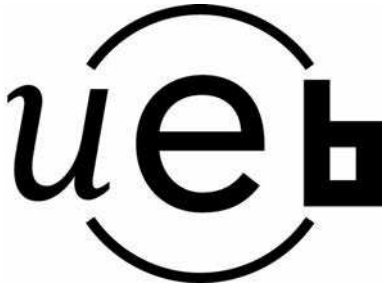
Arithmetic recodings for ECC cryptoprocessors with protections against side-channel attacks

Thomas Chabrier

ANNÉE 2013



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Thomas Chabrier

préparée à l'unité de recherche UMR6074 IRISA
Institut de recherche en informatique et systèmes aléatoires - CAIRN
École Nationale Supérieure des Sciences Appliquées et de Technologie

---

Arithmetic Recodings for ECC Cryptoprocessors with Protections against Side-Channel Attacks

Thèse soutenue à Lannion
le 18 juin 2013

devant le jury composé de :

Lilian BOSSUET
Maître de conférences HDR, Université Jean Monnet, St-Etienne, laboratoire Hubert Curien / rapporteur

Laurent IMBERT
Chargé de recherche CNRS, HDR, LIRMM / rapporteur

Guy GOGNIAT
Professeur, Université de Bretagne Sud, Lab-STICC / examinateur

William MARNANE
Senior Lecturer, University College Cork, Ireland / examinateur

Emmanuel CASSEAU
Professeur, Université de Rennes 1, IRISA / directeur de thèse

Arnaud TISSERAND
Chargé de recherche CNRS, HDR, IRISA / co-directeur de thèse

Résumé

Cette thèse porte sur l'étude, la conception matérielle, la validation théorique et pratique, et enfin la comparaison de différents opérateurs arithmétiques pour des cryptosystèmes basés sur les courbes elliptiques (ECC). Les solutions proposées doivent être robustes contre certaines attaques par canaux cachés tout en étant performantes en matériel, tant au niveau de la vitesse d'exécution que de la surface utilisée. Dans ECC, nous cherchons à protéger la clé secrète, un grand entier, utilisé lors de la multiplication scalaire. Pour nous protéger contre des attaques par observation, nous avons utilisé certaines représentations des nombres et des algorithmes de calcul pour rendre difficiles certaines attaques ; comme par exemple rendre aléatoires certaines représentations des nombres manipulés, en recodant certaines valeurs internes, tout en garantissant que les valeurs calculées soient correctes. Ainsi, l'utilisation de la représentation en chiffres signés, du système de base double (DBNS) et multiple (MBNS) ont été étudiés. Toutes les techniques de recodage ont été validées théoriquement, simulées intensivement en logiciel, et enfin implantées en matériel (FPGA et ASIC). Une attaque par canaux cachés de type *template* a de plus été réalisée pour évaluer la robustesse d'un cryptosystème utilisant certaines de nos solutions. Enfin, une étude au niveau matériel a été menée dans le but de fournir à un cryptosystème ECC un comportement régulier des opérations effectuées lors de la multiplication scalaire afin de se protéger contre certaines attaques par observation.

Summary

This Ph.D. thesis focuses on the study, the hardware design, the theoretical and practical validation, and eventually the comparison of different arithmetic operators for cryptosystems based on elliptic curves (ECC). Provided solutions must be robust against some side-channel attacks, and efficient at a hardware level (execution speed and area). In the case of ECC, we want to protect the secret key, a large integer, used in the scalar multiplication. Our protection methods use representations of numbers, and behaviour of algorithms to make more difficult some attacks. For instance, we randomly change some representations of manipulated numbers while ensuring that computed values are correct. Redundant representations like signed-digit representation, the double- (DBNS) and multi-base number system (MBNS) have been studied. A proposed method provides an on-the-fly MBNS recoding which operates in parallel to curve-level operations and at very high speed. All recoding techniques have been theoretically validated, simulated extensively in software, and finally implemented in hardware (FPGA and ASIC). A side-channel attack called *template attack* is also carried out to evaluate the robustness of a cryptosystem using a redundant number representation. Eventually, a study is conducted at the hardware level to provide an ECC cryptosystem with a regular behaviour of computed operations during the scalar multiplication so as to protect against some side-channel attacks.

Acknowledgements/Remerciements

Ah les remerciements ! C'est une des premières parties du manuscrit, mais en réalité, c'est celle que j'ai rédigée en tout dernier. L'écriture de ces lignes me tient donc particulièrement à cœur pour deux raisons : tout d'abord car elles marquent la fin et la validation de ma thèse, mais surtout parce que je vais pouvoir remercier toutes les personnes qui m'ont accompagné et soutenu durant cette aventure.

En effet, pendant plus de trois ans, mon travail ne s'est pas déroulé tel un long fleuve tranquille ; quelques moments de doute sont venus le parsemer et ce malgré mon naturel joyeux et optimiste. J'ai réussi à traverser les péripéties, les rebondissements et les obstacles grâce à l'action de nombreux personnages : principaux ou secondaires, proches ou éloignés, récurrents ou ponctuels, ils m'ont tous apporté une aide précieuse.

Dans un premier temps, je voudrais remercier Monsieur Guy GOGNIAT, professeur à l'université de Bretagne Sud, pour m'avoir fait l'honneur de présider le jury de cette thèse.

Je remercie également Messieurs Lilian BOSSUET de l'université Jean Monnet à Saint Etienne, et Laurent IMBERT, chargé de recherche CNRS au laboratoire d'informatique, de robotique et de microélectronique de Montpellier (LIRMM), qui ont bien voulu accepter le rôle de rapporteur. En particulier, Laurent IMBERT m'aura permis, de par sa rigueur, d'améliorer la qualité de ce document. Je le remercie donc pour la relecture attentive de ce mémoire, et il ne fait nul doute que ses conseils me seront profitables.

De même, je remercie William MARNANE, professeur à l'université de Cork en Irlande, pour avoir accepté de juger ce travail de thèse. Il m'a accueilli chaleureusement pendant tout un été dans son laboratoire lors d'un séjour de mobilité. Cette expérience a été très enrichissante tant d'un point de vue humain que professionnel.

A présent, je souhaite remercier mon directeur de thèse Monsieur Emmanuel CASSEAU, professeur à l'université de Rennes 1, pour la confiance qu'il m'a accordée en encadrant ma thèse sur un sujet assez éloigné de ses domaines de recherche. En particulier, ses qualités humaines et le recul qu'il m'a aidé à prendre m'ont permis de travailler dans des conditions sereines.

Mes remerciements vont ensuite à mon co-directeur de thèse, Monsieur Arnaud TISSERAND, chargé de recherche CNRS à l'institut de recherche en informatique et systèmes aléatoires (IRISA). Il est difficile de décrire en quelques phrases, l'aide qu'Arnaud m'a apportée. En effet, il m'a guidé, corrigé et même parfois bousculé tout au long de la thèse afin de me faire avancer de manière constructive. Son exigence et sa pédagogie m'ont permis d'approfondir et de préciser mes recherches. Sa porte est toujours restée grande ouverte et je le remercie vraiment pour sa disponibilité et pour le temps qu'il m'a consacré. Je n'imagine pas aujourd'hui avoir pu accomplir ces travaux de thèse sans sa participation.

Je remercie également l'ensemble de mes collègues de travail. En particulier, je remercie Andrianina, Antoine, Jérémie, Renaud, Cécile, Ludovic, Danuta et Karim pour les échanges de

connaissance et les moments de détentes.

Merci à tous ceux qui sont venus jusqu'en Bretagne me rendre visite, mes proches et mes amis, et à tout ceux que j'aurais pu oublier.

Merci à ma belle-famille, ma famille, ma mère, ma sœur Daphné et mes frères Mathieu et David pour leur soutien inconditionnel et leurs encouragements qui ont été autant de bouffées d'oxygène dans les moments de doute. J'ai de plus une pensée toute particulière pour mon père qui, j'en suis sûr, aurait été très fier de moi.

Et voici le meilleur pour la fin ! Je remercie Cindy pour m'avoir accompagné en Bretagne où elle a parfois dû supporter mes humeurs, pour l'intérêt qu'elle a porté à mon travail et surtout pour tous les moments de bonheur et de rire. Elle a toujours été là pour moi en me soutenant dans les moments difficiles, et elle a su faire en sorte que je termine ce manuscrit de thèse avec le sourire.

Contents

List of Acronyms/Notations

$\mathcal{A}$	Affine Coordinates.
`A`	Addition/Subtraction over a finite field.
`ADD`	Point Addition.
ALAP	As-Late-As-Possible.
ALU	Arithmetic Logic Unit.
ASAP	As-Soon-As-Possible.
ASIC	Application Specific Integrated Circuit.
BRAM	Block RAM.
BSD	Binary Signed-Digit.
CMOS	Complementary Metal-Oxide-Semiconductor.
`DBL`	Point Doubling.
DBNS	Double-Base Number System.
DLP	Discrete Logarithm Problem.
DPA	Differential Power Analysis.
DSP	Digital Signal Processing.
EAC	Euclidean Addition Chain.
ECC	Elliptic Curve Cryptography.
ECDLP	Elliptic Curve Discrete Logarithm Problem.
ECDSA	Elliptic Curve Digital Signature Algorithm.
`EPL`	Point Eleventupling.
FF	Flip-Flop.
FIPS	Federal Information Processing Standard.
FPGA	Field Programmable Gate Array.
FSM	Finite State Machine.
gcd	Greatest Common Divisor.

GF	Galois Field.
GMP	GNU Multiple Precision Arithmetic Library.
HDL	Hardware Description Language.
`I`	Inversion over a finite field.
IOB	Input/Output Block.
ISE	Integrated Software Environment.
$\mathcal{J}$	Jacobian Projective Coordinates.
$\mathcal{J}^m$	Modified Jacobian Projective Coordinates.
JTAG	Joint Test Action Group.
LBS	List-Based Scheduling.
lcm	Least Common Multiple.
LSB	Least Significant Bit.
LSW	Least Significant Word.
LT	List of Terms.
LUT	Look-Up Table.
`M`	Multiplication over a finite field.
`mADD`	Mixed Point Addition.
MBNS	Multi-Base Number System.
MI	Montgomery Inversion.
MM	Montgomery Multiplication.
MSB	Most Significant Bit.
`mSUB`	Mixed Point Subtraction.
MSW	Most Significant Word.
`NAF`	Non-Adjacent Form.
NIST	National Institute of Standards and Technology.
`NOP`	No Operation.
$\mathcal{P}$	Standard Projective Coordinates.
PPR	Post Place & Route.
PRNG	Pseudo Random Number Generator.
`QPL`	Point Quintupling.
RAM	Random Access Memory.
RNG	Random Number Generator.

ROM	Read Only Memory.
`S`	Square over a finite field.
SCA	Side-Channel Attack.
$\text{sign}(k_i)$	Sign of the number k at the index i.
SPA	Simple Power Analysis.
`SPL`	Point Septupling.
`SUB`	Point Subtraction.
`TPL`	Point Tripling.
TRNG	True Random Number Generator.
UCC	University College Cork.
UEB	Université Européenne de Bretagne.
VHDL	VHSIC Hardware Description Language.
VHSIC	Very High Speed Integrated Circuit.

Summary in French/Résumé en Français

Cette thèse de doctorat s'intitule « *recodages arithmétiques pour des cryptoprocesseurs ECC robustes aux attaques par observation* ». Elle porte sur l'étude et la validation expérimentale de méthodes de protection contre certaines attaques physiques sur des systèmes cryptographiques basés sur les courbes elliptiques. Les systèmes cryptographiques asymétriques existants sont soumis à de fortes contraintes en termes de sécurité et de performance. En effet, l'utilisation d'algorithmes sûrs théoriquement n'est plus suffisante contre les attaques physiques. Notamment, les attaques par observation représentent un type d'attaque extrêmement efficace contre de tels systèmes non protégés. Étudier différentes méthodes pour se protéger contre ce type d'attaque peut ainsi consister à changer les représentations internes des nombres et les algorithmes de calcul pour rendre plus difficiles certaines attaques ; comme par exemple à rendre aléatoires certaines représentations des nombres calculés tout en garantissant que les valeurs théoriques calculées soient correctes.

Sans le savoir, nous utilisons quotidiennement la cryptographie à clé publique ; par exemple en retirant de l'argent liquide dans un distributeur de billets, ou lors de la saisie et de l'envoi de notre numéro de carte de crédit sur Internet. Des protocoles d'authentification ou de signature numérique permettent par exemple de vérifier l'identité des acteurs d'une transaction électronique. Le cryptosystème RSA (initiales de ses auteurs Rivert, Shamir et Adleman [108]) et le *cryptosystème basé sur les courbes elliptiques* (Elliptic Curve Cryptosystems, ECC) sont ainsi deux des cryptosystèmes de référence pour la plupart des acteurs du monde économique. Mais quelque soit le système cryptographique utilisé, il doit démontrer un niveau de sécurité théorique ne laissant place à aucune attaque réalisable à un coût raisonnable. C'est pourquoi des études sont menées afin d'en connaître le niveau de sécurité actuel. En effet, si l'on suit la célèbre loi de Moore [91] qui mentionne que la puissance de calcul des ordinateurs double environ tous les 18 mois, on peut s'interroger car avec l'augmentation des capacités de calcul de nos ordinateurs, le niveau de sécurité des clés d'une taille donnée, baisse en conséquence. Par exemple, une équipe de chercheurs en cryptographie est parvenue au bout de deux ans et demi à casser le chiffrement RSA 768 bits [69]. Au total, ce sont l'équivalent de 1 700 cœurs qui ont été utilisés pendant un an pour réaliser la factorisation du module RSA, nombre long de 768 bits (soit 232 chiffres décimaux).

De plus, les attaques cryptographiques sont divisées en deux grandes familles, les attaques théoriques et physiques. Les attaques théoriques sont généralement implicitement assimilées à des attaques en « boîte noire » parce que les valeurs des variables temporaires utilisées durant le calcul ne sont pas accessibles par l'attaquant. Cependant, quand les algorithmes sont implantés dans des dispositifs physiques, d'autres attaques deviennent réalisables. Ces attaques, initialement proposées dans [73], consistent à exploiter et observer les caractéristiques physiques d'un cryptosystème, ou bien à modifier le fonctionnement des circuits. Contrairement aux at-

taques théoriques, certaines variables internes aux algorithmes peuvent être lues ou modifiées. Ces attaques se divisent en deux grandes familles : la première contient les *attaques en observation* tandis que la seconde regroupe les attaques par injection de faute. Ces types d'attaque ont récemment vu le jour dans le monde des systèmes embarqués. Les implémentations naïves d'algorithmes cryptographiques peuvent se révéler et ainsi constituer des failles béantes dans la sécurité d'un système. Ces attaques reposent sur l'accès physique que peut avoir un attaquant à un composant manipulant des données secrètes. Le modèle de « boîte noire » souvent utilisé en cryptanalyse s'efface donc ici devant un modèle plus complexe où tous les moyens d'observation peuvent être utilisés par un attaquant pour récupérer de l'information. Un attaquant n'a plus besoin de résoudre le problème mathématique sous-jacent au cryptosystème. La sécurité et l'efficacité des implantations matérielles d'algorithmes cryptographiques constituent donc un véritable défi pour les mathématiciens, informaticiens, électroniciens et les concepteurs de circuits embarqués.

Le système cryptographique choisi doit ainsi utiliser des contre-mesures visant à se protéger des attaques possibles : les algorithmes retenus doivent s'avérer résistants à un certain nombre d'attaques connues et néanmoins rester aussi efficaces que possible en vue d'une implémentation. Les attaques qui nous intéressent dans le cadre de cette thèse sont les attaques par observation, aussi appelées attaques par canaux cachés ou par canaux auxiliaires.

Les attaques par canaux cachés découlent de l'observation pure du composant. Afin d'expliquer ce genre d'attaques, faisons une corrélation avec un exemple clair et parlant : l'histoire d'un vol de documents dans un coffre-fort. Un voleur, équipé d'un stéthoscope, tente d'ouvrir le coffre convoité. Pour cela, il a besoin du code actionnant l'ouverture dudit coffre, code qu'il ne connaît pas mais qu'il va tenter de découvrir. Aussi, il n'a besoin que d'écouter et analyser ce qu'il entend lorsqu'il tourne le bouton. En effet, deux bruits différents peuvent s'entendre ; un « clic » pouvant signifier ainsi qu'un des engrenages est en bonne position, tandis que l'autre bruit signifiant au contraire que l'engrenage ne fait pas partie du code actionnant l'ouverture du coffre. Il ne reste plus qu'à analyser les différents bruits afin de percer le secret du coffre. Le stéthoscope est ici l'outil nécessaire à cette entreprise, il amplifie simplement le son que l'on ne peut entendre à l'oreille. Similairement au stéthoscope qui est utilisé ici pour découvrir le code du coffre-fort, nous pouvons mesurer par exemple l'énergie consommée par le circuit lors d'une exécution d'un algorithme considéré, et ainsi rechercher de l'information, et tenter de trouver la clé secrète d'un algorithme cryptographique. Ainsi, on pourra, via un oscilloscope, visualiser le courant consommé par un circuit. Auparavant, le courant, variable au cours du temps, aura été amplifié car celui-ci étant en effet très faible. Vient ensuite une étape d'analyse où l'on peut observer les différentes opérations utilisées, et si la clé de chiffrement est fonction de ces opérations, celle-ci pourrait être alors révélée. Cet exemple s'appuie sur un type de canal caché, ou canal auxiliaire. Ces canaux peuvent consister en n'importe quelle quantité mesurable en lien avec l'exécution d'un algorithme exploitant de l'information secrète. Certaines sources retenues et étudiées sont le temps d'exécution de l'algorithme, la consommation électrique ou le rayonnement électromagnétique du circuit sur lequel s'exécute l'algorithme.

Dans la cryptographie à clé publique (RSA ou ECC), l'arithmétique joue un rôle important dans la mise en place de cryptosystèmes à la fois efficaces et sûrs. En particulier, il existe essentiellement deux types d'implémentation de cryptosystèmes basés sur les courbes elliptiques : soit la courbe elliptique est définie sur un corps de caractéristique paire ($\mathbb{F}_{2^m}$) soit sur un corps de grande caractéristique première ($\mathbb{F}_{p^m}$ avec $p > 3$). Dans le cadre de cette thèse, nous nous sommes principalement intéressés aux corps premiers $\mathbb{F}_p$ avec p un grand nombre premier (de 160 à 600 bits). Mais nos propositions peuvent être appliquer à d'autres types de courbes.

L'arithmétique des corps finis doit être très rapide étant donnée la quantité de calculs effectués mais tout en nécessitant de ressources limitées (surface de circuit, taille mémoire, consommation

d'énergie). De plus, elle doit offrir un bon niveau de robustesse vis à vis des attaques physiques. Les types de calculs nécessaires sont des additions, soustractions, multiplications (de deux variables ou d'une variable par une ou des constantes), inversions et exponentiations. Ces opérations, qui peuvent paraître plus simples sur des petits entiers ou approximations de nombres réels, sont assez complexes sur des corps finis ($\mathbb{F}_p$ ou $\mathbb{F}_{2^m}$) avec des tailles de nombres de quelques centaines de bits, par exemple de 160 à 600 bits pour les courbes elliptiques en cryptographie [57, App. A]. $\mathbb{F}_p$ est le corps fini à p éléments, où p est un grand nombre premier. Les éléments de $\mathbb{F}_p$ sont manipulés exactement comme des entiers modulo p.

