependencies must be tracked in order to avoid underestimating dependencies and potentially missing leakages.

2.3 Description of the Algorithm

The program implementing the leakage analysis mainly follows the Visitor pattern [63]. It maintains a global state, comprised of the dependency graph and the set of currently leaking values, which is updated after evaluating every node of the AST representing the program to be analyzed. It also computes, for every node, which variables the expression represented by this node depends on.

To define the algorithm, it is thus sufficient to describe how the different tracked objects are updated when evaluating a given node, and how to determine which variables a given expression depends on. Notations that are introduced in this section are summarized below in Table 2.4.

Table 2.4: Notation used in this section

Notation	**Description**
G	Dependency graph
L	Set of leakages
$\langle expr \rangle$	Variables that *expr* depends on
$\langle expr \rangle^R$	Variables that *expr* depends on when considering *read* operations
$\langle expr \rangle^W$	Variables that *expr* depends on when considering *write* operations
$\phi(G, L; inst) = (\phi_G(G, L; inst), \phi_L(G, L; inst))$	Updated dependency graph ($\phi_G(G, L; inst)$) and leaking values ($\phi_L(G, L; inst)$) after the instruction *inst* has been analyzed.

The algorithm verifies that no sensitive values is used in a leaking operation. In a real-world setting, the cache architecture, size and eviction policy have to be taken into account in order to determine if a leakage is exploitable, as well as the implemented countermeasures. In contrast, our algorithm does not take these implementation-specific details into account. As a consequence, a program found to be constant-time will be secure regardless of the configuration of the cache (assuming no leakages were introduced by the compiler). On the other hand, not all leakages found by our tool might be exploitable on every cache configuration.

2.3.1 Abstract Syntax Tree (AST)

An AST is an abstract representation of the source code of a program. In such a graph, every node represents an element of the programming language, and the edges connect

these elements with their arguments. Consider for instance the C code represented in Listing 2.5.

Listing 2.5: Example C code

```
1 int foo (int a, int b) {
2   int c;
3   if (a > 0)
4     c = a+b;
5   else
6     c = a - b;
7   return c;
8 }
```

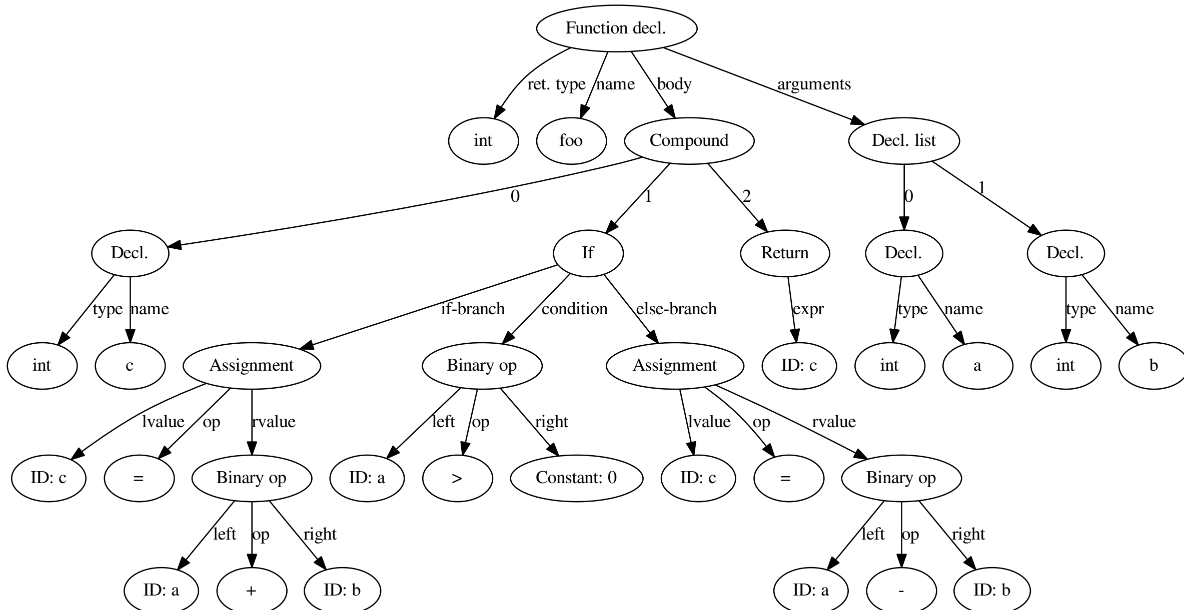


Figure 2.3: AST of the program described in Listing 2.5

The corresponding (simplified) AST is represented in Figure 2.3. It can be noted that this is a direct translation of the source code - there is only ”one“ way to build an AST from a given source code, whereas there can be several ways to compile the source code into, for example, x86 machine code.

2.3.2 Dependency Graph Interpretation

The dependency graph encodes the dependencies between variables and values, and as such, can be queried to determine the set of memory addresses and values that a given variable depends on. However, for a variable `varname`, this set is not equal to $G($`varname`$)$. Indeed, suppose that the dependency graph can be represented as in Table 2.3 and that

the dependencies of `k` need to be determined. A wrong answer would be $G(\texttt{k})$, as this is the empty set. Instead, one should walk down the dependency graph, starting from the node `&&k`, to determine that in fact, the dependencies of `k` are equal to $G(\texttt{i}) \cup G(\texttt{j}) \cup G(\texttt{j}) = \{\texttt{a}^0, \texttt{b}^0\}$.

The algorithm to determine the dependencies of a given variable is described in Algorithm 2.3.1. The result of this algorithm will be denoted by dep(G, `varname`). This notation is extended to a set of variables in the following way:

$$dep(G, S) = \bigcup_{\texttt{varname} \in S} dep(G, \texttt{varname}).$$

Algorithm 2.3.1 Computing the dependency set for a given variable

```
procedure DEPENDENCIES(G, varname)
    declVarname ← varname with indirection level corresponding to that of its declaration
    indirLevelDifference ← indir(declVarname) - indir(varname)
    nodesToExplore ← {declVarname}
    newNodesToExplore ← ∅
    if indirLevelDifference < 0 then
        return varname
    end if
    for i = 1 to indirLevelDifference do
        for node ∈ nodesToExplore do
            newNodesToExplore ← newNodesToExplore ∪ G(node)
        end for
        nodesToExplore ← newNodesToExplore
        newNodesToExplore ← ∅
    end for
    return {G(node)|node ∈ nodesToExplore}
end procedure
```

The example of computing dep(G, `k`) is visualized in Figure 2.4, where the set nodesToExplore is colored in gray.

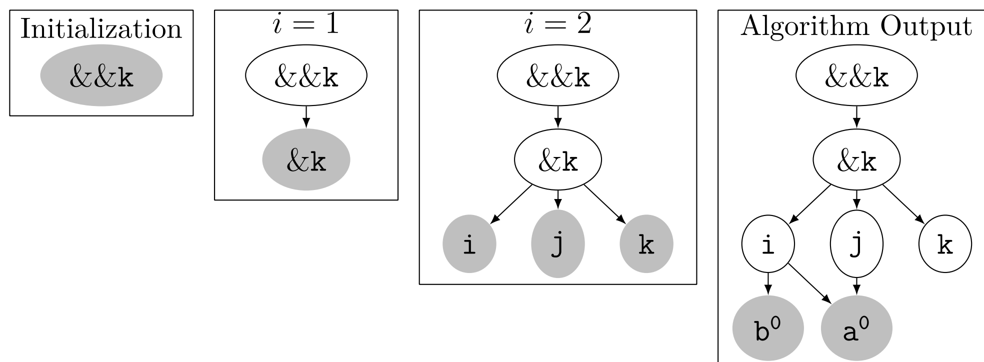


Figure 2.4: Visualization of Algorithm 2.3.1 applied to the dependency graph in Table 2.3

2.3.3 Expression Evaluation

Given an expression of the C language, it is necessary for the analysis to determine which variables this expression refers to. A natural way to define them is explained in Table 2.5.

Table 2.5: Variable dependencies for various expressions

$expr$	$\langle expr \rangle$
$literal$	$\emptyset$
var	$\{\texttt{var}\}$
$expr_1\ op_2\ expr_2$	$\langle expr_1 \rangle \cup \langle expr_2 \rangle$
$op_1\ expr$	$\langle expr \rangle$
$var = expr$	$\langle expr \rangle$
$var\ op_2 {=}\ expr$	$\langle expr \rangle \cup \{\texttt{var}\}$
$\&expr$	$\{\&\texttt{var} \mid \texttt{var} \in \langle expr \rangle\}$
$*expr$	$\{*\texttt{var} \mid \texttt{var} \in \langle expr \rangle\}$
$var[expr]$	$\{*var\} \cup \langle *expr \rangle$

The objective is twofold: for operations that read values from memory, the analysis needs to determine which memory locations are read from. Similarly, for operations that write to memory it needs to determine which memory locations are modified. For the same expression of the C language, these two sets are not necessarily identical.

Consider, for instance, a simple array access operation, arr[index], where arr is some array and index some integer. Our analysis has to consider that `arr` represents a single memory location and therefore, the values that this expression depends on are the same, regardless of the value of index. This does not mean, however, that the value of the expression does not depend of the value of index. For instance, arr could be a fixed array used to implement some permutation. Because its content is fixed, no value is recorded in the memory location corresponding to `arr`. However, the result of the operation does indeed depend on the values recorded at that of `index`.

On the other hand, if arr[index] is part of an assignment operation, such as arr[index] = y, then this operation should only update the values recorded at the memory location corresponding to `arr`, not that of `index`.

In order to distinguish between these two cases, we define the dereference operation on non-pointer variables as yielding a new variable with *negative* indirection level. They can be interpreted as the variables which values are used to compute an address *inside* a single memory location.

The negative indirection level will be indicated by a number of `*` symbols. Thus, a variable with indirection level -1 will be denoted by `*varname`, with -2 by `**varname`, etc.

Thus, for a given expression *expr*, $\langle expr \rangle$ might contain variables of negative indirection level. How they are interpreted depends on whether *expr* represents a memory location to write to, or to read from:

- If *expr* should be read from, then the result might depend on the values of the variables with negative indirection level. Therefore, in order to determine the *read*

set for *expr*, variables of negative indirection level have their indirection level set to 0. The resulting set is denoted by $\langle expr \rangle^R$.

- Conversely, when the memory location for a write operation should be computed, variables of negative indirection level should be *removed*. The resulting set is denoted by $\langle expr \rangle^W$.

Applying these rules to the previous example where the expression was arr[index], we get that $\langle$arr[index]$\rangle = \{$`arr`, `*index`$\}$. Therefore, $\langle$arr[index]$\rangle^R = \{$`arr`, `index`$\}$ and $\langle$arr[index]$\rangle^W$ $= \{$`arr`$\}$.

2.3.4 Instruction Interpretation

The main algorithm visits each node of the AST and updates the dependency graph and set of leaking instructions according to the semantics defined in this section. Indirect dependencies are handled as explained in Section 2.2.4 and ignored in the description of the semantics for the sake of clarity. The pair comprised of the modified dependency graph and set of leakages, after analyzing the instruction *inst*, is denoted by $\phi(G, L; inst)$ where G and L are respectively the initial dependency graph and leakages. The dependency graph of this pair will be denoted by $\phi_G(G, L; inst)$ and the set of leakages by $\phi_L(G, L; inst)$.

Analyzing simple instructions

The modifications on the dependency graph G and leaking values L when analyzing simple instructions are summarized in Table 2.6. ϕ_G (resp ϕ_L) represents the function that computes the dependency graph (resp. the set of leaking values) given the previous state of G and L, and the instruction to analyze. Note that the set L is only comprised of *values*, not of *memory addresses*. As such, if E is a set comprised of both values and memory addresses, $L \cup E$ is understood as the union of L and the *values* of E.

Flagging sensitive variables

As noted in Table 2.6, sensitive variables are flagged using a special PRAGMA directive. This directive creates a dependency between the targeted variable and a special value, `\SECRET`. For instance, the code from Figure 2.5 will result in the dependency graph represented in Table 2.8.

```
1 int foo2(int a, int b) {
2         int i=1, j = a;
3         #pragma STA secret i
4         return i+j;
5 }
```

Figure 2.5: Example of C code with tagged variable

The semantics for variable flagging are described in Table 2.7. Any leakage of the special `\SECRET` value is reported as a violation of the constant-time policy.

Table 2.6: Dependency and leakage semantics for simple instructions

Instruction to analyze		Result
$var = expr$	ϕ_G	$\left[\texttt{var} \rightarrow dep(G, \langle expr\rangle^R)\right] G$
	ϕ_L	L
$var\ op_2 = expr$	ϕ_G	$G \cup \{\texttt{var} \rightarrow dep(G, \langle expr\rangle^R)\}$
	ϕ_L	L
$inst_1; inst_2$	ϕ_G	$\phi_G(\phi_G(G, L; inst_1), L; inst_2)$
	ϕ_L	$\phi_L(\phi_G(G, L; inst_1), \phi_L(G, L; inst_1); inst_2)$
$var[expr_1] = expr_2$	ϕ_G	$G \cup \{\langle *(var + expr_1)\rangle^W \rightarrow dep(G, \langle expr_2\rangle^R)\}$
	ϕ_L	$L \cup dep(G, \langle expr_1\rangle^R) \cup dep(G, \texttt{var})$
return $expr$	ϕ_G	$G \cup \{\texttt{\textbackslash RET} \rightarrow dep(G, \langle expr\rangle^R)\}$
	ϕ_L	L

Table 2.7: Dependency and leakage semantics for simple instructions

Instruction to analyze		Result
#pragma STA secret var	ϕ_G	$G \cup \{\texttt{var} \rightarrow \texttt{\textbackslash SECRET}\}$
	ϕ_L	L

Table 2.8: Dependency graph at the end of function foo2

Variable (memory location)	Values	Pointed memory locations
a	$\texttt{a}^0$	–
j	$\texttt{a}^0$	–
i	\SECRET	–
b	$\texttt{b}^0$	–
\RET	$\texttt{a}^0$, \SECRET	–

Analyzing branching instructions

If-else branches When an `if-else` branch is encountered, the set of leaking values needs to be updated, and the two branches need to be analyzed. The latter part is however not trivial, as the static analysis program cannot determine which branch will be taken by the program flow during the execution. Instead, both branches need to be taken into account *simultaneously*. There are two natural methods to do so:

- Run the analysis sequentially on one branch and then on all the instructions following the branches. Then do the same with the second branch, and finally merge the analysis state.

- Run the analysis on one branch, then on the other, and merge the analysis state. Then continue the analysis on the instructions following the branches.

The first method approximates more closely how a real program execution takes place. However, when applied on a code comprised of n adjacent `if-else` branches, 2^n code paths need to be considered. In the case of n adjacent `if-else` branches nested p levels deep, this number grows to $(2^p)^n = 2^{pn}$. Conversely, the second method might overestimate dependencies, because in a real program execution, only one of the two branches are executed, not both. While nested `if-else` branches are still expensive to analyze, n adjacent if-else branches can be analyzed in time $2n$ only.

It is not clear how the second method could induce false positives for detecting the leakage of sensitive values, and it is clearly computationally more efficient. That's why we choose the second option to be implemented in STAnalyzer. The corresponding semantics are provided in Table 2.9.

Table 2.9: Dependency and leakage semantics for `if-else` branches

Instruction to analyze		Result
if ($expr$) { $inst$ }	ϕ_G	$G \cup \phi_G(G, L; inst)$
	ϕ_L	$L \cup dep(G, \langle expr \rangle^R) \cup \phi_L(G, L; inst)$
if ($expr$) { $inst_1$ } **else** { $inst_2$ }	ϕ_G	$\phi_G(G, L; inst_1) \cup \phi_G(G, L; inst_2)$
	ϕ_L	$L \cup dep(G, \langle expr \rangle^R) \cup$ $\phi_L(G, L; inst_1) \cup \phi_L(G, L; inst_2)$

Switch-case statements `Switch-case` statements represent the second conditional branching statement available in C. Their analysis mostly resembles that of `if-else` branches, and we choose the same time-precision tradeoff. In detail, each **case** instruction block is analyzed separately, and the analysis state is then merged together before analyzing subsequent instructions.

However, `switch-case` statement bring additional complexity because of the *fallthrough* mechanism. If the control flow reaches the end of a **case**-block without encountering a **break**-statement, then the next `case`-block is executed. This behavior needs to be accounted for, as ignoring it might lead to missed leakages - for instance, one `case`-block might set some variable to a sensitive value, and the next block leaks that variable. Furthermore, some `case`-blocks might always terminate with a **break**-statements, some might never execute them, and some other might terminate with a **break**-statements on some executions, and fallthrough on other. The last case is even more complicated, as a precise analysis would need to take into account that only a subset of the instructions of the `case`-block might be performed before the next block is executed. At the risk of introducing some false positives, we simplify the analysis of this third case and consider that all the instructions of a **case**-block are executed before a fallthrough is performed. The semantics of the `switch-case` statement however still depends on whether a **break**-statement will always, never or sometimes be encountered during the execution of a **case**-block.

Additionally, a `switch-case` statement may have a **default** label, which will define the instructions to execute if none of the **case** labels match. However, the **default** label is not mandatory. If it is omitted an no label matches the value of *expr*, then no **case**-block is executed.

In general, for a `switch-case` statement of the form

$$\begin{array}{l}\textbf{switch}(\mathit{expr})\ \{\\ \quad \textbf{case}\ \mathit{expr}_1:\\ \qquad \mathit{inst}_1\\ \qquad \ldots\\ \quad \textbf{case}\ \mathit{expr}_i:\\ \qquad \mathit{inst}_i\\ \}\end{array}$$

the semantics of that statement are defined by

$$\phi_G(G, L; \mathit{inst}) = \begin{cases} \bigcup_i \phi_G^i(G, L) & \text{if a } \textbf{default} \text{ label is present} \\ G \cup \bigcup_i \phi_G^i(G, L) & \text{if no } \textbf{default} \text{ label is present} \end{cases}$$

$$\phi_L(G, L; \mathit{inst}) = L \cup \mathit{dep}(G, \langle \mathit{expr} \rangle^R) \cup \bigcup_i \phi_L^i(G, L)$$

where ϕ_G^i and ϕ_L^i are given in Table 2.10.

Table 2.10: Dependency and leakage semantics for `switch-case` statements

Break encountered in inst_i		**Result**
Always	ϕ_G^i	$\phi_G(G, L; \mathit{inst}_i)$
	ϕ_L^i	$\mathit{dep}(G, \langle \mathit{expr}_i \rangle^R) \cup \phi_L(G, L; \mathit{inst}_i)$
Sometimes	ϕ_G^i	$\phi_G(G, L; \mathit{inst}_i) \cup \phi_G^{i+1}(\phi_G(G, L; \mathit{inst}_i), L)$
	ϕ_L^i	$\mathit{dep}(G, \langle \mathit{expr}_i \rangle^R) \cup \phi_L(G, L) \cup \phi_L^{i+1}(\phi_G^i(G, L), L)$
Never	ϕ_G^i	$\phi_G^{i+1}(\phi_G(G, L; \mathit{inst}_i), L)$
	ϕ_L^i	$\mathit{dep}(G, \langle \mathit{expr}_i \rangle^R) \cup \phi_L(G, L; \mathit{inst}_i) \cup \phi_L^{i+1}(\phi_G^i(G, L), L)$

Analyzing loops

Analyzing requires computing a fixed point in the dependency and leakage analysis. Because our program does not perform any kind of value analysis, it cannot determine

how many times a loop will be executed. Therefore, the worst case situation, leading to the most dependencies and leakages, must be determined. Let's consider a **while**-loop of the type **while** ($expr$) { $inst$ } and define the function ψ as

$$\psi(G, L; expr; inst) = (G \cup \phi_G(G, L; inst), L \cup \phi_L(G, L; inst) \cup dep(G, \langle expr \rangle^R))$$

The function ψ computes the updated dependencies and leakages after one execution of the loop. Let $n \geq 0$, the dependencies and leakages after n loop executions are defined as

$$\begin{aligned} \psi^n(G, L; expr; inst) &= \psi(G', L'; expr; inst) \text{ where} \\ &\qquad (G,' L') = \psi^{n-1}(G, L; expr; inst), \\ \psi^0(G, L; expr; inst) &= (G, L). \end{aligned}$$

Per its definition, the computed dependencies and leakages cannot decrease after applying ψ and are of course upper-bounded. This guarantees that ψ has a fixed point. Therefore, the resulting dependency graph and leakages after analyzing this instruction are equal to

$$\phi(G, L; \textbf{while } (expr) \{ \ inst \ \}) = \lim_{n\to\infty} \psi^n(G, L; expr; inst).$$

Other Loops **do**−**while** loops are similar to **while**-loops, except that the instructions in the loop are executed *at least once*. Similarly, **for**-loops execute a provided *init* statement before first executing the loop instructions, and a *step* instruction after the loop body. Table 2.11 summarizes the dependency graph and leakage set pair after analyzing a given type of loop.

Table 2.11: Dependency and leakage semantics for loops

Instruction	Result
while ($expr$) { $inst$ }	$\lim_{n\to\infty} \psi^n(G, L; \{expr; inst\})$
do { $inst$ } **while** ($expr$);	$\lim_{n\to\infty} \psi^n \left(\phi_G(G, L; inst), L; \{expr; inst\}\right)$
for ($init$; $cond$; $step$) {$inst$}	$\lim_{n\to\infty} \psi^n(G', L'; \{cond; inst; step\})$ where $(G', L') = \phi(G, L; init)$

Overestimating Dependencies The fixed point calculation might lead to false positive in some cases. Consider for instance the C code in Figure 2.6.

Depending on the parity of the variable `c`, the function `bar` returns either `a` or `b`. Because the algorithm cannot determine the parity of `c`, the analysis will conclude that the return value of `bar` depends on `a` and `b` (as well as, of course, `c`).

```
1 int bar(int a, int b, int c) {
2   int i = a, j = b, k = 0;
3   while (c > 0) {
4     k = i; i = j; j = k;
5     c--;
6   }
7   return i;
8 }
```

Figure 2.6: Example C code with variable swapping

Function Calls

Function calls are more complex to handle. Indeed, a function call can modify any global variable and modify the values of the arguments it is called with (in case pointers are passed as arguments). The value returned by the function can also depend on global variables and any function arguments. Finally, any argument or global variable might be leaked. Because the values corresponding to function arguments and global variables are tracked, our algorithm is able to determine their behavior after finishing analyzing any function. When an already analyzed function is called, it is thus only necessary to perform a translation from the called function argument names to the callee expressions in order to determine the effect of the function call on G and L.

In the rest of this section, we consider an expression of the form

$$f(expr_1, expr_2, \ldots, expr_n)$$

where f has been previously declared as

$$f(\&^{l_1} arg_1, \&^{l_2} arg_2, \ldots, \&^{l_n} arg_n).$$

Here, l_i represents the indirection level of argument arg_i.

Definition 10 (Variable translation)**.** Let $I_f = \{(i, j) : i \in [|1; n|] \wedge j \in [|0; l_i|]\}$. For $(i, j) \in I_f$, the *translation function* τ_f is defined on memory addresses as

$$\tau_f(\&^j arg_i) = \langle *^{l_i - j} expr_i \rangle^R$$

and on values as

$$\tau_f(arg_i^0) = \langle *^{l_i} expr_i \rangle^R.$$

The special `\SECRET` value and all global variables are translated as themselves.

The definition of τ_f is naturally extended to a set of values and memory addresses related to the arguments of f as

$$\tau_f\,(S) = \bigcup_{\texttt{varname} \in S} \tau_f(\texttt{varname}).$$

Table 2.12: Dependency and leakage semantics for function calls

Expression	Result
$\phi_L(G, L; f(\ldots))$	$L \cup dep(G, \tau_f(L_f))$
$\phi_G(G, L; f(\ldots))$	$\left[\bigcup_{(i,j>0)\in I_f} \left\{\langle *^j\, expr_i\rangle^W \rightarrow dep(G, \tau_f(dep(G_f, \&^{l_i-j}\, arg_i)))\right\}\right] G$
$\langle f(\ldots)\rangle$	$\tau_f(dep(G_f, \texttt{\\RET}))$

The semantics of function calls can then be defined with the help of τ_f as in Table 2.12. In this table, $f(\ldots)$ is used as shorthand for $f(expr_1, expr_2, \ldots, expr_n)$. Furthermore, L_f and G_f represent respectively the the set of leaking values and the dependency graph computed after analyzing f.

Variadic functions, that is, functions with a variable number of arguments, cannot be analyzed by our tool. However, we implemented a mechanism that allows the user to provide snippets of Python code to describe the behavior of variadic functions that might be encountered during analysis. Finally, function pointers can be handled, if it is clear from the analysis which function they point to. Since it is not easily possible to determine this for function pointers that are passed as function arguments, they are always considered as leaking all their arguments. This minimizes the risk of missing any source of leakage.

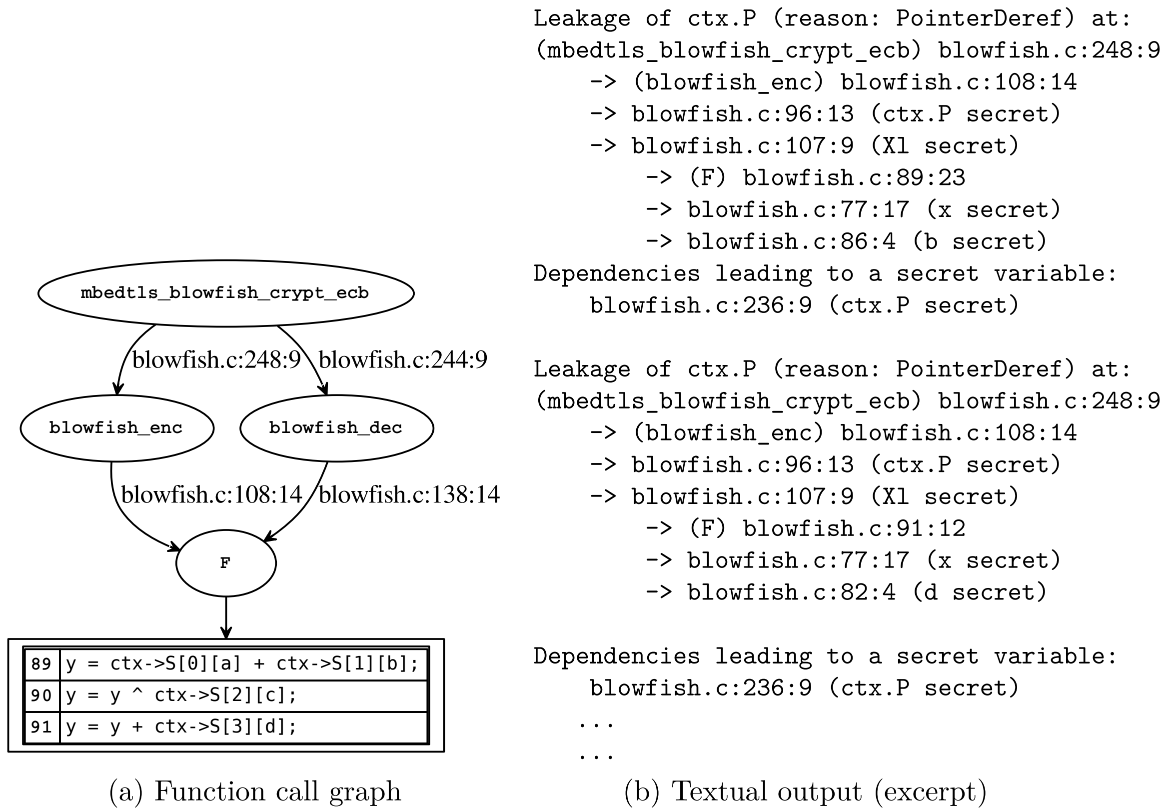
2.3.5 Output Presentation

The aim of our tool is to detect all possible sources of leakages and present them in a way that is easy to understand. To this purpose, two types of outputs are generated after each analysis. The first one is a textual file. For each detected leakage, it describes the function calls, dependencies between variables, and the instruction causing the leakage. The second is an overview of the function call graph leading to the leaking instructions.

In this example, the function `mbedtls_blowfish_crypt_ecb` was chosen as the entry point for our analysis. The variable `ctx.P` was flagged as sensitive, as it contains the Blowfish encryption key. The structure `ctx` is passed to the function `blowfish_enc`. The dependency analysis of that function determined that the variable `Xl` depends on the value of `ctx.P`. The function `F` is then called, with `Xl` corresponding to its `x` argument. Inside this function, `b` depends on the value of `x`, and this variable leaks on line 86. This corresponds to the first reported leak. The second one is very similar, but this time the leakage of the `d`, which also depends on the value of `x`, is reported.

2.3.6 Limitations

Our tool handles memory separation only between members of a structure, not between elements of an array. Consider for instance the code in Figure 2.8. The element at index 1 of the variable `values` is considered sensitive, however, only elements stored at indexes that are an *even* number leak on Line 8-11. Therefore, this code will not leak sensitive data, but because all elements of an array share the same dependencies during our analysis, a leakage will be reported by our tool, resulting in a false positive.



(a) Function call graph (b) Textual output (excerpt)

Figure 2.7: Output from STAnalyzer on mbedtls' Blowfish implementation

```
1  int bar(void) {
2    int values[] = {1,2,3};
3    int j;
4    #pragma STA secret j
5    values[1] = j;
6    for(int i = 0; i < 3; i++)
7    {
8      if (i%2 == 0){
9        // perform some operation
10       // that leaks values[i]
11     }
12   }
13 }
```

Figure 2.8: Example code falsely reported as leaking

2.4 Results

2.4.1 Benchmarks

In order to compare the performance of our approach to existing tools, we ran an analysis of most algorithms mentioned in [57]. The results are given in Table 2.13 and were obtained

on a 2.4 GHz Intel i5 processor.

Implementation	Execution time (s)
curve25519-donna	5.58
nacl_ed25519	7.39
nacl_salsa_20	1.05
nacl_sha512	3.44
mee-cbc	7.03
mbedtls_aes	46.50
mbedtls_sha256	2.87
mbedtls_blowfish	1.16
mbedtls_sha512	2.39
mbedtls_des	12.42
rlwe_sample	4.40

Table 2.13: Verification times for cryptographic algorithms

Violations of the constant-time policy were found for the mbedtls implementations, and the constant-time property was verified for all other implementations. Because our tool lists all constant-time violations, it takes longer to analyze implementations that contain a lot of them. This is for instance the case for `mbedtls_aes` and `mbedtls_des`.

2.4.2 Applications on Other Cryptographic Algorithms

This tool was used in order to detect cache-timing vulnerabilities in the candidates for post-quantum cryptography standardization. A detailed description of this analysis is provided in Section 3.2 and Section 3.3 of the next chapter.

2.5 Conclusion

Folklore has it that you should never implement your own cryptographic algorithms. In addition to the traditional bugs that you could introduce into your software, there are several pitfalls one has to be aware of that are specific to cryptographic implementations, such as side-channel attacks. However, *someone* does need to implement these algorithms, and easy-to-use verification tools that can help to avoid those pitfalls can improve the confidence in your implementation. Thankfully, the rules for avoiding cache-timing leakages are relatively simple—never use sensitive data in a branch or for a memory access—and can be checked through static analysis. Of course, it is impossible to verify this property without introducing false positives. In designing our tool, we have tried to balance the tradeoff between computationally complexity, ease-of-use and false positives, while ensuring that no constant-time violation will be missed. The use cases on real-world code presented in the next chapter show that the tool is useful both to find constant-time violations and to verify their absence.

There a several directions for further improvements. First, additional sources of timing variations could be taken into account, such as divisions, rotations or multiplications taking a variable number of cycles to execute on some platforms [64]. Checking whether these arithmetic operations are performed with sensitive data can be easily done by our too, but this might lead to more reported leakages that are not necessarily exploitable.

In a similar fashion, there is currently no way to rank the severeness of the different constant-time violations that were detected. This could be useful to prioritize the correction of leakages, and could also help to filter out leakages that can be proven to be not exploitable. How to determine useful heuristics for the leakage severity remains an open research direction.

Chapter 3

Applications of the Static Analysis Techniques to Post-Quantum Cryptography and More

Bruce Schneier doesn't use a keylogger. He's standing right behind you.

https://www.schneierfacts.com/

In this chapter, we present applications of our static analyzer tool. The goal is to show that it can be applied on real-world cryptographic code without producing too many false positives and without missing real leakages. Post-quantum algorithms provided an interesting target for this study. Several new algorithms or variants have been designed for the ongoing NIST post-quantum standardization process, and their implementations have not yet been vetted as much as the implementations of more traditional algorithms such as RSA or elliptic curve based cryptography. Furthermore, the implementations are mostly self-contained, facilitating the analysis compared to whole cryptographic libraries .

This chapter is organized as follows. First, in Section 3.1 ground truth is established by comparing the leakages reported by our tool on vulnerable programs to published sources of cache-timing leakages. Section 3.2 is a summary of the findings published in [65] that analyzed the first-round candidates of the NIST post-quantum standardization process. Finally, a more detailed analysis of the second round candidates is provided in Section 3.3.