De plus, la recherche de cryptosystèmes asymétriques sûrs et efficaces a conduit, ces dernières années, à un large développement de l'utilisation des courbes elliptiques. En effet, ces objets mathématiques permettent de concevoir des schémas cryptographiques qui requièrent les longueurs de clefs les plus courtes pour un niveau de sécurité donné, en comparaison à d'autres cryptosystèmes proposés [95]. C'est pourquoi les systèmes cryptographiques basés sur les courbes elliptiques peuvent être très attractifs, notamment pour les applications qui utilisent des ressources limitées (mémoire, puissance, bande passante, ...). L'utilisation des courbes elliptiques en cryptographie a été proposée indépendamment par Koblitz [71] et Miller [86] au milieu des années 80. Depuis, ECC fait l'objet d'études intensives.

Une courbe elliptique peut être définie de manière très générale comme l'ensemble des solutions d'une équation à deux variables. Une courbe elliptique est tout d'abord un objet géométrique : c'est une courbe non-singulière obéissant à l'équation $y^2 = f(x)$, avec f ayant un degré égal à 3 ou 4. Géométriquement, une courbe non-singulière est une courbe qui n'admet qu'une seule tangente en tout point. C'est ce genre de courbes définies sur un corps fini que nous utilisons en cryptographie. De plus, une courbe elliptique est aussi un objet algébrique.

La figure 1 présente une interprétation géométrique de la loi d'addition sur une courbe elliptique. P et Q sont deux points appartenant à la courbe elliptique d'équation $y^2 = x^3 - x + 1$ dans le corps $\mathbb{R}$. L'addition de point $P + Q$ est ainsi représentée différemment lorsque $P \neq \pm Q$ (figure de gauche) et $P = \pm Q$ (figure de droite). Quand $P = Q$, nous avons $P + Q = P + P = [2]P$. Dans ce cas, un doublement de point (`DBL`) est calculé. De plus, les courbes elliptiques sont munies d'un point à l'infini noté $\mathcal{O}$. Quand $P = -Q$, nous avons $P + Q = P - P = \mathcal{O}$. Sinon, quand $P \neq \pm Q$, l'addition de point $P + Q$ est calculée (`ADD`).

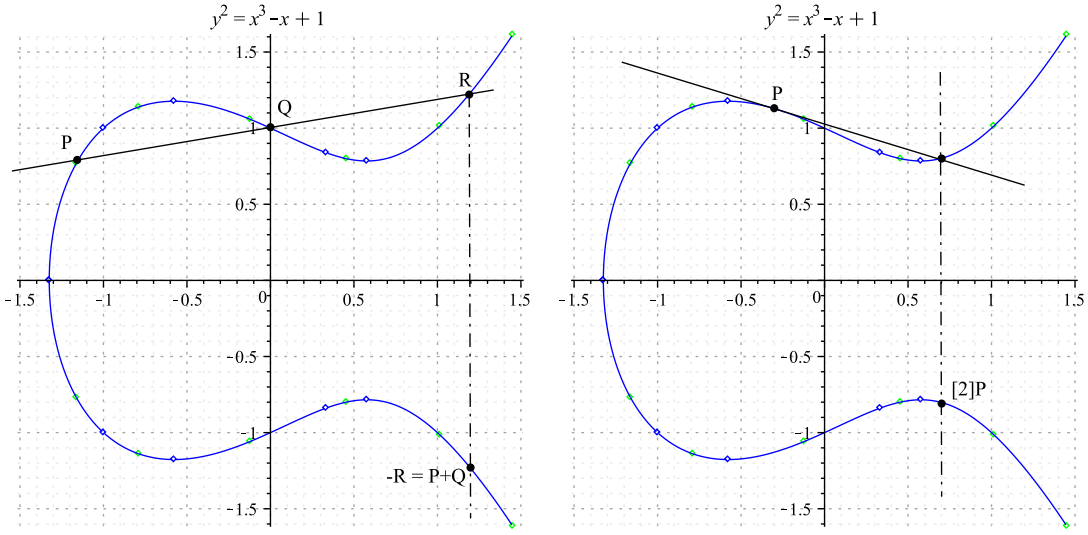


FIGURE 1 – Somme de 2 points appartenant à la courbe elliptique dans $\mathbb{R}$.

Dans les cryptosystèmes basés sur les courbes elliptiques, la clé secrète est utilisée lors de la multiplication scalaire par un point d'une courbe elliptique. Cette opération représente la plus grande part du temps d'exécution d'un protocole ECC. La multiplication scalaire correspond à une suite de doublements (`DBL`) et d'additions (`ADD`) de points qui correspondent à des opérations dans $\mathbb{F}_p$ pour le calcul de

$$[k]P = \underbrace{P + P + \cdots + P}_{k \text{ fois}},$$

où k est un grand entier et P un point d'une courbe elliptique définie sur $\mathbb{F}_p$. La figure 2 présente deux versions algorithmiques du calcul de $[k]P$, utilisant la représentation binaire de k. Ces deux versions consistent à parcourir les bits de k en partant des poids faibles (algorithme de gauche) ou des poids forts (algorithme de droite). L'exécution des algorithmes présentés dans la figure 2 nécessitent en moyenne $(n-1)$`DBL`$+n/2$`ADD`. En effet, pour des raisons de sécurité, il est nécessaire que le scalaire k ait un nombre de 0 équivalent au nombre de 1 dans sa représentation binaire.

entrée : $k = (k_{n-1}k_{n-2}\dots k_1k_0)_2, \quad P \in E(\mathbb{F}_p)$
sortie : $Q = [k]P$

1 :	$Q \longleftarrow \mathcal{O}$		$Q \longleftarrow \mathcal{O}$	
2 :	**pour** i **de** 0 **à** $n-1$ **faire**		**pour** i **de** $n-1$ **à** 0 **faire**	
3 :	**si** $k_i = 1$ **alors** $Q \longleftarrow Q + P$	`ADD`	$Q \longleftarrow [2]Q$	`DBL`
4 :	$P \longleftarrow [2]P$	`DBL`	**si** $k_i = 1$ **alors** $Q \longleftarrow Q + P$	`ADD`
5 :	**retourner** Q		**retourner** Q	

FIGURE 2 – « Doublement et addition » de droite à gauche et de gauche à droite pour le calcul de $[k]P$.

L'algorithme de droite est inspirée de la formule suivante :

$$[k]P = k_0P + [2]\Big(k_1P + [2]\big(k_2P + \cdots + [2](k_{n-1}P)\big)\Big),$$

tandis que celui de gauche applique la formule suivante :

$$[k]P = k_0P + k_1([2]P) + k_2([4]P) + \cdots + k_{n-1}([2^{n-1}]P).$$

Lorsque nous parcourons les bits du scalaire k de gauche à droite, un doublement de point, suivi d'une addition de points sont effectués si le bit du scalaire vaut 1. Sinon, lorsque le bit du scalaire vaut 0, une seule opération au niveau de la courbe est effectuée, un doublement de point. Ces algorithmes présentent clairement la corrélation entre les bits de la clé secrète k et certaines opérations. De plus, dans le cadre de l'exécution d'un cryptosystème à base de courbes elliptiques, le nombre d'opérations dans le corps $\mathbb{F}_p$ effectuées lors d'une addition de deux points (`ADD`) diffère du nombre d'opérations effectuées dans $\mathbb{F}_p$ lors d'un doublement de point (`DBL`). Donc en général, une `ADD` et un `DBL` ont des traces d'exécution différentes. De plus, il n'est pas possible d'observer deux `ADD`s consécutifs : si deux opérations « identiques » se succèdent, il s'agit obligatoirement d'au moins un doublement de point. Ainsi, un attaquant peut savoir si le bit traité vaut 0 ou 1 ; et avec la trace de consommation complète, il peut connaître le scalaire secret.

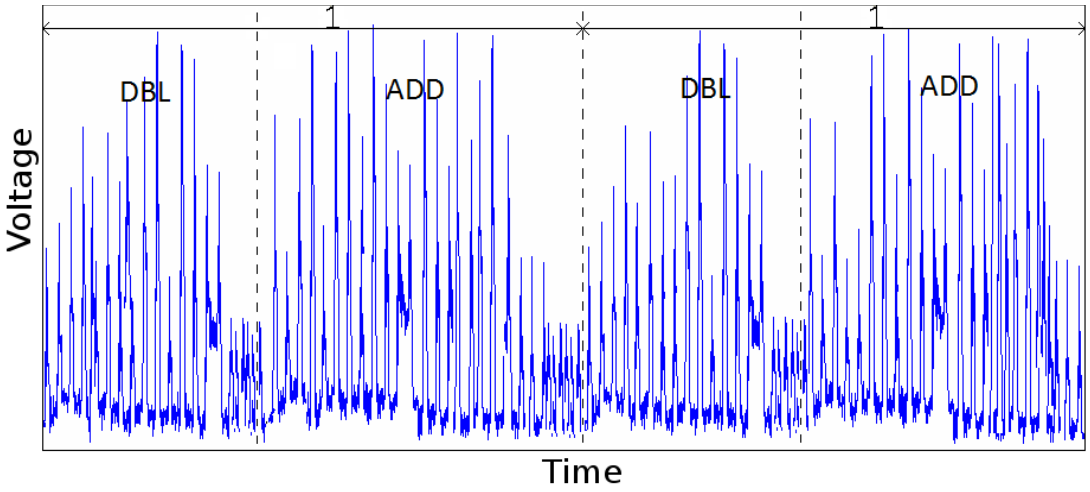


FIGURE 3 – Mesure sur une carte FPGA de la consommation électrique lors d'une multiplication scalaire utilisant l'algorithme « doublement et addition » de gauche à droite.

Pour illustrer cet exemple, nous pouvons voir dans la figure 3 comment une mesure de la consommation électrique d'un circuit effectuant une multiplication scalaire permet d'observer la succession de doublements et d'additions, ce qui permet de retrouver les bits du scalaire k.

Protéger les implémentations de multiplication scalaire contre ce type d'attaques consiste à faire en sorte que le cryptosystème exécute une séquence d'opérations qui ne peut pas être reliée aux bits traités du scalaire k. Les contre-mesures classiques (voir [65] pour plus de détails) utilisent ainsi un algorithme de multiplication scalaire régulier, ou bien rendent l'addition et le doublement de point indistinguables l'uns de l'autre.

L'attaque par observation décrite ci-dessus exploite l'information observable par le bais d'un canal auxiliaire lors de r exécutions de l'algorithme considéré (dans la figure 3, $r = 1$) et rechercher de l'information dans les traces de mesure obtenues. Lorsque r est de l'ordre de 10 ou moins, on parle d'attaques par analyse simple, sinon d'attaques par analyse différentielle ($r \gg 10$). Dans ce cas, on recherche de l'information entre ces r mesures et les données manipulées à l'aide d'outils statistiques. Cette famille d'attaques par observation exploite une fuite d'information plus fine que les attaques par analyse simple qui relient les opérations exécutées et le canal caché mesuré.

Protéger un cryptosystème contre les attaques par analyse simple n'est pas suffisant pour être robuste contre les attaques différentielles par observation. Les contre-mesures classiques consistent à changer la valeur des variables intermédiaires : ajouter de l'aléa sur le scalaire k [35], sur le point de base P [79] ou sur l'équation de la courbe [67] pendant les différents calculs de $Q = [k]P$. Par exemple, Coron dans [35] propose d'effectuer la multiplication scalaire $[k']P$ à la place de $[k]P$ où $k' = k + g\#E(\mathbb{F}_p)$ avec $\#E(\mathbb{F}_p)$ l'ordre de la courbe elliptique $E(\mathbb{F}_p)$ et g un nombre aléatoire. Chaque nouvelle multiplication scalaire permet de rendre imprévisible la valeur de la clé, et donc sa représentation, mais tout en garantissant la valeur exacte du résultat.

Ainsi, l'utilisation de différentes représentations de nombres peut être exploitée comme contre-mesure contre certaines attaques par canaux cachés : une clé de chiffrement pourra utiliser une certaine représentation lors d'une multiplication scalaire, et aura une autre représentation lors d'une autre multiplication scalaire. Le but de ces différentes représentations est de ne pas pouvoir découvrir cette clé ; car ainsi, à chaque nouvelle exécution de l'algorithme de chiffrement, la signature en courant de celui-ci ne pourra être mis en corrélation avec une exécution antérieure.

Dans une représentation redondante, un nombre peut avoir plusieurs représentations possi-

bles. Dans cette thèse, plusieurs représentations redondantes ont été étudiées : en chiffres binaires signés, en double-base et en multi-base. Ces trois représentations sont extrêmement redondantes, et leur utilisation peut permettre la protection d'un cryptosystème contre certaines attaques par canaux cachés. Pour cela, le scalaire k est changé à la volée et aléatoirement en une autre représentation. Ces représentations n'ont rien de nouveau, étant déjà bien connues par la communauté scientifique. Mais bien qu'étudiées, il y a peu d'implémentation logicielles et encore moins d'implémentation matérielles sur de tels recodages pour leur utilisation dans les courbes elliptiques. Pour chaque représentation redondante étudiée, une implémentation matérielle de la méthode permettant de changer aléatoirement un nombre entier en une autre représentation a été réalisée. En effet, nous voulions montrer que l'utilisation de ces représentations est applicable et réalisable en matérielle. En particulier, nous comparons la surface utilisée et la fréquence d'horloge de ces méthodes sur un cryptosystème ECC.

Dans une représentation en chiffres binaires signés, un entier k peut s'écrire :

$$k = \sum_{0}^{n} k_i 2^i \quad \text{avec} \quad k_i \in \{-1, 0, 1\}.$$

Les algorithmes de multiplication scalaires correspondants s'obtient directement en remplaçant les lignes

« **si** $k_i = 1$ **alors** $Q \leftarrow Q + P$ »,

des algorithmes présentés dans la figure 2 par

« **si** $k_i \neq 0$ **alors** $Q \leftarrow Q \ + \ \text{sign}(k_i)\ P$ »,

où $\text{sign}(k_i)$ représente le signe du chiffre de k à l'indice i. Lorsque $k_i = -1$, une soustraction de point est réalisée (`SUB`). En pratique, l'opération et le coût de `ADD` et `SUB` sont considérés équivalents. De plus, cette représentation peut être modifiée pour faire en sorte que l'exécution de la multiplication scalaire soit plus rapide. En effet, la représentation redondante en chiffres binaires signés peut être étendue en une représentation redondante en chiffres signés par fenêtre de w bits.

$$k = \sum_{i=0}^{n} 2^i k_i \quad \text{avec} \quad k_i \in \{0, \pm 1, \pm 3, \ldots, \pm(2^w - 1)\}.$$

Sur w chiffres consécutifs, il est assuré qu'au maximum un chiffre est différent de zéro. Ainsi, il y a plus de 0 dans cette représentation, ce qui permet de réaliser moins d'additions de point, et donc d'accélérer l'exécution de la multiplication scalaire. Mais cette représentation nécessite le pré-calcul des points $P_j = [j]P$ pour tout $j \in \{3, 5, \ldots, 2^w - 1\}$ ($[3]P, [5]P, \ldots, [2^w - 1]P$), ce qui peut être coûteux en terme de surface utilisée. La multiplication scalaire est réalisée en utilisant les additions/soustractions de point (par rapport au signe de k_i) avec les pré-calculs des points P_j. Le pseudo-code correspondant est

« **si** $k_i \neq 0$ **alors** $Q \leftarrow Q \ + \ \text{sign}(k_i)\ P_{|k_i|}$ ».

Différentes représentations d'un nombre en chiffres binaires signés peuvent être obtenues en remplaçant certaines séquences de chiffre par d'autres : $0\bar{1} \Longleftrightarrow \bar{1}1$ et $01 \Longleftrightarrow 1\bar{1}$. Ainsi, différentes représentations d'un entier k peuvent être obtenues, via un nombre aléatoire qui décide du changement de certaines séquences. L'utilisation du scalaire k dans cette représentation

redondante peut ainsi constituer une contre-mesure contre certaines attaques par observation. Cette méthode a été testée en logicielle et implémentée en matérielle (FPGA et ASIC). L'unité de recodage a une fréquence d'horloge plus rapide et sa surface représente environ 20% qu'un cryptoprocesseur ECC.

Afin d'évaluer la robustesse de cette contre-mesure, une attaque par analyse différentielle, une attaque par *template* a été réalisée. L'idée de cette attaque, introduite dans [27], est tout d'abord de construire une base de données mémorisant le comportement du canal caché mesuré pour des opérations connues. La fuite d'information d'un circuit est donc caractérisée : le profil du cryptosystème est établi pour un grand nombre de messages et d'hypothèses de clés. Cette phase peut aussi permettre de recueillir des détails concernant l'implantation de l'algorithme cryptographique. Ensuite, l'attaquant tente de retrouver la clé secrète en analysant des mesures d'un canal caché provenant du cryptosystème attaqué.

Une attaque par template a ainsi été réalisée lors d'un projet de mobilité internationale de trois mois à l'UCC (*University College Cork*) en Irlande. Ce projet de mobilité a été financé par le collège doctoral international (CDI) de l'université européenne de Bretagne (UEB). Lors de l'attaque par template réalisée, le scalaire k est représenté en chiffres binaires signés. Pour chaque multiplication scalaire effectuée, k est changé aléatoirement en une autre représentation.

Dans l'attaque réalisée, nous avons tenté de deviner les trois premiers bits de k utilisés dans la multiplication scalaire, en faisant l'hypothèse que le point de base P ne change pas. Le profil du cryptosystème a été établi en obtenant un grand nombre de mesures (1 000) de la consommation électrique du circuit FPGA pour chacune des clés possibles ($2^3 = 8$ clés possibles). Ensuite, nous avons acquis une centaine de mesures avec une clé secrète inconnue.

Afin de valider ce type d'attaque, nous avons tenté de deviner les trois premiers bits de la clé secrète utilisés avec un algorithme de multiplication scalaire non protégé. Ensuite, une fois que nous avons deviné les bits de la clé secrète, nous avons implanté la contre-mesure dans le cryptosystème. Nous n'avons pas pu deviner les bits attaqués du scalaire k avec l'utilisation de cette contre-mesure. Le fait que l'attaque par template échoue à deviner la clé secrète a donc permis d'évaluer la robustesse de la contre-mesure utilisée, en changeant aléatoirement le scalaire dans une représentation redondante des nombres lors de chaque nouvelle multiplication scalaire.

D'autres représentations redondantes des nombres ont été étudiées dans cette thèse. Par exemple, nous utilisons un système de nombre en double-base [44] (DBNS pour *double-base number system*). Celui-ci permet de représenter les entiers comme une somme de puissances combinées de deux nombres premiers b_1 et b_2. Généralement, les deux bases utilisées sont les bases $(b_1, b_2) = (2, 3)$. Dans ce cas, un nombre entier k s'écrit :

$$k = \sum_{i=0}^{n'-1} k_i b_1^{u_i} b_2^{v_i} \quad \text{avec} \quad k_i \in \{-1, 1\},\ u_i, v_i \geq 0.$$

Pour les calculs ECC en DBNS, une nouvelle opération au niveau de la courbe doit être définie : le triplement de point $[3]P = P + P + P$ (noté `TPL`). En effet, il est plus intéressant d'avoir cette nouvelle opération que d'effectuer un doublement puis une addition de point : un `TPL` est même plus rapide qu'une `ADD`. L'avantage de ce système est que la représentation d'un nombre est creuse (possède beaucoup d'éléments nuls), et n'est pas unique (c'est un système redondant) : il peut même en exister de nombreuses. En effet, 127 a par exemple 783 représentations différentes [46] :

$$127 = 2^2 3^3 + 2^1 3^2 + 2^0 3^0 = 2^2 3^3 + 2^4 3^0 + 2^0 3^1 = 2^3 3^2 + 2^1 3^3 + 2^0 3^0 = \ldots.$$

Cette représentation étant creuse, elle permet d'accélérer l'exécution de la multiplication scalaire car moins d'additions de points seront effectuées. De plus, il est nécessaire d'avoir un

nombre en DBNS avec des exposants décroissants afin de calculer efficacement la multiplication scalaire : $u_0 \geq \cdots \geq u_{n'-1}$ et $v_0 \geq \cdots \geq v_{n'-1}$. Ainsi, un système en double-base avec des exposants décroissant est appelé une chaîne DBNS [40].

L'idée est d'utiliser la redondance qu'offre le système en double-base afin que la représentation du scalaire, préalablement convertie en chaîne DBNS, soit différente au cours du temps, et ce aléatoirement. La conversion d'un entier en une chaîne DBNS se fait par un programme fourni par les auteurs de [46]. Ainsi, ce système de représentation permet de pouvoir rendre aléatoire le scalaire utilisé lors de la multiplication scalaire dans les courbes elliptiques. Ceci est réalisé par l'utilisation de règles telles que $1 + 2 = 3$, ou $1 + 3 = 2^2$ par exemple. Grâce à l'utilisation de ces règles, on va pouvoir développer un terme, ou réduire la chaîne DBNS en choisissant le terme à réduire ou à développer aléatoirement. Les règles d'expansion et de réduction de termes ont été implémentées en matériel. Cette méthode de représentation de la clé secrète peut donc constituer une contre-mesure face à certaines attaques par canaux auxiliaires. Elle a été testée en logicielle et implémentée en matérielle (FPGA et ASIC). L'unité de recodage a une fréquence d'horloge plus rapide et sa surface représente moins de 7% qu'un cryptoprocesseur ECC.

La méthode précédente nécessite d'avoir au préalable le scalaire k déjà en chaîne DBNS. Nous avons donc étudié et implémenté en matériel une méthode permettant de coder un entier en DBNS. Plus généralement, nous avons implémenté en matériel une méthode permettant de recoder un entier en multi-base (MBNS pour *multi-base number system*). DBNS est un cas particulier de MBNS. Un entier en MBNS utilise plusieurs bases $(b_1, b_2, \ldots, b_l)$ et s'écrit :

$$k = \sum_{i=0}^{n'-1} \left(k_i \prod_{j=1}^{l} b_j^{e_{j,i}}\right) \quad \text{avec} \quad k_i = \pm 1.$$

En général, les bases utilisées sont $(2, 3, 5)$ et $(2, 3, 5, 7)$. Comme pour le DBNS, cela peut nécessiter de nouvelles opérations sur la courbe. Par exemple, si les nombres 5 et 7 appartiennent à la multi-base, un quintuplement (`QPL`) et septuplement (`SPL`) de points sont définis.

Ainsi, nous fournissons la première implémentation matérielle de la conversion d'un entier en MBNS. Le codage d'un nombre en MBNS se fait à la volée par une unité de recodage qui opère en parallèle des opérations au niveau de la courbe : par exemple, pendant le calcul d'un doublement de point, nous avons le temps d'obtenir un nouveau terme MBNS et de lancer les opérations au niveau de la courbe. Ceci permet de lancer la prochaine opération au niveau de la courbe sans interruption. De plus, une méthode a été proposée afin de recoder aléatoirement le scalaire k : lors de chaque recodage, la représentation de k en MBNS peut être différente. Encore une fois, ceci peut permettre de procurer une contre-mesure face à certaines attaques par canaux auxiliaires.

Enfin, une méthode de protection contre les attaques par analyse simple a été étudiée et implémentée en matériel. Dans cette méthode, nous utilisons des algorithmes déjà existants de multiplication scalaire. Nous les modifions de manière à pouvoir les effectuer en commençant par les poids faibles (algorithmes de droite à gauche). Ainsi, les opérations au niveau de la courbe peuvent s'effectuer en parallèle. Une multiplication scalaire peut donc effectuer à chaque tour de boucle une même séquence d'opérations au niveau de la courbe. Ainsi, les algorithmes de multiplication scalaire ont un comportement régulier : il ne dépendent pas de la clé secrète. En pratique, il n'y a pas en matériel d'unités qui réalisent les opérations au niveau de la courbe (`ADD`, `DBL`). En effet, ce sont les unités arithmétiques au niveau du corps (par exemple des unités de multiplication et d'addition modulaire) qui sont effectuées en parallèle. Ainsi, ce sont certaines opérations au niveau du corps $\mathbb{F}_p$ qui sont effectuées en parallèle. Dans cette méthode, nous

faisons l'hypothèse que plusieurs unités arithmétiques sont implantées.

Les opérations au niveau de la courbe peuvent être vues comme une succession d'opération au niveau du corps considéré. Nous proposons ainsi des séquences composés d'opérations dans $\mathbb{F}_p$ qui effectuent des opérations au niveau de la courbe. De plus, chaque séquence est composée de plusieurs blocs, et chaque bloc contient un même nombre générique d'unités arithmétiques. Ainsi, nous fournissons plusieurs séquences possibles pour les opérations au niveau de la courbe qui s'effectuent en parallèle. L'idée est alors d'exprimer les opérations effectuées au cours de l'algorithme comme une succession de motifs identiques. Un attaquant effectuant une lecture simple de la consommation énergétique ou du rayonnement électromagnétique pendant la multiplication scalaire ne verrait alors qu'une suite de blocs d'opérations semblables et n'obtiendrait ainsi aucune indication sur les branches conditionnelles suivies par l'algorithme de multiplication scalaire.

Nous proposons ainsi plusieurs séquences qui effectuent des opérations au niveau de la courbe. Lorsqu'une seul séquence est implémentée, l'algorithme de multiplication scalaire n'effectuera qu'une succession de ce bloc. Cette méthode peut servir de contre-mesure contre certaines attaques par observation par analyse simple. Lorsque de nombreuses séquences sont implémentées, cela peut servir de contre-mesure contre certaines attaques par observation par analyse différentielle.

Une multiplication complète a été implémentée en matérielle (FPGA et ASIC) en suivant cette méthode. Ainsi, des tests ont été menés pour connaître la surface utilisée et la vitesse de notre implémentation par rapport au nombre de séquence et d'unité arithmétique. En moyenne, nous utilisons plus de surface (environ 7%) par rapport au cryptosystème ECC avec lequel nous nous comparons, mais en allant tout aussi vite en terme de fréquence d'horloge.

Ainsi, nous avons étudié durant cette thèse des méthodes pour se protéger contre certaines attaques par observation. Notamment, nous avons étudié, testé et implémenté en logiciel et en matériel plusieurs recodages arithmétiques. De plus, une autre méthode a été étudiée et utilisée. Cette méthode s'appuie sur le fait qu'au niveau matériel, un grand nombre de calculs peuvent de faire en parallèle.

Cependant, nous ne pouvons pas garantir l'efficacité réelle de nos contre-mesures. En effet, il n'est pas suffisant d'attaquer un cryptosystème pour conclure de l'efficacité de tel ou tel contre-mesure ou algorithme. Il est en effet nécessaire de connaître théoriquement la robustesse de chaque recodage arithmétique réalisé. Ceci reste un travail et une démarche captivants pour de futures recherches.

Introduction

The history of cryptography is both long and fascinating. Its known origins date from the building of the Egyptian civilization about 4 000 years ago [68]. Since the invention of writing and the first wars, it has always been important to be able to transmit protected messages, that is to say messages which cannot be understood by enemies even in case of interception. *Encryption* is the ability to make a message not understandable. The reverse process, i.e. to make the encrypted information understandable again, is *decryption*. The concepts of confidentiality, authenticity, integrity, and non-repudiation are the core of cryptographic schemes [20]:

- *confidentiality* ensures that both sender and receiver of a message are the only ones who can read it.
- *authentication* is the process of verifying and proving identity of both sender and receiver.
- *integrity* guarantees that the message is not modified during the transfer.
- *non-repudiation* ensures that the sender of a message cannot deny having sent the message, and that the recipient cannot deny having received it.

Encryption is a cryptographic primitive which is used to provide confidentiality. There are two basic techniques for encrypting information: *symmetric cryptography* (also called private-key cryptography) and *asymmetric cryptography* (also called public-key cryptography). In symmetric encryption, two involved parties share a key. To provide privacy, this key needs to be secret. Both sender and receiver can encrypt and decrypt all messages with their secret key, called *private key*. However, the use of symmetric system leads to constraints on keys distribution and management. Keys must be distributed only to the concerned parties [83]. For efficient key management, a trusted third party is often required.

Asymmetric encryption uses a pair of keys. Each sender/receiver has a public and a private key. The public key is available to anyone and must be distributed before the first communications. The private key is only known by the sender. A message encrypted with the public key of a user, can only be decrypted using the corresponding private key of that particular user.

Without knowing it, we use daily public-key cryptography, for example when withdrawing cash, or when using our credit card on the Internet. The RSA (initials of its authors Rivert, Shamir and Adleman [108]) and the elliptic curve cryptosystems [86] [71] are thus two of the reference public-key cryptosystems for most economic actors. However, whatever the cryptosystem used, it must have a theoretical security level such that there are not any practicable attacks in a reasonable amount of time or cost. That is why studies are conducted to evaluate the security level of cryptosystems. Indeed, if one follows Moore's famous law [91] which states that the computing power of computers, more or less doubles every 18 months, one can wonder if cryptosystems security is still enough. For example, an international team of mathematicians, computer scientists and cryptographer researchers managed to factorize the 768-bit (232-digit) number RSA-768 [69] after two and a half years of computations. They occupied the equivalent of 1 700 usual processor cores for a year.

Moreover, the research of secure and efficient asymmetric cryptosystems has led, in recent years, to extensive development of the use of elliptic curves. Indeed, these mathematical objects allow the design of cryptographic schemes which require shorter key lengths for a particular level of security, in comparison to existing cryptosystems such as RSA [106]. Table 1 gives security equivalences between elliptic curves cryptography (ECC) and RSA cryptosystems key lengths [95]. The column labelled "strength" gives the approximate number of bits of security the cryptosystems offer.

security level (bits)		
strength	ECC	RSA
80	160	1 024
112	224	2 048
128	256	3 072
192	384	8 192
256	512	15 360

Table 1: Comparison of ECC and RSA key sizes for different security requirements (from [95]).

An elliptic curve can be a geometric or an algebraic object; and in a very general way, it is defined by the set of solutions of an equation in two variables. In elliptic curves for cryptography, arithmetic is a casual role in the implementation of cryptosystems to ensure efficiency and security. In particular, finite-field arithmetic must be very fast according to the amount of required computations, and use limited resources (circuit area, memory size, power consumption). However, it must also provide a good level of robustness against physical attacks. Required finite-field operations are: additions, subtractions, multiplications (with two variables, or with one variable by one constant) and inversions. These operations, which may seem simple enough over small integers or real numbers approximations are quite complex over finite fields ($\mathbb{F}_p$ or $\mathbb{F}_{2^m}$) with sizes of several hundred bits, for example from 160 to 600 bits for *elliptic curves cryptography* (ECC) [57, Appendix A]. Thus, arithmetic is a key element for designing efficient and secure cryptosystems in elliptic curves cryptography.

In this Ph.D. thesis, we use ECC based on the algebraic structure of elliptic curves over the finite field $\mathbb{F}_p$. All work will have to fit the context of secure and efficient implementations: side-channel attacks and efficient implementations are two important areas of research in ECC.

Arithmetic over the finite-field $\mathbb{F}_p$ should be fast to perform computations on a large amount of operations (addition, subtraction, multiplication, inversion in the finite field) on large numbers. For cost reasons, arithmetic operators should also be area, memory and power efficient. Moreover for security reasons, they should not reveal internal information at runtime using physical attacks such as side-channel analysis.

Table 1 shows that for a same security level, ECC needs key sizes smaller than RSA. Therefore, the use of elliptic curves is a decisive advantage in the context of embedded devices when resources (power, memory, frequency, bandwidth, etc.) are limited. However whereas modern cryptography meets with an excellent security level for authentication, confidentiality, etc., new attacks have recently emerged in the world of embedded systems. Naive implementations of cryptographic algorithms can constitute flaws into the system security. These attacks exploit some correlations between secret values manipulated in the device and physical parameters measured

on the device such as power consumption, electromagnetic emanations or computation timing. Black-box model, often used in cryptanalysis, fades here before so a more complex model where all observation can be used by an attacker to retrieve information. The security and efficiency of implementations is a real challenge for mathematicians, computer scientists and electronic engineers for both hardware and software cryptosystems.

Cryptographic attacks are divided into two large families, theoretical and physical attacks. Theoretical attacks are usually considered as *black box* attacks because the values of temporary variables used during the computation are not accessible by an attacker. However, when algorithms are implemented in embedded systems, other attacks become achievable. These attacks consist in side-channels exploitations or circuit operations modifications. Contrary to theoretical attacks, some internal variables to algorithms can be read or modified. These attacks can be divided into two main groups: the first contains observation attacks, while the second includes perturbation attacks (e.g. fault injections).

Thus, the chosen cryptosystem must use countermeasures to protect itself from possible attacks: chosen algorithms should be resistant to a number of known attacks and nevertheless still remain as efficient as possible in view of an implementation. The attacks which we are interested in are side-channel attacks.

Side-channel attacks come from pure observation of the component. To explain this kind of attack, let us do a correlation with a clear example: the story of the theft of documents in a safety box. A thief, equipped with a stethoscope, tries to open the safety lock. For this, he/she needs the code actuating the opening of the safety box, code that he/she does not know and that he/she will try to discover. Also, he/she only needs to listen to and analyse what it means when he/she turned the knob. Indeed, two different sounds can be heard; whether a "*click*" means one of the gears is in a good position, the other sound means that the gear is not part of the code which unlatches the safety box. The only thing left to do is to analyse the different sounds to discover the secret. The stethoscope is the required tool in this endeavour, it just amplifies the sound that one can hear. Similarly to the way, this stethoscope is used to discover the code of the safety box, one can measure for example the power consumption of a circuit when running an algorithm. With these measurements, one can search for information, and try to find the secret key of an encryption algorithm. Thus, it is possible, via an oscilloscope to measure the consumed power by a circuit. Then there is a stage of analysis where one could observe the different operations executed. If the encryption key is a function of these operations, it could then be revealed. This example is based on a type of side channel. Side channels can consist in any measurable quantity in connection with the execution of an algorithm exploiting secret information. Some used and studied sources are the running time, power consumption or electromagnetic radiation of the circuit on which algorithms run. The aim is to retrieve informations, such as the secret key used in a cryptosystem.

However, there are a-plenty of countermeasures against side-channel attacks in the literature. Some of them are theoretical and use different number representations and algorithms. Calculation algorithms can be changed at the curve level or at the field level. For instance, Pamula [99] performed researches at the CAIRN team by changing algorithms at low level, with hardware implementations and results. However in practice, there are few software results, and even less hardware results.

In this thesis, we study some existing protections at the arithmetic level and we implement them in hardware. We do not change curve parameters, but we change the number representations and the calculation algorithms. Indeed, we want to show that most of solutions are realistic in hardware and implementable. In addition, we want to know what is the costs of the proposed

solutions in term of area or clock frequency.

Thus, we evaluate the cost of some methods and protections. In particular, each method and implementation are validated (at theoretical and practical levels), finely designed in hardware and practically evaluated on different FPGAs (area, clock frequency and robustness). It enables to compare some implementations on several platforms.

The organization of this work is detailed in the following.

We shall study efficient and secure cryptographic implementations. Chapter 1 introduces all necessary concepts for an understanding of the basic elements. Also, it gives the reader concepts about the understanding of subsequent chapters. Thus, chapter 1 deals with elliptic curves elements, arithmetic over finite fields and side-channel analysis. In addition, we work on arithmetic algorithms and various representations of numbers. Speed and area of the various operators are theoretically estimated and practically evaluated on FPGAs. Some algorithms or number representations can have specific characteristics, and modify power consumption during encryption.

Chapter 2 deals with redundant representations, and in particular how to randomly recode numbers. In a redundant system number, some numbers have several representations, thus the name redundant. These representations are known in the literature. We implement the proposed solutions and we propose hardware implementations for on-the-fly random recodings of the scalar digits using two methods, the double-base number system (DBNS) and signed-digit (SD) representation. The very high redundancy of these representations allows us to randomly choose among several representations of the key digits. This may be used as a countermeasure against some side-channel attacks.

Side-channel attacks pose a serious threat to implement cryptographic algorithms. Proposed methods can be countermeasures against some side-channel attacks. In chapter 3, we perform a specific side-channel attack on an implementation using a countermeasure implemented in chapter 2. In particular, a *template attack* is performed to practically evaluate the proposed countermeasure. The advanced statistical attack is performed on an implementation based on ECC. In a template attack, the attacker is assumed to know characteristics of a side channel over some processed data of a device. This characterization is stored and called template. The attacker matches the templates based on different key guess with the recorded traces. Thus, the robustness of the proposed countermeasure based on signed-digit representations which use randomized recoding of a number, is evaluated in this chapter, by performing a practical attack by observation.

Chapter 4 deals with very sparse and redundant representations. Redundant representations based on DBNS or multi-base number system (MBNS) allow to accelerate the encryption. However most of redundant methods require pre-computations and off-line recodings. In this chapter, we provide a fast recoding method which can be fully implemented in hardware with and without pre-computations. It is the first presented hardware implementation. MBNS terms are obtained on-the-fly using a special recoding unit which operates in parallel to curve-level operations and at very high speed. This ensures that all recoding steps are performed fast enough to schedule the next curve-level operations without interruptions. In addition, using such representations enable to add randomness in the scalar recoding. Thus, it can be a protection against some side-channel attacks.

Finally, chapter 5 deals with methods and algorithms which can be performed as a succession of unique patterns. If encryptions have a regular behaviour, attackers may be not allowed to guess the secret key. It can provide countermeasures against some side-channel attacks. When adding randomization in this protection against side-channel attacks, an attack by observation may be even more difficult to perform. A scalar multiplication is implemented with the aforementioned

protections. Thus, we compare cryptosystems with and without countermeasures. In the same way, we compare the evolution of the area and the clock frequency in function of protections and different devices or platforms.

A conclusion finishes this thesis about our overall work, and makes suggestions for future works.

Chapter 1

State of the Art

1.1 Elliptic Curves

The mid-1980s saw the emergence of a fascinating cryptographic idea, that of using elliptic curves in cryptosystems [86], [71]. Basically, *elliptic curve cryptography* (ECC) involves a public curve $E(\mathbb{K})$ where $\mathbb{K}$ is a finite field. In ECC, considered finite fields are $\mathbb{K} = \mathbb{F}_p$ for a large prime p, or binary fields $\mathbb{K} = \mathbb{F}_{2^m}$ for suitable integers m. In this Ph.D. thesis, we only consider elliptic curves over large characteristic fields, i.e. *elliptic curves over the finite field* $\mathbb{F}_p$ with p a large prime. However, all this work can be performed over binary fields with slightly adaptations. The set of points of an elliptic curve E defined over $\mathbb{F}_p$ is denoted $E(\mathbb{F}_p)$.

This section deals with the background on elliptic curves to be able to understand and to use in a cryptographic purpose. This section is particularly based on [114], [115], [74] and [60] for the mathematical aspects of cryptography, and on [34], [64] and [12] for the use of elliptic curves in cryptography.