3.1 Collecting Ground Truth: Rediscovery of Known CVEs

In order to validate the approach of using static analysis techniques to discover cache-timing leakages, we ran the STAnalysis tool on known vulnerable code. The vulnerabilities were chosen from the Common Vulnerabilities and Exposures list `https://cve.mitre.org/`. More precisely, we explored the entries containing the keywords “side-channel”, ignoring the CVEs concerning non cache-timing side channels, implementations in languages other

than C, or vulnerabilities concerning codebases deemed too large for the validation (e.g. web browsers, the Linux kernel, etc.).

3.1.1 CVE-2019-9494 and CVE-2019-9495

These two vulnerabilities concern hostapd version 2.7, a program running on Wifi access points. The vulnerabilities are found in the code that derivates a secret key from the user-provided password. This code did not run in constant time, which could allow an attacker to speed up dictionary attacks against the access point's password.

For our analysis, we looked at the functions `sae_derive_pwe_ecc`, `sae_derive_pwe_ffc` and `compute_password_element`. These functions implement the password derivation and are therefore natural entry points for our analysis. For the ground truth, we looked at the published patches[1] and, by manual inspection, determined which branches and timing-dependent function calls were removed because they could cause timing differences.

True positives First, the patches replaced three calls to the GMP-function `BN_mod_exp` with calls to `BN_mod_exp_consttime`. Two such leaking calls were caught by our tool: those at lines 1298 and 1442 of the file `crypto/crypto_openssl.c`. The third one, at line 552 of the same file, was not found, because it was not called, either directly or indirectly, by the three functions we chose as entry points, and was therefore outside the code analyzed in this experiment.

Second, a call to the (potentially) variable-time `os_memcp` function is replaced by a call with the constant-time `const_time_memcmp`. This leaking function call was detected by our tool.

Third, a number of early returns in various functions were removed, in order to remove timing differences. These are found in `crypto/crypto_openssl.c` at lines 1446 and 1448, in `eap_common/eap_pwd_common.c` at lines 263 and 278, and finally in file `common/sae.c` at line 513 and 237. While performing these corrections, a boolean assignment was also changed in `common/sae.c` at line 260 to guarantee the absence of branching after compiling, as shown in Listing 3.1. All the early returns were detected by our tool, but not the boolean assignment. Such constructions could be reported, at the risk however of increasing the number of false positives.

```
1 res = res == check;
2 // Changed to
3 // mask = const_time_eq(res, check);
4 // res = const_time_select_int(mask, 1, 0);
```

Listing 3.1: Boolean Assignment in hostapd

Finally, some loops that could run a variable number of times have been changes to a fixed number of iterations. These are found in `common/sae.c` at line 602 and 619, and were detected by our analysis tool.

[1] https://w1.fi/security/2019-1/ and https://w1.fi/security/2019-2/

False positives False positives were present in the analysis, and they can be classified into two main categories.

About half of the false positives are caused by the functions `get_random_qr_qnr` and `get_rand_1_to_p_1`. They implement randomness generation in a rejection sampling-like fashion. Although the code branches on randomly generated sensitive values, the sampled value is independent from the direction of the branches taken. Similar false positives are also caused by the function `compute_password_element` while sampling quadratic residues and non-residues.

The second half of the false positives is caused by checking for error conditions - such as failing to allocate memory, or very rare cases where generating random elements might fail. For example, a function that should return a boolean value depending on its (potentially sensitive) inputs, might also return -1 in order to signal a fatal error, which should happen very rarely in practice. Checking this error code would trigger a leakage warning from our tool, although no real leakage is present.

As a summary, about one third of the detected leakages are indeed security-relevant cache-timing leakages, one third are caused by rejection sampling, and one third by checks for error conditions. These false positives do fulfill the conditions of potentially vulnerable code, but are not exploitable. As such, they are an artifact of the general approach and not of the implementation of the static analysis. In order to avoid these kinds of false positives, one could resort to heuristics to filter them out, at the risk of missing real cache-timing leakages, or rely on annotations provided by the developers of the library.

3.1.2 CVE-2018-124049

This vulnerability affected Mozilla's NSS cryptographic library. Variable-timing code exposed the RSA decryption function to a Bleichenbacher-type attack [66] that could be exploited using a cache-timing attack. Several other cryptographic libraries were affected by similar weaknesses, but in this section, we will only focus on the NSS library.

True positives Two functions contain vulnerable code that can be exploited to mount the attack. The first is `mp_to_fixlen_octets` which adds leading zeros to the representation of a large integer. This is not done in constant-time, and this function is applied to the decrypted plaintext. The second is `RSA_DecryptBlock`, which checks whether the decrypted plaintext has been padded correctly according to the PCKS #1 v1.5 standard. Different code is performed for conforming and non-conforming code, which allows an attacker access to different kind of oracles that can be used to decrypt RSA-encrypted ciphertexts.

The leakages in both functions were detected by our tool.

False negatives Numerous other potential leakages were reported by our tool - more than 5000 in total. Some are caused by leakages of randomly generated blinding values, which could potentially lead to multivariate attacks targeting the blinding and blinded values simultaneously. However, no such cache-timing attacks have yet been reported,

and the feasibility of such an attack is unclear. Indeed, it is not possible to reduce the measurement noise related to the blinding values, as they change on each execution.

Most of the other reported leakages concern functions of the big integer library used by NSS. For instance, the code often branches depending on the sign of the operands. This behavior may or may not lead to a cache-timing attack.

Finally, the error code returned by most functions is always checked. While this is good practice in general, it can also lead to numerous false positives. For example, a function could return a non-zero return code (indicating that an error has occurred) if the values supplied as arguments are outside a valid range. If this function is called with sensitive values that will *always* be in this valid range (by construction or because the range is verified beforehand), a potential leakage will be reported by our tool although none can happen in practice. However, one cannot simply ignore branching on *all* function error codes, as they could in other cases cause *real* leakages. More advanced tools can infer the range for the values of certain variables, such as [57]. It has however not yet been demonstrated that this functionality is applicable to large integers handled by libraries such as NSS.

In this example, the vast majority of the potential leakages discovered by our tool were irrelevant to the reported vulnerability. While some of these leakages might be problematic, the vast majority seem to be false positives. Reducing their number would make our tool more useful for analyzing complex libraries. On the other hand, the false positives do not seem to stem from programming errors in our tool, but rather from the type of code found in NSS. As such, this library does not seem to be suitable for static analysis in general, at least not in order to verify the constant-time property of the code.

3.2 Analyzing First Round Post-Quantum Cryptography Candidates

The emergence of quantum computers in the medium term poses a particular threat to public key cryptography. Shor's algorithm [67] promises an exponential speedup for operations such as integer factorization and discrete logarithm, once quantum computers with a sufficient number of quantum bits are available. This would undermine the security of most digital signature, public key encryption and key establishment algorithms deployed today. Private key primitives, such as symmetric encryption or hash functions, would also be affected by quantum computers, due to the Grover search algorithm [68]. However, the speedup provided by this quantum search algorithm is only *quadratic* and can therefore be offset by simpling doubling the key sizes.

It is thus crucial for public key algorithms resisting attacks by quantum computers to be designed, implemented, tested and ultimately standardized in the near future. These types of algorithms are collectively known as "post-quantum" algorithms, and aim at replacing RSA and elliptic curve-based primitives. They are thus based on other hard problems. The most promising candidates problems are:

1. LWE (Learning with Errors) [69], which gives rise to the family of lattice-based

proposals;

2. the decoding problem, used in the code-based family of proposals, initiated by the McEliece cryptosystem [70] and

3. multivariate polynomial systems solving [71], which is the cornerstone of the multivariate cryptography family.

For signatures, systems inspired by the Merkle signature scheme [72] are also proposed [73] while other algorithms are based on more exotics constructions such as supersingular elliptic curve isogenies [74].

In 2016, the Nation Institute of Standards and Technology (NIST) started the standardization process for post-quantum cryptography algorithms. This process is expected to undergo three rounds, the first one which started in 2017 with the publication of the specifications of 69 algorithms, along with implementations for every algorithm. The main focus of the first round was to eliminate broken schemes, and the implementation security was not yet of paramount importance. Nevertheless, we analyzed most of the submitted candidates in order to determine which kind of side-channel leaks post-quantum algorithms might be subject to .

3.2.1 Analysis Methodology

The submissions to the NIST contest have mostly similar structures, owing to predefined templates for implementation. In particular, they all need to implement a `main` function which creates and verifies Known Answer Tests (KATs). This function thus performs either a signature and verification, or an encryption and decryption, with randomly generated key pairs. Executing this function therefore provides a relatively good code coverage. For the tagged variables, we simply choose to taint the randomness sources (which are also required to respect some specific format), as these sources are used to generate the secret keys as well as the randomness used during signature or encryption, which is probably sensitive in most cases. An issue is that the public key would also be considered sensitive. We mitigate this issue by automatically declaring safe any variable called `pk`. Of course, a more individual analysis would be necessary if one wants to guarantee the constant time property of a given implementation or, on the contrary, build a side-channel attack. However, this first analysis allows us to gain first insights about the existing and probable future difficulties of implementing constant time post-quantum schemes.

3.2.2 Result Overview

We present on Figure 3.1 the results for the 52 submissions we were able to perform static analysis on. Out of these 52 candidates, vulnerabilities were found in **42 of them (80.8%)**. 17 candidates have more than 100 vulnerabilities reported and 3 have more than 1000 vulnerabilities reported. Ten submissions (Frodo, Rainbow, Hila5, Saber, CRYSTALS-Kyber, LOTUS, NewHope, ntruprime, ThreeBears and Titanium) were found to be correctly protected. A few submissions were almost perfectly constant time, and replacing, for example, a small number of conditional branches with conditional move operations

would render these implementations perfectly constant-time (EMBLEM, Lima, Giophantus, OKCN-AKCN in the MLWE variant). Also, false positives might have occurred, as it was not possible, given the number of submissions and reported vulnerabilities, to manually remove all false positives. The vast majority of the submissions, however, is not correctly protected against cache attacks, due to recurrent programming oversights in the portions of code handling sensitive data.

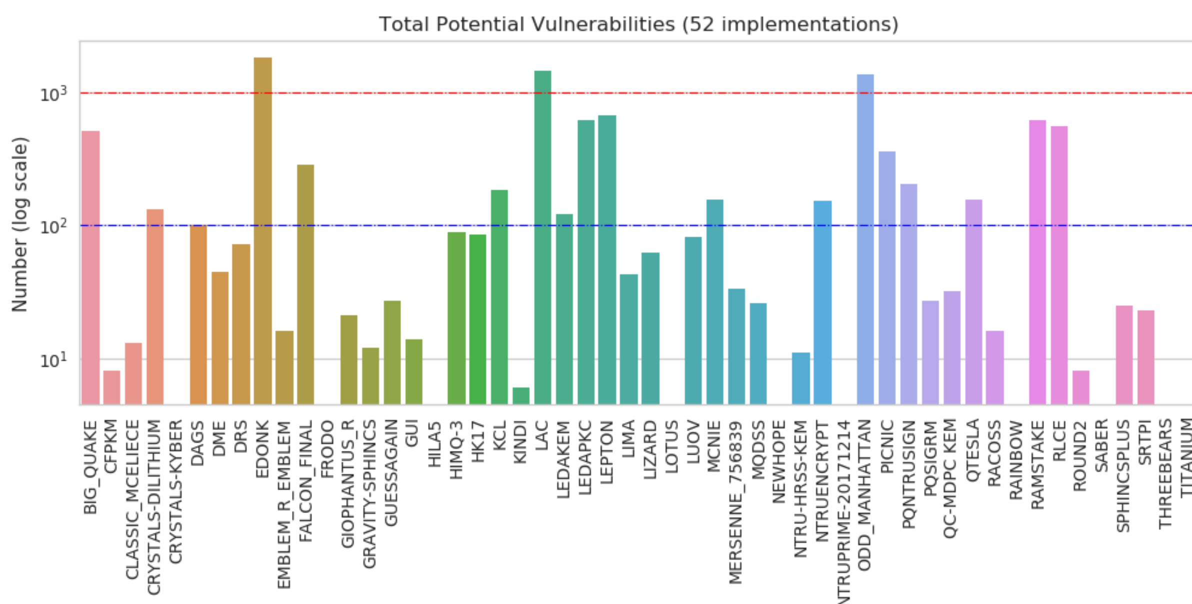


Figure 3.1: Total number of potential vulnerabilities found for each analyzed candidate

3.2.3 Analysis of Vulnerabilities

The detailed reports provided by the tool allowed us to classify the vulnerabilities into several categories, which seem to occur frequently among the submissions:

1. **Gaussian sampling leaks:** similar to the issues reported with BLISS [24], implementing a side-channel leakage free Gaussian sampler is not trivial to achieve. Some proposals, such as Frodo, manage to avoid this issue by slightly modifying the distribution being sampled and implementing this sampling in a constant-time fashion. If the use of a discrete Gaussian sampler is required, constant time implementations [75] should be used.

2. **Other sampling leaks:** in general, when specific distributions need to be sampled, one has to take care to avoid conditional branches and array accesses that depend on the randomness source being used. There is no general solution to these type of leaks, as every submission implements samplers for potentially different kinds of distributions.

3. **GMP library use:** some submissions use GMP. This library does not seem to implement operations in constant time (at least, according to the C code implementing

the version we used for analysis, which is the latest version, at the time of writing, of the `mini-gmp` variant), and thus functions defined by this library should not be used on sensitive data without further inspection of their assembly level implementation. In any case, the portable `mini-gmp` implementation should probably not be used in implementations desiring to achieve the constant-time property.

4. **Operations in finite fields:** several implementations need to handle operations in finite fields, notably multiplications. For small groups, this is often done via log/anti-log tables, which might open these implementations to cache-attacks. Other ways to implement these operations must be considered, such as bit-slicing [76].

5. **Other:** there were some potential leaks that we were not able to fit into one of the other categories. For instance, some submissions provide their own implementation of AES that use table accesses, or perform data-dependent branching for matrix operations such as Gaussian elimination. Custom implementations of AES will probably be replaced with calls to hardware-optimized instructions (such as AES-NI [77]) on the platforms most commonly targeted by cache-timing attacks.

Table 3.1: Breakdown of vulnerabilities per type.

Vulnerability Type	Occurrences
Gaussian sampling	3
Other sampling	13
Use of GMP	4
Unsecure GF operations	12
Other leakage sources	31

As shown in Table 3.1, vulnerable code appears relatively often when implementing finite field operations and sampling specific distributions.

Code-based schemes are mainly affected by leakages due to finite field operations, as they need to perform multiplications in extensions of GF(2). Additions in these groups are less likely to cause side-channel leakages, as they can be implemented using constant time bit shifts and XOR-operations. However a naïve implementation might be vulnerable, as the conditional XOR, used during reduction, must also be executed in constant time, for example by using a conditional move (`cmove`) instruction.

On the other hand, vulnerabilities in randomness sampling are mainly encountered among lattice-based proposals. Perhaps surprisingly, vulnerabilities in Gaussian samplers are relatively rare. The publication of attacks against BLISS [24] might have forced some candidates to take special care in implementing the Gaussian sampler (DRS, qTesla, although the implementation is not entirely constant-time), while others avoided discrete Gaussian distributions altogether (CRYSTALS-Dilithium). However, other sampling routines were not sufficiently protected, for example Bernoulli sampling.

3.3 Analyzing Second Round Post-Quantum Cryptography Candidates

In January 2019, the NIST published a list of 17 candidates for public-key cryptography and key-establishment (which will be referred to as KEM algorithms) and 9 candidates for digital signatures that move to the second round of the post-quantum standardization process [78]. Building on the insights gained during the analysis of first round candidates and improvements on our static analysis tool, we present here a more detailed analysis of 21 post-quantum cryptography algorithms. In this analysis, we strive to clearly present all potential causes of leakages found by our tool, as well as summarize the causes of false positives. This can help to design improved analysis tools that can avoid the false positives, and highlight where manual verification or code annotation is needed for verifying the constant-time property of the implementations.

Compared with the analysis of the first round candidates, we found less vulnerabilities. This is mainly due to two reasons: several submissions were updated with constant-time implementations, and the manual inspection removed reported false positives.

Unless stated otherwise, the analyzed implementations are those submitted to NIST at the beginning of the second round of the post-quantum standardization process. For some proposals, updated implementations that specifically address cache-timing leakages were made available (notably qTesla, Falcon and LAC), and those updated implementations were analyzed instead.

3.3.1 LUOV

LUOV (the Lifted Unbalanced Oil and Vinegar signature scheme) is a post-quantum, multivariate quadratic signature scheme. As such, both signature verification and generation need to perform additions and multiplications in different finite fields, of the form $\mathbb{F}_{2^r}$ - $\mathbb{F}_{2^8}$, $\mathbb{F}_{2^{16}}$, $\mathbb{F}_{2^{32}}$, $\mathbb{F}_{2^{64}}$ and $\mathbb{F}_{2^{80}}$ for the first version, $\mathbb{F}_{2^7}, \mathbb{F}_{2^{47}}, \mathbb{F}_{2^{61}}$ and $\mathbb{F}_{2^{79}}$ for an updated version that addresses weaknesses in the original parameters choices that were discovered during the standardization process [79].

Finite Field multiplication leakage In the optimized as well as the reference implementations, finite field multiplications are **not constant time**. In the older version, multiplications in $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{16}}$ are implemented via log and antilog tables, as shown in Listing 3.2. Multiplications in the larger groups are implemented using multiplications over $\mathbb{F}_{2^{16}}$, and are therefore also vulnerable.

In the newer version, multiplications in $\mathbb{F}_{2^7}$ are also implemented using log and antilog tables, in a similar fashion as Listing 3.2. The larger groups implements carry-less multiplication, as shown in Listing 3.3, which uses table accesses and is potentially vulnerable to cache-timing attacks(see Line 11).

In addition to the reference and optimized implementations, an AVX2 optimized implementation is provided, but only for the variants of LUOV using the smallest group size: $\mathbb{F}_{2^8}$ for the original version, and $\mathbb{F}_{2^7}$ for the updated version. In this version, finite field multiplication is implemented using iterative multiplication with unconditional reduction.

```
1  f8FELT f8multiply(f8FELT a, f8FELT b) {
2    if (a == 0 || b == 0) return 0;
3    return f8antilog((f8log(a) + f8log(b)) % f8units);
4  }
5
6  uint8_t f8log(f8FELT a) {
7    static const uint8_t f8LogTable[256] = {...};
8    return f8LogTable[a];
9  }
10
11 uint8_t f8antilog(uint8_t a) {
12   static const uint8_t f8AntiLogTable[256] = {...};
13   return f8AntiLogTable[a];
14 }
```

Listing 3.2: Implementation of multiplication in $\mathbb{F}_{2^8}$ in LUOV

```
1  uint64_t clmul(uint64_t a, uint64_t b){
2    uint64_t table[16] = {0};
3    table[1] = a;
4    // ...
5    uint64_t b_lower =         b & 0x0f0f0f0f0f0f;
6    uint64_t b_upper = (b >> 4) & 0x0f0f0f0f0f0f;
7
8    unsigned char *b_lower_nibbles = (unsigned char *) &b_lower;
9    unsigned char *b_upper_nibbles = (unsigned char *) &b_upper;
10
11   out_upper = table[b_upper_nibbles[3]];
12   out_lower = table[b_lower_nibbles[3]];
13   //...
14   out = (out_upper << 4) ^ out_lower;
15   return out;
16 }
```

Listing 3.3: Carryless multiplication in LUOV

As shown in [80], this routine could be sped up using Intel's `clmul` instructions and constant-time conditional reduction using `cmov`. In any case, finite field operations in the AVX2 implementation are **constant-time**. However, no constant-time implementation for the other parameter choices is provided.

Other Leaking Instructions After the solution to the multivariate quadratic system has been computed, the solution needs to be transformed, and this transformation is basically a multiplication by the secret linear map `T`. In the reference and optimized implementations, both in the original and updated version, the code performing this operation is shown in Listing 3.4. The code branches on the secret map `T` on Line 4, which might induce a timing vulnerability. The AVX2 optimized code performs the multiplication in a different way and is in fact constant time.

In a similar fashion, on the reference implementation, when building the augmented

```
1 // Convert into a solution for P(x) = target
2 for (i = 0; i <= VINEGAR_VARS; i++) {
3   for (j = 0; j < OIL_VARS; j++) {
4     if (getBit(T[i], j)) {
5       solution[i] = subtract(solution[i],solution[VINEGAR_VARS+1+j]);
6     }
7   }
8 }
```

Listing 3.4: Multiplication by T

matrix representing the multivariate system, another matrix multiplication is implemented in the same way and might potentially leak, as shown in Listing 3.5 (here, `F2` depends on the secret map `T`). The optimized implementation does not contain this branching instruction. It uses strided matrix multiplication instead. However, this code in turn performs array accesses in order to perform the two following functions:

$$b = \sum_{i=0}^{8} b_i \cdot 2^i \in \mathbb{F}_{2^8} \mapsto 255 \cdot \sum_{i=0}^{8} b_i \cdot 256^i$$
$$b \in \mathbb{F}_{2^8} \mapsto 255 \cdot \sum_{i=0}^{8} b \cdot 256^i$$

and is therefore not constant time.

```
1 for (i = 0; i <= VINEGAR_VARS; i++) {
2   for (j = 0; j < OIL_VARS; j++)    {
3     if (getBit(F2[k][i],j)) {
4       A.array[16*x+k][j] = add(A.array[16*x+k][j],vinegar_variables[i]);
5     }
6   }
7 }
```

Listing 3.5: Multiplication by F2

3.3.2 Round5

Round5 is a post-quantum lattice-based KEM scheme. It has the particularity of using constant-time error correction, which prevents timing attacks that would be possible if variable-time decoders were used (e.g. BCH codes). Despite some warnings emitted by our analysis tool, the provided (optimized) implementation is actually **constant-time**. In the rest of this section, we will explain which warnings were emitted, as a way to explain how false positives can emerge.

Random permutation generation The first warning concerns the generation of a random permutation. This is a countermeasure against backdoor attacks [81, §2.7.4].

However, because this random permutation is only applied to a *public* matrix in order to obtain an other *public* matrix, retrieving this permutation (or an equivalent one) is actually possible using public data only, and a side-channel attack would not help a potential attacker. This is therefore a **false positive**. For completeness, the code used to generate this permutation is provided in Listing 3.6.

```
1 for (i = 0; i < params->k; ++i) {
2   do {
3     rnd = drbg_sampler16_2(params->tau2_len);
4   } while (v[rnd]);
5   v[rnd] = 1;
6   row_disp[i] = rnd;
7 }
```

Listing 3.6: Permutation generation

Ternary secret generation The second warning also concerns a randomness generation routine: the generation of a sparse ternary random vector, in order to generate a random matrix. The code that generates that vector in the reference implementation is given in Listing 3.7. Note that the implementation uses rejection sampling (Line 4), which does not fit the exact definition of constant-time but does not, in fact, cause any issue. However, the creation of the ternary vector makes use of an array index at a sensitive location on Line 5, which constitutes the potential leakage.

```
1 for (i = 0; i < h; ++i) {
2   do {
3     idx = drbg_sampler16(len);
4   } while (vector[idx] != 0);
5   vector[idx] = (i & 1) ? -1 : 1;
6 }
```

Listing 3.7: Sparse ternary vector generation

The optimized implementation, when choosing to use the appropriate constant-time matrix operations, is not concerned by this vulnerability, and is indeed **constant-time**.

3.3.3 qTesla

qTesla is a lattice-based post-quantum signature scheme. The signature generation requires to sample from a discrete Gaussian distribution, which is however claimed to be constant-time and thus not vulnerable to attacks such as those targeting BLISS [24, 25] or older versions of Falcon [82]. Both the reference and optimized implementations are claimed to be constant-time, and we will verify that this is the case in this section.

Uniform polynomial generation In addition to the discrete Gaussian sampler, qTesla also implements a uniform sampler in function `sample_y`, in order to generate a polynomial

of N independent coefficients, uniformly distributed in $[-B, B]$. The code that implements this sampler is given in Listing 3.8. This code performs branching on the potentially sensitive values `y_t[0]` to `y_t[4]`. However, what is implemented here is actually rejection sampling, and thus the final coefficients of the polynomial `y` are independent of the branches taken during the execution of this code. Therefore, the uniform polynomial generation is **constant-time**. A very similar implementation is found in function `poly_uniform`, which is **constant-time** for the same reason.

```
1  // initialization: i=0, buf contains random values
2  while (i<PARAM_N) {
3    if (pos >= nblocks*nbytes*4) {
4      cSHAKE((uint8_t*)buf, SHAKE_RATE, dmsp++, seed, CRYPTO_RANDOMBYTES);
5      pos = 0;
6    }
7    y_t[0]  = (*(uint32_t*)(buf+pos)) & ((1<<(PARAM_B_BITS+1))-1);
8    y_t[0] -= PARAM_B;
9    // ... similar for y_t[1], y_t[2], y_t[3]
10   if (y_t[0] != (1<<PARAM_B_BITS))  y[i++] = y_t[0];
11   // ... similar for y_t[1], y_t[2], y_t[3]
12   pos += 4*nbytes;
13 }
```

Listing 3.8: Uniform polynomial generation

Other rejection sampling The main signature code also uses rejection sampling, and branches on the return value of the functions `test_correctness` and `test_rejection`. This does not cause side-channel leakages. `test_rejection` is itself constant-time, while `test_correctness` leaks the position of the invalid coefficient - but per the comments, this does not reveal anything about their values.

Leakage of public output Some functions leak data that depends on the secrets, but is revealed as part of the signature anyways. This is the case for `encode_c`, which leaks the *public* parameter c, as well as `sparse_mul16`, which leaks the positions and signs of the non-zero coefficients of c.

3.3.4 MQDSS

MQDSS (Multivariate Quadratic Digital Signature Scheme) is a post-quantum, multivariate quadratic signature scheme. It implements operations in $\mathbb{F}_{2^{31}}$ in constant-time, for all implementation variants. The rest of the code can also be proven constant-time, with two small exceptions that require manual inspection: rejection sampling is used three times, and public data leaks once. This however does not prevent the analyzed reference and optimized implementations from being considered **constant-time**.

Rejection sampling Rejection sampling is used three times in order to generate random polynomials with coefficients in $\mathbb{F}_{2^{31}}$ in functions `gf31_nrand` and `gf31_nrand_schar`.

The only difference is that `gf31_nrand` generates coefficients in $[0; 30]$ while the coefficients generated by `gf31_nrand_schar` lie in [-15; 15]. The sampling code is provided in Listing 3.9. Finally, the main code in `crypto_sign` implements a similar rejection sampling which we do not reproduce here for brevity.

```
1 while (i < len) {
2   shake256_squeezeblocks(shakeblock, 1, shakestate);
3   for (j = 0; j < SHAKE256_RATE && i < len; j++) {
4     if ((shakeblock[j] & 31) != 31) {
5       out[i] = (shakeblock[j] & 31) /*-15 for gf31_nrand_schar*/;
6       i++;
7     }
8   }
9 }
```

Listing 3.9: Uniform polynomial generation

Leakage of public data As the last step of the signature generation, as per [83, Fig. 7.2], the output of a hash function H_2 is used to determine which commitments c_i^j to write out as part of the signature. The selection code is provided in Listing 3.10 and leaks the output of that hash function, stored in the variable `h1`.

```
1 for (i = 0; i < ROUNDS; i++) {
2   b = (h1[(i >> 3)] >> (i & 7)) & 1;
3   if (b == 0) { gf31_npack(sm, r0+i*N, N); }
4   else if (b == 1) { gf31_npack(sm, r1+i*N, N); }
5   //...
6 }
```

Listing 3.10: Commitment selection

While the value of `h1` (ch_2 in [83, Fig. 7.2]) depends on the secret key sk, it only depends on the *hash* of sk, which is not necessary sensitive. Furthermore, after inspecting the signature generation algorithm, it becomes clear that ch_2 can be computed from public output $\sigma = (R, \sigma_0, \sigma_1, \sigma_2)$, as well as the message M and the public key pk, as

$$\mathsf{ch}_2 = H_2(\mathcal{H}(\mathsf{pk}||R||M), \sigma_0, H_1(\mathcal{H}(\mathsf{pk}||R||M), \sigma_0), \sigma_1)$$

Therefore, the leakage of `h1` constitutes a leakage of *public* data, which is in fact not sensitive. Thus, the provided MQDSS implementations are **constant-time**.

3.3.5 LEDAcrypt

LEDAcrypt is a post-quantum code-based KEM scheme. More specifically, it is based on LDPC (low density parity-check) codes in order to obtain compact key sizes. Our analysis applies to both the reference and optimized implementation provided by the LEDAcrypt implementors. Since we performed the analysis, a new version of the implementation has been made available in March 2020. The updated implementation has not yet been taken into account.

Variable-time decoder The proposed error-decoding algorithm are not constant time. This is the case for both the previously proposed *Q-decoder* [84, §1.5, Alg. 12] as well as the standard bit-flipping decoder referenced by the latest version of the documentation [85, §1.1.1, Alg. 1].

This can be already inferred from the description of the algorithms in the documentation. For [84, §1.5, Alg. 12], on Line 16, *qrow*[`k`] depends on the private matrix `Qtr`, on Line 17, `similarity` depends on *qrow*[`k`] and finally on Line 18, the algorithm branches on the value of `similarity`, which we just showed depends on sensitive values. This leak is also confirmed in the code.

For [85, §1.1.1, Alg. 1], Line 11 branches on the sensitive value `syndrome`, and Line 15 on `unsatParityChecks[i]`, which depends on the sensitive value `syndrome`. Because the code has not been updated to the new specifications yet at the time of the writing, we could not verify that this leakage is also present in the code.

Variable-time finite field operations LEDAcrypt implements various operations on elements in finite fields, notably for elements in extensions of finite fields with a sparse representation in the base field. Several of these operations are in fact variable time.

`left_bit_shift_n` and `left_bit_DIGIT_shift_n` are two primitives used for modular multiplication (function `gf2x_mod_mul_dense_to_sparse`). They shift the representation of an extension field element by n places, implementing the multiplication by X^n. However, both leak the value n: `left_bit_shift_n` takes a shortcut for $n = 0$ (but otherwise performs the shift in constant time), while `left_bit_DIGIT_shift_n` simply iterates a number of times dependent on n.

`gf2x_mod` performs the modulo operation in finite field. A security-relevant part of this function is provided as Listing 3.11. The leaks occurs on Line 4, where the bit-representation of the array `aux` is leaked. This array contains a copy of the input array representing the finite field element for which the modulo operation is to be performed. The function `gf2x_mod` also contains a second, similar block of code, that leaks the values of `aux` for analogous reasons. Therefore, `gf2x_mod` leaks its input parameter.

```
1  for (i = 0; i < nin-NUM_DIGITS_GF2X_MODULUS; i += 1) {
2    for (j = DIGIT_SIZE_b-1; j >= 0; j--) {
3      mask = ((DIGIT)0x1) << j;
4      if (aux[i] & mask) {
5        aux[i] ^= mask;
6        posTrailingBit = (DIGIT_SIZE_b-1-j) + i*DIGIT_SIZE_b + P;
7        maskOffset = DIGIT_SIZE_b-1-(posTrailingBit%DIGIT_SIZE_b);
8        mask = (DIGIT) 0x1 << maskOffset;
9        aux[posTrailingBit/DIGIT_SIZE_b] ^= mask;
10     }
11   }
12 }
```

Listing 3.11: Finite field modulo (excerpt)

`gf2x_mod_mul_sparse` performs the modular multiplication of its two input parameters

A and B in $\mathbb{F}_{2^P}$ for some P, given as the list of the positions of non-zero coefficients of their representative polynomial. Let us denote these positions with as $\{i_1^A, i_2^A, \ldots i_{n_A}^A\}$ and an equivalent notation for B. The multiplication algorithm works as follows:

1. Compute all the sums $i_j^A + i_k^B \mod P$,
2. Sort these values, and
3. Reduce this list by:
   - Deleting the values that appear an *even* number of times
   - Keeping exactly *one* of the values that appear an *odd* number of times

For any implementation, the first step is likely to leak n_A and n_B. The sorting algorithm should be implemented in constant-time in order to avoid leaking the result, and the last step is non-trivial to implement in constant time. The LEDAcrypt implementation performs neither step in constant-time. The code for the first step is given in Listing 3.12 and contains branches on the values of A and B. The sorting algorithm is a non constant-time quicksort. Finally, the third step is not protected, as shown by the code in Listing 3.13.

```
1  for(int i = 0 ; i < sizeA ; i++){
2    for(int j = 0 ; j < sizeB ; j++){
3      uint32_t prod = ((uint32_t) A[i]) + ((uint32_t) B[j]);
4      prod = ( (prod >= P) ? prod - P : prod);
5      if ((A[i] != INVALID_POS_VALUE) && (B[j] != INVALID_POS_VALUE)){
6        Res[lastFilledPos] = prod;
7      } else {
8        Res[lastFilledPos] = INVALID_POS_VALUE;
9      }
10     lastFilledPos++;
11   }
12 }
```

Listing 3.12: Finite field multiplication (step 1)

```
1 while(read_idx < sizeR  && Res[read_idx] != INVALID_POS_VALUE){
2   lastReadPos = Res[read_idx++];
3   duplicateCount=1;
4   while( (Res[read_idx] == lastReadPos) && (Res[read_idx] != ↩
       INVALID_POS_VALUE)){
5     read_idx++;
6     duplicateCount++;
7   }
8   if (duplicateCount % 2) { Res[write_idx++] = lastReadPos; }
9 }
```

Listing 3.13: Finite field modulo (step 3)

gf2x_mod_add_sparse implements sparse addition in $\mathbb{F}_{2^P}$. The function takes as input the list of indices of two elements in $\mathbb{F}_{2^P}$, and output the list of indices of their sum.