1.1.1 Definitions

Formally, an *elliptic curve* E defined over a field $\mathbb{K}$ is the set of $\mathbb{K}$-rational points including a point denoted $\mathcal{O} \in E(\mathbb{K})$, called point at infinity. The elliptic curve E is a *non-singular cubic projective curve*. Below, we explain each word of the previous definition:

- A cubic curve is an *algebraic curve of order three* defined by equation $f(x, y, z) = 0$, where $f(x, y, z)$ is a polynomial in x, y and z arguments in $\mathbb{K}$,
- A *projective 2-space* over a field $\mathbb{K}$, denoted $\mathbb{P}^2(\mathbb{K})$, is the set of all $(2+1)$-tuples such that at least one x_i is non-zero modulo the equivalence relation $\mathcal{R}$

$$(x_0, x_1, x_2) \quad \mathcal{R} \quad (y_0, y_1, y_2) \iff \exists \lambda \in \mathbb{K}^*, (x_0, x_1, x_2) = \lambda(y_0, y_1, y_2),$$

- A curve where all points are non-singular is a *non-singular curve* (or smooth curve). Geometrically, it is a curve with a unique tangent at any point. A point P of a curve E defined by the equation $f(x, y, z) = 0$ is a singular point if the three partial derivatives of f are zeros at point P

$$\left.\frac{\partial f}{\partial x}\right|_P = \left.\frac{\partial f}{\partial y}\right|_P = \left.\frac{\partial f}{\partial z}\right|_P = 0.$$

Figure 1.1 presents graphic illustrations of two singular cubic curves over $\mathbb{R}$. Indeed, the two curves have a singular point at the origin. The *discriminant* Δ of curve equations is zero. Reciprocally, $\Delta = 0$ implies that the curve is singular.

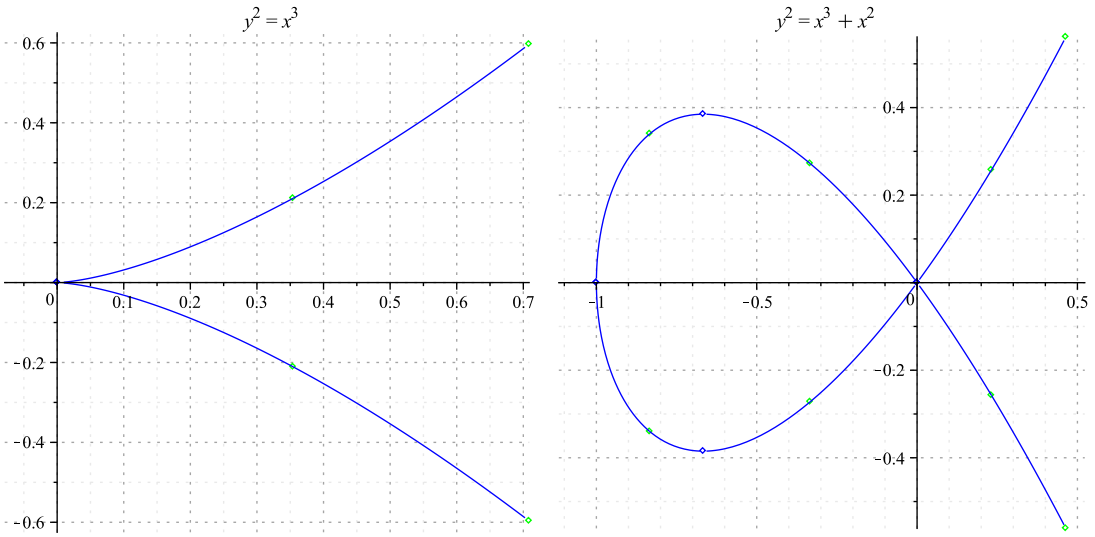


Figure 1.1: Two singular cubic curves (from [114, p. 43]).

1.1.2 Weierstrass Equations

Let an elliptic curve over a field $\mathbb{K}$ be an algebraic non-singular cubic curve. An elliptic curve can be written like a cubic equation of the form (see [34, p. 268])

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3,$$

with $a_1, a_2, a_3, a_4, a_6 \in \mathbb{K}$ and with an additional point, called point at infinity $\mathcal{O} = [0, 1, 0]$.

By a change of variables $x = X/Z$ and $y = Y/Z$, the *Weierstrass equation* of a general elliptic curve over a field $\mathbb{K}$ is of the form

$$E(\mathbb{K}) \ : \ y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

where coefficients a_1, a_2, a_3, a_4, $a_6 \in \mathbb{K}$. With an adequate change of variables, Weierstrass equations defined over a specific $\mathbb{K}$ can be simplified. In particular, if the characteristic of $\mathbb{K}$ is not equal to 2 or 3, the simplified Weierstrass affine equation is

$$E(\mathbb{K}) \ : \ y^2 = x^3 + ax + b,$$

with $a = a_4$ and $b = a_6$.

The curve E is non-singular, and thus is an elliptic curve, if and only if Δ is non-zero over the field $\mathbb{K}$: the *discriminant* Δ of such a curve is $\Delta = -16(4a^3 + 27b^2)$. Thus this curve has a unique tangent at any point.

For faster performances in field-level operations, computations on elliptic curves are often performed with the special case $a = -3$. When using the general case (any a), it is possible to obtain the value $a = -3$ for an isogenous elliptic curve, and to perform computations on this curve rather than on the original one (see [17] for justification).

An isogeny from E_1 and E_2 is a morphism

$$\phi : E_1 \longrightarrow E_2 \qquad \text{satisfying} \qquad \phi(\mathcal{O}) = \mathcal{O}.$$

where E_1 and E_2 are two elliptic curves. E_1 and E_2 are isogenous if there is an isogeny from E_1 to E_2 with $\phi(E_1) \neq \{\mathcal{O}\}$ (it is an equivalence relation).

1.1.3 Group Law

Let E be an elliptic curve given by the Weierstrass equation. Since the equation has degrees three, a line intersects the curve E at exactly three points P, Q and R. When the line is tangent to E, one denotes $P = Q$. Assuming by a natural geometric addition law, the set of points $E(\mathbb{K})$ forms an abelian group. The group of an elliptic curve is the set of $\mathbb{K}$-rational points. $\mathbb{K}$-rational points satisfying Weierstrass equation, with the point at infinity $\mathcal{O}$, form an *abelian group*. Operations in this group are performed using the *chord and tangent rule*. This result is the main reason for the interest in elliptic curves. The *point at infinity* $\mathcal{O}$ is the neutral element (or identity element).

The composition law for curves E defined over $\mathbb{K}$ has the following properties (see [114, p.51] for more details):

- $\forall P, Q \in E$ $\quad P + Q \in E,$ (closure)
- $\forall P \in E$ $\quad P + \mathcal{O} = P,$ (identity element)
- $\forall P, Q \in E$ $\quad P + Q = Q + P,$ (commutativity)
- Let $P \in E$ $\quad \exists (-P) \in E \mid P + (-P) = \mathcal{O},$ (opposite element)
- Let $P, Q, R \in E$ $\quad (P + Q) + R = P + (Q + R).$ (associativity)

Geometrically, the sum of two distinct $\mathbb{K}$-rational points P and Q which differ from $\mathcal{O}$, is obtained by the line which intersects E at a third point R, also a $\mathbb{K}$-rational point. Then $(P + Q) + R = \mathcal{O}$. The opposite of R is the result of the sum $P + Q$. Now finding the opposite of R can be done by setting $R = (x, y)$ giving $-R = (x, -y)$. When P and Q are not distinct, the sum is obtained similarly by considering the tangent to E at that point. This operation is called point doubling. Two operations are the main ones used in cryptography based on elliptic curves: *point addition* $P + Q$ where $P \neq \pm Q$ (denoted by `ADD`) and *point doubling* $[2]P = P + P$ (denoted by `DBL`).

Figure 1.2 illustrates the chord and tangent rule on the elliptic curve $y^2 = x^3 - x + 1$ defined over $\mathbb{R}$. This figure presents the *point addition* $P + Q$ with $P, Q \in E$ where P and Q are distinct (on the left) and where P and Q are the same point (on the right), called *point doubling*.

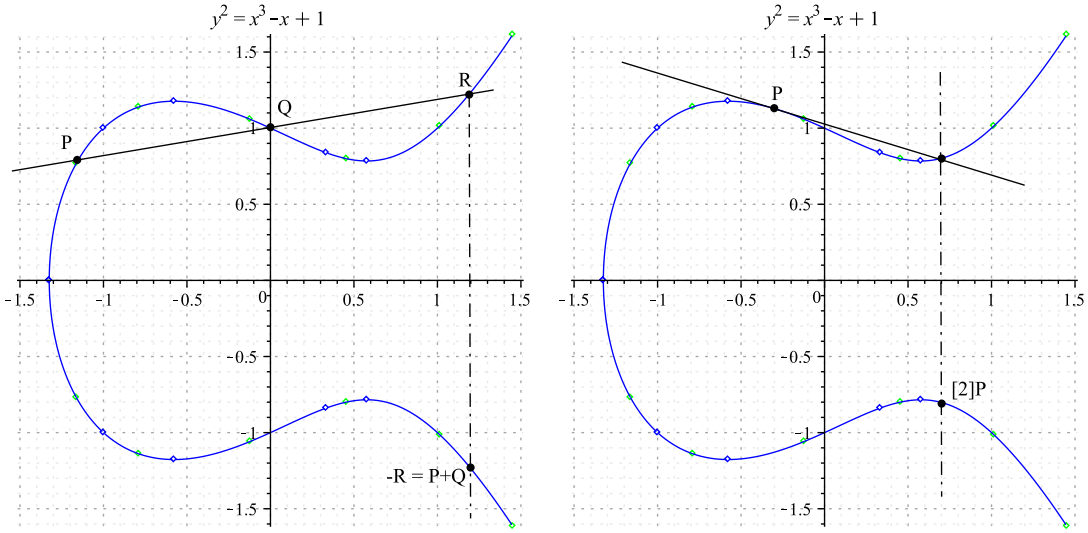


Figure 1.2: Illustration of the group law on $\mathbb{R}$ generated using Maple.

This notation is extended to the *scalar multiplication*, i.e. the computation of

$$[k]P = \underbrace{P + P + \cdots + P}_{k \text{ times}},$$

with $k \in \mathbb{Z}$, a large integer in the range 160–600 bits for typical cryptographic sizes [57, Appendix A]. In addition, we have $[0]P = \mathcal{O}$. For all $k \in \mathbb{Z}$, one can extend the scalar multiplication definition: $[0]P = \mathcal{O}$, $[\#E(\mathbb{F}_p)]P = \mathcal{O}$, and $[k]P = [-k](-P)$ for $k < 0$.

Algebraic formulas for this group law are derived from the geometric description. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points of E. Coordinates of $P_3 = P_1 + P_2$ are:

$$x_3 = \lambda^2 - x_1 - x_2 \qquad y_3 = \lambda(x_1 - x_3) - y_1,$$

$$\text{with } \lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq \pm P_2, \\ \dfrac{3x_1^2 + a}{2y_1} & \text{if } P_1 = P_2. \end{cases}$$

Thus, computing point doublings or point additions is a sequence of operations over the considered field. Note that the presented formulas require one inversion for each curve-level operation.

1.1.4 Discrete Logarithm Problem

Public-key cryptosystems rely on *one-way trapdoor* functions. Computing the inverse of injective functions f^{-1} with $f : A \to B$ is hard except if one knows some specific information. Proving the existence of the one-way trapdoor is still an open problem, but some hard mathematical problems are used as the basis for a trapdoor [116]. In particular, the *discrete logarithm problem* (DLP) relies on the same underlying problem. The security of cryptosystems based on elliptic curves is based on the difficulty of the discrete logarithm problem. Protocols and cryptosystems, like Diffie-Hellman or ElGamal, are also based on the DLP.

The DLP on a multiplicative group G is stated as follows: given an element e of a group and x an element of the order n subgroup generated by e, find the integer $d \in [0, n-1]$ such as $x = e^d$. The elliptic curve discrete logarithm problem (ECDLP) asks for a solution d to the equation $[d]P = Q$ for given points $P, Q \in E(\mathbb{F}_q)$ and $Q \in \langle P \rangle = \{\mathcal{O}, P, [2]P, \cdots, [n-1]P\}$ with $\mathbb{F}_q$ a finite field, and n the order of P.

Up to now, there is no sub-exponential complexity algorithm for solving ECDLP. Some methods solve the ECDLP, like Pollard's ρ (complexity in $O(\sqrt{n})$ [85]) or Shanks' babystep-giantstep (complexity in $O(\sqrt{n})$ but with $O(\sqrt{n})$ storages [32]). The selection of curve parameters is important for security reasons. Curves studied must be robust according to ECDLP.

1.1.5 Security Evaluation

Estimating the number of points on an elliptic curve is a fundamental problem for security evaluation. Indeed, the discrete logarithm problem is easy to solve when the cardinal of the group is smooth, i.e. when numbers are a product of small primes. Thus it is important to verify that the number of points on the curve is divisible by a large prime number for cryptographic applications [34, p. 479]. The number of points in $E(\mathbb{F}_p)$ is denoted $\#E(\mathbb{F}_p)$, and is called the *order* of E over $\mathbb{F}_p$.

The Hasse-Weil interval ensures that the cardinal of a curve is close to p for an elliptic curve. In particular, Hasse's theorem states: $\#E(\mathbb{F}_p) = p + 1 - a_p \quad \text{with} \quad |a_p| \leq 2\sqrt{p}$ with an elliptic curve E defined over a finite field $\mathbb{F}_p$. Thus one has the interval:

$$(\sqrt{p} - 1)^2 \leq \#E(\mathbb{F}_p) \leq (\sqrt{p} + 1)^2.$$

Note that if $\#E(\mathbb{F}_p)$ is prime, $E(\mathbb{F}_p)$ is a cyclic group and any point except $\mathcal{O}$ in $E(\mathbb{F}_p)$ is a generator of $E(\mathbb{F}_p)$ (see [57, p. 84] for an example).

An exponential method to find $\#E(\mathbb{F}_p)$ is presented in [114, p. 372]. In [113] Schoof presents an algorithm which calculates $\#E(\mathbb{F}_p)$ in polynomial time. The idea is to compute the value of $(a_p \bmod l)$ for a great deal of small primes l and then to use the Chinese remainder theorem to reconstruct a_p.

1.1.6 Point Representations

The affine representation (denoted $\mathcal{A}$) is implicitly considered yet such as a point $P = (x, y)$ with $x, y \in \mathbb{F}_p^2$. In this coordinate system, the addition and doubling formulas present sequences of additions (denoted `A`), multiplications (denoted `M`), squares (denoted `S`) and inversions (denoted `I`) over a prime field $\mathbb{F}_p$. Inversion is a relatively expensive operation compared to multiplication in terms of computation time and area. The typical inversion cost assumption in $\mathbb{F}_p$ is about 15 to 45 multiplications ($1\mathtt{I} \approx 15$ to $45\mathtt{M}$) [19]. The use of other coordinate systems enables us to avoid computing modular inversions, and so to accelerate the computation of `ADD` and `DBL`. In practice, the addition cost at the field level is not taken into account because it is a very cheap operation compared to `M`.

The use of *projective coordinates* (denoted $\mathcal{P}$) avoids performing modular inversions. The Weierstrass projective equation over $\mathbb{F}_p$ is:

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3.$$

Projective coordinates solve the modular inversions by adding the third coordinate Z. Inversions are replaced with several other field-level operations. The foundation of these inversion-free coordinate systems can be explained by the concept of *equivalence class*. A point in projective coordinates, denoted $(X : Y : Z)$ is the equivalent class of (X, Y, Z) according to the equivalence relation, that is

$$(X : Y : Z) = \big\{(a, b, c) \in \mathbb{F}_p^3 \backslash (0, 0, 0) \mid \exists \lambda \in \mathbb{F}_p^*,\ (a, b, c) = (\lambda X, \lambda Y, \lambda Z)\big\}.$$

The opposite of point $(X : Y : Z)$ is $(X : -Y : Z)$. The point at infinity $\mathcal{O}$ is the only one which has the coordinate Z equals to zero: $(0, 1, 0)$. Conversion from affine (x, y) to projective coordinates $(X : Y : Z)$ is straightforward. The inverse conversion requires inversions in $\mathbb{F}_p$. In practice, this conversion is only used one time at the end of the computation of $[k]P$ in order to be in affine coordinate.

$$(x, y) \rightarrow (x : y : 1),$$

$$(X : Y : Z \neq 0) \rightarrow (X/Z, Y/Z).$$

Formulas for point addition and doubling are obtained by transforming the projective coordinates into affine coordinates and then applying the formulas in affine. It is then possible to choose the third coordinate Z so as to remove the inversions in $\mathbb{F}_p$. One can find the formulas in [34], [64] or [12].

Several projective coordinate systems have been proposed over $\mathbb{F}_p$ to speed up computation of `ADD` and `DBL` operations:

- standard projective coordinates (denoted $\mathcal{P}$),
- Jacobian projective coordinates (denoted $\mathcal{J}$),
- modified Jacobian coordinates (denoted $\mathcal{J}^m$),
- etc.

Table 1.1 presents the properties of some coordinate systems ("coord"). Note that in all cases, the opposite of the point $(X : Y : Z)$ is $(X : -Y : Z)$.

references	coord	representation of P	conv in $\mathcal{A}$	curve equation	$\mathcal{O}$
[57, pp. 86–95]	$\mathcal{P}$	$(X : Y : Z)$	$(X/Z, Y/Z)$	$Y^2Z = X^3 + aXZ^2 + bZ^3$	$(0 : 1 : 0)$
	$\mathcal{J}$	$(X : Y : Z)$	$(X/Z^2, Y/Z^3)$	$Y^2 = X^3 + aXZ^4 + bZ^6$	$(1 : 1 : 0)$
[33]	$\mathcal{J}^m$	$(X : Y : Z : aZ^4)$	$(X/Z^2, Y/Z^3)$	$Y^2 = X^3 + aXZ^4 + bZ^6$	$(1 : 1 : 0)$

Table 1.1: Properties of different projective coordinate systems.

The choice of the coordinate system is determined by the number of operations at the field level to perform during the computation of point doubling and addition. Table 1.2 compares the cost of `DBL` and `ADD` for several coordinate systems in $\mathbb{F}_p$. These costs come from the web site Explicit-Formulas Database `http://hyperelliptic.org/EFD`. We apply the typical cost assumption used in many references: $1\mathtt{S} = 0.8\,\mathtt{M}$ and general parameters for the curve constants a and b. Note that there is no coordinate system that provides both fastest `DBL` formula and fastest `ADD` formula.

coordinate	ADD	DBL
$\mathcal{A}$	$3\mathtt{M} + 1\mathtt{I}$	$4\mathtt{M} + 1\mathtt{I}$
$\mathcal{P}$	$9.8\mathtt{M}$	$13.6\mathtt{M}$
$\mathcal{J}$	$7.4\mathtt{M}$	$15\mathtt{M}$
$\mathcal{J}^m$	$7\mathtt{M}$	$16.6\mathtt{M}$

Table 1.2: Costs in multiplication and inversion for point doubling and addition in several coordinate systems (from EFD).

To always use the best cost for `DBL` and `ADD`, operations can be performed in mixed coordinates. Cohen *et al.* in [33] change the coordinate system during the point computation to select the most appropriate one. The choice is then based on the efficiency with respect to the concerned operation and the cost of transformation from one coordinate system to another one. For instance, when using projective coordinates and starting from a given affine point (x, y), one easily converts to projective coordinates by adding a one at the third coordinate, i.e. having the projective point $(X : Y : 1)$. To recover the affine point from $(X : Y : Z) \neq \mathcal{O}$, one must first compute Z^{-1} in the considered field. Then, one obtains the affine point (XZ^{-1}, YZ^{-1}).

Let Crd_1, Crd_2 and Crd_3 be three coordinate systems. The mixed addition (denoted `mADD`) of two points in coordinate Crd_1 and Crd_2 respectively and the result is in coordinate Crd_3 is denoted $Crd_1 + Crd_2 \rightarrow Crd_3$. For instance, $\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$ is a point addition with a point in Jacobian and the second in affine coordinate. The result of this point addition is in Jacobian coordinate. The use of mixed coordinates can accelerate the computations.

In this study, we focus only on Jacobian coordinates system and the mixed addition $\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$. Table 1.3 reports computation costs, given in field operations (`M`, `S`) for various curve-level operations over $\mathbb{F}_p$ from literature. For multiple publications from a group of authors, we only

report the best results. λDBL denotes a sequence of λ successive DBL operations, as for the computation of $[2^\lambda]P$. λDBL formula is used when it is faster than their equivalent sequence of DBL: λDBL formula is faster than successive DBL from $\lambda > 8$.

curves	references	curve-level operations		
		ADD	mADD	DBL
$a \neq -3$	EFD	**11**M + **5**S	**7**M + **4**S	**1**M + **8**S
	[40], [41], [87], [75]	12M + 4S	8M + 3S	4M + 6S
	[78], [76]	**11**M + **5**S	**7**M + **4**S	2M + 8S
	[54]	n/a	n/a	**1**M + **8**S
$a = -3$	EFD, [78], [77], [76], [75]	**11**M + **5**S	**7**M + **4**S	**3**M + **5**S
	[40], [41]	12M + 4S	8M + 3S	4M + 4S
curves	references	λDBL		
$a \neq -3$	[40], [41], [61], [87]	4λM + $(4\lambda + 2)$S		

Table 1.3: Costs of curve-level operations from literature and curves over $\mathbb{F}_p$.

1.1.7 Scalar Multiplication

Scalar multiplication is the most time consuming operation in ECC based protocols. Let E be an elliptic curve defined over $\mathbb{F}_p$ and $P \in E(\mathbb{F}_p)$ be a point on E, with $\mathbb{F}_p$ a large prime field. Scalar multiplication is denoted by $[k]P$ where P is a curve point and k a scalar, i.e. a large integer used as private/public key in protocols.

Figure 1.3 illustrates the typical number of operations required at each level. One $[k]P$ requires hundreds of curve-level operations (k is n bits long with n about 160–600). Each curve-level operation (ADD, DBL) requires a sequence of 8–12 field-level operations. Finally, each field operation requires tens (for large operators) to hundreds (for small iterative operators) of clock cycles.

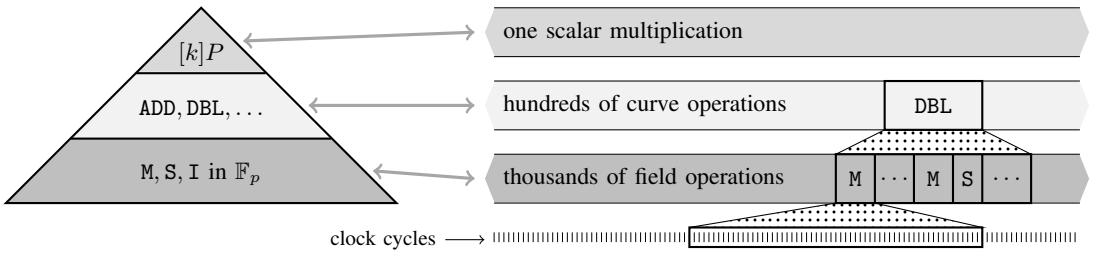


Figure 1.3: Pyramid of operations in a scalar multiplication (arbitrary scale).

The scalar k is an n-bit integer $k = (k_{n-1}k_{n-2}\dots k_1k_0)_2$. Usually n and the size of p (#bits) are similar and in the range 160–600 bits for typical cryptographic sizes.

Algorithm in figure 1.4 presents the two classic $[k]P$ algorithms, called *double-and-add*, starting from least and most significant bits of k. They are based on the following observation:

$$\begin{aligned} [k]P &= k_0P + k_1([2]P) + k_2([4]P) + \cdots + k_{n-1}([2^{n-1}]P), \\ [k]P &= k_0P + [2]\Big(k_1P + [2]\big(k_2P + \cdots + [2](k_{n-1}P)\big)\Big). \end{aligned}$$

In this algorithm, the point addition operation $Q \leftarrow Q + P$ is only performed when the ith bit of the scalar k equals to 1.

input: $k = (k_{n-1}k_{n-2}\dots k_1k_0)_2, \quad P \in E(\mathbb{F}_p)$

output: $Q = [k]P$

1:	$Q \longleftarrow \mathcal{O}$		$Q \longleftarrow \mathcal{O}$	
2:	**for** i **from** 0 **to** $n-1$ **do**		**for** i **from** $n-1$ **downto** 0 **do**	
3:	**if** $k_i = 1$ **then** $Q \longleftarrow Q + P$	(ADD)	$Q \longleftarrow [2]Q$	(DBL)
4:	$P \longleftarrow [2]P$	(DBL)	**if** $k_i = 1$ **then** $Q \longleftarrow Q + P$	(ADD)
5:	**return** Q		**return** Q	

Figure 1.4: Right-to-left and left-to-right binary "double-and-add" algorithms to compute $[k]P$.

The double-and-add algorithm is not unique to elliptic curves. It is applicable to any group. When the group law is written multiplicatively, the double-and-add algorithm is called "square-and-multiply". The average runtime of double-and-add algorithm to compute $[k]P$ is:

$$\lfloor \log_2 k \rfloor \, \mathtt{DBL} + \frac{1}{2}\big(1 + \lfloor \log_2 k \rfloor\big)\mathtt{ADD} = (n-1)\mathtt{DBL} + \frac{n}{2}\mathtt{ADD},$$

since the binary expansion of the integer k has in average $n/2$ number of zeros. The theoretical security of ECDLP comes from the fact that given two points P and Q such that $Q = [k]P$, finding the integer k is not feasible in practice (for well chosen curves), see [57, Sec. 4.1] for details and theoretical attacks.

The computation of the opposite of a point $(-Q)$ is straightforward. Thus point subtraction (SUB) $P - Q$ is as efficient as point addition ($\mathcal{A}$: $-P = (x, -y)$ and $\mathcal{J}$: $-P = (X : -Y : Z)$ for curves over $\mathbb{F}_p$). This motivates the use of signed digits. A *non-adjacent form* (NAF) of k is a representation where in two consecutive signed digits $k_i \in \{-1, 0, 1\}$, maximum one digit is different to zero, i.e., $k_{i+1}k_i = 0$ for all $i \geq 0$ (see [57, Sec. 3.3.1] for details and conversion algorithm). A scalar k has a unique NAF representation.

Scalar multiplication using NAF recoding is straightforward: replace lines

"**if** $k_i = 1$ **then** $Q \leftarrow Q + P$",

in algorithm on figure 1.4 by

"**if** $k_i \neq 0$ **then** $Q \leftarrow Q \; + \; \mathrm{sign}(k_i)\; P$",

where $\mathrm{sign}(k_i)$ is the sign of the digit k_i. The computation cost is on average $\frac{n}{3}\,\mathtt{ADD} + n\,\mathtt{DBL}$ (cost for point subtraction is ADD). Indeed, the average number of non-zero digits among all NAF representations is $n/3$ (see [14] for an analysis of the NAF density).

Another optimization, called wNAF, processes a window of w digits of k (with larger k_i values). The representation wNAF uses digits $k_i \in \{0, \pm 1, \pm 3, \pm 5, \dots, \pm 2^{w-1} - 1\}$, and at most one of any w consecutive digits is non-zero (see [57, Sec. 3.3.1], NAF is wNAF for $w = 2$). Thus wNAF guarantees that on average there is a density of $\frac{1}{w+1}$ non-zero digits in the key representation, that is $\frac{n}{w+1}$ point additions.

$$\mathtt{wNAF}(k) = \sum_{i=0}^{n} k_i 2^i \qquad \text{with } |k_i| < 2^{w-1}.$$

Thus if $w = 2$, $k_i \in \{0, \pm 1\}$; if $w = 3$, $k_i \in \{0, \pm 1, \pm 3\}$; if $w = 4$, $k_i \in \{0, \pm 1, \pm 3, \pm 5, \pm 7\}$; etc. Multiples of P, the constant points $P_j = [j]P$, must be pre-computed and stored for all $j \in \{3, 5, 7, \ldots, 2^{w-1} - 1\}$.

For instance, if $k = 763 = (1011111011)_2$, the wNAF representation of k for $2 \leq w \leq 4$ is:

$$\begin{aligned} 2\text{NAF}(k) &= 10\bar{1}00000\bar{1}0\bar{1}, \\ 3\text{NAF}(k) &= 0030000\bar{1}003, \\ 4\text{NAF}(k) &= 00300000005\bar{}. \end{aligned}$$

In practice, wNAF is used with $w \leq 4$ for limited storage overhead. Scalar multiplication is done using point addition/subtraction (depending of the sign of k_i) of pre-computed multiple P_j. The corresponding pseudo-code is

"**if** $k_i \neq 0$ **then** $Q \leftarrow Q + \text{sign}(k_i)\, P_{|k_i|}$".

The computation cost is on average $\frac{n}{w+1}$ ADD + n DBL without the pre-computation step. This type of pre-computation may be interesting if the same point P is reused.

The representation wNAF has a window of w digits and can skip consecutive zeros after a non-zero digit k_i is processed. In this sense, an optimization of the simple wNAF scalar multiplication can be performed by a sliding window algorithm for scalar multiplication [34, Sec. 13.1.2]. Sliding method is more efficient than the scalar algorithm with wNAF, but with a higher storage of pre-computed points.

1.2 Double-Base and Multi-Base Number System

Various methods have been proposed to speed up scalar multiplication. Among them scalar recoding is popular: non-adjacent form (wNAF), double-base number system (DBNS, typically with bases 2 and 3), multi-base number system (MBNS with bases such as $(2, 3, 5, 7)$) [42]. DBNS and MBNS represent numbers as the sum of terms such as $\pm 2^u 3^v 5^c 7^d$. Fast recoding methods can require pre-computations: multiples of base point for wNAF and off-line conversion for DBNS and MBNS. In this section, we present DBNS and MBNS for ECC.

1.2.1 Double-Base Number System

Double-base number system (DBNS) was initially introduced in [38], used for modular exponentiation in [43], for signal processing in [44] and for ECC in [40], [46], [6], [7], [45] and [41] (which is very complete).

The concept of DBNS has some advantages in implementing elliptic curve scalar multiplication. The DBNS simultaneously uses two bases for representing numbers. In most works, bases are 2 and 3. In this work, we assume the same choice (i.e. $\mathcal{B} = (2, 3)$). Thus an integer k is a sum of n' terms and is represented by:

$$k = \sum_{i=0}^{n'-1} s_i 2^{u_i} 3^{v_i},$$

where $s_i = \pm 1$ and $(u_i, v_i) \in \mathbb{N}$. The size (or length) of a DBNS expansion is equal to the number of terms n', which can be different to the number of bits n of the integer. Most of the time, n' is small compared to n: one has $n' \ll n$. Triplets (s_i, u_i, v_i) are called terms and are the "digits" of the DBNS expansion. DBNS is a sparse number system (i.e. the number of terms is small).

Whether one considers signed ($s_i = \pm 1$) or unsigned ($s_i = 1$) expansions, this representation is a redundant number system (i.e. some numbers have several representations). An integer which can be written as a sum of n' terms, but cannot be represented with $(n'-1)$ or fewer terms is called canonic double-base representation. For instance, the value 127 has 783 different unsigned DBNS representations among them six canonic representations which can be written as a sum of three terms [8]:

$$\begin{aligned} 127 &= 2^2 3^3 + 2^1 3^2 + 2^0 3^0 \\ &= 2^2 3^3 + 2^4 3^0 + 2^0 3^1 \\ &= 2^3 3^2 + 2^1 3^3 + 2^0 3^0 \\ &= \cdots \end{aligned}$$

The total number $f(k)$ of unsigned DBNS for an integer $k > 0$ is given by the following recursive function (see [41] for details):

$$f(k) = \begin{cases} 1 & \text{if} \quad k = 1, \\ f(k-1) + f(k/3) & \text{if} \quad k \equiv 0 \pmod 3, \\ f(k-1) & \text{otherwise.} \end{cases}$$

For instance, 10, 100 and 1 000 have 5, 402 and 1 295 579 different DBNS representations, respectively. For large numbers, finding a representation of minimal length, in a reasonable amount of time, is a very large problem. One can use the greedy approach presented in figure 1.5 to quickly find a representation, based on the determination of the best default approximation of k by a term, i.e. the largest integer $\leq k$ of the form $2^u 3^v$.

Converting Numbers to DBNS

Finding one of the best DBNS expansion of an integer in an efficient way can be done by the following algorithm in figure 1.5 from [44]. LT denotes the list of terms which stores the DBNS recoding of k. Each term corresponds to a triple (s, u, v) with $s = \pm 1$ such that $k = \sum_{i=0}^{n'-1} s_i 2^{u_i} 3^{v_i}$.

input: a positive integer k
output: list of terms LT which stores the DBNS recoding of k
$\text{LT} = \big((s_1, u_1, v_1), (s_2, u_2, v_2), \dots\big)$

```
1: LT ← ∅
2: s ← 1
3: while k ≠ 0 do
4:     find the best approximation of k of the form 2^u 3^v
5:     LT ← LT ∪ (s, u, v)
6:     if k < 2^u 3^v then
7:         s ← −s
8:     k ← |k − 2^u 3^v|
9: return LT
```

Figure 1.5: Greedy algorithm to convert integers into DBNS (from [44]).

The greedy algorithm provides expansions of length n'. It terminates after $n' = \left(\frac{\log k}{\log\log k}\right)$ steps [43], and enables to very quickly find a DBNS expansion of a given integer. This algorithm finds the best approximation of the form 2^u3^v of k, and then computes $|k - 2^u3^v|$. This method is applied until reaching zero. See the following example from [46] for an illustration.

Example. *Let $k = 841\,232$. One has the sequence of approximations:*

$$\begin{aligned} 841\,232 &= 2^73^8 + 1\,424 \\ 1\,424 &= 2^13^6 - 34 \\ 34 &= 2^23^2 - 2 \end{aligned}$$

Therefore, the recoding is: $841\,232 = 2^73^8 + 2^13^6 - 2^23^2 + 2^1$.

This approach sometimes fails in finding a minimal representation. For example, the minimal representation of 41 is $32 + 9 = 2^53^0 + 2^03^2$ whereas algorithm in figure 1.5 returns $41 = 36 + 4 + 1 = 2^23^2 + 2^23^0 + 2^03^0$.

The crucial problem of this algorithm is to find the best default approximation of k (line 3 of algorithm in figure 1.5) of the form 2^u3^v. A method has been proposed in [93] using lookup-tables with specific addressing scheme. However this approach may be too costly for cryptographic sizes, and unrealistic for hardware implementation. Another method, presented in [9], uses continued fractions, Ostrowski's number systems and diophantine approximation. This method can be reformulated as finding integers u and v

$$u\log 2 + v\log 3 \leq \log k,$$

such that no other integer gives a better approximation to $\log k$. An approach can consist in scanning points with integer coordinates near the line $y = -k\log_3 2 + \log_3 k$. One only keeps the best approximations, that is points which have the smallest vertical distance to the right line [9].

Another method, a tree based approach is proposed in [45] for binary to DBNS conversion. Assuming that the number k is coprime to 6, a tree is built by considering $k-1$ and $k+1$: the powers of 2 and 3 are removed from $k-1$ and $k+1$ as much as possible. For each node, 1 is added and subtracted, and then the process is reapplied until reaching 1. Repeating this will create a binary tree.

These methods allow to recode an integer into DBNS. This recoding leads to a very sparse representation. This inherent spareness of DBNS leads to fewer curve-level operations in elliptic curves. That is why DBNS can be very attractive when considering ECC.

Double-Base Number System for Elliptic Curve Cryptography

In curve-based cryptosystems, the core operation that needs to be optimized as much as possible is the scalar multiplication $[k]P$. For ECC computations in DBNS, a new curve-level operation needs to be defined: *point tripling* $[3]P = P+P+P$ (denoted by `TPL`). It is faster than an addition point (`ADD`) in Jacobian coordinates (see below in table 1.5 for costs of curve-level operations). Naive scalar multiplications in DBNS can be computed in a simple way: for each term (s_i, u_i, v_i) in the expansion, required operations are u_i `DBL`s, v_i `TPL`s and `ADD`/`SUB` accordingly to the symbol of s_i, $\text{sign}(s_i)$. Several variants with close performances exist. DBNS is a very sparse representation. Then, the number of point additions is reduced: the computing cost of the scalar multiplication follows a sublinear complexity of less than $(\log k/\log\log k)$ additions [43], where

k is the scalar.

In [40] and [46] a special type of DBNS recoding is proposed, called *DBNS chain*. The scalar multiplication can be efficiently computed using DBNS with non-increasing exponents, where u_0 is considered the largest exponent of powers of 2, while $u_{n'-1}$ is the smallest one (the same notation applies for v_i exponents and powers of 3)

$$u_0 \geq \cdots \geq u_{n'-1} \quad \text{and} \quad v_0 \geq \cdots \geq v_{n'-1}.$$

Finding a DBNS expansion with non-increasing exponents [41] can be computed using the greedy algorithm in figure 1.6. LT denotes the list of terms which stores the DBNS recoding of k.

input: a positive integer k, and $u_{max}, v_{max} > 0$ the largest allowed binary and ternary exponents
output: list of terms LT which stores the DBNS recoding of k
$\text{LT} = \big((s_1, u_1, v_1), (s_2, u_2, v_2), \dots\big)$, with $u_i \leq u_{i-1}$ and $v_i \leq v_{i-1}$ for $i \geq 1$

1: $\text{LT} \leftarrow \emptyset$
2: $s \leftarrow 1$
3: **while** $k > 0$ **do**
4: Find the best approximation of k of the form $2^u 3^v$ with $0 \leq u \leq u_{max}$ and $0 \leq v \leq v_{max}$
5: $\text{LT} \leftarrow \text{LT} \cup (s, u, v)$
6: $u_{max} \leftarrow u \qquad v_{max} \leftarrow v$
7: **if** $k < 2^u 3^v$ **then**
8: $s \leftarrow -s$
9: $k \leftarrow |k - 2^u 3^v|$
10: **return** LT

Figure 1.6: Greedy algorithm to convert integers into DBNS with non-increasing exponents (from [41]).

Parameters u_{max} and v_{max} are the upper bounds for the exponents in the expansion of k. Clearly, one can suppose that inputs u_{max} and v_{max} require $u_{max} < \log_2(k) < n$ and $v_{max} < \log_3(k)$.

Let $t_i = 2^{u_i} 3^{v_i}$. Using a non-increasing sequence of exponents, the value $k = \sum_{i=0}^{n'-1} s_i t_i$ can be factorized by $t_{n'-1}$:

$$k = \left(\sum_{i=0}^{n'-2} s_i t'_i + s_{n'-1} \right) \times t_{n'-1},$$

where $t'_i = 2^{u_i - u_{n'-1}} 3^{v_i - v_{n'-1}}$. In the same way, one can factorize $\sum_{i=0}^{n'-2} s_i t'_i$. This computation can be applied until reaching ± 1. This method is similar to Horner scheme.

For instance, $k = 15\,679$ can be recoded into DBNS with non-increasing exponents:

$$15\,679 = 2^6 3^5 + 2^2 3^3 + 2^1 3^2 + 2^0 3^0 = 2^1 3^2 \big(2^1 3^1 (2^4 3^2 + 1) + 1\big) + 1.$$

Without considering non-increasing exponents, the computation cost of $[15\,679]P$ is $3\texttt{ADD} + 9\texttt{DBL} + 10\texttt{TPL} = (n'-1)\texttt{ADD} + \big(\sum_{i=0}^{n'-1} u_i\big)\texttt{DBL} + \big(\sum_{i=0}^{n'-1} v_i\big)\texttt{TPL}$ operations. When one considers non-increasing exponents, the cost of $[15\,679]P$ reduces to only

$3\texttt{ADD} + 6\texttt{DBL} + 5\texttt{TPL} = (n' - 1)\texttt{ADD} + u_0\texttt{DBL} + v_0\texttt{TPL}$ operations.