As expected, this function involves quite a lot of comparing the input indices and is not constant-time. This function could be rewritten by using techniques derives from constant-time sorting algorithms.

Lastly, a less severe leakage appears in `gf2x_transpose_in_place_sparse` where `A[0]` might leak, and `rand_circulant_sparse_block` implements a kind of rejection sampling for generating random elements and matrices, which triggers a warning from our tool, but no leakage actually happens.

AES re-implementation LEDAcrypt ships with an AES implementation, used in the randomness generation code. It uses a T-table implementation which is not protected from cache-timing leaks. However, a real installation of LEDAcrypt would very likely use a system-specific and probably protected AES implementation, provided by OpenSSL or similar libraries.

3.3.6 Picnic

Picnic is a post-quantum signature scheme based on zero-knowledge proofs-of-knowledge (ZK-PoK), made non-interactive using standard transforms (Fiat-Shamir or the Unruh transform). The particular ZK-PoK exploited in picnic consists in simulating a multi-party computation (MPC) protocol T times, and selectively revealing some of the transcripts of the MPC as part of the signature. Two variants, picnic and picnic2 are proposed, the latter exploiting a more efficient ZK-PoK protocol. The principal advantage of this construction is that the security of the scheme only relies on standard symmetric primitives (cryptographic hash-functions and block ciphers). Thus, it is fairly unlikely that devastating attacks breaking novel assumptions will be discovered any time soon. As such, is can be seen as an alternative to stateless hash-based signature schemes, with which is shares the advantages (confidence in the security) and drawbacks (fairly long signature times and sizes compared with lattice-based solutions).

Two variants, picnic1 and picnic2, are implemented in the code submitted to NIST. Our analysis applies to both versions and shows that they are **constant-time**.

Leakage of the commitment hashes The analysis revealed that the commitments to the MPC traces can leak during various operations. They are stored inside a Merkle tree, and search operations on the Merkle tree may leak the values of the commitments that are searched for. However, because the commitments are either part of the signature, or can be reconstructed from it, they should be considered as *public output*. Once the commitments are flagged as public output, almost all leakages disappear. For the picnic1 variant, the values of the challenges e_t also leak during the proof-assembling phase (see [86, § 6.2; Step 5], the implementations branch on the values of e_t). However, since $e = (e_t)_t$ is actually part of the signature, this is obviously a leakage of public output.

For the picnic2 variant, the only remaining leakages are concerning verifications against corrupted public keys or fault injection attacks resulting in a failed MPC simulation (see [86, § 8.4]). However, this behavior can be directly observed by interacting with the program performing the signature, and no side-channel is required in order to determine

whether a failure took place. The only remaining risk would be a simultaneous fault and cache-timing attack: a fault attack could corrupt the secret key or other sensitive variables, and the side-channel attack could reveal the faulted commitments. However, even in this case, it is not clear how an attacker could exploit the knowledge of the commitments in order to gain knowledge about sensitive values. In any case, fault attacks are out-of-scope in this analysis, so the provided implementations are in fact **constant-time**.

Rejection sampling-like hashing The picnic1 variant uses a ternary hash function `H3` that generates a sequence of elements in $\{0, 1, 2\}$. As shown in [86, § 6.4.5], the ternary hash function is build from a binary hash function by iterating over pairs of bits. The pair $(1, 1)$ is discarded, while the other pairs are converted into ternary elements. This implies a branching operation on the pair of bits, which is potentially sensitive, as confirmed by the code implementing this conversion in the reference implementation given in Listing 3.14 (the code for the optimized implementation is different but also branches on the value of the bit pairs). However, this branch is actually benign: as in rejection sampling, the values returned by `H3` cannot be inferred from the direction of the branching operations. Furthermore, the output of `H3` is public, so this could also be considered as a leakage of public output.

```
1 for (int j = 0; j < 8; j += 2) {
2   uint8_t bitPair = ((byte >> (6 - j)) & 0x03);
3   if (bitPair < 3) {
4     setChallenge(challengeBits, round++, bitPair);
5     if (round == params->numMPCRounds) { goto done; }
6   }
7 }
```

Listing 3.14: Ternary output generation

3.3.7 NTSKEM

NTSKEM is a code-based post-quantum key-encapsulation scheme build on Goppa codes. It proved to be a challenge to analyze for three reasons:

- It makes heavy use of *function pointers*. While the initial version of our analysis tool had very limited capabilities for handling function pointers, these were expanded in an effort to try to handle the NTSKEM submissions. However, this ultimately proved unsuccessful, in part because the NTSKEM code makes use of function pointers to functions using *themselves* function pointers. This could be handled but requires a greater rewriting effort. Thankfully, only a handful of such functions are used by NTSKEM, exclusively for finite-field operations. These functions were manually inspected and easily found to be constant-time. They could then be excluded from the analysis.

- The implementations try to hide or abstract some implementation details via **void** $*$-typed fields of structures related to the private key. These fields will themselves

hold an other structure, also related to the private key. Some of these secondary fields, hidden behind the **void** ∗-field, are sensitive. However, because the information about the structure fields is lost, our analysis tool must consider all fields as being sensitive, which yields more false positives.

- The code makes use of generic data structures. More specifically, it makes use of a stack implementation which is used for the Fast Fourier Transform (FFT). Polynomials are saved into that stack, and the coefficients of some of them are to be considered sensitive. However, the structure representing the coefficients also hold non-sensitive data, such as their degree. Again, after passing by the **void** ∗ interface of the stack implementation, all fields of the polynomial structure are considered sensitive, which yields an unusually high amount of false positives.

A manual inspection found that most warnings emitted by our tool stem from false positives due to the reasons mentioned above. However, in addition to some leakages during key generation (which may be considered an issue or not), one potential leakage could be confirmed.

Permutation leakage As part of the key-generation process, a permutation **p** is generated, which is part of the private key. The generation of the permutation itself is done via Fisher-Yates shuffle [87] and is not constant-time. More importantly, the permutation is used during the error-recovery step of the decapsulation to index the `e_prime_ptr` array, as shown in the code of Listing 3.15 at Line 7, 12, and 18. This leakage is present in both the reference and the optimized implementations of NTSKEM.

```
1  #define bit_value(v,i) (((v)[(i)>>LOG2] & (1ULL<<((i) & MOD))) >> ((i) & ↩
     MOD))
2
3  //...
4
5  for (i=0; i<NTS_KEM_PARAM_A; i++) {
6    a = p[i];
7    bit_value = bit_value(e_prime_ptr, a);
8    bit_set_value(e_ptr, i, bit_value); /* Permute e_prime */
9  }
10 for (; i<NTS_KEM_PARAM_K; i++) {
11   a = p[i];
12   bit_value = bit_value(e_prime_ptr, a);
13   bit_set_value(e_ptr, i, bit_value); /* Permute e_prime */
14   bit_toggle_value((packed_t *)k_e, i-NTS_KEM_PARAM_A, bit_value); /* Step↩
         8: recovering k_e */
15 }
16 for (; i<NTS_KEM_PARAM_N; i++) {
17   a = p[i];
18   bit_value = bit_value(e_prime_ptr, a);
19   bit_set_value(e_ptr, i, bit_value); /* Permute e_prime */
20 }
```

Listing 3.15: NTSKEM error decoding

3.3.8 LAC

LAC (LAttice-based Cryptosystems) is a ring-LWE based post-quantum public-key encryption and key-encapsulation scheme. The implementations submitted for the second round NIST post-quantum contest made use of a variable-time BCH decoding algorithm and was subsequently updated. Here, we therefore analyze the updated "LAC-v3a" variant which, as we will see, is still non constant-time.

Finite field operations Multiplication in finite fields is implemented with table lookups, similar to other non-constant-time implementations. For instance, the code for finite field multiplication is given in Listing 3.16. This implementation avoids branching in order to check if both `a` and `b` are zero and uses a ternary check as well as masking. While it cannot be guaranteed that the compiled code does not contain any branching, we could verify their absence by inspecting the compiled code. However, this property needs to be checked again when using another compiler or compilation target. In any case, the problem of a secret-dependent table access remains (see Line 4). Similar leakages are also present in the functions `gf_sqr` and `a_log`.

```
1 static inline unsigned int gf_mul(unsigned int a, unsigned int b)
2 {
3   unsigned int tmp,mask;
4   tmp=a_pow_tab[mod_s(a_log_tab[a]+a_log_tab[b])];
5   mask=(a && b)? 0xffffffff:0x0;
6   return  tmp&mask;
7 }
```

Listing 3.16: LAC finite field multiplication

Finite-field polynomial multiplication is also affected by secret-dependent memory accesses. The secret corresponds to a sparse vector of $+1$ and -1 coefficients. In the sparse representation, only the positions of those non-zero coefficients are stored. The core multiplication code is given in Listing 3.17, where `s_one` (resp. `s_minusone`) represent the positions of the coefficients equal to $+1$ (resp. -1). The address of both `v1_p` and `v2_p` depends on these sensitive positions, and the array accesses at Line 5 resp. Line 6 thus constitute secret dependent memory accesses.

```
1 for(i=0;i<NUM_ONE;i++) {
2     v1_p=v+DIM_N-s_one[i];
3     v2_p=v+DIM_N-s_minusone[i];
4     for(j=0;j<vec_num;j++) {
5         sum1[j] += v1_p[j];
6         sum2[j] -= v2_p[j];
7     }
8 }
```

Listing 3.17: LAC finite field polynomial multiplication

BCH decoder array accesses and branching While the BCH decoder has been improved for LAC-v3a, the provided code is still potentially vulnerable against cache-timing attacks. The issues causing the vulnerabilities include:

- Use of primitives for finite fields that use table accesses (see above);
- “Solving” timing issues by duplicating code (see Listing 3.18), which does not protect from e.g. BranchScope [46]. This is even more problematic as the compiled code for the second branch is, for some reason, located far away from the rest of the function code, making this section particularly vulnerable to cache-timing attacks in both the reference and optimized implementation;
- Use of the ternary operator for masking, for the same reasons as exposed above.

```
1 if (tmp > elp->deg) { //memcpy cause 70 cycles gap
2   gf_poly_copy(&pelp, &elp_copy,i);
3   elp->deg = tmp;
4
5 } else {//just for constant time
6   gf_poly_copy(&elp_copy, elp,i);
7   tmp =  elp->deg;
8 }
```

Listing 3.18: LAC branch balancing

Decapsulation failure check The verification whether the received ciphertext is correctly decapsulated is not done in constant time (see Listing 3.19). This probably leads to an attack against this implementation [88].

```
1  if(memcmp(c,c_v,CIPHER_LEN)!=0)
2  {
3    //k=hash(hash(sk)|c)
4    hash((unsigned char*)sk,SK_LEN,buf);
5    hash(buf,MESSAGE_LEN+CIPHER_LEN,k);
6  }
```

Listing 3.19: LAC decapsulation failure check

3.3.9 SIKE

SIKE (Supersingular Isogeny Key Encapsulation) is a post-quantum KEM scheme that relies on the hardness assumption of finding isogenies between supersingular elliptic curves. This is the only NIST-PQC candidate based on this novel assumption. Its main advantage are short public keys and ciphertexts. This was one of the more complicated submissions to analyze, for two reasons:

- SIKE relies on GMP [89], a library for handling large integers. More specifically, it relies on a subset implemented in a single 3400 lines-of-code file, `mini-gmp.c`, which has to be analyzed by our tool. This somewhat increases the duration of the analysis.
- The SIKE reference implementation makes liberal use of function pointers which are passed around in structures representing the algorithm parameters. Thus, the analysis tool had to be improved in to better handle function pointers, but the support of this functionality is still experimental.

While the analysis is probably not as accurate as for the other candidates, we could nevertheless make out two sources of potential leakages. These are only present in the reference implementation; the optimized implementation is, at the best of our knowledge, constant-time.

Point multiplication Point multiplication is implemented via the double-and-add method and provided in the function `mont_double_and_add`. The core of this function is presented in Listing 3.20. This code branches on the value of `k`, which represents the secret key.

```
1 for (i = msb - 1; i >= 0; i--) {
2   xDBL(curve, &kP, &kP);
3   if (fp_IsBitSet(p, k, i)) {
4     xADD(curve, &kP, P, &kP);
5   }
6 }
```

Listing 3.20: Point multiplication via double-and-add

Use of GMP Because GMP is a generic multiprecision arithmetic library, it supports operations with numbers, and specifically integers, of various sizes. Thus, the duration of most operations depends on the size (in bytes or words) of the numbers being handled, and their size depends on the value. Therefore, most GMP operations on sensitive inputs are reported as leaking by our tool. However, whether this leakage actually weakens the security of the implementation is not clear and would require further investigation on what information can actually be learned from cache-timing attacks targeting GMP.

3.3.10 Other Analyzed Candidates

FrodoKEM FrodoKEM requires an AES implementation. Three variants can be used: OpenSSL, AES-NI, or a provided standalone implementation. This last one is not constant-time; it is a straightforward implementation that leaks during S-BOX access. However, in practice, one of the other two options would be used, and they would be constant-time. The rest of the code is constant time, with one exception: checking for decapsulation failure is **not** done in constant-time, leading to an exploitable timing variation [88]. The code has since been patched.

```
1 // Is (Bp == BBp & C == CC) = true
2 if (memcmp(Bp, BBp, 2*PARAMS_N*PARAMS_NBAR) == 0 && memcmp(C, CC, 2*↩
     PARAMS_NBAR*PARAMS_NBAR) == 0) {
3   // Load k' to do ss = F(ct  k')
4   memcpy(Fin_k, kprime, CRYPTO_BYTES);
5 } else {
6   // Load s to do ss = F(ct  s)
7   memcpy(Fin_k, sk_s, CRYPTO_BYTES);
8 }
```

Listing 3.21: Decapsulation failure check

SPHINCS The SPHINCS-Haraka implementations ship with their own AES implementation which leaks the S-BOX accesses, but a real deployment is very likely to use a protected implementation or the AES-NI instruction set. Furthermore, warnings were reported for the Haraka variant, but this is due to a limitation in our tool: the implementation of this hash function represents the internal state with an array `s_inc` of type `uint8_t[65]`, where the last byte represents the number of bytes that were not processed yet. This is not a sensitive information, but because our analysis tool does not distinguish between dependencies inside the same array, it is considered sensitive and leakages are reported, but they are false positives. These false positives are also reported for the shake256 variant, for similar reasons. Other than that, no leaks were reported, and we conclude that the provided implementations are **constant time**.

Falcon While the original second round implementations included a variable-time Gaussian sampler, it was replaced by a constant-time sampler in an updated version [90]. We analyzed this update, and the only reported warnings concerned rejection sampling. Thus, we verified that the updated implementation is indeed **constant time**.

NTRUPrime NTRUPrime uses AES, and the provided implementations exploit OpenSSL's implementation, which we assume is constant-time. The rest of the code was verified to be **constant time**.

NTRU The provided implementation was verified to be **constant time**.

NewHope The rejection sampling used in NewHope triggers a warning from our tool, but this is expected. Other than that, the code is **constant time**.

CRYSTALS-Kyber The rejection sampling used in CRYSTALS-Kyber triggers a warning from our tool, but this is expected. Other than that, the code is **constant time**.

CRYSTALS-Dilitihium The use of rejection sampling in several functions, as well as using recursive `goto` instructions to implement it, caused an unusually high number of false positives. Manual inspection of these false positives confirmed that the code is **constant time**.

ThreeBears The provided implementation was verified to be **constant time**.

rainbow The small number of false positives reported by our tool were causes by difficulties handling `union` constructs. The provided implementation was nevertheless verified to be **constant time**.

McEliece Ignoring the code present only for debugging purposes, the provided implementation was verified to be **constant time**.

SaberKEM The provided implementation was verified to be **constant time**.

3.3.11 Summary

Most round 2 candidates could be shown to be constant-time. Only LAC, LEDAcrypt and NTS-KEM do not have constant-time implementations. For LUOV, not all parameter sets have constant-time implementations, and these use the AVX2 instruction set. Therefore, on platforms with no AVX2 support, no constant-time implementation is currently available. Finally, the reference implementation of Round5 and SIKE are not constant-time, but the provided optimized implementations are in fact constant-time. A summary of those findings is provided in Table 3.2 for the KEM algorithms, and Table 3.3 for the signature schemes.

3.4 Perspectives

The NIST post-quantum standardization process will undergo a third round before the winners will be announced. New implementations will be provided, and analyzing the new implementations will be necessary in order to verify the absence of cache-timing leaks. Additionally, key generation could also be investigated. Depending on the usage that is made of the cryptographic algorithms, new keys might be frequently generated, and the cryptographic systems might be broken if an attacker gains knowledge about the keys. To our knowledge, no cache-timing attacks on key generation have yet been published. This could also be due to the fact that an attacker would have access to only one trace per key generation, making attacks much harder. However, even partial information about the key could lower the security provided by the schemes.

Table 3.2: NIST PQC Round 2 Submissions (KEM Algorithms)

Algorithm Name	Reference Implementation	Optimized Implementation	Additional Notes
BIKE	?	?	Uses C11 features.
ClassicMcEliece	✓	✓	
CRYSTALS-KYBER	✓	✓	
FrodoKEM	✗	✗	
HQC	?	?	Uses C++.
LAC	✗	✗	
LEDAcrypt	✗	✗	
NewHope	✓	✓	
NTRU	✓	✓	
NTRU Prime	✓	✓	
NTS-KEM	✗	✗	Only one leakage found.
ROLLO	?	?	Uses C++.
Round5	✗	✓	
RQC	?	?	Uses C++.
SABER	✓	✓	
SIKE	✗	✓	
Three Bears	✓	✓	

✓ (green) Implementation verified to be constant time.
✗ Constant-time violations found.
✓ (black) Implementation assumed to be constant-time.
? Not analyzed.

Table 3.3: NIST PQC Round 2 Submissions (Signature Algorithms)

Algorithm Name	Reference Implementation	Optimized Implementation	Additional Notes
CRYSTALS-DILITHIUM	✓ (green)	✓	
FALCON	✓ (green)	✓	
GeMSS	?	?	Uses C++.
LUOV	✗	✗	AVX2 Implementation is constant-time.
MQDSS	✓ (green)	✓	
Picnic	✓ (green)	✓	
qTESLA	✓ (green)	✓	
Rainbow	✓ (green)	✓	
SPHINCS+	✓ (green)	✓	Assuming AES-NI is available.

✓ (green) Implementation verified to be constant time.
✗ Constant-time violations found.
✓ Implementation assumed to be constant-time.
? Not analyzed.

Chapter 4

On the Stochastic Model of Physically Unclonable Functions

Scientists have calculated that the chances of something so patently absurd actually existing are millions to one. But magicians have calculated that million-to-one chances crop up nine times out of ten.

Terry Pratchett.

4.1 An Introduction to Physically Unclonable Functions (PUFs)

PUFs are electronic circuits whose behavior is unpredictable from one instantiation to another. However, ideally, the behavior of a given manufactured PUF device is deterministic and constant in time. These devices can be used as building blocks for hardware enabled security protocols [91], requiring little computing power, and thus suitable for embedded devices. However, the security and reliability properties are not yet well understood, undermining the usability of protocols based on such hardware primitives. In the remainder of this thesis, we will motivate the use of a stochastic model to predict and improve the reliability and security of PUFs, with applications to the Loop-PUF [92] and other PUFs sharing the same stochastic model (RO-sum PUF [93], Arbiter PUF [94], Ring-oscillator PUF [95]).

4.1.1 "Weak" and "Strong" PUFs

There are two black-box ways to model a PUF. The first way consist in considering a PUF as a device without input that produces an *identifier* when powered up. This is called a **weak PUF**. This identifier can then be used, for instance, to generate a cryptographic key, or be used to detect counterfeit devices.

The second way consists in considering a PUF as a device that takes an input, called *challenge*, and in response to that challenge produces an *output* bit. Such a PUF is called a **strong PUF**, and can be used, for instance, in challenge-response authentication protocols.

Of course, one can always consider a weak PUF that produces an identifier of n bits as a strong PUF with n possible challenges, and a strong PUF can become a weak PUF if the set of admissible challenges is fixed and part of the PUF specification. However, in general, a PUF is considered "weak" if the number of possible output bits is linear or sub-linear in the PUF "size", and "strong" if it is larger (for instance, exponential in the PUF "size").

4.1.2 Toy Example: The Ring Oscillator

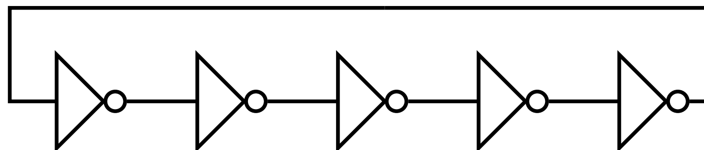


Figure 4.1: An oscillating chain of 5 inverters

As an example of how to design a PUF, let's consider a chain of an odd number of inverters (NOT gates), where the output of the last gate is taken as the input of the first gate. This is called a *ring oscillator*. After an impulse is applied as the input of the first inverter, the voltage at any given point in the circuit will oscillate. The exact oscillation frequency will depend on the physical characteristics of the inverters, and thus slightly vary from one chain to another. For a given circuit, however, this frequency should be relatively stable among several measurements. The frequency measurement can then be converted into response bits. One could, for instance, take the most significant digit of the measurement. However, it is not guaranteed that this number is uniformly distributed, and it might be subject to measurement noise and intrinsic jitter [95]. Another solution is to take *two* oscillators and compare their frequencies, yielding one output bit. This forms the basis of the so-called Ring Oscillator PUF.

4.1.3 Description of the Analyzed PUF Designs

In the next chapters, we will analyze the stochastic model applicable to several PUF designs exploiting the difference of delays in electronic circuits ("delay PUFs")

Ring Oscillator PUF The Ring Oscillator PUF [91] (RO PUF) consists of N ring oscillators with the same, odd number of inverters. It is considered a weak PUF, and there are mainly two ways to extract an identifier from a RO PUF: either comparing the frequencies of all ring oscillators, or comparing only the frequencies of $\lfloor N/2 \rfloor$ ring oscillator pairs. In the first case, the output of the PUF consists in a permutation of N elements, and assuming the oscillator frequencies follow an i.i.d. law, produces $\log_2(N!) \simeq N\log(N)$

bits of entropy. However, encoding the permutation into $\log_2(N!)$ independent bits is not trivial. In the second case, the output are directly $\lfloor N/2 \rfloor$ independent bits. The circuit description is summarized in Figure 4.2.

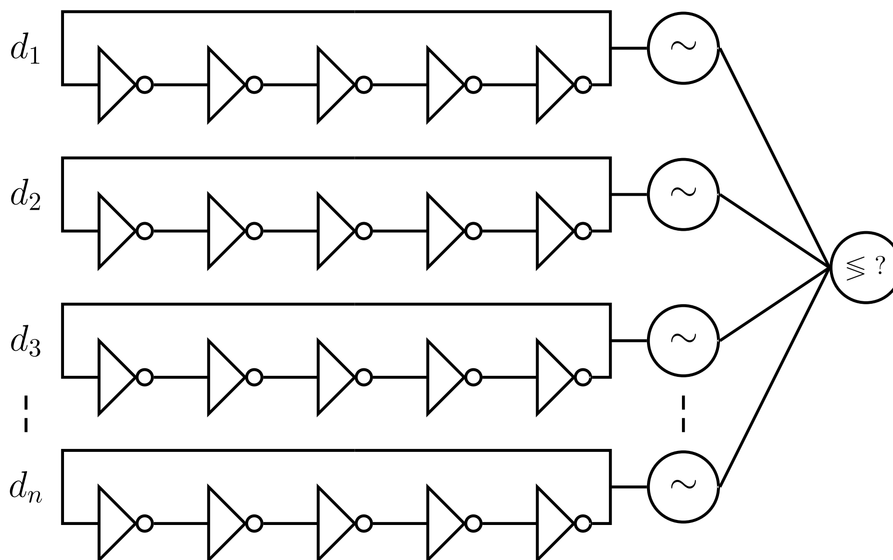


Figure 4.2: Ring Oscillator PUF

RO PUF output: We suppose here that the RO PUF is used to output $n = \lfloor N/2 \rfloor$ independent bits. Let d_i denote the delay of the ring oscillator i. The RO PUF output is then equal to

$$b = (\mathrm{sign}(d_{2i+1} - d_{2i+2}))_{i \in [0,n-1]} \,.$$

Arbiter PUF The arbiter PUF [91] consists of two series of delay paths, with n mux elements on each path, as depicted in Figure 4.3. Depending on the challenge bits $c_1, c_2, \ldots, c_n$, the signals either cross from one path to another if $c_i = -1$, or stay at the same path if $c_i = 1$. One bit of output is produced by determining whether the signal injected into the circuit arrives first on the last mux of the upper path or of the lower path.

Since there are 2^n different possible challenges, the arbiter PUF is considered a strong PUF. Compared to the RO PUF, it is however more complicated to implement, as all sections of the delay path between consecutive mux elements need to be perfectly balanced, as well as the cross links between the upper and lower path.

Arbiter PUF output: Let's denote by d_1^i (resp. d_{-1}^i) the delay incurred by the signal traversing the i-th upper (resp. lower) delay element. The arbiter PUF output for a given challenge $c = (c_1, c_2, \ldots, c_n)$ is then equal to

$$b(c) = \mathrm{sign}\left(\sum_{i=1}^{n} d^i_{c_i \cdot c_{i+1} \cdot \ldots \cdot c_n} - \sum_{i=1}^{n} d^i_{-c_i \cdot c_{i+1} \cdot \ldots \cdot c_n}\right) = \mathrm{sign}\left(\sum_{i=1}^{n} \left(\prod_{j=i}^{n} c_j\right) (d_1^i - d_{-1}^i)\right).$$

Note that this slightly simplified model does not consider differences in delays caused by the connections between delay elements and muxes. However, as in [96], these extra delay differences can be taken into account into a model that is also linear in $\left(\prod_{j=i}^{n} c_j\right)_i$.

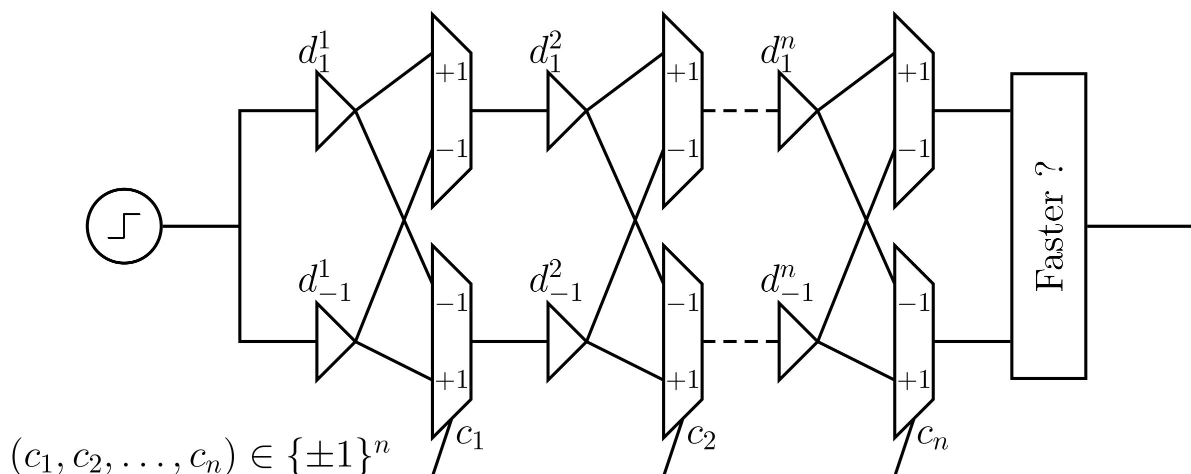


Figure 4.3: Arbiter PUF

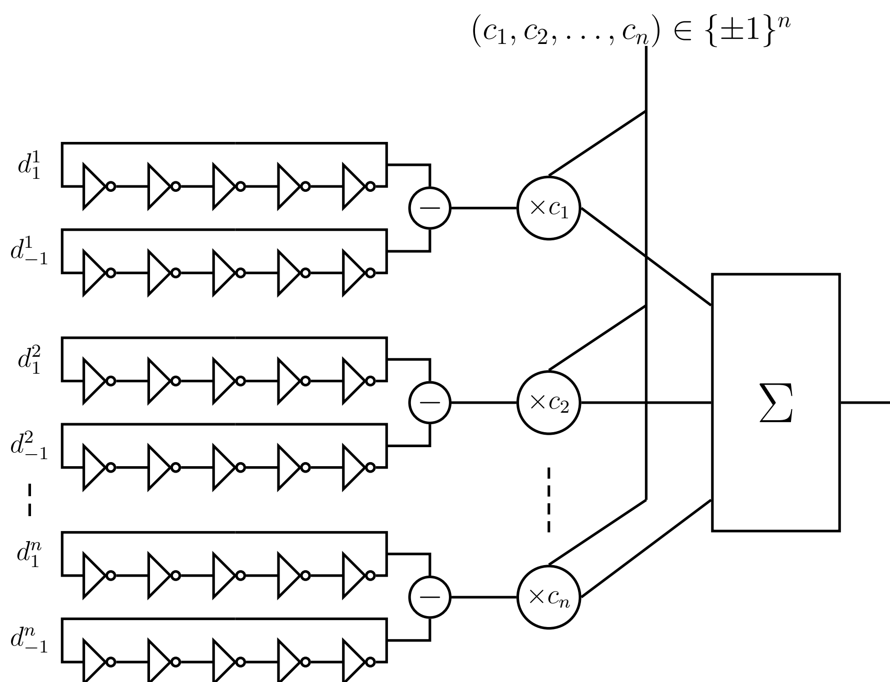


Figure 4.4: RO-sum PUF

RO-sum PUF The RO-sum PUF [93] consists of n pairs of identical ring oscillators. The frequency measured for one oscillator of a pair is subtracted from the frequency of the other oscillator in the pair, multiplied by $+1$ or -1 depending on a challenge bit, and summed with the other frequency differences, as described in Figure 4.4. The output bit is then the sign of this sum. This design transforms $2n$ weak RO PUFs into one strong PUF. The output is linear in the delays, as in the arbiter PUF. Using ring oscillators allows to improve the reliability by improving the frequency measurement of the oscillator's frequency, whereas the arbiter PUF measurement cannot be improved. However, the RO-sum PUF design requires $2n$ ring oscillators, n subtractors and multipliers, as well as

a summation element, and is thus more costly to build than an arbiter PUF.

RO-sum PUF output: Let's denote by d_1^i and d_{-1}^i the delay of the two ring oscillators of the i-th pair. The RO-sum PUF output for a given challenge $c = (c_1, c_2, \dots, c_n)$ is then equal to

$$b(c) = \text{sign}\left(\sum_{i=1}^{n} c_i \cdot (d_1^i - d_{-1}^i)\right).$$

Loop-PUF The loop-PUF [92], described in Figure 4.5, is a combination between the arbiter PUF and the RO-sum PUF. It consists of n pairs of delay elements, separated by n mux elements. Depending on the value of the i-th challenge bit, each mux will choose to output the signal delivered by one or the other delay element of the previous pair. This design is easier to implement than an arbiter PUF, since once a balanced pair of delay elements has been designed, they are fairly easy to replicate. Similarly to the RO-sum PUF, it is possible to amplify the differences in delay of the different delay elements by measuring several oscillation cycles, improving the reliability.

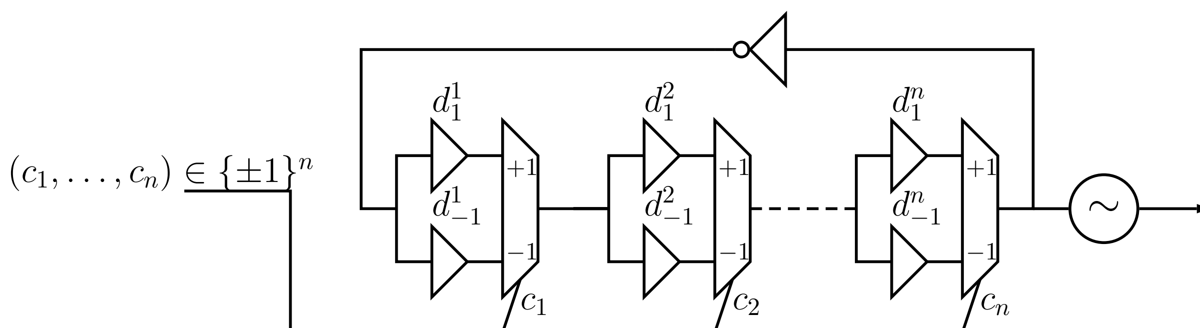


Figure 4.5: Loop PUF

There are several ways in which one can obtain an output bit from a loop-PUF. We will use the method described in [97] which consists in measuring the the delay for a codeword $c = (c_1, c_2, \dots, c_n) \in \{\pm 1\}^n$ and its complimentary $-c$, and then taking the sign of the difference in delays.

Loop PUF output: Let's denote by d_1^i (resp. d_{-1}^i) the delay incurred by the signal traversing the i-th upper (resp. lower) delay element. The loop-PUF output for a given challenge $c = (c_1, c_2, \dots, c_n)$ is then equal to

$$b(c) = \text{sign}\left(\sum_{i=1}^{n} c_i \cdot (d_1^i - d_{-1}^i)\right).$$

4.2 Stochastic Models for PUFs

4.2.1 Delay Distribution

For the stochastic model of these PUFs, we will suppose, as in [98], that all delays d_i^j follow a **i.i.d. Gaussian distribution**. This assumption is justified by the statistical analysis of the PUF circuit delays [92, 99].

Strong PUFs As a consequence, the PUF output for the strong PUFs (arbiter PUF, RO-sum PUF and loop-PUF) can be rewritten as follows:

$$b(c) = \text{sign}(c \cdot x)$$

where $x = (d_1^1 - d_{-1}^1, d_1^2 - d_{-1}^2, \ldots, d_1^n - d_{-1}^n)$ is distributed as an i.i.d. Gaussian vector. For the arbiter-PUF, a one-to-one transformation of the challenge needs to considered before the scalar product with the delay differences is taken. As this does not change the security and reliability considerations, we will ignore this transformation for the sake of simplicity.

In general, the response bits for different challenges are **not independent**. Although there can be as many as 2^n possible challenges, we showed that the entropy cannot be higher than about n^2. Furthermore, for small values of $n \leq 10$, the Shannon entropy is actually close to its theoretical maximal value. These findings, along with a description of the methods that allowed us to perform the simulations, are presented in Chapter 6, which is based on two arte icles, one presented at *Allerton Conference on Communication, Control, and Computing (2019)* [100], the other published in *Advances in Mathematics of Communications (2020)* [101].

Weak PUFs The three aforementioned PUFs can be turned into weak PUFs by restricting the set of applicable challenges. As proven in e.g. [97], using a Hadamard matrix as the challenge matrix allows to obtain independent response bits [1]. Therefore, the model of the weak PUF is

$$b = (\text{sign}(x_1), \text{sign}(x_2), \ldots, \text{sign}(x_n))$$

where $x = (x_1, x_2, \ldots, x_n)$ is distributed as a vector of i.i.d Gaussian random variables.

4.2.2 Measurement Noise Distribution

PUF delay measurements are subject to several sources of noise. One such source is the *measurement noise*, altering the measured delays. It can be modeled as an additive Gaussian noise, independent from the circuit delays. Thus, the weak PUF model with noise becomes

$$b = (\text{sign}(x_1 + y_1), \text{sign}(x_2 + y_2), \ldots, \text{sign}(x_n + y_n))$$

where $(y_1, y_2, \ldots, y_n)$ is distributed as a vector of i.i.d Gaussian random variables with variance σ^2, and similarly for the strong PUF model.

An analysis of the weak PUF model with noise, as well as the effect of reliability-enhancing measures, is provided in Section 5.1, based on work published at *Digital System Designs (2018)* [102]. Follow-up work that does not reduce the entropy as part of the reliability-enhancing measures, but uses a function more complicated than the sign function, is presented in Section 5.4 and is based on work published at *IEEE International Workshop on Advances in Sensors and Interfaces (2019)* [103].

[1] Assuming that the delays are distributed as Gaussian random variables

Chapter 5

On the Reliability of Physically Unclonable Functions

Insanity is doing the same thing over and over again and expecting different results.

Unknown.

This chapter is based on the two articles "An improved analysis of reliability and entropy for delay PUFs", published in *21st Euromicro Conference on Digital System Design (DSD)* (2018) [102], and "Two-Metric Helper Data for Highly Robust and Secure Delay PUFs", published in *IEEE 8th International Workshop on Advances in Sensors and Interfaces* (IWASI) (2019) [103], with coauthors Sylvain Guilley, Olivier Rioul and Jean-Luc Danger.

5.1 An Improved Analysis of Reliability and Entropy for Delay PUFs

Designing a PUF involves a three-way tradeoff between entropy, reliability and complexity (e.g., circuit size). Firstly, entropy is increased by adding more elements such as RAM cells or oscillators, at the expense of an increased circuit size. Also, reliability is enhanced by error-correcting codes (ECC), but their redundancy generally decreases the entropy. For a given PUF design, it is not obvious how to precisely characterize the tradeoff between these three parameters (entropy, reliability and circuit complexity). For instance, fuzzy extraction [104] using error-correcting codes is implemented in the PUFKY [105] based on the ROPUF [106]. However, for this design, and fuzzy extraction in general, it is very hard to determine the bit error rate (BER) theoretically. Therefore, the actual parameter selection for the fuzzy extractor is not straightforward. Bit-filtering [107] can also improve the reliability but since the number of output bits is reduced as a result of the filtering, this technique also decreases the entropy of the PUF. Thus, this technique, similar to fuzzy extraction, is subject to a tradeoff between reliability and entropy.

The aim of this chapter is to build a framework to analyze this tradeoff for delay PUFs. Maes [108] proposed such a framework for the reliability of SRAM PUFs which was *ad hoc*

for a given PUF architecture and where the parameters' identification was performed on experimental data. Bhargava et al. [109] also perform filtering to improve the reliability of their PUF design, but provide no theoretical model to predict the reliability that might be obtained. In contrast, we aim at deriving a generic model using elementary assumptions, where the three-way tradeoff is not fully determined by real measurements, but given instead by closed-form expressions involving the signal-to-noise ratio (SNR). In this way, additional estimations of the SNR yield *new predictions* for the tradeoff.

Our framework is applied to three popular delay PUFs: the RO-PUF [91], the RO sum PUF [93] and the Loop PUF [92]. Two methods are chosen to improve the reliability: bit-filtering in a manner similar to the η-out-of-λ scheme of Škoric et al. [107], and a novel "two-metrics" method. Our contributions are as follows:

- a generic tradeoff analysis framework for delay PUFs;
- closed-form expressions for the BER and entropy for these PUFs, with and without bit-filtering;
- an alternative method to improve reliability, based on two distinct metrics, as well as reliability estimations for this method;
- an analysis of the RO-PUF, the RO sum PUF and the Loop PUF, using this framework;
- real measurements of the delay PUFs on ASIC confirming our theoretical results.

The remainder of this chapter is organized as follows. Section 5.2 presents a theoretical model for the delay PUFs. Closed-form expressions for reliability and entropy under bit-filtering are derived in Section 5.3. The "two-metric" method is presented in Section 5.4. The analysis framework is applied to various delay PUFs in Section 5.5. Section 5.6 provides an experimental validation on silicon. Section 5.8 concludes.

5.2 Delay PUF Model

In this section, we provide a black-box analysis for a generic delay PUF. Throughout this chapter we use the following notations.

n	number of delay elements in the circuit
i	index of a delay element
t	index of a measurement
T	total number of measurements
M	total number of challenges
m	index of a challenge

J	number of circuits
j	index of a circuit
c_i^m	i-th challenge bit
C^m	m-th challenge $C^m = (c_i^m)_i$
$d_{C,t}^j$	total delay for challenge C (at measure t, for circuit j)
$\delta_{C,t}^j$	$\delta_{C,t}^j = d_{C,t}^j - d_{\neg C,t}^j$
δ_C^j	$\delta_C^j = \frac{1}{T}\sum_{t=1}^{T}\delta_{C,t}^j$
Δ_C	random variable modeling δ_C
Z	additive Gaussian measurement noise

For simplification, sub- and superscripts (such as m, t, or j) may be dropped when this does not introduce any confusion.

We model an ideal (noiseless) delay PUF as a deterministic algorithm $\mathcal{P}^I$ that takes a challenge C as input, and outputs a delay difference δ_C:

$$\mathcal{P}^I : C \mapsto \delta_C.$$

This delay is then, in general, discretized in order to extract one (or more) bit(s). Thus, the final output is some function of the measured delay difference. For the sake of simplicity, we consider the *sign* function as the bit-output of the PUF:

$$b = \operatorname{sign}(\delta_C).$$

The delay difference δ_C for a given challenge stems from a multitude of small delay variations caused by technology dispersion, and is thus seen as a realization of a random variable Δ_C. Similarly to Lim et al. [98], we model this random PUF variable as Gaussian $\Delta_C \sim \mathcal{N}(0, \Sigma^2)$ for some positive deviation $\Sigma > 0$.

Such a delay PUF model is ideal since in practice, measurement noise is always present. Following e.g., [98] we model this noise as additive and independent Gaussian. Our PUF model becomes a *probabilistic* algorithm:

$$\mathcal{P} : C \mapsto \delta_C + Z \qquad b = \operatorname{sign}(\delta_C + Z) \tag{5.1}$$

where $Z \sim \mathcal{N}(0, \sigma^2)$ for some $\sigma > 0$. Since $\mathcal{P}(C)$ is the sum of a "signal" Δ_C and noise Z, the signal-to-noise ratio (SNR) can be defined

$$\text{SNR} = \frac{\mathbb{E}[\Delta_C^2]}{\mathbb{E}[Z^2]} = \frac{\Sigma^2}{\sigma^2}. \tag{5.2}$$

and the bit error rate is defined as

$$\text{BER}(\delta_C) = \mathbb{P}(\operatorname{sign}(\delta_C + Z) \neq \operatorname{sign}(\delta_C)). \tag{5.3}$$

To simplify the reliability analysis, we make the additional assumption that *all PUF responses δ_C are mutually independent*. In general this will only be satisfied approximately. As shown below for each specific PUF, the independence assumption will hold accurately for specific sets of challenges (at the order of n).

In the model proposed by Maes [108], δ_C would correspond to the *process variables* and Z to the *noise variable*. However, the output bits from a delay PUF do not precisely correspond to a measurement of the process variables and further analysis is needed to apply the Maes model to delay PUFs. Furthermore, rather than estimating the BER from experimental data and then find the parameters using a top-down approach, we find it more convenient to derive the BER from measures of simple system parameters such as the SNR, in a bottom-up approach, as described in the next section. We feel that such a determination is better theoretically justified since it requires less *ad hoc* assumptions.

5.3 Delay PUF Reliability and Entropy

When considering n challenges to generate n response bits, there is a high probability that unreliable response bits are obtained. Katzenbeisser et al. [110] showed that there is 2% to 15% unreliable bits, depending on the environment. Here we consider the proportion of faulty bits, or, equivalently, the average probability that a PUF bit flips, as a metric to characterize the PUF reliability. In contrast to an SRAM PUF, for which only the output bit values are available, delays can be measured in a delay PUF to detect unreliable bits, as we will explain in the next sections.

5.3.1 Reliability Assessment

The reliability of a delay PUF is directly related to the absolute value $|\delta_C|$ of the delay difference δ_C associated to each challenge C. Indeed, the larger the value, the smaller the probability to have a bit flip of the measured δ_C sign due to measurement error. More formally, if we consider the Gaussian noise $Z \sim \mathcal{N}(0, \sigma^2)$ added to δ_C, the BER is the probability to have a bit flip for challenge C, and is given by the following

Lemma 1. *One has*

$$\mathrm{BER}(\delta_C) = \mathbb{P}\Big(\mathrm{sign}(\delta_C + Z) \neq \mathrm{sign}(\delta_C)\Big) = Q\Big(\frac{|\delta_C|}{\sigma}\Big), \tag{5.4}$$

where $Q(x) = \frac{1}{2}\,\mathrm{erfc}(\frac{x}{\sqrt{2}})$.

Proof. Let $Z \sim \mathcal{N}(0, \sigma^2)$ and δ_C be a fixed value. Then

$$\begin{aligned}
\mathrm{BER} &= \mathbb{P}[\mathrm{sign}(\delta_C + Z) \neq \mathrm{sign}(\delta_C)] \\
&= \mathbb{P}[(\delta_C + Z > 0, \delta_C < 0)] + \mathbb{P}[(\delta_C + Z < 0, \delta_C > 0)] \\
&= \mathbb{P}[Z > |\delta_C|, \delta_C < 0] + \mathbb{P}[-Z > |\delta_C|, \delta_C > 0] \\
&= \mathbb{P}[Z > |\delta_C|] = Q\Big(\frac{|\delta_C|}{\sigma}\Big)
\end{aligned}$$

since Z is symmetrically distributed. □

Figure 5.1 illustrates the distribution of Δ_C and the noise distribution around the value δ_C associated to the challenge C. In this example, an error occurs when $\delta_C + Z$ is negative.

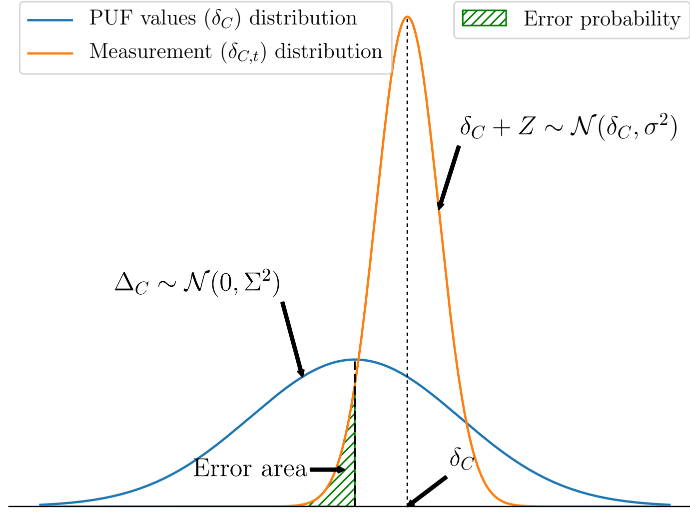


Figure 5.1: pdf of Δ and noise for a given challenge C.

$\lvert\delta_C\rvert/\sigma$ value	0	1	2	3	4
BER	0.5	$1.6\ 10^{-1}$	$2.3\ 10^{-2}$	$1.3\ 10^{-3}$	$3.2\ 10^{-5}$
$\lvert\delta_C\rvert/\sigma$ value	5	6	7	8	9
BER	$2.9\ 10^{-7}$	$9.9\ 10^{-10}$	$1.3\ 10^{-12}$	$6.2\ 10^{-16}$	$1.1\ 10^{-19}$

Table 5.1: BER for one bit according to the $|\delta_C|/\sigma$ value.

Table 5.1 gives the BER one can expect for a given challenge. For a set of challenges, the BER has to be assessed on all the δ_C values, which are assumed to be independent.

The average proportion of bit flips is the expectation of the BER over Δ_C, and is given by the following

Lemma 2. *One has*

$$\widehat{\mathrm{BER}} = \mathbb{E}[\mathrm{BER}(\Delta_C)] = \frac{1}{\pi}\arctan(\frac{1}{\sqrt{\mathrm{SNR}}}). \tag{5.5}$$

Proof. As shown in the proof of Lemma 1,

$$\begin{aligned}\widehat{\mathrm{BER}} &= \mathbb{P}[\mathrm{sign}(\Delta_C + Z) \neq \mathrm{sign}(\Delta_C)] \\ &= \mathbb{P}[Z > |\Delta_C|] \\ &= \mathbb{P}[\frac{Z}{\sigma} > |\frac{\Delta_C}{\Sigma}|\sqrt{\mathrm{SNR}}].\end{aligned}$$

Note that this probability is taken jointly over Δ_C, Z and that these are independent Gaussian variables, $\Delta_C \sim \mathcal{N}(0, \Sigma^2), Z \sim \mathcal{N}(0, \sigma^2)$. Therefore, $X = \frac{\Delta_C}{\Sigma}$ and $Y = \frac{Z}{\sigma}$ are independent and follow standard normal distributions, and the formula becomes

$$\widehat{\mathrm{BER}} = \mathbb{P}[Y > |X|\sqrt{\mathrm{SNR}}].$$

Since the probability distribution of (X, Y) is isotropic, it is easily seen that $\widehat{\text{BER}}$ equals

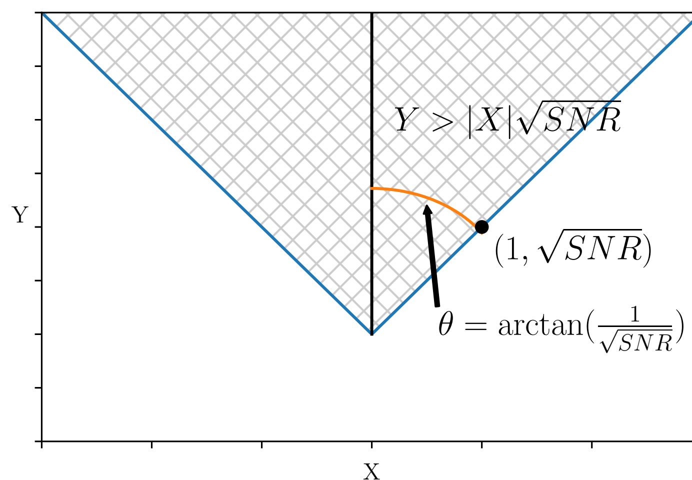


Figure 5.2: Polar representation of X and Y.

the proportion of the hatched area on Fig. 5.2. This proportion is simply $2\theta/2\pi$, where $\tan(\theta) = 1/\sqrt{\text{SNR}}$ by the geometric definition of the tan function. Thus, we simply have that

$$\widehat{\text{BER}} = \frac{1}{\pi} \arctan(\frac{1}{\sqrt{\text{SNR}}}). \qquad \square$$

The expected BER is represented as a function of the SNR in Fig. 5.3. Although the expected BER (5.3.1) vanishes with the noise:

$$\lim_{\text{SNR} \to +\infty} \widehat{\text{BER}} = 0,$$

it is easily seen that the expected BER remains quite high, $> 10^{-3}$, even for large values of SNR (several thousands).

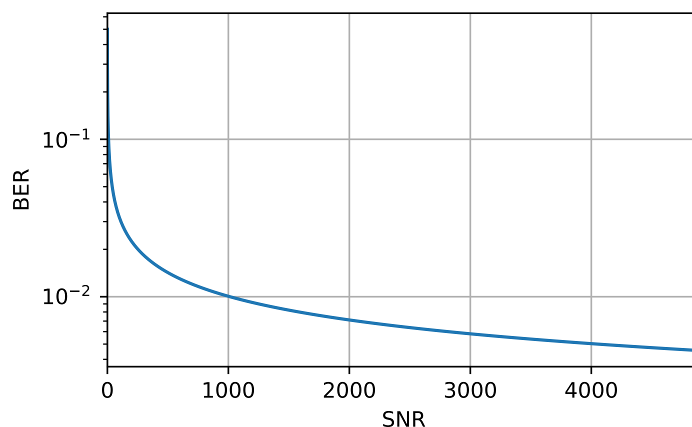


Figure 5.3: Expected BER as a function of the SNR.

5.3.2 Reliability Enhancement by Delay Knowledge

A classical and efficient method to enhance (reduce) the BER is to take advantage of ECCs, like the secure sketch methods presented by Dodis [104] and exploited by reliable architectures like PUFKY [105]. With this method, an enrollment phase takes place once, just after manufacturing, in order to build a public "helper data". The helper data, also called "secure sketch", can be either a n bit code-offset or a $n-k$ bit syndrome. During PUF usage, noise might corrupt the PUF value, but thanks to the secure sketch, the potential errors can be corrected by the ECC decoder.

We will investigate here another method to improve the reliability of the PUF that uses the knowledge of the δ_C values to filter out unreliable bits. Therefore, ECC may not be necessary or at least less complex, which helps to reduce circuit complexity.

The BER can be decreased discarding the challenges which generate unreliable bits. These challenges are recorded during the enrollment phase in the helper data. This helper data is then used during the reconstruction phase of the PUF. This construction resembles the η-out-of-λ scheme by Škoric et al. [107]. However, to make the computations tractable, instead of removing a fixed number of challenges, we remove bits that whose reliability is lower than a given threshold. Below we compute the resulting average reliability in terms of mean BER, and the average remaining entropy after bit-filtering.

Security Considerations From the security point of view, this helper data does not unveil any information of the response bits, since the PUF responses to challenges are assumed independent. However, if an attacker could modify the helper data, she could reconstruct the PUF response, for example using an attack similar to the one described by Hiller et al. [111].

An attacker could use the unreliable bits as a pivot to retrieve the key generated by the PUF. Indeed, suppose the attacker can exchange the reliability of bit i in and bit $i-1$ in the helper data, as shown in Fig. 5.4. If she noticed that the cryptographic result has not changed, that means that the key bits $i-1$ and i have the same value on average, as the unreliable bit is always biased towards '1' or '0' (the probability that it is exactly balanced is negligible). If she does the same with bits i and $i+1$, she can deduce whether key bits i and $i+1$ are the same, and consequently (by transitivity) if the reliable bits $i-1$ and $i+1$ are the same. This method allows the attacker to retrieve the whole key.

For the attack to work, it suffices that the attacker knows whether the key is properly generated or not. In this respect, any prior knowledge about the value of the (secret) key is unnecessary, only whether or not it can be used, e.g., whether a *secure boot* (based on a master key derived from the PUF) unfolds well. In this respect, this attack is equivalent to a *safe error* attack in the field of embedded devices security [112]. A natural countermeasure would be to have the helper data be checked for integrity (e.g., with a CRC). However, in an adversarial context, the attacker can change the helper data along with its CRC. Therefore, we have to assume that the helper data can only be read by an attacker, but cannot be modified. This can, for example, be achieved by storing it on ROM memory on the PUF.

The declaration of "unreliability" is given at enrollment phase when the delay $|\delta_C|$ for a challenge C is below a threshold Th, which has to be chosen to take into account the noise

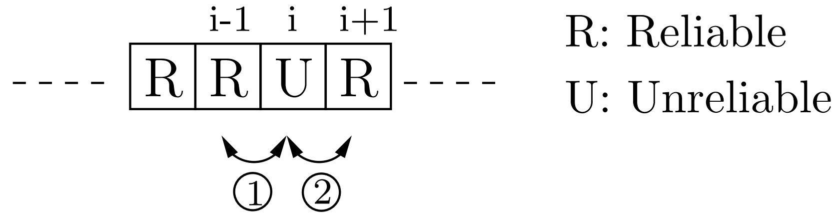


Figure 5.4: Attacking the helper data by using unreliable bits as pivot

level σ. In what follows, we set $Th = W \cdot \sigma$, where W expresses the capacity to filter the unreliable bits. Increasing W decreases the BER, but reduces the number of bits, hence the entropy.

Figure 5.5 illustrates the distributions of Δ_C and the noise. It points out the unreliable area in the window $[-Th, +Th]$ of width $2W\sigma$.

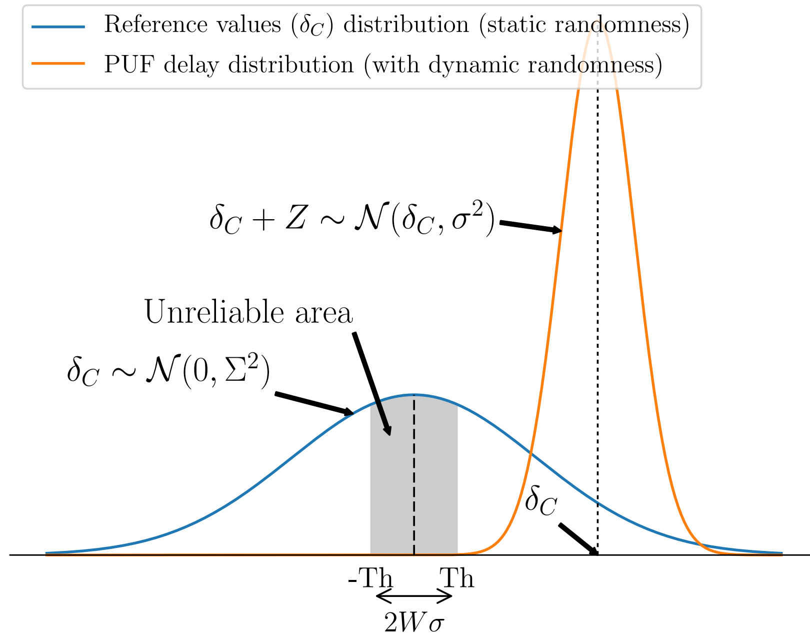


Figure 5.5: Unreliable area vs distributions of Δ_C and the noise Z.

The average BER reduction after filtering the unreliable bits depends directly on $Th = W\sigma$ and is given by the following

Lemma 3.

$$\widehat{\mathrm{BER}}_{filt} = \frac{2}{\mathrm{erfc}(\frac{W}{\sqrt{2}\sqrt{\mathrm{SNR}}})}\Bigg(T(W, \frac{1}{\sqrt{\mathrm{SNR}}}) + \frac{1}{4}\,\mathrm{erf}(\frac{W}{\sqrt{2}\sqrt{\mathrm{SNR}}})(\mathrm{erf}(\frac{W}{\sqrt{2}}) - 1)\Bigg) \quad (5.6)$$

where T represents Owen's T function:

$$T(h,a) = \frac{1}{2\pi}\int_0^a \frac{e^{-\frac{1}{2}h^2(1+x^2)}}{1+x^2}\,\mathrm{d}x.$$

Proof. For the sake of simplicity, we will drop the subscript C from the random variable Δ_C.

By definition of the filtered BER, we have that:

$$\widehat{\mathrm{BER}}_{filt} = \int_{-\infty}^{+\infty} p(\Delta \mid |\Delta| > Th)\cdot \mathrm{BER}(\Delta)\,\mathrm{d}\Delta.$$

This generic formulation, or very similar ones, have already been found before, for example by Delvaux [113] (Eq 4.41). However, we will apply it here to a specific PUF, and can therefore derive a more explicit formulation. Indeed, we can find a closed form of $\mathbb{E}(\mathrm{BER}_{filt})$ after filtering the bits as:

$$\begin{aligned}
\widehat{\mathrm{BER}}_{filt} &= \int_{-\infty}^{+\infty} p(\Delta \mid |\Delta| > Th)\mathrm{BER}(\Delta)\,\mathrm{d}\Delta \\
&= \int_{-\infty}^{+\infty} \mathbb{1}_{|\Delta|>Th}(\Delta)\frac{p(\Delta)}{\mathbb{P}(|\Delta|>Th)}\mathrm{BER}(\Delta)\,\mathrm{d}\Delta \\
&= \frac{2}{\mathbb{P}(|\Delta|>Th)}\int_{Th}^{+\infty} p(\Delta)\cdot\mathrm{BER}(\Delta)\,\mathrm{d}\Delta \\
&= \frac{2}{\mathbb{P}(|\Delta|>Th)}\frac{1}{2\sqrt{2\pi}\Sigma}\int_{Th}^{+\infty} e^{\frac{-\Delta^2}{2\Sigma^2}}\,\mathrm{erfc}(\frac{\Delta}{\sigma\sqrt{2}})\,\mathrm{d}\Delta.
\end{aligned}$$

Using the following integral value for $x, k > 0$:

$$\begin{aligned}
&\int e^{-x^2}\,\mathrm{erfc}(kx)\,\mathrm{d}x = \\
&-\frac{1}{2}\sqrt{\pi}(4T[\sqrt{2}kx,\frac{1}{k}] + \mathrm{erf}(x)(\mathrm{erf}(kx)-1)+1) + \text{constant}
\end{aligned}$$

(where T is Owen's T function, first introduced by Owen [114]), and using a change of variables, we get that

$$\begin{aligned}
\widehat{\mathrm{BER}}_{filt} = &\frac{2}{\mathbb{P}(|\Delta|>Th)}(T(\frac{Th}{\sigma},\frac{1}{\sqrt{\mathrm{SNR}}})+ \\
&\frac{1}{4}\,\mathrm{erf}(\frac{Th}{\sqrt{2}\Sigma})(\mathrm{erf}(\frac{Th}{\sqrt{2}\sigma})-1))
\end{aligned}$$

or, since $\mathbb{P}(|\Delta|>Th) = \mathrm{erfc}(\frac{W}{\sqrt{2}\sqrt{\mathrm{SNR}}})$ and $\frac{Th}{\Sigma} = \frac{W\sigma}{\Sigma} = \frac{W}{\sqrt{\mathrm{SNR}}}$,

$$\begin{aligned}
\widehat{\mathrm{BER}}_{filt} = &\frac{2}{\mathrm{erfc}(\frac{W}{\sqrt{2}\sqrt{\mathrm{SNR}}})}\Bigg(T(W,\frac{1}{\sqrt{\mathrm{SNR}}}) + \\
&\frac{1}{4}\,\mathrm{erf}(\frac{W}{\sqrt{2}\cdot\sqrt{\mathrm{SNR}}})(\mathrm{erf}(\frac{W}{\sqrt{2}})-1)\Bigg).
\end{aligned}$$

□

5.3.3 Entropy After Filtering Out Unreliable Bits

The proportion of unreliable bits is given by

$$\begin{aligned}\mathbb{P}(Bit\ unreliable) = \mathbb{P}(|\Delta| < Th) &= \mathrm{erf}(\frac{Th}{\sqrt{2\Sigma}}) \\ &= \mathrm{erf}(\frac{W}{\sqrt{2\mathrm{SNR}}}).\end{aligned} \tag{5.7}$$

In other words, the average remaining entropy of a circuit with n elements (thus, of complexity proportional to n) is equal to

$$H(n, W)_{SNR} = n \cdot \mathrm{erfc}(\frac{W}{\sqrt{2\mathrm{SNR}}}). \tag{5.8}$$

With this method, it is necessary to increase the number of elements to generate a given entropy. The expected number of elements n, with $n > h$, to consider in order to obtain h bits of entropy is given by:

$$n = \frac{h}{1 - \mathbb{P}(Bit\ unreliable)} = \frac{h}{\mathrm{erfc}(\frac{W}{\sqrt{2\mathrm{SNR}}})}. \tag{5.9}$$

Figure 5.6 represents the average remaining entropy for a circuit, depending on the SNR and the target BER. This characterizes the tradeoff between reliability and entropy.

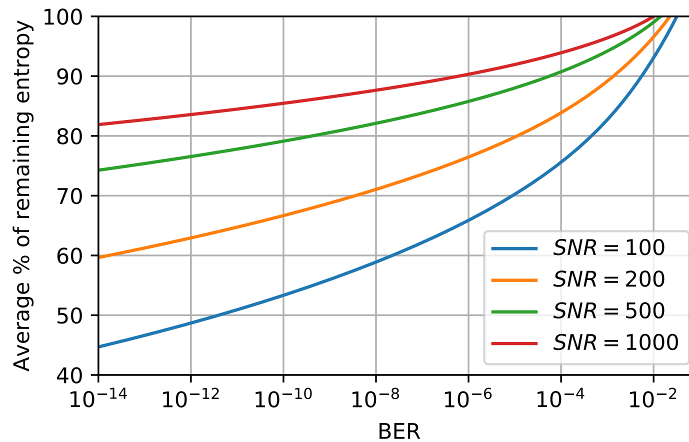


Figure 5.6: Remaining average entropy after filtering unreliable bits as a function of the BER to reach.

If ever it is not possible to reach the required entropy, the device is discarded. The probability of this happening can also be computed. Since the obtained delays are independent (when choosing a Hadamard matrix for the challenges), the number of unreliable bits is given by a binomial distribution $\mathcal{B}(n, \mathrm{erf}(\frac{W}{\sqrt{2\mathrm{SNR}}}))$ on a PUF with n elements. Thus,

$$
\begin{aligned}
p_{discard} &= \sum_{i=n-h+1}^{n} \binom{n}{i} \operatorname{erf}\Big(\frac{W}{\sqrt{2\mathrm{SNR}}}\Big)^{i} \operatorname{erfc}\Big(\frac{W}{\sqrt{2\mathrm{SNR}}}\Big)^{n-i} \\
&= I_{p_d}(n-h+1, h) \qquad (5.10)
\end{aligned}
$$

where $I_x(a, b)$ is the regularized incomplete beta function and p_d is the probability of discarding a bit, $p_d = \operatorname{erf}(\frac{W}{\sqrt{2\mathrm{SNR}}})$.

5.4 The “Two-Metric” Method

5.4.1 Motivation and Definition

Instead of extracting one PUF bit from the delay difference δ_C as the *sign* of δ_C, it can also be done by considering the interval between the first and third quartile of the Δ_C distribution:

$$b = \mathbb{1}\{\delta_C \in [\delta_{1/4}, \delta_{3/4}] = [-a\Sigma, +a\Sigma]\}$$

where $a = 0.674489...$ Let us call this method the “metric” M0.

Figure 5.7 with $\Sigma = 1$ illustrates the bit extraction by the method M0 according to the value of Δ. The entropy is still of one bit with this metric M0 as the area for bit=0 and bit=1 is the same, namely 1/2.