In [40], a scalar multiplication algorithm dedicated to DBNS with non-increasing exponents is proposed. This algorithm is given in figure 1.7 with k in DBNS chain. The cost of this algorithm is

$$(n' - 1)\texttt{ADD} + u_0\texttt{DBL} + v_0\texttt{TPL},$$

operations at the curve level. The number of terms in the DBNS expansion is represented by n'. To evaluate this cost at the field level (number of multiplication), one must consider what formulas are used. Indeed, when one uses the formula $\lambda\texttt{DBL}$ (resp. $\lambda\texttt{TPL}$) for λ successive DBL (resp. TPL), u_0 (resp. v_0) cannot be replaced by λ to have the cost at the field level: λ must be replaced by each factorized term ($\sum_{i=0}^{n'-2} u_i - u_{i+1}$) for point doublings ($\sum_{i=0}^{n'-2} v_i - v_{i+1}$ for point triplings). This is due to the fact that the formulas of $\lambda\texttt{DBL}$ and $\lambda\texttt{TPL}$ are not a product of a function of λ. For instance, a complexity in Jacobian for $\lambda\texttt{DBL}$ is $4\lambda\texttt{M} + (4\lambda + 2)\texttt{S}$:

$$5\texttt{DBL} = 20\texttt{M} + 22\texttt{S} \quad \neq \quad 3\texttt{DBL} \quad + \quad 2\texttt{DBL} \quad = \quad 20\texttt{M} + 24\texttt{S},$$

$$\underbrace{\hspace{8em}}_{}\!\!\overset{5\texttt{DBL}}{\vdash\!\!\!\longrightarrow\!\!\dashv} \neq \overset{3\texttt{DBL}}{\vdash\!\!\longrightarrow\!\!\dashv} + \overset{2\texttt{DBL}}{\vdash\!\!\longrightarrow\!\!\dashv}.$$

---

input: a point $P \in E(\mathbb{F}_p)$, and an integer $k = \sum_{i=0}^{n'-1} s_i 2^{u_i} 3^{v_i}$, with $s_i = \pm 1$, such that $u_0 \geq \cdots \geq u_{n'-1}$ and $v_0 \geq \cdots \geq v_{n'-1}$
output: the point $Q = [k]P \in E(\mathbb{F}_p)$

$Q \leftarrow s_0 P$
for i **from** 0 **to** $n' - 2$ **do**
  $u \leftarrow u_i - u_{i+1}$ $v \leftarrow v_i - v_{i+1}$
  $Q \leftarrow [2^u]Q$ $Q \leftarrow [3^v]Q$
  $Q \leftarrow Q + s_{i+1}P$
if $u_{n'-1} \neq 0$ **then**
  $Q \leftarrow [2^{u_{n'-1}}]Q$
if $v_{n'-1} \neq 0$ **then**
  $Q \leftarrow [3^{v_{n'-1}}]Q$
return Q

Figure 1.7: Right-to-left scalar multiplication algorithm (from [40]).

Table 1.4 presents performance of DBNS scalar multiplication from literature in terms of costs in field-level operations, evaluation for $a = -3$ and $a \neq -3$ (any a), over $\mathbb{F}_p$, in Jacobian coordinates, and $n = 160$ bits. Results can be different according to the authors because the input values u_{max} and v_{max} in figure 1.6 can be different when one converts an integer into DBNS with non-increasing exponents. The length of a representation can increase, but the maximum powers of 2 and 3 can be smaller.

1.2.2 Multi-Base Number System

Multi-base number system (MBNS) is a generalization of DBNS with more than two bases [87], [75], [77], and [102]. There are several other works on MBNS but without comparisons to state-of-the-art (e.g. [103]). A multi-base $\mathcal{B}$ is a tuple of l co-prime integers $(b_1, b_2, \ldots, b_l)$. Number

curves	references	number of required multiplications	pre-computations
$a \neq -3$	[40]	1 863.0M	$\emptyset$
	[41]	1 722.3M	$\emptyset$
	[7]	1 558.4M	7 points
	[46]	1 615.3M	$\emptyset$
$a = -3$	[46]	1 563.2M	$\emptyset$
	[7]	1 504.3M	7 points

Table 1.4: Comparison of DBNS scalar multiplication methods (curves with $n = 160$).

x is represented as the sum of terms $x = \sum_{i=0}^{n'-1} \left(s_i \prod_{j=1}^{l} b_j^{e_{j,i}} \right)$ with $s_i = \pm 1$. MBNS is a very sparse and redundant representation. In literature, proposed multi-bases are often $(2, 3, 5)$ and $(2, 3, 5, 7)$. MBNS are even shorter and more redundant than DBNS. The number of representations grows very fast in the number of base elements (see the formula in [88] for details). For example (from [87]), 100 has 402 DBNS representation with the bases $(2, 3)$, 8 425 representations using the bases $(2, 3, 5)$, and has 43 777 representations using the bases $(2, 3, 5, 7)$ (considering only unsigned representations, i.e. $s_i = 1$).

For ECC computations in MBNS, new curve-level operations need to be defined: *point quintupling* $[5]P$ (QPL), *point septupling* $[7]P$ (SPL), *point eleventupling* $[11]P$ (EPL), etc. These new operations are more efficient than equivalent sequences of ADD, DBL and TPL operations.

Table 1.5 reports computation costs, given in field-level operations (M, S) for various curve-level operations over $\mathbb{F}_p$ from literature. For multiple publications from a group of authors, we only report the best results. We remind that λDBL (resp. λTPL) denotes a sequence of λ successive DBL (resp. TPL) operations (e.g. $k = 2^\lambda$ or $k = 3^\lambda$).

In [87] and [102] conversion into a MBNS chain uses good/best approximations of k using terms of form $\pm \prod_{j=1}^{l} b_j^{e_j}$ similarly to DBNS conversion. Using such a strategy is possible but is very difficult: the conversion in a MBNS chain is away from the optimal. In [75] and [77] conversion uses an adaptation of wNAF to multi-base (with detection of b_j multiples into a limited window) but it requires pre-computations and additional storage. Digits of the conversion algorithm are generated by repeatedly dividing the positive integer k by the bases, allowing remainders of $\left\{0, \pm 1, \ldots, \pm \left\lfloor \frac{b_j^2-1}{2} \right\rfloor \right\} \setminus \left\{ \pm b_j, \pm 2b_j, \ldots, \pm \left\lfloor \frac{b_j-1}{2} \right\rfloor \right\}$ for each base b_j. The conversion ensures that no consecutive digits are non-zero. Similarly to NAF, the authors define a window w non-adjacent form for multi-base representations. To our knowledge, [77] and [75] provide the best MBNS results but without hardware implementation.

Once a scalar is converted into MBNS, scalar multiplication is similar to DBNS algorithms with more curve-level operations (e.g. QPL, SPL, etc.). MBNS helps to reduce the total cost of curve-level operations compared to DBNS, but it has the same limitation: the need for off-line conversion with huge tables and/or long pre-computations.

1.3 Arithmetic in a Large Prime Field

Efficient implementation of arithmetic operations on finite fields is an important pre-requisite in ECC. Indeed, operations on elliptic curves are performed using operations in the prime field $\mathbb{F}_p$. Figure 1.3 shows that each curve-level operation requires several operations at the field level. We will therefore focus in this section on modular arithmetic for large elements (160–600 bits).

curves	references	curve-level operations			
		TPL	QPL	SPL	EPL
$a \neq -3$	EFD	**5M + 10S**	n/a	n/a	n/a
	[40], [41]	10M + 6S	n/a	n/a	n/a
	[87]	10M + 6S	15M + 10S	n/a	n/a
	[78]	6M + 10S	10M + 14S	17M + 14S	27M + 18S
	[54]	**5M + 10S**	**7M + 16S**	15M + 24S	**17M + 30S**
	[76]	6M + 11S	9M + 15S	**13M + 18S**	n/a
	[75]	9M + 7S	14M + 10S	19M + 12S	29M + 16S
$a = -3$	EFD	**7M + 7S**	n/a	n/a	n/a
	[40], [41]	10M + 6S	n/a	n/a	n/a
	[87]	n/a	15M + 8S	n/a	n/a
	[78]	**7M + 7S**	11M + 11S	18M + 11S	**28M + 15S**
	[77], [76]	7M + 8S	**10M + 12S**	**14M + 15S**	n/a
	[75]	9M + 5S	14M + 8S	19M + 10S	29M + 14S

curves	references	λTPL
$a \neq -3$	[40], [41], [61]	$(11\lambda - 1)$M + $(4\lambda + 2)$S
	[87]	10λM + $(6\lambda - 5)$S

curves	references	λTPL / λ'DBL
$a \neq -3$	[40], [41]	$(11\lambda + 4\lambda' - 1)$M + $(4\lambda + 4\lambda' + 3)$S

Table 1.5: Costs of curve-level operations from literature and curves over $\mathbb{F}_p$.

A natural number x is represented in a positional system with a radix of 2 on n bits. Thus we have

$$x = (x_{n-1}x_{n-2}\cdots x_1x_0)_2 = \sum_{i=0}^{n-1} x_i 2^i.$$

Notation $()_2$ means that elements into brackets are the representation bits with radix of 2, least significant on the right.

1.3.1 Definitions and Properties

Let p be a large prime integer. $\mathbb{F}_p$ is the field $\mathbb{Z}/p\mathbb{Z}$ with $\mathbb{Z}/p\mathbb{Z} = \{r + p\mathbb{Z}, r \in [0, p-1]\}$. In other words, one has:

$$\forall x \in \mathbb{F}_p,\ \exists! r \in [0, p-1] \ \text{ such as } \ r \equiv x \pmod{p}.$$

The field $\mathbb{F}_p$ has a finite cardinal of p elements. Thus, the *characteristic* of $\mathbb{F}_p$, denoted $char(\mathbb{F}_p)$, is the additive order of 1. Thus with p a prime number, $char(\mathbb{F}_p) = p$, and $\mathbb{F}_p$ is a prime field of characteristic p.

1.3.2 Modular Addition

The classic modular addition is presented in Fig 1.8. For two elements $(a, b) \in \mathbb{F}_p^2$, the algorithm computes $a + b$ and $a + b - p$. It returns $a + b$ when $a + b - p < 0$, else $a + b - p$.

---

input: p a prime number, and $(a, b) \in \mathbb{F}_p^2$
output: $(a + b) \bmod p$

1: $T \leftarrow a + b$
2: $T' \leftarrow T - p$
3: **if** $T' < 0$ **then**
4: **return** T
5: **else**
6: **return** T'

Figure 1.8: Classic modular addition.

Omura [96] optimizes modular addition by avoiding making a comparison and by calculating only additions. Line 2 of the classical algorithm is replaced by $T' \leftarrow T - m$ with $m = 2^n - p$. The Omura modular addition returns $(T' \bmod 2^n)$ if $T' \geq 2^n$, else $T = a + b$. This comparison and the operation $\pmod{2^n}$ are straightforward when one considers radix 2.

The two previous modular additions are dependent on the carry propagations. Redundant representations like *carry save* and *borrow save* can be advantageous in avoiding the delay of carry or borrow propagation. However whereas conversion from binary to redundant representations is straightforward, the inverse conversion is not. In addition, comparison of two numbers in redundant representation is more complex than in binary. Thus, these representations can be used only if several successive additions must be performed. For more details on these representations, addition algorithms and corresponding implementations, one can see [92, Chap. 2] for more details.

Methods for *modular subtraction* are similar to *modular addition* methods. For two elements $(a, b) \in \mathbb{F}_p^2$, a modular subtraction preforms $a - b$ and $a - b + p$. It returns $a - b$ when $a - b \geq 0$, or $a - b + p$ otherwise.

1.3.3 Montgomery Method

Montgomery [89] introduced a new way to represent elements of $\mathbb{Z}/p\mathbb{Z}$. Numbers are represented in a so called *Montgomery representation* which is especially used for modular multiplication.

Modular multiplication over $\mathbb{F}_p$ consists in computing $T = (a \times b) \bmod p$ with $(a, b) \in \mathbb{F}_p^2$. The classical algorithm for performing such a modular multiplication consists in computing the multiple-precision multiplication $a \times b$, and then performing modular reduction. The classic reduction is the Euclidean division. For an integer x and a positive integer p, a basic relation is

$$x \bmod p = x - p \lfloor x/p \rfloor ,$$

which is equivalent to the quotient/remainder decomposition $x = qp + r$, where r is the modulo result $(x \bmod p)$. Thus, the division operation which is the number q begets a modulo. By this equation, it is possible to find a power of the base $\mathcal{B}^m$ such that $\mathcal{B}^m p \leq x \leq \mathcal{B}^{m+1} p$. The quotient $\lfloor x/\mathcal{B}^m p \rfloor \in [1, \mathcal{B} - 1]$. Thus one can replace x by $x - \mathcal{B}^m p \lfloor x/\mathcal{B}^m p \rfloor$, divide $\mathcal{B}^m p$ by $\mathcal{B}$ (that is a

right shift by one digit), and repeat this method until one has the result of the division $\lfloor x/p \rfloor$ (see [36] for details). No multiplications are performed when one considers $\mathcal{B}$ as the base.

Methods exist for which no divisions are required: the Montgomery method eliminates the division step by computing $(xR^{-1} \bmod p)$, with R a power of the base. Let p and R be two coprime positive integers and $p' \equiv -p^{-1} \bmod R$. Then, for any integer x, the number $y = x + p(xp' \bmod R)$ is divisible by R with $y/R \equiv xR^{-1} \bmod p$. In addition, if $0 \leq x \leq Rp$, the difference $y/R - (xR^{-1} \bmod p)$ is either 0, or p. In our case, p is a prime number, and when R is a power of the base 2, the operation $\pmod R$ is trivial: it is a division by R to obtain y:

$$y/R \equiv xR^{-1} \bmod p = (x + p(\underbrace{(xp')\&(R-1)}_{xp' \bmod 2^s})) \gg s,$$

with $R = \mathcal{B}^s = 2^s$ and "$\gg s$" a right shift of s bits.
Hence for $0 \leq x \leq Rp$, one has a way to compute $xR^{-1} \bmod p$. It is called the *Montgomery reduction* of x. Montgomery reduction is a generalization of an old method due to Hensel in [59] (see [15] for more details). With a suitable choice of R, a Montgomery reduction can be efficiently computed. The typical choice for R is 2^n with p is n-bit long. It implies that $R > p$ and $gcd(p, R) = 1$.

If R and p are coprime and $0 \leq x \leq p$, the residue class (R, p) of x, or *Montgomery representation*, denoted by square brackets, of x is

$$[x] = xR \bmod p.$$

1.3.4 Modular Multiplication

Modular multiplication operation is widely used in the context of public key cryptography, including RSA and ECC. Modular multiplication over a prime field $\mathbb{F}_p$, with p a big prime, consists in computing $(a \times b \bmod p)$ with $(a, b) \in \mathbb{F}_p^2$.

The *Montgomery multiplication*, denoted MM of integers a and b is defined by

$$\mathrm{MM}(a, b) = abR^{-1} \bmod p.$$

One has the following properties with $0 \leq a, b \leq p$:
- $\mathrm{MM}([a], [b]) = [ab]$,
- $\mathrm{MM}([a], 1) = a \bmod p$.

Figure 1.9 presents a naive algorithm for multiplying two integers a and b modulo a prime p, by applying the reduction of Montgomery.

Below, we explain possible improvements and provide corresponding algorithms (from [49]). Instead of computing the product of two integers and then performing the reduction of the result, one can alternate partial products and their accumulations with reductions. It allows to compute $p'_0 = p^{-1}$ modulo the radix, instead of p'. In addition, we consider binary representations. Thus $p = \sum_{i=0}^{n-1} p_i 2^i$ with $p_i \in \{0, 1\}$, $R = 2^n$, and $p'_0 \equiv 1 \bmod 2$. Whereas the algorithm in figure 1.9 can be in subquadratic time (there exist subquadratic methods for multiplication), we use a method which is in quadratic time. Indeed, such a method can be easily parallelizable and more efficient in hardware. Moreover, the operation $\pmod{2^n}$ is just a truncation. The algorithm in

input: p a prime number, $(a, b) \in \mathbb{F}_p^2$, and $R = b^n$
with $gcd(p, b) = 1$, all in radix b representation
output: $\text{MM}(a, b) = TR^{-1} \bmod p$ with $T = ab \bmod p$

1: $p' \leftarrow -p^{-1} \bmod R$
2: $T \leftarrow ab$
3: $M \leftarrow Tp' \bmod R$
4: $T \leftarrow T + Mp$
5: **return** T/R

Figure 1.9: Montgomery multiplication (from [49]).

input: p a prime number, and $(a, b) \in \mathbb{F}_p^2$ on n bits
output: $\text{MM}(a, b) = TR^{-1} \bmod p$ with $T = ab \bmod p$

1: $T \leftarrow 0$
2: **for** i **from** 0 **to** $n - 1$ **do**
3: $\quad T \leftarrow T + a_i b$
4: $\quad parity = T_0$
5: $\quad T = (T + parity \cdot p)/2$
6: **if** $T \geq p$ **then** $T \leftarrow T - p$
7: **return** T

Figure 1.10: Improved Montgomery multiplication (from [49]).

figure 1.10 presents another version of the algorithm in figure 1.9.

To compute a Montgomery multiplication, the multiplications at line 3 and the reductions at line 5 are alternated at each loop iteration. The computation of the *parity* value (line 4) is equivalent to the computation of the variable M of the previous algorithm in figure 1.9.

The algorithm in figure 1.11 taken from [119] was chosen and implemented for the product of two integers modulo a prime number. It follows the Montgomery multiplication method, but instead of using the data a and p directly, we cut them into words. The algorithm scans the multiplicand b word by word, and the multiplier a digit by digit:
$b = (b^{(t-1)} \cdots b^{(1)} b^{(0)})$,
$a = (a_{n-1} \cdots a_1 a_0)$,
$p = (p^{(t-1)} \cdots p^{(1)} p^{(0)})$,
with $t = \lceil n + 1/\omega \rceil$. The number of bits per word is represented by ω, and the number of words by t. Thus $b_i^{(j)}$ is the ith bit of the jth word, $b_{i..j}$ represents the b vector bits of position i to j, and $(x|y)$ is the concatenation of two bits sequences.

The computed value *carry* is on two bits, and the result of the modular multiplication T is on $n+1$ bits. Line 7 performs the left shift (division by two). The algorithm in figure 1.10 performs this operation at line 5. In our implementation, lines 6 and 7 are performed simultaneously:

$$\left(carry, \left(T_{\omega-2}^{(j)} \cdots T_0^{(j)} T_{\omega-1}^{(j-1)}\right)\right) \leftarrow a_i b^{(j)} + carry + T^{(j)} + parity \cdot p^{(j)}.$$

input: p a prime number, and $(a, b) \in \mathbb{F}_p^2$ on n bits
output: $\mathrm{MM}(a, b) = TR^{-1} \bmod p$ with $T = ab \bmod p$

1: $T \leftarrow 0$
2: **for** i **from** 0 **to** $n-1$ **do**
3: $\quad parity = T_0^{(0)} + a_i b_0^{(0)}$
4: $\quad (carry, T^{(0)}) \leftarrow a_i b^{(0)} + T^{(0)} + parity \cdot p^{(0)}$
5: $\quad$ **for** j **from** 0 **to** $t-1$ **do**
6: $\quad\quad (carry, T^{(j)}) \leftarrow a_i b^{(j)} + carry + T^{(j)} + parity \cdot p^{(j)}$
7: $\quad\quad T^{(j-1)} \leftarrow (T_0^{(j)} | T_{\omega-1..1}^{(j-1)})$
8: $\quad T^{(t-1)} \leftarrow (carry | T_{\omega-1..1}^{(t-1)})$
9: **if** $T \geq p$ **then** $T \leftarrow T - p$
10: **return** T

Figure 1.11: Montgomery multiplication (from [119]).