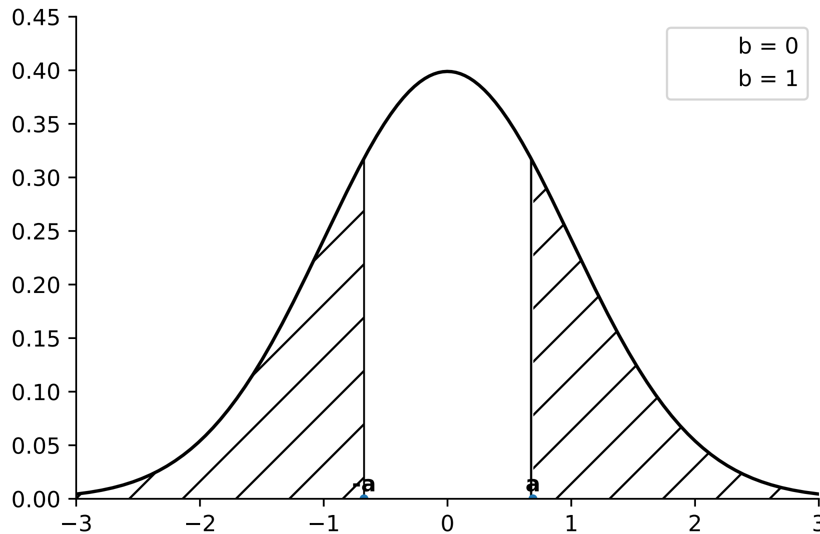


Figure 5.7: Metric M0: Bit extraction according to a and the pdf of Δ

But this alternative extraction method does improve neither the security against the helper data attack nor the reliability as there is still uncertainty around the values $-a$ and a.

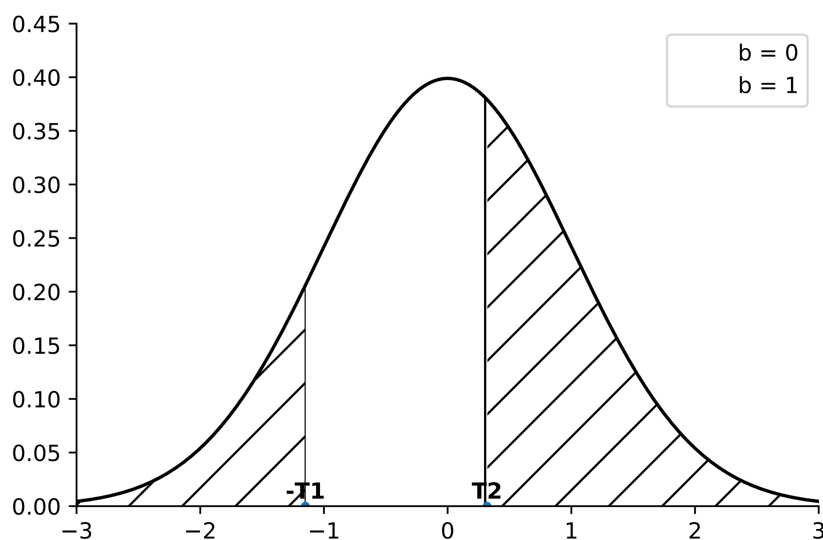


Figure 5.8: Metric M1: Bit extraction according to $-T_1, T_2$ and the pdf of Δ_C

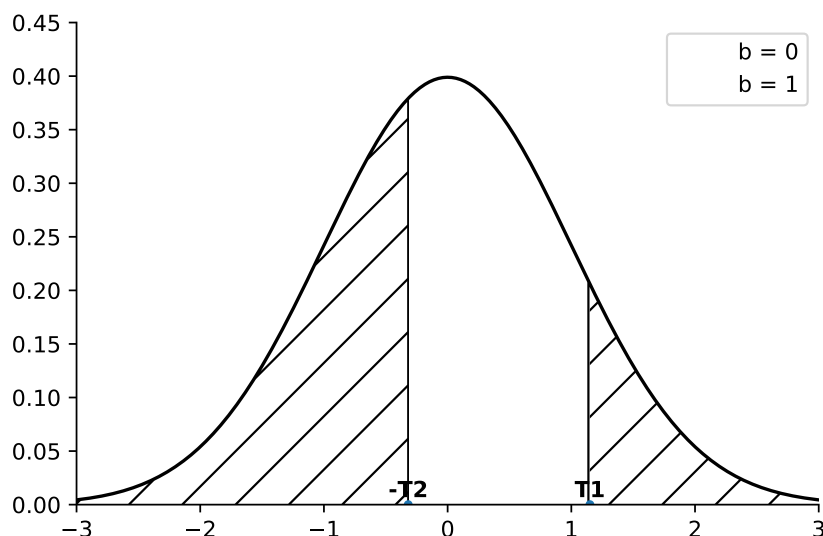


Figure 5.9: Metric M2: Bit extraction according to $-T_2, T_1$ and the pdf of Δ_C

Let us consider now two metrics M1 and M2 as illustrated by Fig. 5.8 and Fig. 5.9 respectively. These metrics use two constants, $T_1 > T_2$, such that

$$\int_{-T_2}^{T_1} \phi(x)\,\mathrm{d}x = \int_{-T_1}^{T_2} \phi(x)\,\mathrm{d}x = \frac{1}{2}$$

and

$$\int_{0}^{T_2} \phi(x)\,\mathrm{d}x = \frac{1}{8}.$$

Hence, as shown in Fig. 5.10, there are eight octiles in the pdf of Δ which are:

$$[-\infty, -T_1[, [-T_1, -a[, [-a, -T_2[, [-T_2, 0], \\ ]0, T_2],]T_2, a],]a, T_1],]T_1, +\infty].$$

Using this observation, we can now define a more refined bit extraction method, dubbed the "two-metric" method.

During the PUF enrollment phase, if we determine that the PUF response would be unreliable with metric M1, i.e. δ_C is around $-T_1$ or T_2, we choose the metric M2. On the other hand, if δ_C is around $-T_1$ or T_2, we choose the metric M2. The helper data now indicates the type of metric rather than the reliability. More formally, as $\int_0^{T_2} \phi(x)\,\mathrm{d}x = \int_{T_2}^{a} \phi(x)\,\mathrm{d}x$, the choice of the metric is given by Alg. 5.4.1 and summarized in Fig. 5.10.

Algorithm 5.4.1 Proposed PUF enrollment

Input: Threshold a, related to the expected reliability
Output: Vector of metric choices $M \in \{M1, M2\}$
  $\mathsf{metric} \in \{M1, M2\}^N$
  for $i \in \{1, \dots, N\}$ **do** $\Delta \leftarrow \mathsf{PUF}(c_i)$
    if $\Delta \in [-\mathsf{a}, 0]$ or $\Delta > \mathsf{a}$ **then**
      $\mathsf{metric}_i \leftarrow M1$
    else
      $\mathsf{metric}_i \leftarrow M2$
    end if
  end for
  return metric

5.4.2 Reliability of the The "Two-Metric" Method

Supposing that the initial measurement Δ is done without noise, we can deduce from Fig. 5.10 that the challenges with the highest bit error rate BER are those where Δ is either close to 0, or around $\pm a$. Computing those error rates, we find that the challenges with Δ around 0 are actually those with the highest BER.

In the general case, where a, T_1 and T_2 must be multiplied by Σ, the probability to get a bit error is thus upper-bounded by

$$\begin{aligned} BER &< \int_{T_2\Sigma}^{+\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}\,\mathrm{d}x + \int_{T_1\Sigma}^{+\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}\,\mathrm{d}x \\ &< \frac{1}{2}\,\mathrm{erfc}(\frac{T_2\sqrt{SNR}}{\sqrt{2}}) + \frac{1}{2}\,\mathrm{erfc}(\frac{T_2\sqrt{SNR}}{\sqrt{2}}) \end{aligned}$$

For instance, for a typical value $\sqrt{SNR} = 15$ ($SNR \approx 220$, see Section 5.6), this yields

$$BER < 10^{-6}\,.$$

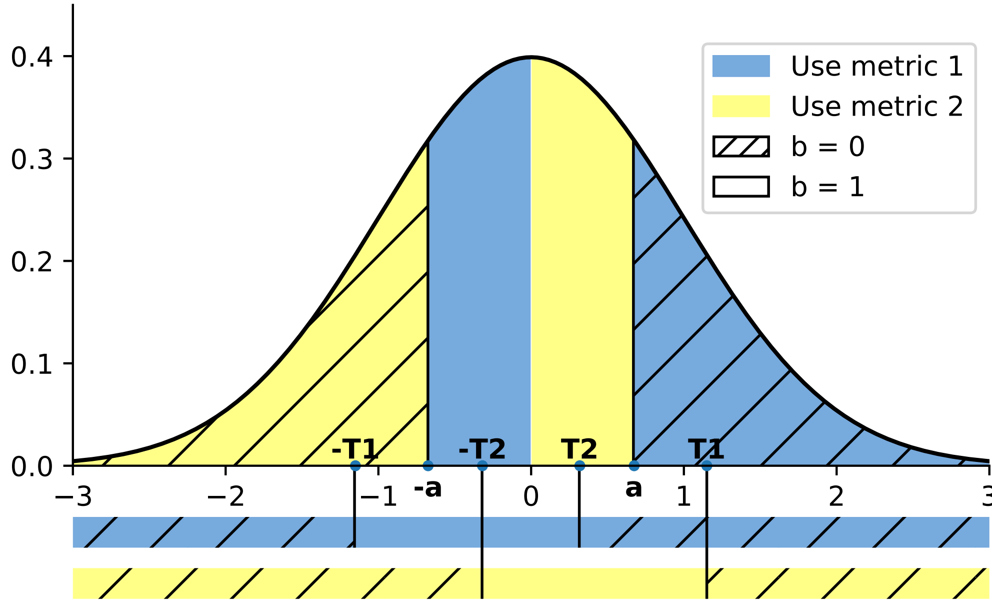


Figure 5.10: Choice of metric and extracted bit value.

A simulation shows that the average BER, for $\sqrt{SNR} = 15$, is actually less than 10^{-8}. The results of these simulations are shown in Fig. 5.11, and compared to the BER rates without helper data. These results have a relative margin of error of less than 1% for all estimates at the 95% confidence level[1].

5.4.3 Security

The Helper Data contains the choice of a metric amongst M1 and M2. The question is to know if an attacker can retrieve the key by inverting the metric of the Helper Data, as explained above and illustrated in Fig. 5.4. Here we assume the attacker changes the metric $metric_i$ of bit i from M1 to M2. Let us call bit^{-1}, the bit value before the attack and bit the bit value after the attack. The attacker can not retrieve any information if:

$$\mathbb{P}(bit = 0|bit^{-1} = 0) = \mathbb{P}(bit = 1|bit^{-1} = 1)$$

and

$$\mathbb{P}(bit = 1|bit^{-1} = 0) = \mathbb{P}(bit = 0|bit^{-1} = 1)$$

For the two-metric method, these probabilities are all equal to $\frac{1}{2}$ regardless of the value of the key bit when changing from one metric to the other. Therefore, there is no possibility for the attacker to gain any information about the key bits by changing the metric. Consequently the two-metric method is natively robust against such attacks.

[1] That is, for any BER estimate b, the confidence interval is included in the interval $[b - 0.01b, b + 0.01b]$.

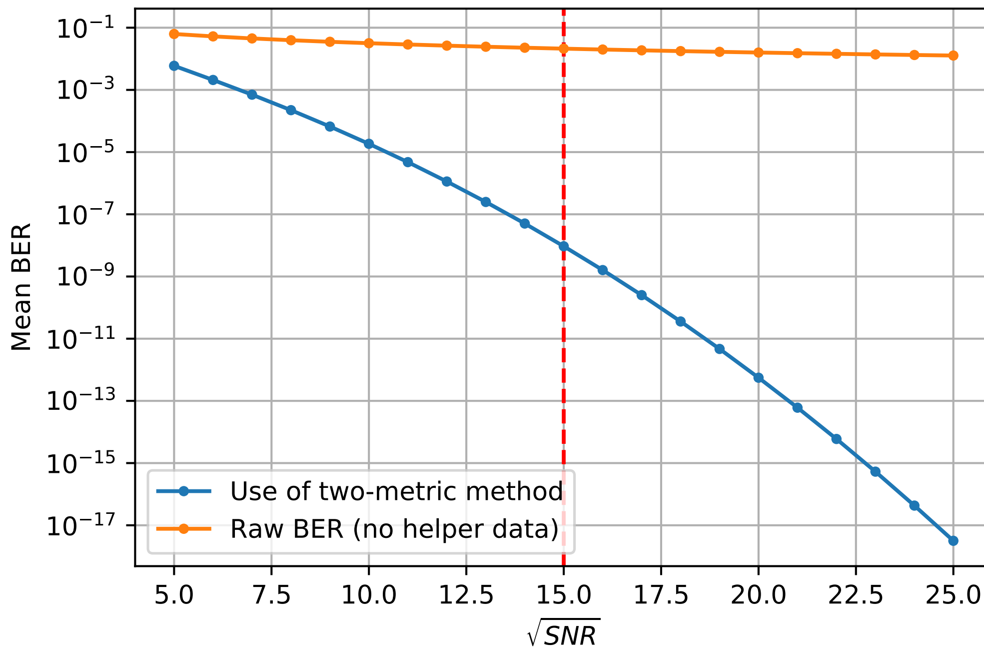


Figure 5.11: Average BER with and without two-metric helper data

5.4.4 Entropy

As all the bits are used, there is no loss of entropy as it is the case when using a single metric.

5.5 Translation for Various PUF Architectures

When applying our results on real PUF architectures, we must suppose that the output bits are independent and non-biased. By restricting the set of possible challenges, we can prove, under the assumption that the theoretical model describing the PUF is accurate, that the remaining output bits are indeed independent and non-biased. However, this is no longer true when the PUF behavior deviates from that predicted by its model. It is possible to ensure that the output bits are not biased and uniform using standard test suites, for instance those provided by NIST [115]. If these tests fail, bias and correlation can be corrected by applying fuzzy extractor techniques [116] prior to using reliability-enhancing methods.

5.5.1 RO-PUF

The RO-PUF has been first described by Suh and Devadas [91]. In the general case, it uses a certain number of oscillating loops for which the oscillation frequencies are measured and compared. In the setting that we will analyze, and that had already been described in this seminal work, we will use $2n$ ring oscillators to generate n bits. To describe this PUF

in our unified framework, we will define a challenge C as any n-bit string with Hamming weight exactly 1. If C^m is such that $c_i^m = 1$ iff $m = i$, then the delay difference δ_{C^m} will correspond to the frequency difference between oscillators $2m$ and $2m+1$. Thus, the δ_{C^m} will be mutually independent. Therefore, our framework can be directly applied in order to estimate the reliability-entropy tradeoff in case filtering is used.

5.5.2 RO sum PUF

The RO sum PUF, or recombined oscillator, has been proposed by Yu and Devadas [93]. Instead of comparing the oscillator frequencies, they are measured, added or subtracted, before one bit is generated from the sign. More precisely, the $2n$ oscillators are divided into n pairs. Let $C = (c_i)_i$ be a challenge of length n. If d_i is the delay difference for the two oscillators of the i-th pair, then the total delay is obtained as

$$\delta_C = \sum_{i=1}^{n} d_i(-1)^{c_i}.$$

Here, d_i should be modeled as a realization from a normal law, with variance Σ_0^2. Therefore, we will have that $\Delta_C \sim \mathcal{N}(0, n\Sigma_0^2 = \Sigma^2)$. There are 2^n possible challenges, however, the delays for all these challenges will not be independent. It has been shown by Rioul et al.[97], for a different PUF but the same delay model, that the challenges are mutually independent if, when converted to $\{\pm 1\}$ vectors instead of $\{0, 1\}$ vectors, they are orthogonal. We can therefore find a subset of n challenges that are independent if a Hadamard matrix of rank n exists. This is always the case if n is a power of two or a multiple of 4 smaller than 668 [117]. Assuming this is the case, we can choose any such subset of challenges for the n possible challenges. Our framework can then be applied to this PUF.

5.5.3 Loop PUF

The Loop PUF, described by Cherif et al. [92], strongly resembles the RO sum PUF, with the exception that one configurable ring oscillator is used, instead of $2n$ simple ROs for the RO sum PUF. For the Loop PUF, each RO comprises n configurable and balanced delay element pairs. During delay measurement, the signal only passes through one half of the delay elements, this half being determined by the input challenge. The same measurement is then done for the complementary challenge, so that the signal passes through the other half of the delay elements, and the delay difference is then computed. The mathematical model is thus very similar to that of the RO sum PUF, with some minor differences. For example, in the RO sum PUF, the delays for the individual ring oscillators are first quantified and then added, which might lead to some rounding errors. This is less the case for the Loop PUF, since a total delay is directly measured. Thus, there are only two delay quantifications for the Loop PUF.

As shown by Rioul et al. [97], in order to obtain independent delay differences, and thus independent bits, the challenges need to be orthogonal, in the same sense as before. Thus, an entropy of n bit can be obtained by choosing a $n \times n$ Hadamard matrix for the challenges, if a Hadamard matrix of this size exists.

5.6 Experiments and Validation with Real Silicon

In this section, results are presented only for bit filtering. Because the two-metric method is not "tunable" and the SNR of the tests circuits is high, too many measurements would have been necessary in order to obtain a sufficient number of bit-flips to reliably estimate the BER.

5.6.1 Architecture of the Test Circuit

We used Loop PUFs with $n = 64$ delay cells for our experiment. The cells use 65 nm CMOS technology, and each test chip contains 49 PUFs, embedded in a 7×7 matrix. We performed the delay measurements during $L = 2^{14}$ oscillation periods of the reference clock at $f_{ref} = 100$ MHz. This allows us to simulate:

- 49 Loop PUFs with 64 delay elements, or
- 64 RO-PUFs with 24 delay elements, or
- 64 RO sum PUFs with 48 delay elements.

Following [97], we choose a 64×64 Hadamard matrix as the challenge matrix to control the 49 Loop PUFs. The 64 challenge responses can therefore be considered independent. We perform $T = 1000$ measurements for each challenge and each PUF per chip. This directly yields the responses for the Loop PUF. In order to simulate a RO-PUF, we fix a challenge index m and consider the $24 \times T$ response delays:

$$\left\{\delta_{m,t}^{2j} - \delta_{m,t}^{2j-1},\ j \in [1, 24]\right\}.$$

In a similar fashion, for the RO sum PUF we choose a 48×48 Hadamard matrix $\widetilde{C}$. For a fixed challenge index m of the Loop PUF, we then obtain $48 \times T$ response delays:

$$\left\{\widetilde{C} \cdot \begin{pmatrix} \delta_{m,t}^{1} \\ \delta_{m,t}^{2} \\ \vdots \\ \delta_{m,t}^{48} \end{pmatrix}\right\}.$$

5.6.2 BER and Entropy Measurement

Results

Six test chips have been analyzed, and the measured BER and remaining entropy have been plotted in Fig. 5.12.

The error bars represent the range of values obtained among the tested chips. Although they do not share the exact same SNR, a middle value has been chosen, so that a simple comparison is possible. The SNR was calculated by estimating the variance of Δ_C and Z from the delay measurements of the test chips. Moreover, the range of measured SNRs is relatively small (between 180 and 250).

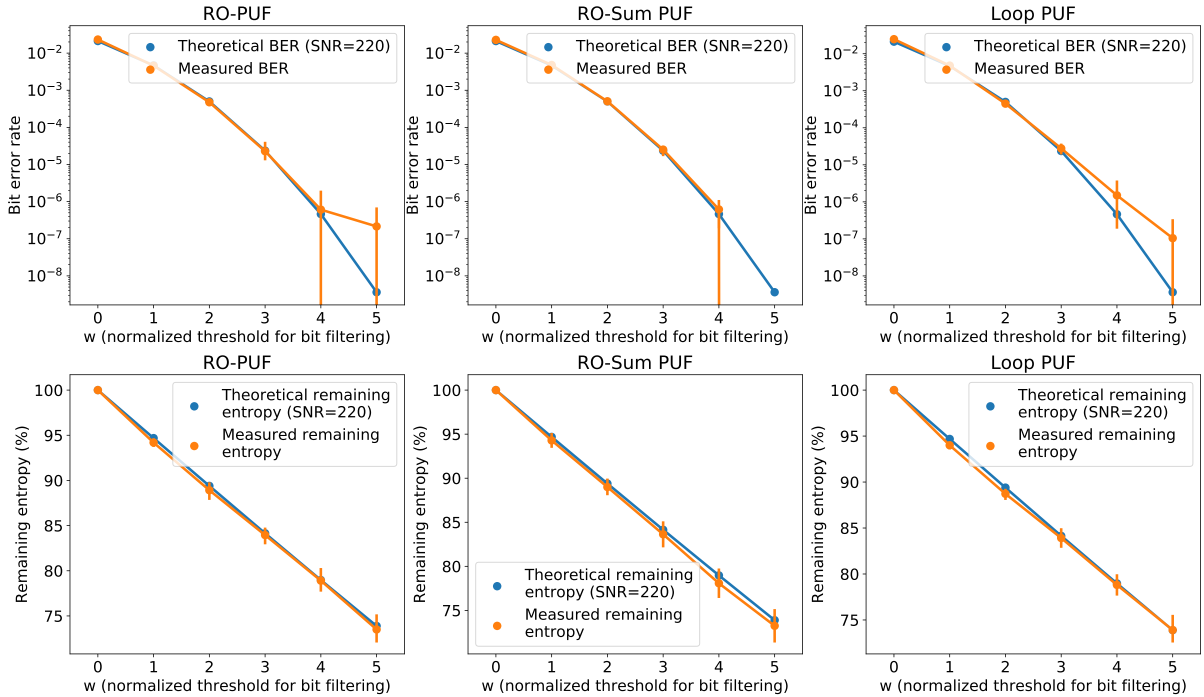


Figure 5.12: Experimental validation of the SNR and remaining entropy.

Discussion

For the remaining entropy, the measured and predicted values match quite closely. This seems to confirm the hypothesis of a Gaussian distribution for the average delay values. For the bit error rate however, the interpretation of the results seems more complicated. Indeed, while the BER for small filtering thresholds, and thus "large" BERs, seems to match our prediction, this is not the case for larger thresholds, at least for the RO-PUF simulation and the Loop PUF. We can see two explanations for this:

First, the sample size is probably not large enough to reliably estimate probabilities around 10^{-8}. Indeed, for each chip, we record about 3 million samples, and thus, even one bit error would yield a BER, for that circuit, of more than $3 \cdot 10^{-7}$. Therefore, the BERs for parameters $W \geq 4$ come with a fairly large uncertainty.

The small sample size does not explain everything, however. When further analyzing the delay measurements, we notice that the noise distribution does not perfectly follow a Gaussian distribution. Indeed, on some chips, we observe multiple measurements that are more than 7σ away from the computed mean delay value, as taken over 1000 measurements. This should not happen more than once in about 500 million measurements, if the noise was truly Gaussian. Thus we must admit that the noise is not exactly Gaussian. More exactly, it seems to be more heavy-tailed than a Gaussian noise. This could be an artifact of our experimental setup. Indeed, it forces us to wait a relatively long time span between measures, and the outliers could be explained, for example, with voltage fluctuations (the Loop PUF is relatively sensible to supply voltages changes). On the other hand, it should not come to a great surprise that a physical phenomenon does not exactly follow a Gaussian distribution. In order to derive a more precise model, other types of noise

distributions need to be considered.

One can further notice that the divergence from the expected BER is almost absent from the RO sum PUF simulation. This can be easily explained. A simulated delay measurement for the RO sum PUF corresponds to the sum of 48 independent Loop PUF delay measurements. If only one Loop PUF measure is an outlier relative to the expected Gaussian noise distribution, this will less affect the whole sum. This also explains why the RO-PUF exhibits less divergent behavior than the Loop PUF, as any outlier will be summed with another delay measurement. These results, however, are possibly artifacts of our experimental setup, if we suppose that external factors cause these outlier measures. Indeed, in a real RO-PUF or RO sum PUF, all measures would certainly be done in parallel, and might be affected by the same glitch at the same time. Therefore, this does not say anything about the intrinsic robustness of these three PUF types.

5.7 Effect of Environmental Changes: Temperature

PUFs are not necessarily used in the same environmental conditions they were enrolled at. Mainly two factors seem to be able to affect their behavior: temperature and input voltage [118]. We will assume that the input voltage can be controlled, via an voltage regulator for instance, and not further investigate in this direction. However, it is more complicated to control the temperature at which the PUF will be used, and it would therefore be helpful if it was possible to model the PUF-response dependency on temperature. In this section, we propose and test such a model. A similar model has been proposed by Maes [108], but for delay PUFs, it is possible to more directly test the model and make more straightforward predictions.

5.7.1 Assumptions

Given our experiments, we think that it is safe to make the following assumption: The delay response of a given oscillator is linearly dependent on the temperature, but the proportional constant might vary among ring oscillators. Testing on the Loop PUF circuits yielded a linear regression R^2 score above 0.999 for every oscillator. In addition, we will assume that this proportional constant follows a normal law. The curve for different oscillators in Figure 5.13 seems to validate this kind of distribution. Since in general, only the differences between ring oscillators are being considered (for the RO-PUF as well as the RO sum PUF), we can suppose that the probability distribution is centered. More formally, let's denote the temperature by θ, and the linear dependency coefficient by ℓ, where ℓ is a realization of a random variable $L \sim \mathcal{N}(0, \sigma_\theta)$. We therefore have the model for the temperature dependent PUF:

$$\mathcal{P}_\theta : C \mapsto \delta_C + Z + \ell\theta,\ b = \mathrm{sign}(\delta_C + Z + \ell\theta) \tag{5.11}$$

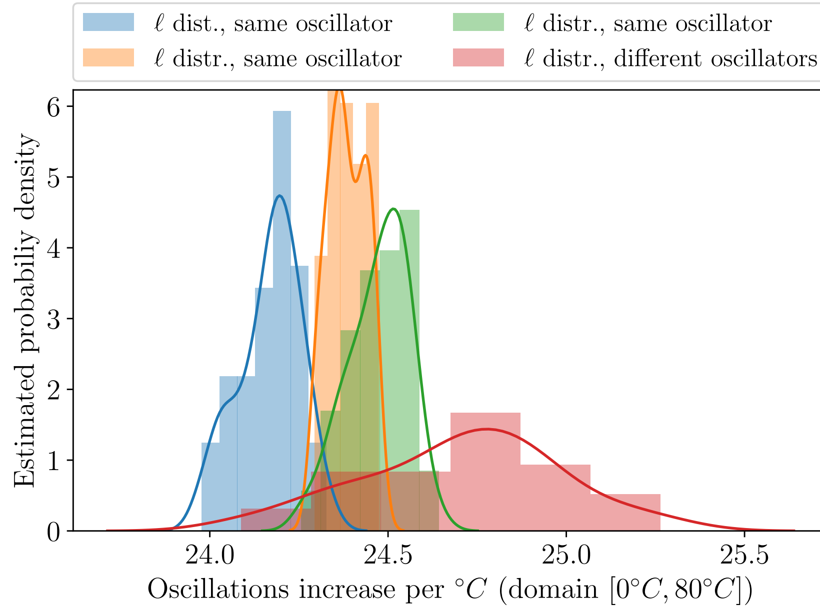


Figure 5.13: Distribution of temperature dependency coefficients (for 49 distinct oscillators, as well as 64 challenges of the same oscillator, for three different oscillators)

5.7.2 Average BER

We can now try to compute the average bit error rate over all average delays δ_C and dependency coefficients ℓ. As a reminder, the BER is defined here as

$$\widehat{BER}_\theta = P[\text{sign}(\Delta + Z + L\theta) \neq \text{sign}(\Delta)] \tag{5.12}$$

Since Z and $L\theta$ are two centered independent Gaussian random variables, with variance respectively σ^2 and $\theta^2\sigma_\theta^2$, the sum is also a Gaussian random variable with variance $\sigma^2 + \theta^2\sigma_\theta^2$. Therefore, the result for the average BER obtained in 5.3.1 can be directly applied, by replacing σ with $\sqrt{\sigma^2 + \theta^2\sigma_\theta^2}$:

$$\widehat{BER}_\theta = \frac{1}{\pi}\arctan\left(\frac{\sqrt{\sigma^2 + \theta^2\sigma_\theta^2}}{\Sigma}\right) \tag{5.13}$$

Thus, for the average BER, using the PUF at a temperature that is different from the enrollment temperature is equivalent to a loss of SNR. Of course, for individual delay measurements, this is not true, as the BER can exceed 0.5 if an inversion of the average sign happens due to the temperature difference, but it remains true for the average BER.

5.7.3 Effect on Delay PUFs

The RO-PUF and RO sum PUF are equally affected by the temperature dependency of the ring oscillators on the temperature. Indeed, for the RO-PUF, the delay difference is simply the difference of delay among two oscillators, and the model can be directly

applied as is. For the RO sum PUF, the total delay difference is actually the sum of a larger number of ring oscillator-pair delays. However, since the sum of Gaussian random variables still follows a Gaussian distribution, the same formula applies for the RO sum PUF, where σ, σ_θ and Σ are simply multiplied by the square root of the number of ring oscillator pairs. Since the average BER only depends on the ratio between these quantities, the BER formula is unchanged.

The case of the Loop PUF is a little different. Indeed, the delay differences are measured on the same oscillator, and different challenges should have a similar temperature dependency. However, as Figure 5.13 shows, this is not exactly the case. While the temperature dependency coefficients vary less between challenges of the same oscillator than between oscillators, the variance is not zero. The model seems also applicable to the Loop PUF, albeit with a lower standard deviation σ_θ.

5.7.4 Impact on the "Two-Metric" Method

The "two-metric" method is more sensitive to a change in the variance of the delays, as both the helper data generation and the bit extraction depend on it. One solution is to systematically assess the value of Σ before transforming the delay difference into a response bit. For instance the estimation can be carried out as per:

$$\widehat{\Sigma} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \Delta_i^2}$$

where N is the number of challenges and Δ_i the time difference of the Delay PUF measured for each challenge i.

Assuming a change in temperature and voltage only amounts to a change in Σ and σ, one can determine the new BER by simulation. The BER actually depends on the number of challenges, as shown in Fig. 5.14, because this number determines the precision of the estimate $\hat{\Sigma}$.

While the BER is increased in this setting, it stays below $5 \cdot 10^{-8}$ for $\sqrt{SNR}$ equal to 15 and 128 challenges, and below $6 \cdot 10^{-7}$ for 64 challenges.

Because Σ is estimated using noisy measurements, $\hat{\Sigma}$ will actually overestimate Σ. Indeed, it is easy to see that

$$\mathbb{E}[\hat{\Sigma}] = \sqrt{\Sigma^2 + \sigma^2}.$$

If the SNR is expected to be constant across different voltages and temperatures, this information could be used to improve the estimation of Σ. In any case, this requires more experiments to really assess the voltage and temperature impact on the SNR. Furthermore, because the value of Σ also influences the helper data generation, it is not easy to determine how the "two-metric" method is affected when enrollment and response generation are performed under different environmental conditions.

5.8 Conclusion

This chapter first presents the formalism to express the entropy and reliability of multiple delay PUFs: the RO-PUF, the RO sum PUF and the Loop PUF. We obtained a closed-

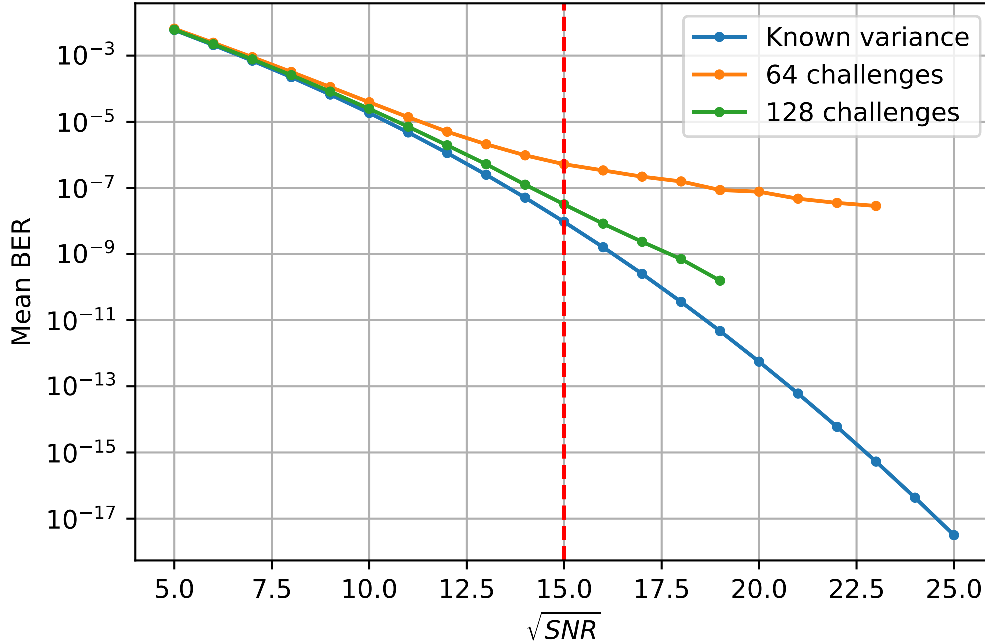


Figure 5.14: Average BER when Σ is known or unknown at measurement stage

form expression of the reliability which shows that the BER cannot go lower than about 10^{-3} even with large SNRs. The gain provided by the bit-filtering method that discards unreliable bits at enrollment phase has been formalized, giving a BER which can go to less than 10^{-10}.

The tradeoff between BER, entropy and complexity has been characterized. The resulting parameter selection for a given application is quite straightforward and simple. Practical experiments on few hundred PUFs designed in 65 nm CMOS process validate the theory. Testing the effect of temperature on the different types of PUFs is difficult when simulating with Loop PUFs. Tests with "native" PUFs might be necessary for a more thorough validation.

A second improvement technique, the “two-metric” method has also been proposed. This method is immune against helper-data tampering attacks and preserves the whole entropy while obtaining very low bit error rates. However, it is more sensitive to changes in environmental conditions. This poses an interesting challenge which, once solved, could make this method more practical to use.

The Gaussian model for process and noise variables are validated by these experiments up to a certain threshold. Beyond, the Gaussian model may not be valid at the far tail of the noise distribution, and an adequate model for the noise distribution is a subject for future work. Such a model would allow more efficiently designs of PUFs with very low error rates. In particular, this model for reliability (also sometimes termed steadiness) is a suitable metric for stochastic models being developed in ISO/IEC 20897 project [119].

5.A Verifying the Reliability of a PUF

Once a PUF has been designed, regardless of the methods employed to enhance its reliability, it is natural to verify that the key (or bit) error rate is sufficiently low for its intended application. In general, when given a PUF, the goal is to test whether its failure probability p is below some threshold ε. Suppose that we consider the PUF as a black box here: all we can do is request an identifier and check if the generated identifier is the expected one. How many times (say n) do we need to generate an identifier that is wrong no more than t times in order to be certain that the needed reliability is attained ?

Actually, as such, the problem is not well-posed. There is always the chance for an unreliable PUF to pass any stochastic testing, and for a reliable PUF to fail it. Therefore, we must specify the following parameters:

- What proportion of reliable PUFs ($p \leq \varepsilon$) are allowed to be rejected ? Let us denote this proportion with α.
- What proportion of unreliable PUFs ($p > \varepsilon$) are allowed to pass ? Let us denote this proportion with β.

Now, the problem is as a hypothesis testing problem, but it is *still* ill-posed. That is because for any stochastic testing method that takes finite time, the probability of rejecting a PUF is a *continuous function* of its failure probability p. However, if $\alpha < 1 - \beta$ (which should be the case for any reasonable choice of these parameters), then the rejection probability would be discontinuous for $p = \varepsilon$, which is not possible. The second requirement thus needs to be somewhat relaxed. The final, well-posed problem is thus the following: given ε and δ, α and β, such that

- a proportion α of reliable PUFs ($p \leq \varepsilon$) might be rejected and
- a proportion β of unreliable PUFs ($p > \varepsilon + \delta$) might pass,

how should the stochastic testing parameters n and t be chosen ?

5.A.1 Illustrating Example

For example, take $n = 10^7$ and $t = 120$, that is, 10^7 identifiers are generated by a PUF, and the PUF is rejected when there is a mismatch between the expected and generated identifier more than 120 times. The PUF rejection probability, as a function of its true error rate p, is computed in Figure 5.15. It shows a steep transition from a rejection rate of almost 0 to a rejection rate of almost 1 around $p = \frac{t}{n} = 1.210^{-5}$. For instance, a PUF with error probability 10^{-5} will be *rejected* with probability 2.27% while a PUF with error probability $1.5 \cdot 10^{-5}$ will *pass* with probability 0.66%.

5.A.2 Finding n and t

As shown in the previous example, for fixed ε, δ, n and t, it is easy to compute α and β. However, it is more useful to fix α and β, and then compute the corresponding n and t.

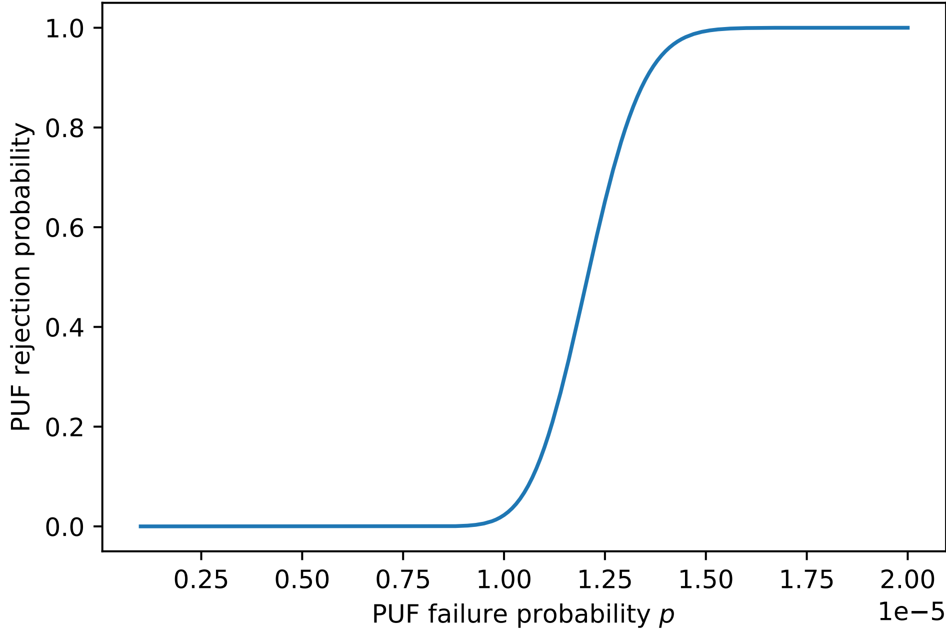


Figure 5.15: Rejection probability as a function of PUF error rate ($n = 10^7, t = 120$)

An exact solution is possible but it requires testing several values of n and t. For this procedure to be more efficient, it is better to start with a good approximate solution, and then exploring neighboring values. The simplest way to do so is to approximate the binomial distribution $\mathcal{B}(n, p)$ with the Gaussian distribution $\mathcal{N}(np, \sqrt{np(1-p)^2})$. Then, t can be fixed as a function of n.

In order to do so, the *z-score* corresponding to α needs to be computed first. In other words, we need to determine z_α such that

$$\mathbb{P}(\mathcal{N}(0,1) > z_\alpha) = \alpha$$

This value can be found in standard tables or computed using the inverse error function provided in numerous mathematics-oriented function libraries. When using standard tables, it is important to mention that most are *two-tailed* standard normal tables, while a *one-sided* z-score is needed here. This corresponds to the z-score for $2 * \alpha$ in a *two-tailed* standard normal table such as `https://www.sheffield.ac.uk/polopoly_fs/1.43999!/file/tutorial-10-reading-tables.pdf`.

Because of the scale invariance property of the normal distribution, we also have that

$$\mathbb{P}[\mathcal{N}(np, np(1-p)) > np + z_\alpha\sqrt{np(1-p)}] = \alpha.$$

Setting $p = \varepsilon$, the minimum number of failures to tolerate among n tries in order to reject less than a proportion α of reliable PUFs can be easily determined:

$$t \geq \lfloor n\varepsilon + z_\alpha\sqrt{n\varepsilon(1-\varepsilon)} \rfloor.$$

A similar reasoning can be applied to fix n given β. First, a value z_β needs to be determined that verifies

$$\mathbb{P}(\mathcal{N}(0,1) > z_\beta) = \beta.$$

We then have that

$$\mathbb{P}[\mathcal{N}(np, np(1-p)) < np - z_\beta\sqrt{np(1-p)}] = \beta.$$

Setting $p = \varepsilon + \delta$, we then find a second constraint for t that is required in order to pass no more than a proportion β of non-reliable ($p \geq \varepsilon + \delta$) PUFs:

$$t \leq \lceil n(\varepsilon+\delta) - z_\beta\sqrt{n(\varepsilon+\delta)(1-(\varepsilon+\delta))}\rceil$$

Ignoring the fact that both t and n need to be integers, we can solve for n and find an approximation as

$$n \approx \frac{1}{\delta^2}\left(z_\alpha^2\varepsilon(1-\varepsilon) + z_\beta^2(\varepsilon+\delta)(1-\varepsilon-\delta) + z_\alpha z_\beta\sqrt{\varepsilon(1-\varepsilon)(\varepsilon+\delta)(1-\varepsilon-\delta)}\right)$$

As an example, for $\varepsilon = 10^{-5}, \delta = 10^{-6}, \alpha = \beta = 1\%$, we get $z_\alpha = z_\beta \approx 2.326$ and $n = 227168250, t = 2382$. The graph for those parameters provided in Figure 5.16. The passing and rejection probabilities are respectively 1.044% and 0.9574%, which is quite close to the target values of α and β.

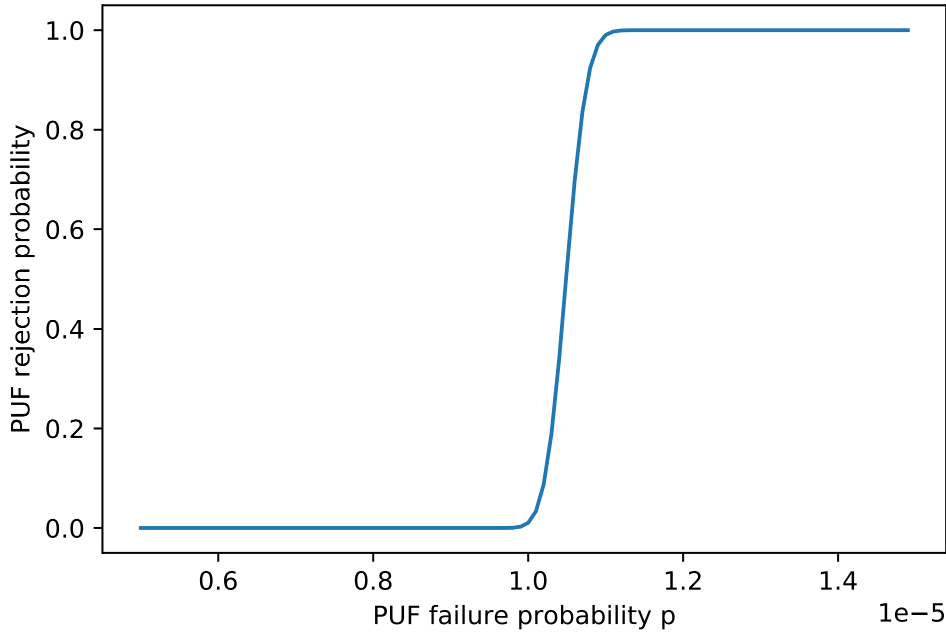


Figure 5.16: Rejection probability as a function of PUF error rate ($n = 227168250, t = 2382$)

Chapter 6

Entropy Estimation of Physically Unclonable Functions via Chow Parameters

Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.

John von Neumann

This chapter merges the contributions published in collaboration with Olivier Rioul, Jean-Luc Danger, Sylvain Guilley and Joseph J. Boutros.

A physically unclonable function (PUF) is an electronic circuit that produces an intrinsic identifier in response to a challenge. These identifiers depend on uncontrollable variations of the manufacturing process, which make them hard to predict or to replicate. Various security protocols leverage on such intrinsic randomness for authentication, cryptographic key generation, anti-counterfeiting, etc. Evaluating the entropy of PUFs (for all possible challenges) allows one to assess the security properties of such protocols.

In this chapter, we estimate the probability distribution of certain kinds of PUFs composed of n delay elements. This is used to evaluate relevant Rényi entropies and determine how they increase with n. Such a problem was known to have extremely high complexity (in the order of 2^{2^n}) and previous entropy estimations were carried out up to $n = 7$. Making the link with the theory of Boolean threshold functions, we leverage on the representation by Chow parameters to estimate probability distributions up to $n = 10$. The resulting Shannon entropy of the PUF is close to the max-entropy, which is asymptotically quadratic in n.

6.1 Introduction

Suppose we are given a (nonlinear) (n, M) code C with M codewords $c_i \in \{\pm 1\}^n$ and n i.i.d. standard Gaussian variables $X_1, X_2, \ldots, X_n \sim \mathcal{N}(0, 1)$. Let $X = (X_1, X_2, \ldots, X_n)$

and consider the scalar products

$$c_i \cdot X = \sum_{j=1}^{n} c_{i,j} X_j \qquad (i = 1, 2, \ldots, M) \tag{6.1}$$

and the associated sign bits

$$B_i = \operatorname{sign}(c_i \cdot X) \in \{\pm 1\} \qquad (i = 1, 2, \ldots, M). \tag{6.2}$$

The question addressed in this chapter is the following: What is the joint entropy of the sign bits

$$H(C) = H(B_1, B_2, \ldots, B_M)? \tag{6.3}$$

In particular, can we evaluate the *maximum entropy* $H(n) = \max_C H(C)$ attained for the full universe code $C = \{\pm 1\}^n$? Despite appearances, this problem turns out to be largely combinatorial as shown below.

6.1.1 Notations and Definitions

Definition 11 (Challenge code)**.** Let $n > 0, M > 0$ be two integers. A (n, M) *challenge code* C is a subset $C \subseteq \{-1, +1\}^n$ of cardinality M. The elements of this subset are called *codewords*, and the i-th codeword is denoted by c_i. By an abuse of notation, we identify the challenge code with the $n \times M$ matrix C, called the *challenge matrix*, which rows contain all codewords exactly once. The i-th row is c_i, and conversely, for any codeword $c \in C$, $i(c)$ denotes its row index.

The motivation for this problem comes from hardware security. Modern secure integrated circuits make use of hardware primitives called *physically unclonable functions* (PUFs) that can generate unique identifiers from challenges, such as described, for example, by Maes [120]. More formally, a PUF is a function that takes several challenges $c_1, c_2, \ldots, c_M$ (the so-called *challenge code*) as inputs and returns the bitvector identifier $(b_1, b_2, \ldots, b_M)$. PUFs exploit small, uncontrollable physical variations of the manufacturing process that cannot be replicated, hence the name "physically unclonable".

Definition 12 (Physically unclonable function (PUF))**.** Let C be an (n, M) challenge code. Let $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ such that the scalar product $c \cdot x \neq 0$ for all codewords $c \in C$. Then the *physically unclonable function* (PUF) with parameter x is the function $f^x : C \to \{-1, +1\}$ defined as

$$f^x(c) = \operatorname{sign}(c \cdot x), \tag{6.4}$$

where sign is the sign function and $\cdot$ denotes the usual scalar product. Equivalently, f^x is given by the *sign vector* $b = (b_1, b_2, \ldots, b_M) \in \{-1, +1\}^M$ such that

$$b_i = \operatorname{sign}(c_i \cdot x) \qquad (i = 1, \ldots, M). \tag{6.5}$$

The following notion of *randomized* PUF coincides with that of a PUF at a design stage, when it is not yet instantiated by a foundry fabrication process (cf. [121, Fig. 1]).

Definition 13 (Randomized PUF)**.** For a fixed (n, M) challenge code, we define the *random PUF* as f^X, where $X = (X_1, X_2, \ldots, X_n)$ and X_i are i.i.d. standard normal random variables $X_i \sim \mathcal{N}(0, 1)$.

The corresponding random sign vector is then $B = (B_1, \ldots, B_M)$, where

$$B_i = \operatorname{sign}(c_i \cdot X) \qquad (i = 1, \ldots, M) \tag{6.6}$$

with probability distribution

$$\mathbb{P}_b = \mathbb{P}[B = b] = \mathbb{P}[B_1 = b_1, B_2 = b_2, \ldots, B_M = b_M] \qquad (b \in \{-1, +1\}^M).$$

We denote $|\operatorname{supp}(\mathbb{P}_b)|$ the cardinality of the support of $\mathbb{P}_b$.

To assess the security of a PUF, it is necessary that the entropy of the identifier's distribution is sufficiently high. The most natural definition is the Shannon entropy, characterizing the uncertainty about the PUF distribution. Depending on the desired application, other kinds of entropies may be relevant. The most conservative view is to consider the *min-entropy* H_∞, which can be interpreted as the "cloning" entropy in the *worst* case, when the PUF to clone is obtained with probability $\max_{b \in \{\pm 1\}} \mathbb{P}_b$. When using a PUF to generate a key, the min-entropy also characterizes the security of the key, as shown for example in [122]. In other settings, the *collision entropy* allows for a more accurate security bound on the key derivation, as suggested by Skorski [123] and Dodis *et al.* [124]. It accounts for PUF uniqueness, since it is related to the probability that no two generated keys are the same. In contrast, the max-entropy H_0 has no obvious practical interest apart from being an easily computable upper-bound of the Shannon entropy (and of all other Rényi entropies).

Definitions for the different kinds aforementioned entropies are given below. Each depends on the choice of a challenge code C.

Definition 14 (Rényi entropies [125])**.** For $\alpha \geq 0$, the *Rényi entropy* of order α is defined as

$$H_\alpha(C) = \frac{1}{1 - \alpha} \log_2 \sum_{b \in \{\pm 1\}^M} \mathbb{P}_b^\alpha.$$

As special cases (taking the limits when α approaches 1 or infinity) we have

$$\begin{aligned}
H_0(C) &= \log_2 |\operatorname{supp}(\mathbb{P}_b)| && \text{(max-entropy)} \\
H_1(C) = H(C) &= \sum_{b \in \{\pm 1\}^M} \mathbb{P}_b \log_2 \frac{1}{\mathbb{P}_b} && \text{(Shannon entropy)} \\
H_2(C) &= -\log_2 \sum_{b \in \{\pm 1\}^M} \mathbb{P}_b^2 && \text{(collision entropy)} \\
H_\infty(C) &= -\log_2 \max_{b \in \{\pm 1\}^M} \mathbb{P}_b && \text{(min-entropy).}
\end{aligned}$$

A well-known property of the Rényi entropies is that H_α is non-increasing in α. Thus, for any code C, $H_\infty(C) \leq H_2(C) \leq H(C) \leq H_0(C)$. It is also easily seen that $H_2(C) \leq 2H_\infty(C)$.

Definition 15 (Full entropy)**.** For any $\alpha \geq 0$, we define the *full entropy* $H_\alpha(n)$ as the Rényi entropy for the $(n, 2^n)$ challenge code that contains all possible codewords.

The *full entropy* is highest among all codes, as shown in the following lemma.

Lemma 4 (Full entropy is maximal)**.** *For any $\alpha \geq 0$ and any challenge code C,*

$$H_\alpha(n) \geq H_\alpha(C).$$

Proof. We prove a stronger result: For any challenge matrix C of an (n, M) challenge code and challenge matrix C' of an $(n, M+1)$ challenge code where the first M lines are identical to C, $H_\alpha(C') \geq H_\alpha(C)$.

Let b be a sign vector associated with C such that $\mathbb{P}_b > 0$, and let b^+ (resp. b^-) the sign vector associated with C' equal to $(b_1, \ldots, b_M, 1)$ (resp. $(b_1, \ldots, b_M, -1)$). By definition of C', one has $\mathbb{P}_b = \mathbb{P}_{b^+} + \mathbb{P}_{b^-}$.

Assume $\alpha > 1$. To prove that $\mathbb{P}_b^\alpha \geq \mathbb{P}_{b^+}^\alpha + \mathbb{P}_{b^-}^\alpha$, consider $\frac{\mathbb{P}_{b+}}{\mathbb{P}_b}$ and $\frac{\mathbb{P}_{b-}}{\mathbb{P}_b}$. Since $0 \leq \frac{\mathbb{P}_{b+}}{\mathbb{P}_b}, \frac{\mathbb{P}_{b-}}{\mathbb{P}_b} \leq 1$, we know that $(\frac{\mathbb{P}_{b+}}{\mathbb{P}_b})^\alpha \leq \frac{\mathbb{P}_{b+}}{\mathbb{P}_b}$ and $(\frac{\mathbb{P}_{b+}}{\mathbb{P}_b})^\alpha \leq \frac{\mathbb{P}_{b+}}{\mathbb{P}_b}$. Therefore,

$$(\frac{\mathbb{P}_{b+}}{\mathbb{P}_b})^\alpha + (\frac{\mathbb{P}_{b-}}{\mathbb{P}_b})^\alpha \leq \frac{\mathbb{P}_{b+}}{\mathbb{P}_b} + \frac{\mathbb{P}_{b-}}{\mathbb{P}_b} = 1. \tag{6.7}$$

which implies $\mathbb{P}_{b^+}^\alpha + \mathbb{P}_{b^-}^\alpha \leq \mathbb{P}_b^\alpha$. Summing over all $\mathbb{P}_b$ we obtain

$$\sum_{b\in\{\pm1\}^M} \mathbb{P}_b^\alpha = \sum_{b\in\{\pm1\}^M} (\mathbb{P}_{b^+} + \mathbb{P}_{b^-})^\alpha \geq \sum_{b\in\{\pm1\}^M} \mathbb{P}_{b^+}^\alpha + \mathbb{P}_{b^-}^\alpha \geq \sum_{b\in\{\pm1\}^{M+1}} \mathbb{P}_b^\alpha. \tag{6.8}$$

The assertion follows by taking the logarithm on both sides of this inequality and multiplying by the negative constant $\frac{1}{1-\alpha}$.

The case $\alpha < 1$ is similar: The inequalities (6.7) and (6.8) are reversed because $x^\alpha \geq x$ for $x \in [0, 1]$, but the constant $\frac{1}{1-\alpha}$ is positive. Therefore, the same assertion follows. The cases $\alpha = 1$ and $\alpha = \infty$ are established by taking limits. □

Notice that the maximum entropy $H^\alpha(n)$ is always attained by a $(n, 2^{n-1})$ challenge code, by the following symmetry argument: since $\text{sign}(c \cdot x) = -\text{sign}((-c) \cdot x)$, the set $\{\pm1\}^n$ can be partitioned into two opposite sets where codewords in the second set bring no additional entropy. Indeed, adding a codeword c to a code C which already contains $-c$ does not change the probabilities of the sign vectors, only their labeling. This leaves all Rényi entropies unchanged. Therefore, it is possible to obtain the maximum entropy with any $(n, 2^{n-1})$ code $\mathcal{C}$ satisfying $c \in \mathcal{C} \implies -c \notin \mathcal{C}$.

Table 6.1 summarizes the notations used in the remainder of this chapter.

6.1.2 Motivation

Definitions 12 and 13 correspond to a particular PUF that exploits the variability of n distinct delay elements (a so-called "Loop PUF"), where $X_1, X_2, \ldots, X_n$ are independent *Gaussian delay* differences. This type of PUF has been first described by Cherif *et al.* [92]. A previous modelization of the Loop PUF, obtained via Monte-Carlo simulations of the possible circuit behaviors, showed a distribution of delays close to a Gaussian distribution,

Table 6.1: Summary of Notations.

Notation	Explanation
n	number of delay elements in a PUF
X_i	Gaussian random variable representing the delay difference of the i-th delay element ($i \in [1, n]$)
X	$X = (X_1, X_2, \ldots, X_n)$
x_i	realization of X_i
M	number of challenges
C	challenge code, a matrix defined by its rows $(c_i)_{i \in [1,M]}$
sign	$\text{sign}(x) = 1$ if $x > 0$, $\text{sign}(x) = -1$ if $x < 0$, and $\text{sign}(0) = 0$.
B_i	$B_i = \text{sign}(c_i \cdot X)$
B	$B = (B_1, B_2, \ldots, B_M)$
b_i	realization of B_i
b	realization of B
$\mathbb{P}_b$	$\mathbb{P}_b = \mathbb{P}[B = b]$

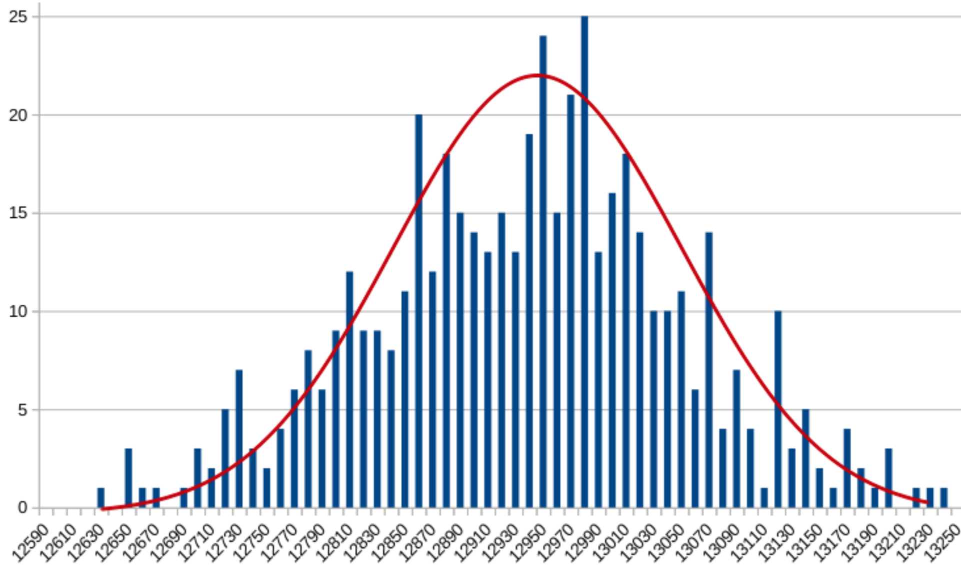


Figure 6.1: Distribution of delays obtained via circuit simulation

as shown in Figure 6.1. Other types of simulations also suggest a Gaussian distribution of process variations, and thus delay differences, in electronic circuits [99]. This motivates the choice of modeling the delay differences of the Loop-PUF as independent Gaussian variables.

Because they share the mathematical model with the Loop-PUF, definitions 12 and 13 also apply to the Arbiter PUF [94], for which the Gaussian model has been confirmed [126, 127], and to the RO-sum PUF [93].

These process variations can then be exploited in different ways. For example, it is possible to build authentication protocols based on PUFs: an authentication server queries a PUF via a set of challenges and checks the PUF answer against a whitelist. In this

way, counterfeit or overproduced chips can be detected. This requires no implementation of costly asymmetric cryptography primitives, and is therefore adapted to low-cost IoT devices. The PUF can also be used to generate a secret cryptographic key that is required for secure storage or communications with other devices. Using a PUF is more secure than directly storing the cryptographic key into memory, from where it might potentially be read or written by an attacker.

6.1.3 State of the Art

Results on the min-entropy

An upper bound of the min-entropy has been derived for the so-called RO-sum PUF by Delvaux *et al.* in [1]. Since this PUF shares the same mathematical bound as the Loop-PUF, this result is also relevant for our analysis. The following upper-bound is valid for odd values of n:

$$H_\infty(n) \leq -\log_2\left(\frac{1}{2}\left(1-\sqrt{\frac{n-1}{n}}\sum_{i=0}^{(n-3)/2}\frac{(2i)!}{(i!)^2(4n)^i}\right)\right). \tag{6.9}$$

This expression is not easy to interpret, but we have the following bound for practical values of n:

$$H_\infty(n) \leq 4n \quad \text{for } n \leq 251. \tag{6.10}$$

The min-entropy is therefore at most linear in n for $n \leq 251$. Because of the inequality $H_2 \leq 2H_\infty$, valid for any distribution, we deduce the following bound on the collision entropy:

$$H_2(n) \leq 8n \quad \text{for } n \leq 251. \tag{6.11}$$

Exact values for small n

Exact results for the entropy and probability distribution of the Loop-PUF have been obtained in certain special cases. Rioul *et al.* showed in [97] that the optimal challenge code when $M \leq n$ is given by a Hadamard code[1] C for which one can attain a uniform distribution of the Loop-PUFs, giving

$$H(C) = n.$$

The exact calculation of the PUF distribution of n delay elements for $M \geq n$ can be carried out only for very small values of n. Rioul *et al.* [97] give the exact values of the Loop-PUF distribution, and thus $H(C)$ for all $n, M \leq 3$ using well-known closed-form formulas for orthant probabilities of bi- and trivariate normal distributions (see Lemma 5).

Results on the max-entropy

The max-entropy $H_0(n)$ is simply the logarithm of the number of different Loop-PUFs of n delay elements. This number has been computed for small values of $n \leq 10$, because it

[1] When such a Hadamard code exists, which implies that $n = 1, 2$ or a multiple of 4.

actually corresponds to the number of so-called Boolean Threshold Functions (BTF) of $n-1$ variables. This number was determined up to $n = 8$ by Winder [128], up to $n = 9$ by Muroga *et al.* [129] and finally up to $n = 10$ by Gruzling [130]. Asymptotic estimates have also been published [131]. These results are recalled in Section 6.6.1.

Unfortunately, the quadratic behavior in n of the max-entropy $H_0(n)$ somehow overestimates the security of the PUFs, since it is much higher than the min-entropy, which is approximatively linear in n.

6.1.4 Our Contributions

In this work, we extend previous results in two directions.

First, we provide the exact values of the distribution of the Loop-PUF (for all possible challenges) for $n = 3$ and $n = 4$. This allows us to compute the exact values of all entropies in these cases. Such an exact computation comes as a surprise since no closed-form expression exists for the orthant probabilities of an M-dimensional Gaussian vector for $M \geq 4$. In our computation, we leverage on the discrete nature of the challenge code to determine these probabilities up to $M = 8$.

Second, we introduce a novel algorithm for the simulation of equivalence classes (SEC). The SEC algorithm also finds all equivalence classes of challenge codewords corresponding to the same value of joint probabilities $\mathbb{P}_b$. Interestingly, this problem is purely of discrete combinatorial nature. The actual values of the corresponding probabilities are then estimated by Monte Carlo simulation, which allows us to compute all relevant entropies. We provide the resulting values of the entropies $H_0(n)$, $H(n)$, and $H_2(n)$ up to $n = 10$.

The remainder of this chapter is organized as follows. Section 6.2 presents exact values of the distributions and entropies for the cases $n = 3$ and $n = 4$. Section 6.3 recalls results obtained from the study of Boolean threshold functions which allow to efficiently characterize the symmetries of the PUF distribution. The SEC algorithm is presented in Section 6.6 along with the entropies up to $n = 10$. Section 6.7 concludes.

6.2 Closed-form expressions

6.2.1 Preliminaries

In order to determine the closed-form expressions of the PUF distributions up to $n = 4$, we need the following lemmas.

Lemma 5 (Orthant probabilities for the bi- and trivariate normal distribution)**.** *Let $n > 0$, c_1 and c_2 two challenges, and $Y_1 = c_1 \cdot X, Y_2 = c_2 \cdot X$. Let $\rho = \frac{\mathbb{E}[Y_1 Y_2]}{n}$ the correlation coefficient of Y_1 and Y_2. Then*

$$\mathbb{P}[Y_1 > 0, Y_2 > 0] = \mathbb{P}_{++} = \frac{1}{4} + \frac{\arcsin(\rho)}{2\pi}. \tag{6.12}$$

Let c_3 be a third challenge vector and $Y_3 = c_3 \cdot X$, and denote the correlation coefficients between Y_i and Y_j by $\rho_{i,j} = \frac{\mathbb{E}[Y_i Y_j]}{n}$. Then

$$\mathbb{P}[Y_1 > 0, Y_2 > 0, Y_3 > 0] = \mathbb{P}_{+++} = \frac{1}{8} + \frac{\arcsin(\rho_{1,3}) + \arcsin(\rho_{1,2}) + \arcsin(\rho_{2,3})}{4\pi}. \tag{6.13}$$

The bivariate case was already known to Hermite [132]. The extension to the trivariate case is a lesser known extension and can be found, for instance, in [133]. A short proof of both formulas is given by Rioul *et al.* in [97].

Lemma 6 (Zero probabilities). *Let $b = (b_i)_{i\in[1;M]}$ be a sign vector. Then $\mathbb{P}_b = 0$ if and only if there exists $\alpha = (\alpha_1, \ldots, \alpha_M) \in \mathbb{R}^M \backslash \{0\}^M$ such that $\operatorname{sign}(\alpha_i) = b_i$ when $\alpha_i \neq 0$ and $\sum_{i=1}^M \alpha_i c_i = 0$.*

Proof. Suppose that such a vector α exists. There is at least one component that is different from 0. Without loss of generality, suppose that $\alpha_1 \neq 0$. We then have

$$c_1 = -\frac{1}{\alpha_1}\sum_{i=2}^{M} \alpha_i c_i.$$

In particular, this implies that

$$X \cdot c_1 = -\frac{1}{\alpha_1}\sum_{i=2}^{M} \alpha_i (c_i \cdot X).$$

Now, if $\forall i > 1,$ such that $\alpha_i \neq 0, \operatorname{sign}(\alpha_i) = \operatorname{sign}(c_i \cdot X)$, the sign of the right-hand side of the expression is the opposite sign of α_1. Thus, $\alpha_1 = -\operatorname{sign}(c_1 \cdot X)$, which contradicts our hypothesis.