This algorithm is suitable for hardware implementation because it requires only shifts and additions, and is parallelizable [119]. This parallelization is described in figure 1.12: process C represents lines 3 and 4, while process D lines 6 and 7. Each column describes each value of i which can be a parallelizable process. Except the first column, each column begins 2 cycles after the previous column and takes as input the result of the previous cycle. Indeed, it is necessary for example that the second process C takes as input the value of $T^{(0)}$ corresponding to the line 4 of the algorithm in figure 1.11.

1.3.5 Modular Inversion

The modular inversion $a^{-1} \bmod p$ corresponds to finding the integer b such as $(a \times b \equiv 1 \bmod p)$. Fermat's little theorem states that if p is a prime number, then for any integer a, the number $a^p - a$ is an integer multiple of p. That is

$$a^{p-1} \equiv 1 \bmod p \quad \Longleftrightarrow \quad a^{-1} \equiv a^{p-2} \bmod p.$$

The modular inverse corresponds to a modular exponentiation. Another method uses the greatest common divisor (gcd) [118]. This method consists by computing $(a^{-1} \equiv X \bmod p)$ such as

$$aX + pY = 1.$$

An efficient algorithm, called binary Euclidean [70, Sec 4.5.2], uses the gcd method in radix 2 to compute modular inversion. Montgomery modular inversion method [72] is similar to the gcd method.

The *Montgomery inversion* (denoted MI) of an integer a is

$$\mathrm{MI}(a) = a^{-1} 2^n \bmod p,$$

with $a < p$, where p is a prime number, and $n = \lfloor \log_2 p \rfloor + 1$. The inversion algorithm consists of two phases. The output of phase 1 (algorithm in figure 1.13) is the integer r such that $r = a^{-1} 2^k \bmod p$, where $n \leq k \leq 2n$. This result is then corrected using phase 2 (algorithm in figure 1.14) to obtain the Montgomery inverse $x = a^{-1} 2^n \bmod p$.

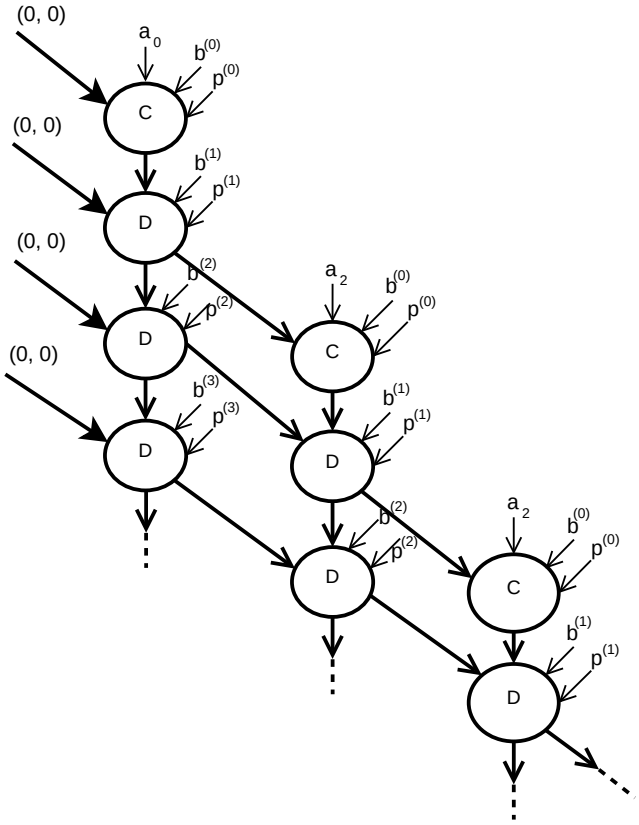


Figure 1.12: Diagram of Montgomery multiplication algorithm dependency (from [119]).

The interest for Montgomery inversion is that it computes an inverse element with the product of the quantity R. Thus, when computing an inversion followed by a multiplication, one does not need to perform a correction step for having the expected result:

$$a^{-1}b \bmod p = \mathrm{MM}(\mathrm{MI}(a), b) = \mathrm{MI}(a)b \times R^{-1} = a^{-1}R \times b \times R^{-1} = a \times b.$$

One has the following properties with $0 < a < p$:
$a^{-1} = \mathrm{MM}(\mathrm{MI}(a), 1) = \mathrm{MM}(a^{-1}R, 1) = a^{-1} \cdot R \cdot R^{-1}$,
$a^{-1}b^{-1} = \mathrm{MI}(\mathrm{MM}([a], [b]) = \mathrm{MI}([ab]) = \mathrm{MI}(abR) = (abR)^{-1} \cdot R = (ab)^{-1}$.

1.4 Side-Channel Attacks and Countermeasures

The use of embedded modules for cryptographic applications introduced a new cryptanalytic approach. Indeed, cryptographic protocols and algorithms were considered to be secure from a theoretical point of view if parameters were well chosen. However, the advent of embedded systems has created new links between cryptography, electronic and physical security. Passing from a high-level mathematical description to a netlist of gates and then to a silicon chip introduces hidden weaknesses and information leakages.

Side-channel attacks (SCAs) are attacks which try to break cryptosystems by exploiting information leakages from hardware implementations. A large number of attacks, such as *Simple Power Analysis* (SPA) and *Differential Power Analysis* (DPA) have been reported on a wide variety of cryptographic implementations [107], [80], [35], [84], [30]. SCAs exploit some correlations between secret values manipulated in the device and physical parameters measured from

input: p a prime number, and $a \in \mathbb{F}_p$
output: r and k such as $r = a^{-1}2^k \bmod p$ and $n \leq k \leq 2n$

```
1:  u ← p, v ← a, r ← 0, s ← 1
2:  k ← 0
3:  while v > 0 do
4:      if u is even then u ← u/2, s ← 2s
5:      elsif v is even then v ← v/2, r ← 2r
6:      elsif u > v then u ← (u − v)/2, r ← r + s, s ← 2s
7:      else v ← (v − u)/2, s ← r + s, r ← 2r
8:      k ← k + 1
9:  if r ≥ p then r ← r − p
10: return r
```

Figure 1.13: Montgomery inversion – Phase 1 (from [72]).

input: p a prime number, k such as $n \leq k \leq 2n$, and $r \in \mathbb{F}_p$ from phase 1
output: r such as $r = \mathrm{MI}(a) = a^{-1}2^n \bmod p$

```
1: for j from 1 to k − n do
2:     if r is even then r ← r/2
3:     else r ← (r + p)/2
4: return r
```

Figure 1.14: Montgomery inversion – Phase 2 (from [72]).

the device such as power consumption, electromagnetic emanations or computation timing. One can refer to [80] for a complete introduction on power analysis based SCAs.

In this section, we focus on two kinds of SCAs. Simple and differential side-channel analysis rely on the following physical property: a circuit is physically made of thousands of logical gates switching differently depending on the executed operations and on the manipulated data. The cryptographic device is attacked by observing physical properties of the device. Therefore power consumption and electromagnetic emanation and may leak information on both operations and data. By monitoring a device performing cryptographic operations, an observer may infer information on the implementation of the program executed and on the secret data involved.

In ECC, scalar multiplication $[k]$P is the key operation. The scalar k is considered as the secret key. This section explains how these attacks can be used to recover the secret key k. An attacker can measure a trace of $[k]P$ computation. A trace is a succession of values over time. For instance a power trace is a graph of consumed power over time. Such a trace is measured from a cryptographic device and recorded using a digital oscilloscope. Oscilloscopes can be controlled remotely by a computer via a general-purpose interface bus or an Ethernet interface. Based on this interface, the recorded traces can also be transferred to a PC and then, they can be thoroughly analysed.

1.4.1 Simple Side-Channel Analysis

Simple side-channel analysis was introduced by Kocher *et al.* in [73]. Secret key information is extracted from one or very few traces due to leakage from the execution of key dependent code.

Power consumption or electromagnetic radiation of a cryptosystem is different following on the performed operation and handled operands. For instance, a modular multiplication requires more clock cycles than an addition, and this difference may be visible on measured trace of side channel. In particular, costs of curve-level operations in table 1.3 indicate that a point addition needs more field-level operations than point doubling.

Some implementations of the scalar multiplication are particularly vulnerable to SCAs since formulas for point additions and doublings have different behaviour. The execution path for scalar multiplication algorithms presented in section 1.1.7 is determined by bits of the secret key. Consequently, under these conditions, cryptosystems generate traces of side channels that can be easily distinguishable. In the "double-and-add" algorithm in figure 1.15, the duration (the size on the trace) of each operation can indicate if it is a point doubling `DBL` (short pattern) or a point addition `ADD` (longer pattern).

input: $k = (k_{n-1}k_{n-2}\dots k_1k_0)_2, \quad P \in E(\mathbb{F}_p)$
output: $Q = [k]P$

1: $Q \longleftarrow \mathcal{O}$
2: **for** i **from** $n-1$ **downto** 0 **do**
3: $\quad Q \longleftarrow [2]Q$ (`DBL`)
4: $\quad$ **if** $k_i = 1$ **then** $Q \longleftarrow Q + P$ (`ADD`)

Figure 1.15: Left to right binary "double-and-add" algorithms to compute $[k]P$.

Recovering the secret bits from the sequence of operations is straightforward: considering the left-to-right implementation, a `DBL` followed by an `ADD` corresponds to the digit key 1, and a `DBL` followed by another `DBL` corresponds to the digit key 0. For instance, figure 1.16 presents an in-house power consumption leakage based on a scalar multiplication implementation with the double-and-add algorithm in figure 1.15. Figure 1.16 was measured at University College Cork (UCC, Ireland) during a stay of three months. An attacker can distinguish a power trace generated by point additions and point doublings. Thus, one can guess some key bits because patterns `DBL` + `ADD` (resp. one `DBL`) correspond to the manipulated key bit 1 (resp. 0).

Two main countermeasures preventing simple SCAs are about regular and atomic algorithms. Regular algorithms always perform the same sequence of operations. To achieve this property, the maximal number of operations that can be required per scalar bit (one `DBL` and one `ADD`) is always performed. *Double-and-add-always* and *Montgomery ladder* [90] [66] [65] belong to this category. The second countermeasure is about atomic algorithms. Computed operations during the scalar multiplication algorithm can be done as a succession of identical atomic patterns. The measured side channel would be a succession of similar patterns. Several loop iterations may be necessary to the processing of a scalar bit. For instance, an atomic method in [28] performs one loop iteration for a bit 0 and two loop iterations for a bit 1. Each iteration is composed of several operations at the field level.

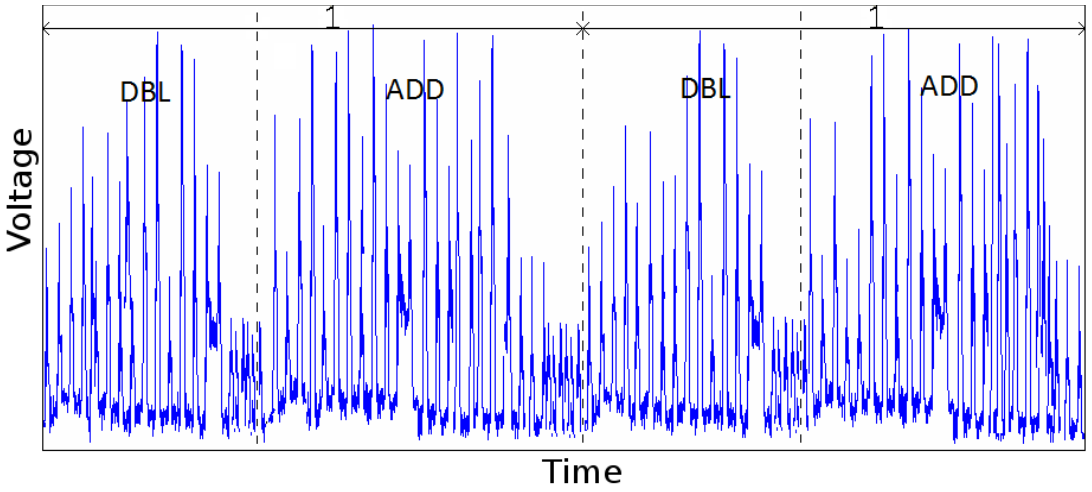


Figure 1.16: Scalar multiplication power consumption leakage trace.

Simple SCAs can be difficult to perform. Indeed, traces are often influenced by the cryptographic device and experimental noise.

1.4.2 Differential Side-Channel Analysis

A more sophisticated attack, called differential side-channel analysis, can be performed. In contrast to simple SCAs, this type of attack requires a large amount of measurements from a side channel to determine the key. These measurements are processed using statistical tools. Simple SCAs exploit the relationship between the performed operations and the side channel, whereas differential SCAs exploit the relationship between the processed data and the measured side channel. Traces are measured from side channels where the same device is used to operate on different data. The large number of traces is also used to reduce noise by averaging.

This attack was initially proposed in [73]. Differential SCAs use statistics to reveal behavioural differences which are difficult to find with simple SCAs. Differential SCAs basically consist in acquiring many leakage traces of a large number of executions in which a constant secret key is manipulated. These traces are stored along with the known input value. Then, a key hypothesis is made on some bits of the secret key. A statistical process is performed according to the key hypothesis and the known input on one hand, and the collected traces on the other hand. Reproducing this process for all possible considered pieces of the key reveals the correct one, i.e. the value of the targeted key bits. This process is eventually iterated on the remaining unknown key bits until all of them are recovered, or until the last ones can be found by exhaustive search.

Classical countermeasures used for protecting scalar multiplication implementations against differential SCAs consist in blinding (i.e. randomizing) the internal variables of the computation using arithmetic properties at different levels: group arithmetic, point representation, and modular arithmetic. All countermeasures lead to randomization on the curves [67], on the point P [79] or on the scalar k [35].

For instance, Coron in [35] computes the scalar multiplication $[k']P$ instead of $[k]P$ where $k' = k + r\#E(\mathbb{F}_p)$ where $\#E(\mathbb{F}_p)$ is the order of $E(\mathbb{F}_p)$ and r a random number. Another method consists in applying a multiplicative blinding to the point coordinates using the projective representation redundancy [4].

In addition, randomization can be lead at the field level [48] instead of performing blinding

countermeasures at the curve level.

DPA countermeasures are provided in most ECC based protocols, or ECC cryptographic schemes, which are based on randomization. In practice, very few ECC protocols, such as elliptic curve digital signature algorithm (ECDSA), do not incorporate any DPA countermeasures.

A SCA will be performed in this Ph.D. thesis on a scalar multiplication, operation which dominates the computation time in elliptic curve cryptography protocols. The concerned attack is a particularly interesting class of a probabilistic SCA, called *template attacks* which was introduced in [27]. In a template attack, the attacker is assumed to know characteristics of a side channel of some processed data of a device. This characterization is stored and called template. The attacker matches the templates based on different key hypotheses with the recorded traces.

Chapter 2

Hardware Implementations of Scalar Random Recoding Countermeasures

In a conventional non-redundant number system, all numbers are represented in a unique way (canonical representation). A redundant number system allows multiple representations of some numbers, thus the name redundant. We propose hardware implementations for on-the-fly random recodings of the scalar k using two methods, the double-base number system (DBNS) and the signed-digit (SD) representation. The hardware implementation of DBNS recoding was published in [25].

The very high redundancy level of these representations allows us to randomly choose among several representations of the scalar digits k_i. Redundant representations, and in particular theoretical solutions, are a common knowledge. In this chapter, we deal with hardware implementations which provide a small overhead. Then the number and order of operations at curve level (point additions, doublings and triplings) are randomized. This may be a protection against some side-channel attacks.

This kind of protection requires a random number. In this way, another representation can be randomly chosen from the output of a random number generator (RNG) block. In practice, random bits are produced by a RNG. Ideally, the randomness is produced by a true random generator (TRNG). However, a pseudo random number generator (PNRG) may be sufficient for our applications with the use of seeds generated by a TRNG.

2.1 Random Number Generator (RNG)

Randomization implies the use of RNG. A RNG can be described as being true random number generator (TRNG) or as being pseudo random number generator (PRNG). TRNGs use a physical noise source such as radioactive decay, meta-stability, thermal noise or jitter variations in free running oscillators, to produce a random signal. One can use hybrid RNGs: a PRNG, characterized by high speeds, produces random numbers, and a TRNG produces the random seed.

In this Ph.D. thesis, we do not implement a RNG. However, a RNG was developed in the team (CAIRN team), with on-line randomness monitoring to provide the random bits required. This kind of RNG is based on rings oscillators sampling for the physical noise source (random jitter produced by one or several free running oscillators). See [111] and [110] for details. In practice, a RNG is used in many cryptographic protocols.

2.2 Double-Based Number System Random Recoding

A double-base expansion of k is of the form

$$k = \sum_{i=0}^{n'-1} s_i 2^{u_i} 3^{v_i},$$

with $s_i = \pm 1$. In this chapter, we only consider DBNS chains where exponents are non-increasing: $u_0 \geq \cdots \geq u_{n'-1}$ and $v_0 \geq \cdots \geq v_{n'-1}$ (see section 1.2.1). When n is the length of k in binary, n' corresponds to the number of terms. In general, $n' \ll n$. That is why DBNS is a sparse representation.

2.2.1 Proposed Arithmetic Countermeasure

In this work, we use random representations of k in DBNS. This is possible because the DBNS is extremely redundant: for instance, 10, 100 and 1 000 have 5, 402 and 1 295 579 different DBNS representations, respectively [41]. The idea is to use this natural redundancy to produce on-the-fly another representation of k in DBNS. From now on, we consider to already have k in DBNS chain (conversion from the standard binary representation to DBNS is studied in chapter 4). Random recodings are applied using the following identities [44]:

$$\begin{aligned}
\mathrm{I}_1 &: \quad 1 + 2 = 3, \\
\mathrm{I}_2 &: \quad 1 + 3 = 2^2, \\
\mathrm{I}_3 &: \quad 1 + 2^3 = 3^2, \\
\mathrm{I}_4 &: \quad 1 + 1 = 2.
\end{aligned}$$

Those identities can be applied in two different ways: one can reduce (e.g. $1 + 2 \rightarrow 3$) or expand (e.g. $3 \rightarrow 1 + 2$) DBNS terms. Reductions may accelerate the $[k]P$ computation time due to the fact that it reduces the number of curve-level operations. Thus there are two recoding rules for each presented identity:

$$\mathrm{I}_1 \Rightarrow \begin{cases} 2^{i+1}3^{j-1} + 2^i 3^{j-1} = 2^i 3^j & \mathrm{R}_1, \\ 2^{i-1}3^{j+1} - 2^{i-1}3^j = 2^i 3^j & \mathrm{R}_2, \end{cases}$$

$$\mathrm{I}_2 \Rightarrow \begin{cases} 2^{i-2}3^{j+1} + 2^{i-2}3^j = 2^i 3^j & \mathrm{R}_3, \\ 2^{i+2}3^{j-1} - 2^i 3^{j-1} = 2^i 3^j & \mathrm{R}_4, \end{cases}$$

$$\mathrm{I}_3 \Rightarrow \begin{cases} 2^{i+3}3^{j-2} + 2^i 3^{j-2} = 2^i 3^j & \mathrm{R}_5, \\ 2^{i-3}3^{j+2} - 2^{i-3}3^j = 2^i 3^j & \mathrm{R}_6, \end{cases}$$

$$\mathrm{I}_4 \Rightarrow \begin{cases} 2^{i-1}3^j + 2^{i-1}3^j = 2^i 3^j & \mathrm{R}_7, \\ 2^{i+1}3^j - 2^i 3^j = 2^i 3^j & \mathrm{R}_8. \end{cases}$$

When several rules can be applied, one can randomly recode the scalar k amongst possible rules. Thus, it may introduce randomness in the power traces. In addition, expansions slow down the scalar multiplication by increasing the number of terms. There should be a trade-off

between the computation time and the amount of randomness introduced by expansions. Figure 2.1 illustrates some examples of DBNS recodings for the scalar value $k = 140\,400$. In this figure, we can see that the DBNS representation of k change according to what rule is applied. In addition, we can see the $[k]P$ computation time at the curve level for each representation. In this figure, `mADD` and `mSUB` stand for mixed point addition and subtraction.