Conversely, suppose that $\mathbb{P}_b = 0$. Therefore, the Gaussian vector $(c_i \cdot X)_i$ is degenerate, and its support is included in a sub-space of $\mathbb{R}^M$ of dimension $< M$. In particular, it is included in some hyperplane of equation $\sum_{i=1}^M a_i x_i = 0$, where the a_i are not all 0. Since $\mathbb{P}_b = 0$, this hyperplane is disjoint from the orthant defined by the signs of b, that is the set $x_1 b_1 > 0, x_2 b_2 > 0, \ldots, x_M b_M > 0$. Therefore, we have that all $a_i b_i$ have the same sign, that we can take positive. Since the support is included in the hyperplane defined before, we must have

$$\sum_{i=1}^{M} a_i (c_i \cdot X) = (\sum_{i=1}^{M} a_i c_i) \cdot X = 0$$

for all $X \in \mathbb{R}^n$, and therefore $\sum_{i=1}^M a_i c_i = 0$. By setting $\alpha_i = a_i$, the α_i have the same signs as the b_i and are not all 0, which concludes the proof. □

Lemma 7 (Equivalence classes). *Suppose that after permuting and/or changing the signs of certain* columns *of C, one obtains a matrix C' that can be obtained by permuting, and then optionally changing the signs, of certain* lines *from C. Denote the corresponding permutation of the lines by $\sigma \in S_M$, and the following change of signs of the lines by $s_i \in \{\pm 1\}^M$. Then for any sign vector $b = (b_i)_{i\in[1;M]}$, b has the same probability as*

$$b' = (s_1 b_{\sigma(1)}, s_2 b_{\sigma(2)}, \ldots, s_M b_{\sigma(M)}).$$

Such b and b' are then said to be in the same equivalence class, or simply equivalent.

Proof. Permuting the columns or changing corresponds to a permutation or sign changes of the X_i. For any $s = (s_1, s_2, \ldots, s_n) \in \{-1, +1\}^n$ and $\sigma \in S_n$, the joint distribution of $X = (X_i)_i$ and $(s_i X_{\sigma(i)})$ are the same. □

6.2.2 Case $n = 3$

By considering the challenge matrix $C_3 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & - \\ 1 & - & 1 \end{pmatrix}$, exact probabilities $\mathbb{P}_b$ can be derived by using the formula for trivariate Gaussian, recalled in Equation (6.13). This yields an entropy of

$$H(C_3) = -\Big(\frac{1}{4} - 3\frac{\arcsin\frac{1}{3}}{2\pi}\Big)\log\Big(\frac{1}{8} - 3\frac{\arcsin\frac{1}{3}}{4\pi}\Big). \tag{6.14}$$

For the matrix with four challenges $C_4 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & - \\ 1 & - & 1 \\ - & 1 & 1 \end{pmatrix}$ and the two sign vectors $+---$ and $-+++$, we have that

$$\mathbb{P}_{+---} = \mathbb{P}_{-+++} = 0.$$

By exploiting symmetries, it follows that eight sign vectors satisfy

$$\mathbb{P}_{++++} = \mathbb{P}_{++--} = \mathbb{P}_{+-+-} = \mathbb{P}_{+--+} = \mathbb{P}_{----} = \mathbb{P}_{--++} = \mathbb{P}_{-+-+} = \mathbb{P}_{-++-} = p$$

and for the six remaining sign vectors

$$\mathbb{P}_{+++-} = \mathbb{P}_{++-+} = \mathbb{P}_{+-++} = \mathbb{P}_{---+} = \mathbb{P}_{--+-} = \mathbb{P}_{-+--}.$$

Furthermore, by adding complementary challenges, we have that $p = p + 0 = \mathbb{P}_{+--+} + \mathbb{P}_{+---} = \mathbb{P}_{+--\cdot} = \frac{1}{8} - 3\frac{\arcsin\frac{1}{3}}{4\pi}$ using the generic formula for trivariate normal distributions.

These findings are summarized in the table below.

Table 6.2: Distribution for $n = 3$

Size of equivalence class	Probability per element
8	$\frac{1}{8} - 3\frac{\arcsin\frac{1}{3}}{4\pi}$
6	$\frac{\arcsin\frac{1}{3}}{\pi}$

Therefore,

$$H(C_4) = H(3) = \\ -\left(1 - 6\frac{\arcsin\frac{1}{3}}{\pi}\right)\log\left(\frac{1}{8} - 3\frac{\arcsin\frac{1}{3}}{4\pi}\right) - 6\left(\frac{\arcsin\frac{1}{3}}{\pi}\right)\log\left(\frac{\arcsin\frac{1}{3}}{\pi}\right). \tag{6.15}$$

6.2.3 Case $n = 4$

Similar techniques have been employed in order to compute entropies with $n = 4$. Because $\frac{\arcsin(\frac{1}{2})}{\pi}$ is a rational number (it is in fact equal to $\frac{1}{6}$), the results for $n = 4$ are much simpler, compared to the case $n = 3$.

In order to compute the distribution for the maximal challenge code, we first determine the distributions of smaller codes. Sign vectors with zero probability are determined

according to Lemma 6. Those of equal probability are found with the help of Lemma 7. Using recurrence relations between the probabilities, we are then able to deduce the sign vector distribution for larger codes, when adding one codeword each time.

The first four codewords that are chosen are the lines of a Hadamard matrix of order 4:

$$C_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ S1 & - & 1 & - \\ 1 & 1 & - & - \\ 1 & - & - & 1 \end{pmatrix}.$$

As recalled before, the sign vector distribution is uniform for this challenge matrix.

The results when adding one additional codeword are summarized below. For the sign vectors, it is understood that the opposite sign vectors are also present in each probability class.

- Additional codeword $(1\ 1\ 1\ -)$:

Probability per sign vector	Sign vectors
$\frac{11}{192}$	+++++,++−−+,+−+−+,+−−+−
$\frac{1}{192}$	++++−,++−−−,+−+−−,+−−++
$\frac{1}{32}$	++−++,++−+−,+−+++,+−++−,+−−−+,+−−−−
$\frac{1}{16}$	+++−+

- Additional codeword $(1\ 1\ -\ 1)$:

Probability per sign vector	Sign vectors
$\frac{10}{192}$	++++++,++−−+−,+−+−++,+−−+−+
$\frac{1}{192}$	+++++−,++++−+,++−−++,++−−−−, +−+−+−,+−+−−+,+−−+++,+−−+−−
$\frac{1}{32}$	+++−++,+++−+−,+−++++,+−++−+
$\frac{1}{64}$	++−+++,++−++−,++−+−+,++−+−−, +−−−++,+−−−+−,+−−−−+,+−−−−−

- Additional codeword $(1\ -\ 1\ 1)$:

Probability per sign vector	Sign vectors
$\frac{3}{64}$	+++++++,++−−+−+,+−−+−++,+−+−++−
$\frac{1}{192}$	+++++−+,+−+−+−−,+−−++++,+−+−+++, +−−+−−+,++−−−−+,++++++−,++++−++, +−−−−−−,++−−+−−,+−−−+−+,+−−+−+−, ++−−+++,+−−−++−,+−+−−+−,+−−−−++
$\frac{1}{96}$	+−−−+++,+−−−−−+,+−−−−+−,+−−−+−−
$\frac{1}{64}$	++−+−++,+−++−++,+++−+−+,+++−++−, ++−++−+,+−++++−,+−++−+−,++−+−−+, +++−+++,+−+++++,+++−+−−,++−++++

- Additional codeword ($-$ 1 1 1): This code maximizes the entropies. As there are too many sign vectors per equivalence class to enumerate them all, we give here only their numbers.

Probability per sign vector	Number of sign vectors
$\frac{1}{24}$	8
$\frac{1}{192}$	64
$\frac{1}{96}$	32

In summary for $n = 1, 2, 3, 4$:

Table 6.3: Exact entropies for $n \leq 4$

n	1	2	3	4
$H(n)$	1	2	3.6655...	6.2516...
$H_0(n)$	1	2	3.8073...	6.7004...
$H_2(n)$	1	2	3.54615...	5.71049...
$H_\infty(n)$	1	2	3.20858...	4.58496...

6.3 The Chow Parameters of PUFs

In the remainder of this section, we will consider PUFs for which the challenge code $\mathcal{C}$ contains *all* possible challenges. Seen as boolean functions, they correspond to a specific type of so-called boolean threshold functions (BTF).

Definition 16 (Self-dual BTF [134])**.** Let $x \in \mathbb{R}^n$ be such that for all $c \in \{\pm 1\}^n$, $c \cdot x \neq 0$. The self-dual BTF of *size* n and *weight* sequence x is the function $f_x : \{-1, +1\}^n \to \{-1, +1\}$ defined as

$$f_x(c) = \operatorname{sign}(c \cdot x) \tag{6.16}$$

where $c \cdot x = \sum_{i=1}^{n} c_i x_i$ is the usual scalar product.

This definition coincides with that of a PUF with n variables. In the remaining sections of this chapter, we will thus use the term "PUF" to mean "self-dual BTF". BTFs have been studied since the 1950's as building blocks for Boolean circuits [135] and also find applications in machine learning [136]. Leveraging the correspondence between PUFs and BTFs, we adapt fundamental results from BTF theory to conveniently characterize PUFs.

6.3.1 All PUFs are Attainable

Recall that in our framework, the PUF parameters $x \in \mathbb{R}^n$ are realizations of a random vector $X \in \mathbb{R}^n$. Under this probabilistic model a PUF becomes a randomized mapping f_X such that $f_X(c) = \operatorname{sign}(c \cdot X)$ for any (deterministic) challenge $c \in \{\pm 1\}^n$.

Lemma 8. *For every PUF f_x, we have $\mathbb{P}(f_X = f_x) > 0$.*

In other words, every PUF f_x can be reached by a realization of weights X with positive probability (even though one has $\mathbb{P}(X = x) = 0$).

Proof. By assumption all components of X are i.i.d. with symmetric density of support S containing 0. Hence the support S^n of the density of X is an n-dimensional manifold containing the origin in its interior. Let $x \in \mathbb{R}^n$ be fixed and let C_x be the cone (scale-invariant set) of all $y \in \mathbb{R}^n$ such that $f_x = f_y$. This cone C_x has apex 0 and contains the intersection of all half-spaces $\{y \mid \text{sign}(c \cdot y) = \text{sign}(c \cdot x)\}$ where $c \in \{\pm 1\}^n$. Therefore, it is a n-dimensional manifold which intersects S^n with positive volume. Hence $\mathbb{P}(f_X = f_x) = \mathbb{P}(X \in C_x \cap S^n) > 0$. □

6.3.2 Chow Parameters Characterize PUFs

First introduced by Chow [137] and later studied by Winder [135] who gave them their name, the so-called Chow parameters uniquely define a Boolean threshold function. Their definition is especially simple for PUFs:

Definition 17 (Chow parameters)**.** The *Chow parameters* $p = (p_1, \dots, p_n) \in \mathbb{Z}^n$ of a PUF f of size n is defined as

$$p = \sum_{c \mid f(c)=1} c \tag{6.17}$$

where the vector sum is carried out componentwise.

We remark that for $n \geq 2$, all Chow parameters are *even integers*. This is due to the fact that a sum of even number of elements ± 1 must be even. More precisely,

$$\begin{aligned} p_i \bmod 2 &\equiv \sum_{c \mid f(c)=1} c_i \bmod 2 \equiv \sum_{c \mid f(c)=1} 1 \bmod 2 \\ &\equiv 2^{n-1} \bmod 2 \\ &\equiv 0 \bmod 2. \end{aligned}$$

Theorem 6.3.1 (Chow's theorem [137])**.** *Two PUFs with the same Chow parameters are identical.*

For completeness, we give a new proof of Chow's theorem rewritten in our PUF framework. Such proof turns out to be very simple.

Proof. Let f_x and f_y be two PUFs with identical Chow parameters:

$$\sum_{c \mid f_x(c)=1} c = \sum_{c \mid f_y(c)=1} c. \tag{6.18}$$

Simplifying this expression by $\sum_{\substack{c \mid f_x(c)=1, \\ f_y(c)=1}} c$, we obtain

$$\sum_{\substack{c \mid f_x(c)=1, \\ f_y(c)=-1}} c = \sum_{\substack{c \mid f_x(c)=-1, \\ f_y(c)=1}} c, \tag{6.19}$$

which is equivalent to

$$\sum_{c|f_x(c)\neq f_y(c)} f_x(c)c = 0. \tag{6.20}$$

Taking the scalar product with x, we get

$$\sum_{c|f_x(c)\neq f_y(c)} f_x(c)c \cdot x = \sum_{c|f_x(c)\neq f_y(c)} |c \cdot x| = 0 \tag{6.21}$$

which implies $c{\cdot}x = 0$ whenever $f_x(c) \neq f_y(c)$. Now we assumed that $c \cdot x$ is never zero by Def. 16. Thus $f_x = f_y$. □

6.3.3 Consequence on the Max-Entropy

An upper bound on the max-entropy can be easily deduced from Chow's theorem.

Corollary 1. *There are no more than 2^{n^2} PUFs of size n, i.e., the max-entropy of the PUF of size n satisfies*

$$H_0(n) \leq n^2 \qquad (\forall n \geq 2). \tag{6.22}$$

A more refined version, which can be rewritten as $H_0(n) \leq (n-1)^2 + 1$ for $n > 1$, can be found in [138, Corollary 10.2]. The proof of (6.22) is again particularly simple for PUFs.

Proof. The Chow parameters p_i, $i = 1 \ldots n$, satisfy

$$p_i = \sum_{c|f(c)=1} c_i \leq \sum_{\substack{c|f(c)=1,\\ c_i=1}} 1 \leq 2^{n-1} \tag{6.23}$$

and similarly, $p_i \geq -2^{n-1}$. Since there are $2^{n-1}+1$ even integers between -2^{n-1} and 2^{n-1}, there can only be $(2^{n-1}+1)^n \leq 2^{n^2}$ different values taken by the Chow parameters. The conclusion follows from Chow's Theorem 6.3.1. □

A lower bound on H_0 is also easily found from the representation of Definition 16, as given by the following Proposition. The corresponding bound for the number of BTFs was first established independently by Smith [139] and Yajima et al. [140] in the 1960s.

Proposition 1. *The max-entropy satisfies*

$$H_0(n) > \frac{(n-2)^2}{2} \quad (\forall n \geq 2). \tag{6.24}$$

Proof. Recall from Lemma 8 that every PUF f_x can be reached by a realization of weights X with positive probability. Hence it is sufficient to consider all f_x for all $x \in \mathbb{R}^n$ in order to lower-bound the total number of PUFs.

Let f_x a PUF of size n. Applying some small perturbation on x if necessary (without affecting f_x) we may always assume that all the $c \cdot x$ ($c \in \{\pm 1\}^n$) take distinct values.

Now let $x_{n+1} \in \mathbb{R}$ be such that $2x_{n+1}$ is different from all the $c \cdot x$, and define $x' = (x_1, \cdots, x_{n-1}, x_n - x_{n+1}, x_{n+1})$. For any challenge $c' = (c_1, \ldots, c_n, c_{n+1})$, we have

$$f_{x'}(c') = \begin{cases} f_x(c_1, \ldots, c_n) & \text{if } c_n = c_{n+1} \\ \text{sign}(\sum_{i=1}^{n} c_i x_i - 2c_n x_{n+1}) & \text{otherwise.} \end{cases} \tag{6.25}$$

Depending on how many of the 2^{n-1} values of $c \cdot x$ are smaller/larger than $2c_n x_{n+1}$, we can construct $2^{n-1} + 1$ different PUF functions of size $n + 1$. Hence each PUF of size n gives rise to more than 2^{n-1} PUFs of size $n + 1$. Therefore, $H_0(n + 1) > n - 1 + H_0(n)$. The result follows by finite induction:

$$H_0(n) > \frac{(n-1)(n-2)}{2} + H_0(2) > \frac{(n-2)^2}{2}.$$

□

More recently, Zuev [131] has shown that, asymptotically, $H_0(n) > n^2(1 - \frac{10}{\ln(n)})$. Therefore, for the max-entropy, we have that $H_0(n) \sim n^2$. As a result, instead of evaluating the probabilities of 2^{2^n} different PUFs, we will only have to evaluate about 2^{n^2}.

As apparent in the proof of Zuev [131, Theorem 1] although through different geometrical considerations on normal vectors of hyperplanes, we can further reduce the number of PUFs to be considered down by a factor of about $2^n n!$. Section 6.4 will derive the exact compression factor using the equivalence classes on Chow parameters.

6.3.4 Order and Sign Stability of Chow Parameters

An important property of the Chow parameters p is that their share the same signs and relative order as the weights x.

Lemma 9. *Let $f = f_x$ be a PUF with weight $x \in \mathbb{R}^n$, and $p \in \mathbb{Z}^n$ be the corresponding Chow parameters. Then*

- $x_i \geq 0 \implies p_i \geq 0$ *and* $x_i \leq 0 \implies p_i \leq 0$.
- $x_i \leq x_j \implies p_i \leq p_j$.

A similar result was shown by Chow in [137], although with another definition of Chow parameters. Again we give a simplified proof in the PUF framework.

Proof. We first prove that $x_i \geq 0 \implies p_i \geq 0$, the other case $x_i \leq 0 \implies p_i \leq 0$ being similar. Suppose that $x_i \geq 0$. Let E_i^+ (resp. E_i^-) be the set $\{c \mid f(c) = 1, c_i = 1\}$ (resp. $\{c \mid f(c) = 1, c_i = -1\}$). By definition,

$$p_i = \sum_{c|f(c)=1} c_i = |E_i^+| - |E_i^-|. \tag{6.26}$$

We show the existence of an injective mapping from E_i^- to E_i^+. Consider the one-to-one mapping $\phi : \{\pm 1\}^n \to \{\pm 1\}^n$ defined by

$$\phi(c)_j = \begin{cases} c_j, & j \neq i \\ -c_j, & j = i \end{cases} \tag{6.27}$$

For any $c \in E_i^-$, $c_i = -1$, $\phi(c)_i = +1$ and

$$\begin{aligned}\sum_{j=1}^{n} \phi(c)_j x_j &= \sum_{j \neq i}^{n} c_j x_j + x_i \\ &= \underbrace{\sum_{j=1}^{n} c_j x_j}_{>0} + \underbrace{2x_i}_{\geq 0} > 0.\end{aligned}$$

Therefore, $f(\phi(c)) = 1$ and $\phi(c) \in E_i^+$. Hence, the bijection ϕ induces an injection from E_i^- to E_i^+. This implies that $|E_i^+| \geq |E_i^-|$ hence $p_i \geq 0$.

To prove the second part, assume that $x_i \leq x_j$ for $j \neq i$. Let $f' : \{\pm 1\}^{n-1} \to \{\pm 1\}$ be a PUF given by $f'(c') = \operatorname{sign}(c' \cdot x')$, where $c' \in \{\pm 1\}^{n-1}$ is obtained from c by dropping c_i, $x'_\ell = x_\ell$ for any $\ell \neq j$, and $x'_j = x_j - x_i \geq 0$. Say the Chow parameters of f' is p'. According to the first part of this lemma, we have $p'_j \geq 0$. Now, expand the expression of $p_j - p_i$ as

$$\begin{aligned}p_j - p_i &= \sum_{c|f(c)=1} c_j - \sum_{c|f(c)=1} c_i \\ &= 2 \sum_{c|f(c)=1, c_j=-c_i} c_j \\ &= 2 \sum_{c'|f'(c')=1} c'_j = 2p'_j \geq 0.\end{aligned}$$

□

6.4 Equivalence Classes and Chow Parameters

Since the X_i are i.i.d. symmetric random variables, the joint probability distribution of the weights $X = (X_1, \ldots, X_n)$ is invariant under permutations and sign changes. Therefore, all PUFs f_x that can be obtained from one another by permuting or changing signs of their weights $x_1, x_2, \ldots, x_n$ can be clustered together into equivalence classes of PUFs with the same probability $\mathbb{P}(f_X = f_x)$.

We now establish several properties of these equivalence classes for PUFs, known as "self-dual" classes [134] in the context of BTFs. Zuev [131] had already mentioned $2^n n!$ elements per class in a special case. Our generalization (Theorem 6.4.2) is mentionned in a different form in [130, § 3.1.2] for calculating the total number of BTFs, yet we couldn't find formal proofs published in the literature.

We give a formal definition of the equivalence classes by the action of the group

$$G_n = S_n \times \{-1, +1\}^n \tag{6.28}$$

where S_n is the symmetric group of order $n!$. An element $g = (\sigma, s) \in G_n$ is determined by the permutation $\sigma \in S_n$ and the sign changes $s \in \{-1, +1\}^n$.

Proposition 2. *For any* $x = (x_1, \dots, x_n) \in \mathbb{R}^n$ *and* $g = (\sigma, s) \in G_n$ *define* $g \cdot x : G_n \times \mathbb{R}^n \to \mathbb{R}^n$ *such that*

$$(g \cdot x)_i = s_i x_{\sigma(i)}. \tag{6.29}$$

This defines a group action of G_n *on* $\mathbb{R}^n$*, where the inner product in* G_n *is defined by*

$$(\sigma_1, s^1) \cdot (\sigma_2, s^2) = (\sigma_1 \circ \sigma_2, (s_i^1 s^2_{\sigma_1(i)})_i). \tag{6.30}$$

Proof. G_n is clearly a group with identity $e = (id, (1, \cdots, 1))$. For any $(\sigma_1, s^1), (\sigma_2, s^2) \in G_n$ and $x \in \mathbb{R}^n$,

$$\begin{aligned}(\sigma_1, s^1) \cdot ((\sigma_2, s^2) \cdot x) &= (\sigma_1, s^1) \cdot (s_i^2 x_{\sigma_2(i)})_i \\ &= (s_i^1 s^2_{\sigma_1(i)} x_{\sigma_1(\sigma_2(i))})_i \\ &= (\sigma_1 \circ \sigma_2, (s_i^1 s^2_{\sigma_1(i)})_i) \cdot x \\ &= ((\sigma_1, s^1) \cdot (\sigma_2, s^2)) \cdot x.\end{aligned}$$

This shows that $g \cdot x$ defines a group action of G_n on $\mathbb{R}^n$. □

Thus we can say that the group G_n acts on the PUFs of size n, the action being defined as

$$g \cdot f_x = f_{g \cdot x}. \tag{6.31}$$

In keeping with Lemma 9, we now show that the group action is carried over to Chow parameters:

Theorem 6.4.1. *Let* f_x *a PUF of Chow parameters* p*, and let* $g \in G_n$*. The Chow parameters of* $f_{g \cdot x}$ *is* $g \cdot p$*.*

Proof. Let $g = (\sigma, s) \in G_n$. For any challenge c, we have that $f_x(g^{-1} \cdot c) = f_{g \cdot x}(c)$. Thus,

$$\begin{aligned}\sum_{c | f_{g \cdot x}(c)=1} c_i = \sum_{c | f_x(g^{-1} \cdot c)=1} c_i = \sum_{c | f_x(c)=1} (g \cdot c)_i = \sum_{c | f_x(c)=1} s_i c_{\sigma(i)} \\ = s_i p_{\sigma(i)} = (g \cdot p)_i.\end{aligned}$$

□

Changing the signs of the weights or permuting them is reflected by the same operation on the Chow parameters. This allows us to compute the size of the equivalence classes:

Theorem 6.4.2. *Let* f *be a PUF with Chow parameters* p*. Let* $m_p(k)$ *be the number of Chow parameters equal to* k *or* $-k \in \mathbb{Z}$*, and let* $\operatorname{Orb}(f) = \{g \cdot f \mid g \in G_n\}$ *the orbit of* f *by* G_n*, that is, the equivalence class containing* f*. Then*

$$|\operatorname{Orb}(f)| = 2^n n! \Big(2^{m_p(0)} \prod_{k \in \mathbb{N}} m_p(k)!\Big)^{-1}. \tag{6.32}$$

Proof. By applying the well-known *orbit-stabilizer theorem* (see for instance [141, p. 89]), we have

$$|\operatorname{Orb}(f)| = \frac{|G_n|}{|\operatorname{Stab}(f)|} = \frac{|\{\pm 1\}^n| \times |S_n|}{|\operatorname{Stab}(f)|} = \frac{2^n n!}{|\operatorname{Stab}(f)|} \tag{6.33}$$

where $\operatorname{Stab}(f) = \{g \in G_n \mid g \cdot f = f\}$ is the *stabilizer* of f. The size of the orbit of f can therefore be deduced from the size of its stabilizer. Now the latter can be easily computed: Let $g = (\sigma, s) \in G_n$ such that $g \cdot f = f$. Since $g \cdot p = p$, we have $\sigma(i) = j \iff p_i = s_i \cdot p_j$ and $s_i = \operatorname{sign}(p_i) \cdot \operatorname{sign}(p_{\sigma(i)})$ except when $p_i = 0$, in which case s_i is unconstrained. The number of such g is exactly $2^{m_p(0)} \prod_{k \in \mathbb{N}} m_p(k)!$. □

6.5 Monte-Carlo Algorithm

As seen in the introduction to the previous section, all PUFs in one equivalence class have the same probability. It follows that the probability of any particular PUF can be deduced from the probability of the class to which it belongs. Therefore, to determine the various entropies, it suffices to find a method that estimates the probabilities of the various equivalence classes.

In this section, we propose an algorithm that exploits a definition of a *canonical* PUF in any equivalence class in such a way that for given any PUF, it is trivial to determine the corresponding canonical PUF. As expected, only about $2^{n^2}/2^n n!$ probabilities need to be estimated, instead of approximatively 2^{n^2}.

Definition 18 (Canonical PUF)**.** A *canonical* PUF of n variables is a PUF whose Chow parameters satisfy

$$p_1 \geq p_2 \geq \cdots \geq p_n \geq 0. \tag{6.34}$$

The *canonical form* of a PUF f is the canonical PUF belonging to the same class, i.e., $f' = g \cdot f$ where $g \in G_n$ is such that f' is canonical.

This notion was first introduced by Winder [135] and is related to the concept of "prime" functions independently studied by Chow [137].

Proposition 3 (Unicity of the canonical PUF)**.** *Two canonical PUFs in the same class are equal.*

Proof. Since f and f' are in the same equivalence class, their Chow parameters are identical up to sign changes and order. Since both are canonical, the signs and order are fixed. Their Chow parameters are thus identical and $f = f'$. □

Proposition 4. *Let $x = (x_1, \ldots, x_n)$ be a weight sequence of a PUF $f = f_x$, and let $g \in G_n$ such that $g \cdot x = (x'_1, \ldots x'_n)$ satisfies*

$$x'_1 \geq x'_2 \geq \ldots \geq x'_n \geq 0. \tag{6.35}$$

Then $g \cdot f$ is the canonical form of the PUF f.

Proof. Let us denote by p (resp p') the Chow parameters of f (resp $g \cdot f$). The PUF obtained from weights x' is $g \cdot f$. From Lemma 9, the p'_i satisfy the same ordinal relations and have the same signs as the x'_i. Therefore, f' is a canonical PUF. □

These results allow us to efficiently estimate the PUF distribution by Monte-Carlo methods, as described in Algorithm 6.5.1. Such an algorithm can be used for any i.i.d. weight distribution with symmetric densities (not necessarily Gaussian).

Algorithm 6.5.1 How to estimate the PUF distribution.

Input: $n > 0, \mathsf{nbRounds} > 0$
Output: Estimation of PUF probability distribution
  Initialize HashMaps $\mathsf{counts}, \mathsf{proba}, \mathsf{size}$
  for $i \in [1, \mathsf{nbRounds}]$ **do**
    Generate n realizations $x_1, \ldots, x_n$
    Sort the absolute values of the x_i to obtain x'
    Compute the Chow parameters p of $f_{x'}$
    if $\mathsf{p} \in \mathsf{counts}$ **then**
      $\mathsf{counts}[\mathsf{p}] \leftarrow \mathsf{counts}[\mathsf{p}] + 1$
    else
      $\mathsf{counts}[p] \leftarrow 1$
    end if
  end for
  for $\mathsf{p} \in \mathsf{counts}$ **do**
    $\mathsf{size}[\mathsf{p}] \leftarrow \dfrac{2^n n!}{2^{m_p(0)} \prod_k m_p(k)!}$
    $\mathsf{proba}[\mathsf{p}] \leftarrow \dfrac{\mathsf{counts}[\mathsf{p}]}{\mathsf{size}[\mathsf{p}] * \mathsf{nbRounds}}$
  end for
  return $(\mathsf{proba}, \mathsf{size})$

6.6 Entropies Estimation

In this section, we present the simulation results in the Gaussian case where the weights X_i are i.i.d. $\sim \mathcal{N}(0, 1)$. Exact values were already determined up to $n = 4$ in [142].

6.6.1 Estimating the Max-Entropy H_0

According to Lemma 8, every PUF can be attained by some realization of weights. Therefore, the max-entropy of the PUF distribution is simply the logarithm of the total number of PUFs with n weights. This number is equal to the total number of BTFs of $n - 1$ variables and has been computed up to $n = 10$ in [130, § 3.1.2], see Table 6.4.

Table 6.4: Exact values of H_0

n	# PUFs	H_0 (bits)
1	2	1
2	4	2
3	14	3.8074...
4	104	6.7004...
5	1882	10.8781...
6	94572	16.5291...
7	15028134	23.8411...
8	8378070864	32.9640...
9	17561539552946	43.9974...
10	144130531453121108	57.0001...

6.6.2 Estimating the Shannon Entropy H_1

For any PUF f, let $[f]$ denote the equivalence class of f with cardinality $|[f]|$, $\mathbb{P}(f)$ its probability, F_n the set of all PUFs and F_n/G_n the quotient group induced by the action of the group G_n. Then, letting $\mathbb{P}([f']) = \sum_{f\in[f']} \mathbb{P}(f)$, one has

$$H_1(n) = -\sum_{f\in F_n} \mathbb{P}(f)\log(\mathbb{P}(f)) \tag{6.36}$$

$$= -\sum_{f'\in F_n/G_n}\sum_{f\in[f']} \mathbb{P}(f)\log(\mathbb{P}(f)) \tag{6.37}$$

$$= -\sum_{f'\in F_n/G_n} \mathbb{P}([f'])\log(\mathbb{P}(f')) \tag{6.38}$$

$$= -\sum_{f'\in F_n/G_n} \mathbb{P}([f'])\log(\mathbb{P}([f'])) + \mathbb{E}[\log(|[f_X]|)]. \tag{6.39}$$

In other words, the Shannon entropy of the PUF distribution is simply the sum of the entropy of the equivalence classes and the average of their logarithmic size. The latter term can be estimated using the unbiased empirical mean, where a confidence interval can be determined using Student's t-distribution [143]. The former term, however, is an entropy, for which no unbiased estimator exists [144]. The NSB estimator [145] has a reduced bias and a low variance. However, because we generated much more PUFs than equivalence classes (by a factor of at least 100000), the plug-in estimator, based on the empirical frequency estimates, performs quite well: Its bias can be upper bounded as described in [144] and was found to be less than 0.01 bit. The results are summarized in Table 6.5.

6.6.3 Estimating the Collision Entropy H_2

The collision entropy was estimated using an unbiased estimator adapted from [146, § 1.4.2]. Let $N_{[f]}$ be the number of PUF samples that belong to the equivalence class of $[f]$

Table 6.5: Confidence intervals at the 95% level for H_1 (exact values up to $n = 4$).

n	PUF Sample size	H_1 (bits)
1	—	1
2	—	2
3	—	3.6655...
4	—	6.2516...
5	10^{10}	10.0134 – 10.0156
6	10^{10}	15.1903 – 15.1925
7	10^{10}	21.9856 – 21.9879
8	$2 \cdot 10^{10}$	30.5628 – 30.5645
9	$2 \cdot 10^{10}$	41.0367 – 41.0384
10	$3 \cdot 10^{12}$	53.4737 – 53.4740

among a number of Poisson-distributed PUFs with parameter N, and $N_{[f]}^{\underline{2}} = N_{[f]} \cdot (N_{[f]} - 1)$. We can compute

$$\mathbf{E}\Big[\sum_{f \in F_n/G_n} \frac{N_{[f]}^{\underline{2}}}{|[f]|N^2}\Big] = \sum_{f \in F_n/G_n} \mathbf{E}\Big[\frac{N_{[f]}^{\underline{2}}}{N^2}\Big]\frac{1}{|[f]|} \tag{6.40}$$

$$= \sum_{f \in F_n/G_n} \frac{\mathbb{P}([f])^2}{|[f]|} \tag{6.41}$$

$$= \sum_{f \in F_n} \mathbb{P}(f)^2 \tag{6.42}$$

where we used the fact that $\mathbf{E}\,[\frac{N_{[f]}^{\underline{2}}}{N^2}] = \mathbb{P}([f])^2$ from [146, § 2.2]. It follows that

$$\sum_{f \in F_n/G_n} \frac{N_{[f]}^{\underline{2}}}{|[f]|N^2}. \tag{6.43}$$

is an unbiased estimator for the power-sum $\sum_{f \in F_n} \mathbb{P}(f)^2$. As can be also checked, the variance of this estimator admits the same upper bound as the one described in [146, § 1.4.2]. This allows us to determine confidence intervals for the collision entropy as shown in Table 6.6.

6.6.4 Estimating the Min-Entropy H_∞

In order to determine the min-entropy of the PUF distribution, one needs to estimate the probability of the most likely PUF. Our experiments, as well as those of Delvaux et al. [1], strongly suggest that for a Gaussian distribution of the weights, the most likely PUFs are the $2n$ PUFs corresponding to the Boolean functions c_i and $\overline{c_i}$, $i = 1...n$.

The maximum likelihood estimator of that probability is simply the sample frequency, which is an unbiased estimator. A confidence interval for this estimator can be obtained

Table 6.6: Confidence intervals at the 95% level for H_2 (exact values up to $n = 4$)

n	PUF Sample size	H_2 (bits)
1	—	1
2	—	2
3	—	3.5462...
4	—	5.7105...
5	10^{10}	8.4551 – 8.4568
6	10^{10}	11.5977 – 11.6023
7	10^{10}	14.8819 – 14.89805
8	$2 \cdot 10^{10}$	18.5201 – 18.5753
9	$2 \cdot 10^{10}$	22.0309 – 22.4067
10	$3 \cdot 10^{12}$	25.9070 – 26.1983

using the Wilson score interval [147], which yields a confidence interval for the min-entropy H_∞.

Because we have already determined that there are exactly $2n$ PUFs in the equivalence class of the most likely PUF, we only need to estimate a confidence interval on the sample frequency of the *equivalence class*. Once such an interval was obtained, for instance $[p_-, p_+]$, then the confidence interval for the min-entropy is given by

$$[-\log_2(p_+) + \log_2(2n), -\log_2(p_-) + \log_2(2n)].$$

The confidence intervals of the min-entropy are presented in Table 6.7.

Table 6.7: Confidence intervals at the 95% level for H_∞ (exact values up to $n = 4$)

n	PUF Sample size	H_∞ (bits)
1	—	1
2	—	2
3	—	3.2086...
4	—	4.5850...
5	10^{10}	6.1006 – 6.1008
6	10^{10}	7.7352 – 7.7354
7	10^{10}	9.4731 – 9.4735
8	$2 \cdot 10^{10}$	11.3020 – 11.3024
9	$2 \cdot 10^{10}$	13.2123 – 13.2132
10	$3 \cdot 10^{12}$	15.1899 – 15.1901

The results of the simulation, up to $n = 10$, are presented in Figure 6.2. The results show that the Shannon entropy is close to the max-entropy, which as seen in Section 6.3 is asymptotically equivalent to n^2 as n increases.

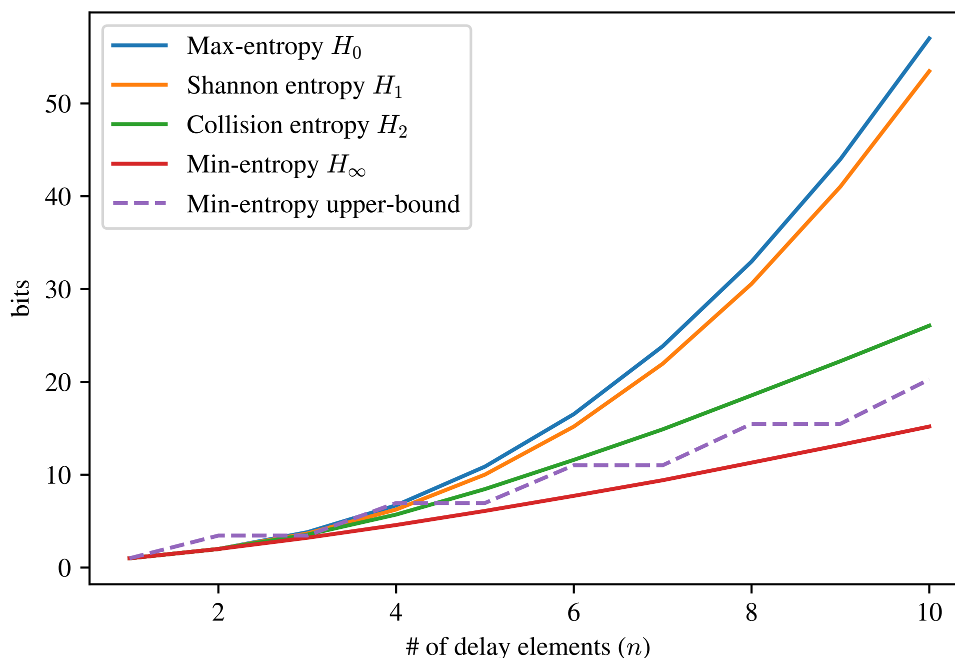


Figure 6.2: Entropy estimates for $n \leq 10$. The upper bound of the min-entropy (dashed line) is taken from [1].

6.7 Conclusions and Perspectives

While it had been previously shown [97] that the entropy of the loop-PUF of n elements could exceed n, the exact values were only known for very small values of n. Making the link with BTF theory using Chow parameters, we have extended these results to provide accurate approximations up to $n = 10$. Our results suggest that the entropy of the loop-PUF might be *quadratic* in n: This would be a very positive result for circuit designers, since it implies that the PUF has a very good resistance to machine learning attacks. However, because the min-entropy and collision entropy are much smaller (on the order of n) the resistance to cloning may not be as high as expected.

Two interesting theoretical aspects of the PUF entropy are still open: First, to what extent does the entropy of the PUF stay close to the max-entropy for larger values of n? Second, is it possible to obtain a quasi-quadratic entropy in n when choosing a small subset of all 2^n possible challenges? The latter point is of great practical interest since it would reduce the time required to obtain the PUF identifier while maintaining a high resistance to machine learning attacks.

For values of n larger than 10, our method seems to become too costly in space and time to produce accurate estimates of the PUF probability distributions under reasonable conditions. One could perhaps have recourse to entropy estimation methods that dispense with learning the distribution itself, such as the NSB estimation [145]. This could be used to check the predicted trend of the PUF entropy for increasing n.

Bibliography

[1] J. Delvaux, D. Gu, and I. Verbauwhede, "Upper bounds on the min-entropy of RO sum, arbiter, feed-forward arbiter, and S-ArbRO PUFs," in *Hardware-Oriented Security and Trust (AsianHOST), IEEE Asian*, pp. 1–6, IEEE, 2016.

[2] 7-cpu.com, "Intel Haswell." `https://www.7-cpu.com/cpu/Haswell.html`. Accessed: 2020-10-12.

[3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pp. 1–19, 2019.

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, (Berkeley, CA, USA), pp. 973–990, USENIX Association, 2018.

[5] Intel, "Intel 64 and IA-32 architectures optimization reference manual." `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`. Accessed: 2019-11-06.

[6] AMD, "AMD Ryzen 9 3950X." `https://www.amd.com/fr/products/cpu/amd-ryzen-9-3950x`. Accessed: 2019-11-06.

[7] J. Daemen and V. Rijmen, "AES proposal: Rijndael." `https://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf`, 1999.

[8] D. J. Bernstein, "Cache-timing attacks on AES." `http://palms.ee.princeton.edu/system/files/Cache-timing+attacks+on+AES.pdf`, 2005.

[9] N. J. Al Fardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in *2013 IEEE Symposium on Security and Privacy*, pp. 526–540, IEEE, 2013.

[10] M. R. Albrecht and K. G. Paterson, "Lucky microseconds: A timing attack on Amazon's s2n implementation of TLS," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 622–643, Springer, 2016.

[11] P. C. Kocher, “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems,” in *Annual International Cryptology Conference*, pp. 104–113, Springer, 1996.

[12] D. Brumley and D. Boneh, “Remote timing attacks are practical,” *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[13] Y. Yarom and N. Benger, “Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack.,” *IACR Cryptology ePrint Archive*, vol. 2014, p. 140, 2014.

[14] P. L. Montgomery, “Modular multiplication without trial division,” *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[15] O. Acıiçmez and W. Schindler, “A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL,” in *Cryptographers Track at the RSA Conference*, pp. 256–273, Springer, 2008.

[16] C. Chen, T. Wang, Y. Kou, X. Chen, and X. Li, “Improvement of trace-driven I-cache timing attack on the RSA algorithm,” *Journal of Systems and Software*, vol. 86, no. 1, pp. 100–107, 2013.

[17] D. Johnson, A. Menezes, and S. Vanstone, “The elliptic curve digital signature algorithm (ECDSA),” *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[18] bushing, marcan, segher, and sven, “Console hacking 2010 - PS3 epic fail.” `https://web.archive.org/web/20200212033236/https://fahrplan.events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf`. Accessed: 2020-05-19.

[19] K. Michaelis, C. Meyer, and J. Schwenk, “Randomly failed! The state of randomness in current Java implementations,” in *Topics in Cryptology – CT-RSA 2013* (E. Dawson, ed.), pp. 129–144, Springer Berlin Heidelberg, 2013.

[20] N. Benger, J. Van de Pol, N. P. Smart, and Y. Yarom, ““Ooh aah... just a little bit”: A small amount of side channel can go a long way,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 75–92, Springer, 2014.

[21] S. Fan, W. Wang, and Q. Cheng, “Attacking OpenSSL implementation of ECDSA with a few signatures,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1505–1515, 2016.

[22] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky, “Lattice signatures and bimodal Gaussians,” in *Annual Cryptology Conference*, pp. 40–56, Springer, 2013.

[23] L. Ducas, “Accelerating BLISS: the geometry of ternary polynomials.” https://eprint.iacr.org/2014/874.pdf, 2014.

[24] L. G. Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, “Flush, Gauss, and reload — A cache attack on the BLISS lattice-based signature scheme,” in *International Conference on Cryptographic Hardware and Embedded Systems*, pp. 323–345, Springer, 2016.

[25] P. Pessl, L. G. Bruinderink, and Y. Yarom, “To BLISS-B or not to be: Attacking strongSwan’s implementation of post-quantum signatures,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1843–1855, 2017.

[26] “strongSwan VPN.” `https://www.strongswan.org/`.

[27] A. K. Lenstra, H. W. Lenstra, and L. Lovász, “Factoring polynomials with rational coefficients,” *Mathematische Annalen*, vol. 261, pp. 515–534, 1982.

[28] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, “Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript,” in *International Conference on Financial Cryptography and Data Security*, pp. 247–267, Springer, 2017.

[29] H. Eldib and C. Wang, “Synthesis of masking countermeasures against side channel attacks,” in *International Conference on Computer Aided Verification*, pp. 114–130, Springer, 2014.

[30] M. Joye, P. Paillier, and B. Schoenmakers, “On second-order differential power analysis,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 293–308, Springer, 2005.

[31] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, “Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks.” Cryptology ePrint Archive, Report 2017/564, 2017. `https://eprint.iacr.org/2017/564`.

[32] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of AES,” in *Cryptographers Track at the RSA Conference*, pp. 1–20, Springer, 2006.

[33] O. Aciiçmez, B. B. Brumley, and P. Grabher, “New results on instruction cache attacks,” in *Cryptographic Hardware and Embedded Systems, CHES 2010* (S. Mangard and F.-X. Standaert, eds.), (Berlin, Heidelberg), pp. 110–124, Springer Berlin Heidelberg, 2010.

[34] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 605–622, IEEE, 2015.

[35] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, “A high-resolution side-channel attack on last-level cache,” in *Proceedings of the 53rd Annual Design Automation Conference*, p. 72, ACM, 2016.

[36] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES," in *2015 IEEE Symposium on Security and Privacy*, pp. 591–604, May 2015.

[37] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in Intel processors," in *2015 Euromicro Conference on Digital System Design*, pp. 629–636, IEEE, 2015.

[38] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *International Symposium on Recent Advances in Intrusion Detection*, pp. 48–65, Springer, 2015.

[39] A. Andreou, A. Bogdanov, and E. Tischhauser, "Cache timing attacks on recent microarchitectures," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 155–155, May 2017.

[40] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX," in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 51–67, 2017.

[41] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack.," in *USENIX Security Symposium*, pp. 719–732, 2014.

[42] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 279–299, Springer, 2016.

[43] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: a timing attack on OpenSSL constant-time RSA," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[44] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 538–570, 2019.

[45] Intel, "Developer reference for Intel integrated performance primitives cryptography." `https://software.intel.com/en-us/ipp-crypto-reference-symmetric-cryptography-primitive-functions`. Accessed: 2019-11-20.

[46] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *ASPLOS*, vol. 53, pp. 693–707, Mar. 2018.

[47] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on AES to practice," in *2011 IEEE Symposium on Security and Privacy*, pp. 490–505, IEEE, 2011.

[48] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 870–887, IEEE, 2019.

[49] T. Allan, B. B. Brumley, K. Falkner, J. Van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pp. 422–435, ACM, 2016.

[50] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*, pp. 69–90, Springer, 2017.

[51] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjá vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 7–18, 2017.

[52] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs.," in *NDSS*, 2017.

[53] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, (Vancouver, BC), USENIX Association, Aug. 2017.

[54] "TIS-CT." `https://trust-in-soft.com/tis-ct/`. Accessed: 2019-02-20.

[55] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, p. 4, 2015.

[56] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 1–11, ACM, 1988.

[57] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations.," in *USENIX Security Symposium*, pp. 53–70, 2016.

[58] S. Blazy, D. Pichardie, and A. Trieu, "Verifying constant-time implementations by abstract interpretation," in *European Symposium on Research in Computer Security*, pp. 260–277, Springer, 2017.

[59] Intel, "Intel 64 and IA-32 architectures software developer's manual." `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`. Accessed: 2019-02-20.

[60] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[61] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, “Sparse representation of implicit flows with applications to side-channel detection,” in *Proceedings of the 25th International Conference on Compiler Construction*, pp. 110–120, ACM, 2016.

[62] D. E. Denning and P. J. Denning, “Certification of programs for secure information flow,” *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[63] E. Gamma, *Design patterns: elements of reusable object-oriented software.* Pearson Education India, 1995.

[64] T. Pornin, “Why constant-time crypto?.” `https://bearssl.org/constanttime.html`. Accessed: 2020-05-22.

[65] A. Facon, S. Guilley, M. Lec’Hvien, A. Schaub, and Y. Souissi, “Detecting cache-timing vulnerabilities in post-quantum cryptography algorithms,” in *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, pp. 7–12, IEEE, 2018.

[66] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom, “The 9 lives of Bleichenbacher’s CAT: New cache attacks on TLS implementations,” in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 435–452, IEEE, 2019.

[67] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[68] L. K. Grover, “A fast quantum mechanical algorithm for database search,” in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC ’96, (New York, NY, USA), pp. 212–219, Association for Computing Machinery, 1996.

[69] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” *Journal of the ACM (JACM)*, vol. 56, no. 6, p. 34, 2009.

[70] R. J. Mceliece, “A public-key cryptosystem based on algebraic coding theory,” *Coding Thv*, vol. 4244, pp. 114–116, 1978.

[71] T. Matsumoto and H. Imai, “Public quadratic polynomial-tuples for efficient signature-verification and message-encryption,” in *Workshop on the Theory and Application of of Cryptographic Techniques*, pp. 419–453, Springer, 1988.

[72] R. C. Merkle, “A certified digital signature,” in *Conference on the Theory and Application of Cryptology*, pp. 218–238, Springer, 1989.

[73] J. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya, and C. Vuillaume, “Merkle signatures with virtually unlimited signature capacity,” in *Applied Cryptography and Network Security*, pp. 31–45, Springer, 2007.

[74] D. Jao and L. De Feo, “Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies,” in *International Workshop on Post-Quantum Cryptography*, pp. 19–34, Springer, 2011.

[75] A. Karmakar, S. S. Roy, O. Reparaz, F. Vercauteren, and I. Verbauwhede, "Constant-time discrete gaussian sampling," *IEEE Transactions on Computers*, 2018.

[76] M. Matsui and J. Nakajima, "On the Power of Bitslice Implementation on Intel Core2 Processor," in *CHES 2007, Vienna, Austria, September 10-13, 2007*, pp. 121–134, 2007.

[77] S. Gueron, "Advanced encryption standard (AES) instructions set." `http://www.ferretronix.com/unigroup/intel_aes_ni/aes-instructions-set_wp.pdf`, 2008.

[78] N. I. of Standards and Technology, "Post-quantum cryptography: Round 2 submissions." `https://csrc.nist.gov/Projects/post-quantum-cryptography/round-2-submissions`, 2019.

[79] J. Ding, Z. Zhang, J. Deaton, K. Schmidt, and F. Vishakha, "New attacks on lifted unbalanced oil vinegar," in *The 2nd NIST PQC Standardization Conference*, 2019.

[80] J.-L. Danger, Y. El Housni, A. Facon, C. T. Gueye, S. Guilley, S. Herbel, O. Ndiaye, E. Persichetti, and A. Schaub, "On the performance and security of multiplication in GF (2n)," *Cryptography*, vol. 2, no. 3, p. 25, 2018.

[81] H. Baan, S. Bhattacharya, J. H. Cheon, S. Fluhrer, O. Garcia-Morchon, P. Gorissen, T. Laarhoven, R. Olayer, R. Rietman, M.-J. O. Saarinen, Y. Son, L. Tolhuizen, J. L. Torre-Arce, and Z. Zhang, "Round5:kem and pke based on (ring) learning withrounding." `https://round5.org/doc/Round5_Submission042020.pdf`.

[82] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over NTRU." `https://falcon-sign.info/`, 2018. Accessed: 2020-04-29.

[83] M.-S. Chen, A. Hülsing, J. Rijneveld, S. Samardjiska, and S. Peter, "MQDSS specifications." `http://mqdss.org/files/MQDSS_Ver2.pdf`.

[84] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAcrypt specification revision 2.0." `https://www.ledacrypt.org/documents/LEDAcrypt_spec_latest.pdf`.

[85] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAcrypt specification revision 2.5." `https://www.ledacrypt.org/documents/LEDAcrypt_spec_2_5.pdf`.

[86] M. Chase, D. Derler, S. Goldfeder, J. Katz, V. Kolesnikov, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, X. Wang, and G. Zaverucha, "The picnic signature algorithm - specification." `https://github.com/microsoft/Picnic/blob/master/spec/spec-v2.1.pdf`.

[87] R. A. Fisher and F. Yates, *Statistical tables: For biological, agricultural and medical research.* Oliver and Boyd, 1938.

[88] Q. Guo, T. Johansson, and A. Nilsson, “A key-recovery timing attack on post-quantum primitives using the fujisaki-okamoto transformation and its application on frodokem,” in *Annual International Cryptology Conference*, pp. 359–386, Springer, 2020.

[89] T. Granlund *et al.*, “GMP: GNU multi precision library.” `https://gmplib.org/`, 2014. Accessed: 2020-04-29.

[90] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, “Updated falcon reference implementation.” `https://falcon-sign.info/falcon-20190918.tar.gz`, 2019. Accessed: 2020-04-29.

[91] G. E. Suh and S. Devadas, “Physical unclonable functions for device authentication and secret key generation,” in *44th ACM/IEEE Design Automation Conference*, pp. 9–14, 2007.

[92] Z. Cherif, J. Danger, S. Guilley, and L. Bossuet, “An easy-to-design PUF based on a single oscillator: The loop PUF,” in *2012 15th Euromicro Conference on Digital System Design*, pp. 156–162, Sep. 2012.

[93] M.-D. M. Yu and S. Devadas, “Recombination of physical unclonable functions,” *35th Annual GOMACTech Conference*, March 2010. Reno, NV, USA.

[94] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, “Delay-based circuit authentication and applications,” in *Proceedings of the 2003 ACM Symposium on Applied Computing*, pp. 294–301, ACM, 2003.

[95] A. Hajimiri, S. Limotyrakis, and T. H. Lee, “Jitter and phase noise in ring oscillators,” *IEEE Journal of Solid-state circuits*, vol. 34, no. 6, pp. 790–804, 1999.

[96] J. Delvaux and I. Verbauwhede, “Side channel modeling attacks on 65nm arbiter PUFs exploiting CMOS device noise,” in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 137–142, IEEE, 2013.

[97] O. Rioul, P. Solé, S. Guilley, and J.-L. Danger, “On the entropy of physically unclonable functions,” in *2016 IEEE International Symposium on Information Theory (ISIT)*, pp. 2928–2932, IEEE, 2016.

[98] D. Lim, J. Lee, B. Gassend, G. Suh, M. van Dijk, and S. Devadas, “Extracting secret keys from integrated circuits,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, pp. 1200 –1205, oct. 2005.

[99] H. Chang and S. S. Sapatnekar, “Statistical timing analysis considering spatial correlations using a single PERT-like traversal,” in *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design*, p. 621, IEEE Computer Society, 2003.

[100] A. Schaub, O. Rioul, and J. J. Boutros, "Entropy estimation of physically unclonable functions via Chow parameters," in *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 698–704, IEEE, 2019.

[101] A. Schaub, O. Rioul, J.-L. Danger, S. Guilley, and J. Boutros, "Challenge codes for physically unclonable functions with Gaussian delays: A maximum entropy problem," *Advances in Mathematics of Communications*, 2020.

[102] A. Schaub, J.-L. Danger, S. Guilley, and O. Rioul, "An improved analysis of reliability and entropy for delay PUFs," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, pp. 553–560, IEEE, 2018.

[103] J.-L. Danger, S. Guilley, and A. Schaub, "Two-metric helper data for highly robust and secure delay PUFs," in *2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI)*, pp. 184–188, IEEE, 2019.

[104] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data," *SIAM J. Comput.*, vol. 38, no. 1, pp. 97–139, 2008.

[105] R. Maes, A. Van Herrewege, and I. Verbauwhede, "Pufky: A fully functional puf-based cryptographic key generator," in *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'12, (Berlin, Heidelberg), pp. 302–319, Springer-Verlag, 2012.

[106] L. Bossuet, X. T. Ngo, Z. Cherif, and V. Fischer, "A puf based on a transient effect ring oscillator and insensitive to locking phenomenon," *Emerging Topics in Computing, IEEE Transactions on*, vol. 2, pp. 30–36, March 2014.

[107] B. Škoric, P. Tuyls, and W. Ophey, "Robust key extraction from physical uncloneable functions," in *Applied Cryptography and Network Security*, vol. 3531, pp. 407–422, Springer, 2005.

[108] R. Maes, "An accurate probabilistic reliability model for silicon PUFs," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 73–89, Springer, 2013.

[109] M. Bhargava and K. Mai, "An efficient reliable PUF-based cryptographic key generator in 65nm CMOS," in *Proceedings of the conference on Design, Automation & Test in Europe*, p. 70, European Design and Automation Association, 2014.

[110] S. Katzenbeisser, U. Kocabaş, V. Rožić, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann, "PUFs: Myth, Fact or Busted? A Security Evaluation of Physically Unclonable Functions (PUFs) Cast in Silicon," in *CHES 2012*, vol. 7428 of *Lecture Notes in Computer Science*, pp. 283–301, Springer Berlin Heidelberg, 2012.

[111] M. Hiller, M. Weiner, L. Rodrigues Lima, M. Birkner, and G. Sigl, "Breaking Through Fixed PUF Block Limitations with Differential Sequence Coding and Convolutional

Codes," in *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, TrustED '13, (New York, NY, USA), pp. 43–54, ACM, 2013.

[112] S.-M. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *IEEE Trans. Computers*, vol. 49, no. 9, pp. 967–970, 2000. DOI: 10.1109/12.869328.

[113] J. Delvaux, *Security Analysis of PUF-Based Key Generation and Entity Authentication.* PhD thesis, Shanghai Jiao Tong University, China, 2017.

[114] D. B. Owen, "Tables for computing bivariate normal probabilities," *The Annals of Mathematical Statistics*, vol. 27, no. 4, pp. 1075–1090, 1956.

[115] L. E. Bassham III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, *et al.*, "Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications," 2010.

[116] J. Delvaux, D. Gu, I. Verbauwhede, M. Hiller, and M.-D. M. Yu, "Efficient fuzzy extraction of puf-induced secrets: Theory and applications," in *International Conference on Cryptographic Hardware and Embedded Systems*, pp. 412–431, Springer, 2016.

[117] D. Ž. Đoković, "Hadamard matrices of order 764 exist," *Combinatorica*, vol. 28, no. 4, pp. 487–489, 2008.

[118] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, (New York, NY, USA), pp. 148–160, ACM, 2002.

[119] S. Guilley, S. Hamaguchi, and Y. Kang, "ISO/IEC NP 20897. Information technology – Security techniques – Security requirements, test and evaluation methods for physically unclonable functions for generating nonstored security parameters." http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=69403.

[120] R. Maes and I. Verbauwhede, "Physically unclonable functions: A study on the state of the art and future research directions," in *Towards Hardware-Intrinsic Security*, pp. 3–37, Springer, 2010.

[121] J.-L. Danger, S. Guilley, P. Nguyen, and O. Rioul, "PUFs: Standardization and evaluation," September 23 2016. Proc. 2nd IEEE Workshop on Mobile System Technologies (MST 2016 – http://www.mstworkshop.eu/), Milano, Italy. [Online version]. DOI: 10.1109/MST.2016.11.

[122] Y. Dodis, K. Pietrzak, and D. Wichs, "Key derivation without entropy waste," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 93–110, Springer, 2014.

[123] M. Skorski, “Key derivation for squared-friendly applications: Lower bounds.,” *IACR Cryptology ePrint Archive*, vol. 2016, p. 157, 2016.

[124] Y. Dodis and Y. Yu, “Overcoming weak expectations,” in *Theory of Cryptography*, pp. 1–22, Springer, 2013.

[125] A. Rényi, “On measures of entropy and information,” in *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley: University of California Press, 1961.

[126] M. Majzoobi, F. Koushanfar, and M. Potkonjak, “Lightweight secure pufs,” in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pp. 670–673, IEEE Press, 2008.

[127] S. Tajik, E. Dietz, S. Frohmann, J.-P. Seifert, D. Nedospasov, C. Helfmeier, C. Boit, and H. Dittrich, “Physical characterization of arbiter pufs,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 493–509, Springer, 2014.

[128] R. O. Winder, “Enumeration of seven-argument threshold functions,” *IEEE Transactions on electronic computers*, no. 3, pp. 315–325, 1965.

[129] S. Muroga, T. Tsuboi, and C. R. Baugh, “Enumeration of threshold functions of eight variables,” *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 818–825, 1970.

[130] N. Gruzling, “Linear separability of the vertices of an n-dimensional hypercube,” Master’s thesis, University of Northern British Columbia, 2008.

[131] Y. A. Zuev, “Methods of geometry and probabilistic combinatorics in threshold logic,” *Discrete Mathematics and Applications*, vol. 2, no. 4, pp. 427–438, 1992.

[132] T. J. Stieltjes, “Extrait d’une lettre adressée à M. Hermite,” *Bulletin of Science and Mathematics, 2nd Series*, vol. 13, no. 2, pp. 170–72, 1889.

[133] I. Abrahamson *et al.*, “Orthant probabilities for the quadrivariate normal distribution,” *The Annals of Mathematical Statistics*, vol. 35, no. 4, pp. 1685–1703, 1964.

[134] E. Goto and H. Takahasi, “Some theorems useful in threshold logic for enumerating boolean functions.,” in *IFIP Congress*, pp. 747–752, 1962.

[135] R. O. Winder, “Single stage threshold logic,” in *Symposium on Switching Circuit Theory and Logical Design*, pp. 321–332, IEEE, 1961.

[136] T. M. Cover, “Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition,” *IEEE Transactions on Electronic Computers*, no. 3, pp. 326–334, 1965.

[137] C.-K. Chow, “On the characterization of threshold functions,” in *Symposium on Switching Circuit Theory and Logical Design*, pp. 34–38, 1961.

[138] S.-T. Hu, *Threshold Logic.* Univ of California Press, 1965.

[139] D. R. Smith, "Bounds on the number of threshold functions," *IEEE Transactions on Electronic Computers*, pp. 368–369, June 1966.

[140] S. Yajima and T. Ibaraki, "A lower bound of the number of threshold functions," *IEEE Transactions on Electronic Computers*, vol. EC-14, pp. 926–929, Dec 1965.

[141] T. W. Hungerford, *Algebra*, vol. 73 of *Graduate Texts in Mathematics.* New York: Springer-Verlag, 1980.

[142] A. Schaub, O. Rioul, J. J. Boutros, J.-L. Danger, and S. Guilley, "Challenge codes for physically unclonable functions with Gaussian delays: A maximum entropy problem," *Latin American Week on Coding and Information, UNICAMP-Campinas, Brazil*, pp. 22–27, 2018.

[143] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.

[144] L. Paninski, "Estimation of entropy and mutual information," *Neural computation*, vol. 15, no. 6, pp. 1191–1253, 2003.

[145] I. Nemenman, F. Shafee, and W. Bialek, "Entropy and inference, revisited," in *Advances in Neural Information Processing Systems*, pp. 471–478, 2002.

[146] J. Acharya, A. Orlitsky, A. T. Suresh, and H. Tyagi, "The complexity of estimating Rényi entropy," in *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1855–1869, SIAM, 2014.

[147] E. B. Wilson, "Probable inference, the law of succession, and statistical inference," *Journal of the American Statistical Association*, vol. 22, no. 158, pp. 209–212, 1927.

ECOLE DOCTORALE

Titre : Méthodes formelles pour l'analyse de canaux cachés logiciels et la génération de clés dans les implémentations cryptographiques

Mots clés : Informatique, sécurité, canaux auxilliaires, cache, PUF

Résumé :
La cryptographie est omniprésente dans notre monde actuel hyperconnecté, protégeant nos communications, sécurisant nos moyens de paiement. Alors que les algorithmes cryptographiques sont en général bien compris, leurs implémentations ont été vérifiées avec moins d'insistance. Cela a mené à des attaques contre les implémentations de la plupart des primitives modernes: AES, RSA, ECDSA... En bref, la sécurité des implémentations pourrait fortement bénéficier de meilleurs garanties théoriques.
Dans cette thèse, nous appliquons ce raisonnement à deux sujets différents, l'un portant sur la sécurité logicielle, l'autre sur la sécurité matérielle.œ
La première moitié de la thèse explore les canaux auxiliaires logiciels dits "cache-timing". Ce genre de vulnérabilités apparaît lorsque la durée d'une opération cryptographique, ou l'état du cache après cette opération, dépend d'une information sensible. C'est le cas lorsqu'une opération de branchement dépend d'une information secrète comme une clé privée, ou si la mémoire est accédée à une adresse qui dépend de ce secret.
Nous avons développé un outil pour détecter et prévenir ce genre de fuites dans des programmes écrits en C, et l'avons appliqué à la plupart des candidats du processus de standardisation de cryptographie post-quantique initié par le NIST. Ce processus vise à remplacer des primitives cryptographiques traditionnelles comme RSA ou ECDSA, vulnérables aux ordinateurs quantiques, par des alternatives sûres. Ces nouveaux algorithmes étant relativement récents, leurs implémentations ont été moins scrutées. Dans cette thèse, nous appliquons notre outil à la plupart de ces algorithmes pour détecter des fuites d'information potentielles, et expliquons comment les éviter.
La deuxième moitié de la thèse est consacrée aux "physically unclonable functions" (PUFs). De ces circuits, on peut extraire des identifiants imprédictibles mais stables, grâce à de petites variations incontrôlables dans les propriétés des semi-conducteurs. Des garanties théoriques pour deux caractéristiques fondamentales de certains PUFs sont présentées dans cette thèse: la stabilité de l'identifiant, perturbée par des bruits de mesure, et l'entropie disponible, dérivée du modèle mathématique du PUF.

Title : Formal methods for the analysis of cache-timing leaks and key generation in cryptographic implementations

Keywords : Computer science, security, side-cannel, cache, PUF

Abstract : Cryptography is ubiquitous in today's interconnected world, protecting our communications, securing our payment systems. While the cryptographic algorithms are generally well understood, their implementations have been less subject to formal verification. This has lead to successful breakages of implementions of most modern primitives: AES, RSA, ECDSA... In general, cryptographic implementations would benefit from stronger theoretical guarantees.
In this thesis, we apply this line of reasoning to two different topics, one in software security, and the other in hardware security. The first half of this thesis explores cache-timing side channel vulnerabilities that arise when the time taken by a cryptographic operation, or the cache state after this operation, depends on sensitive information. This occurs if any branching operation depends on secret information such as a private key, or if memory is accessed at an address that depends on that secret.
We developed a tool to detect and prevent such leaks in programs written in the C programming language. This tool is applied on most candidates of NIST's post-quantum standardization process in order to find cache-timing leakages. This process aims at replacing traditional cryptographic primitives such as RSA or ECDSA, broken by quantum computers, by safer alternatives. The development of such primitives is on the way, but the security of their implementations has received less scrutiny. We show how our tool is able to detect potential cache-timing leaks in a majority of the implementations and what mitigations are possible.
The subject of the second half of this thesis are the so-called physically unclonable functions, or PUFs: elementary circuits from which stable but unpredictable identifiers can be extracted. They rely on small, uncontrollable changes in the semiconductor properties to exhibit unpredictable behavior. Theoretical guarantees concerning two fundamental characteristics of PUFs are derived in this thesis, for a large family of PUFs: the stability of the identifier, related to circuit noise, and the exploitable entropy, derived from the mathematical PUF model.

Institut Polytechnique de Paris
91120 Palaiseau, France