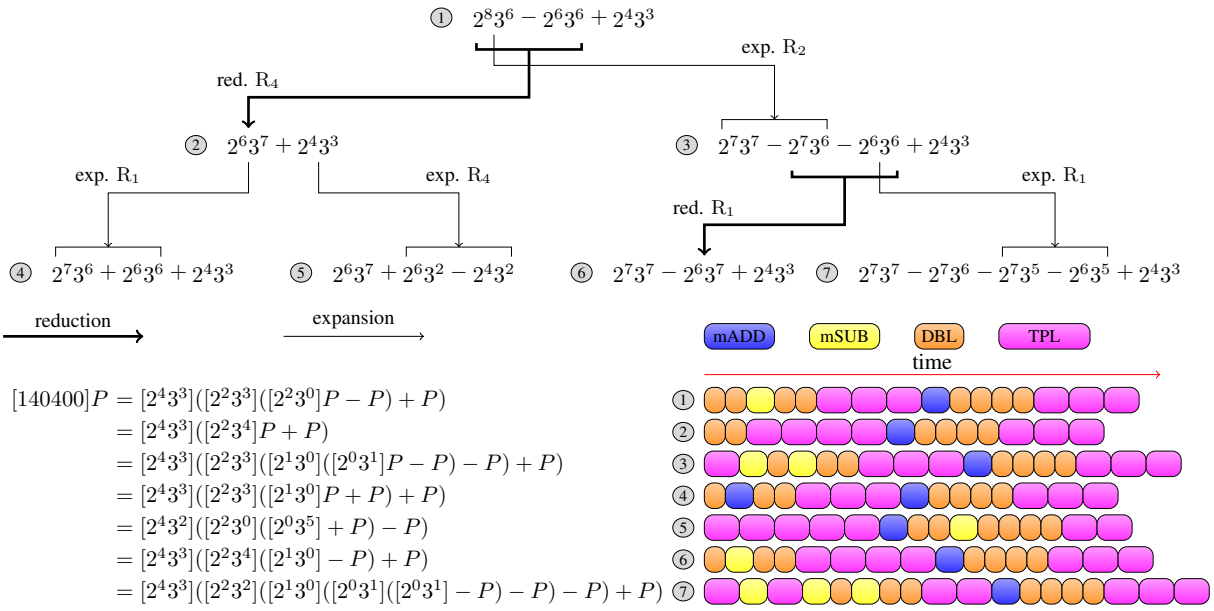


Figure 2.1: Examples of some possible DBNS recodings for $k = 140\,400$.

In some cases, some rules cannot be applied due to neighbouring terms. For instance, if two consecutive terms $k_i = (s_i, u_i, v_i)$ and $k_{i-1} = (s_{i-1}, u_{i-1}, v_{i-1})$ share the same exponent (e.g. $u_i = u_{i-1}$) then reductions based on rule R_1 cannot be applied. Even for expansions, some limits exist due to the fact that the sequence of exponents must be kept non-increasing: $u_0 \geq \cdots \geq u_{n'-1}$ and $v_0 \geq \cdots \geq v_{n'-1}$ (DBNS chain). To always ensure that recodings only produce non-increasing sequences of exponents, three consecutive terms must be read for each rule. The second read term is recoding according to the previous and the next one. With this method one can ensure to always have a DBNS chain. Indeed, only rules implying non-increasing exponent are applied. This has been validated by numerous tests, but a future work could consist by providing a proof.

More recoding identities and rules can be applied like $2^2 + 2 = 3^2 - 3$ for instance. This type of recoding does not change the number of terms. Indeed, this identity transforms two terms into other two terms.

Below we provide a complete example of a situation where some rules should be discarded. We use the k value presented in figure 2.1. For instance, when starting with $k = 140\,400 = 2^7 3^7 - 2^7 3^6 - 2^7 3^5 - 2^6 3^5 + 2^4 3^3$ (which is the recoding 7 in figure 2.1), one may think that the reduction $2^7 3^6 + 2^7 3^5 \xrightarrow{R_3} 2^9 3^5$ can be applied. However, this reduction would produce a non-non-increasing sequence of exponents for this specific value of k. Indeed the new recoded DBNS representation would be $2^7 3^7 - 2^9 3^5 - 2^6 3^5 + 2^4 3^3$ which is not a non-increasing exponents sequence of terms. The previous DBNS expansion gives the following computation sequence $2^4 3^3 \Big(2^2 3^2 \big(2^1 3^0 (-2^2 3^0 + 2^0 3^2) - 1\big) + 1\Big)$. Thus, the DBNS scalar multiplication in the algorithm

in figure 1.7 section 1.2.1 cannot be applied with this DBNS expansion.

A same DBNS recoding does not have only one antecedent. Let LT be the list of terms T which stores the DBNS recoding of k. Each term T is in the shape of $T_i = (s_i, u_i, v_i)$. Randomizing a pattern LT $= (T_{n'-1} \cdots T_0)$ allows to have several DBNS chains $(T'_{n'-1} \cdots T'_0)$ such as $(T_{n'-1} \cdots T_0) \neq (T'_{n'-1} \cdots T'_0)$. The strength of this random recoding is that even with only one random recoding of a DBNS chain, the operations for $[k]P$ computation can be totally different. It is sufficient to have several differences in exponents of one term to have a completely different computation for $[k]P$. For instance, recodings 1, 4, 5 and 6 in figure 2.1 seem to be very similar and have a same number of terms, but the operations during the scalar multiplication are totally different. This is due to the fact that the scalar multiplication algorithm computes a DBNS chain with k in the Horner scheme.

In practice, several random recodings are performed. We statistically note that we do not provide a final recoding which was ever produced with such a strategy. As an experiment, we randomly recoded one thousand times a scalar k, and we repeated it for one thousand different scalars where k was 160 and 224-bit long. This ensures that none of the recodings of k was produced by another recoding. However a more important experimental and a theoretical study have still to be carried out.

2.2.2 Experiment Results and Implementation

The experiments reported below have been realized using the curve P-224 provided by NIST (FIPS 186-2), see [57, appendix A.2.1] for details. The prime field $\mathbb{F}_p$ is defined with $p = 2^{224} - 2^{96} + 1$ ($n = 224$ is the size of the prime field). The experimental setup was as follows. For each scalar k, the binary representation of k is converted into DBNS using the algorithm provided by Dr. Christophe Doche, with the condition $s_i \in \{-1, 1\}$. Then 10 000 random recodings among all applicable rules, from R_1 to R_8, are applied to recode k. For each DBNS recoding, the number of curve-level operations (`ADD`, `DBL`, `TPL`) is counted. The number of expansion and reduction rules which can be applied is taken into account.

Table 2.1 presents the obtained statistics for 1 000 scalars k and 10 000 random DBNS recodings for each scalar. The average value (avg.) and the standard deviation (std. dev.) of each rule number which can be applied are reported for reductions and for expansions. In addition, we give the average in percent of each rule according to the two possible rules (reduction and expansion).

rules		R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
reductions	avg.	594.4	750.3	544.1	450.0	157.5	166.8	975.9	1 010.9
	std. dev.	48.5	81.0	62.8	46.9	34.9	38.3	67.8	86.6
	avg. [%]	12.8	16.1	11.7	9.7	3.4	3.6	21.0	21.7
expansions	avg.	561.8	743.0	537.4	437.8	150.6	163.2	1 039.2	1 031.8
	std. dev.	45.4	82.3	62.8	47.4	34.3	37.3	71.8	88.6
	avg. [%]	12.0	15.9	11.5	9.4	3.2	3.5	22.3	22.1

Table 2.1: Experiment results of DBNS random recodings for the number of times each rule can be applied.

Rules R_7 and R_8 are the most often used rules both for reductions and expansions. Note that

given three successive terms, in 6.7 % of cases no rule can be applied. The average length of the obtained DBNS chain n' is 62 with a standard deviation equals to 4.1. Rules R_5 and R_6 are the less used. The probability that an expansion can be computed by these two rules is smaller than the other rules because the exponents used by these rules are greater for rules R_5 and R_6. That is why adding more rules can be inefficient, because the added rules would often not be used. In addition, the hardware implementation could be more complex and slower.

The maximal exponent of powers of 2 and 3 are 112 and 71, respectively. Then the average cost for computing $[k]P$ using the random DBNS recoding is

$$62\,\texttt{ADD} + 112\,\texttt{DBL} + 71\,\texttt{TPL}.$$

The average cost of $[k]P$ has been computed using the PARI/GP (`http://pari.math.u-bordeaux.fr/`) program kindly provided by Dr. Christophe Doche. This program generates a signed DBNS chain using pre-computations and where approximations are obtained by a search table. When no random recoding is performed, the average cost for computing $[k]P$ is

$$50\,\texttt{ADD} + 112\,\texttt{DBL} + 71\,\texttt{TPL},$$

that is 12 more point additions when one recodes randomly a signed DBNS chain with $n = 224$. It corresponds to 133 multiplications over a prime field when using mixed addition ($\mathcal{J} + \mathcal{A} \to \mathcal{J}$) of which the cost is $7\texttt{M} + 4\texttt{S}$.

Note that the first term (s_0, u_0, v_0) is never transformed for efficiency purposes. Indeed, our random recoding unit does not impact the maximum exponents of the bases. The number of point doublings and triplings does not change. Otherwise, the number of `DBL` and `TPL` would become larger, and thus the $[k]P$ computation time can be changed and increased.

	# `ADD` with			additional cost in `M`
n	[46]	random recoding	difference	of the random recoding
160	35	46	11	112.2
192	43	56	13	132.6
224	50	62	12	122.4
256	57	69	12	122.4
384	84	95	16	163.2
521	114	134	20	204.0

Table 2.2: Costs in `M` using random DBNS recoding.

Table 2.2 presents the number of point additions for 1 000 scalars. For each scalar, 10 000 random recodings have been performed. We use the standard cost approximation $\texttt{S} \approx 0.8\,\texttt{M}$ to calculate the additional cost in multiplication of the random recodings. The number of point additions determines the length of a DBNS chain. The difference in the number of point additions for the two methods is evaluated. For instance, the initial DBNS chain has in average 35 terms for $n = 160$, while the DBNS chain has in average 46 terms after 10 000 random recodings. It corresponds to 11 additional terms, which is 112.2 multiplications over $\mathbb{F}_p$ (`M`) for an initial scalar k in DBNS. In average, the additional cost represents about 7% more multiplications (at the field level) in the scalar multiplications with random recodings.

Table 2.2 shows the difference according to the multiplication number for the random recoding method with the PARI/GP program provided by [46]. This difference is similar for all

prime fields, except for $n = 521$. For $n = 521$, the number of terms is much higher than for the other value of n. Indeed, the DBNS chain for $n = 521$ is sparser. Thus, reduction rules are not performed as many the other prime fields. Therefore, the DBNS chain increases with expansion rules until the recoded chain is so big that reduction rules can be practically applied on each recoding.

In practice, the length n' of a random recoding DBNS chain increases up to about 80 recodings. When one considers more than 80 recodings for one initial DBNS chain, every length of DBNS chain recoding is similar with a change of ± 1. Figure 2.2 presents the length variation of one initial DBNS chain, from 1 to 100 recodings on the prime field with $n = 224$. Reduction rules are often performed from 80 random recodings. Experimentally, we note that the length or the number of point additions practically does not change when one randomly recodes 1 000 times a DBNS chain. When one considers a large number of DBNS chain and perform random recodings, the length evolution follows the figure 2.2 without peaks.

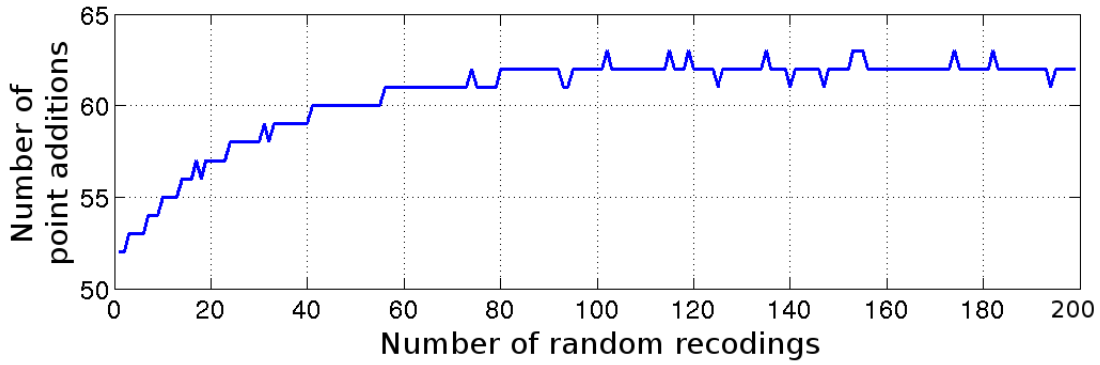


Figure 2.2: Length evolution of one DBNS chain randomly recoded on P-224.

Figure 2.3 presents the architecture of the implemented recoding unit. This unit randomly recodes the scalar k on-the-fly. The scalar k, represented in a DBNS chain, is stored in a specific register as a sequence of terms (s_i, a_i, b_i). Two specific blocks check which rules, from R_1 to R_8, can be applied. One block is dedicated to reductions and the other to expansions. The both blocks take three consecutive terms as inputs.

In case of a reduction, one of the original two terms is modified accordingly to the selected rule (addition/subtraction on the exponents and sign adjustment), while the other term is deleted from the DBNS chain.

In case of an expansion, the block checks whether inserting a new term still leads to a non-increasing sequence of exponents. Using 3 random bits produced by a RNG, one rule is randomly selected among the set of allowed rules and then applied. For all applicable rules, one is selected; the block checks if application of rules will produce a valid sequence of exponents (non-increasing exponents).

When a reduction is possible, the controller selects the outputs of the reduction block, else those of the expansion block. This allows to randomly recode a DBNS chain by reducing as much as possible the length of the DBNS chain.

A global controller (CTRL) generates all high-level control signals for the architecture units (these signals and clock are not represented on the figure). It also determines the times number the scalar must be randomly recoded. Then, it provides the global control with informations on

which curve-level operations must be launched.

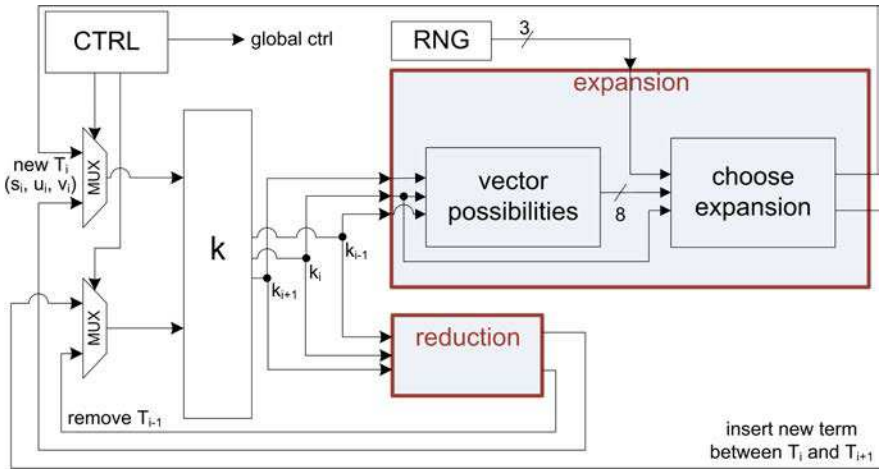


Figure 2.3: Architecture of the on-the-fly recoding unit with random DBNS chains.

2.2.3 FPGA Implementation

All hardware implementations reported in this section have been described in VHDL and implemented on a XC5VLX50T FPGA using ISE 12.4 from Xilinx with standard efforts for synthesis, place and route. We report numbers of clock cycles, best clock frequencies and numbers of occupied slices. We also report numbers of look-up tables (LUTs with 6 inputs in Virtex 5) and flip-flops (FFs) for area. A XC5VLX50T contains 7 200 slices with 4 LUTs and 4 flip-flops per slice. We use flip-flops for all storage elements.

FPGA implementation results are reported in table 2.3 for the proposed recoding. The area required by the recoding unit represents less than 7% compared to the complete ECC processor studied in appendix A. The recoding unit can work at a clock frequency greater than 250 MHz which is faster than the clock frequency of our arithmetic units in $\mathbb{F}_p$.

block	area slices (FF/LUT)	freq. MHz
expansion	102 (93/352)	325
reduction	23 (26/101)	289
complete DBNS recoding	147 (186/431)	289

Table 2.3: FPGA Implementation results of the DBNS recoding unit.

2.2.4 ASIC Implementation

All ASIC results reported in this thesis have been synthesized into gate-level netlists using standard V_{th} (SVT) cells of an industrial 130nm bulk CMOS technology library using Synopsys Design Compiler G-2012.06-SP5. The standard cells used were restricted to a set {nand2, nor2, xor2, inv} of logic gates without loss of generality.

The DBNS recoding implementation is applied with a maximum path delay constraint of 10ns and 5ns from all inputs to all outputs. Tables 2.4 and 2.5 report area in μm^2 and power estimation in μW. The area required by the recoding unit is very small compared to the total area of the complete ECC processor (see appendix A).

delay	combinational	buf/inv	non combinational	total
10	3 425.2	413.5	1 849.7	5 274.9
5	3 427.2	413.5	1 849.7	5 276.9

Table 2.4: Area results (in μm^2) of ASIC implementation for DBNS recoding.

delay	cell internal	net switching	total dynamic	cell leakage
10	92.6	17.2	109.8	1.6
5	185.2	34.4	219.6	1.7

Table 2.5: Power results in (μW) of ASIC implementation for DBNS recoding.

An on-the-fly recoding unit in DBNS has been implemented in FPGA. In addition, the cost of the DBNS random recoding unit is very small compared to the total cost of a complete ECC processor both in clock frequency and silicon area aspects. This recoding unit provides a countermeasure which changes on-the-fly a DBNS signed representation of an integer. A future work could be to evaluate the security of such a recoding.

In this section, we provide random recodings from a DBNS chain. It is our initial condition. We will see in chapter 4 the conversion from an integer into DBNS. The recoding of the scalar can be used as a countermeasure against some side channel attacks, by referring to the second countermeasure of Coron (see [35] for details). In the following, we have studied another redundant system which deals with signed-digit representation.

2.3 Signed-Digit Representations

Carry-free addition property lies in the redundancy of signed-digit number systems. However, we just use this system for its redundancy to change the representation of numbers on-the-fly. We first present relevant questions concerning random binary signed-digit representations and how to reduce the numbers of 0s of a number binary representation in order to speed up the scalar multiplication on elliptic curves.

2.3.1 Avizienis System

Avizienis introduced in [5] a redundant notation for some number representations. Initially, these redundant representations were proposed for computing additions without carry propagation. This redundancy obviously leads to the representation of an integer in different ways, and needs more bits. For instance, if one chooses to represent numbers in radix 10 with digits in $\{\bar{5}, \bar{4}, \bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3, 4\}$, one can write the number $(9\,999)_{10} = (10\,00\bar{1})_{10} = 1.10^4 + (-1).10^0$. In the following, we will see which parameters we can consider to have a redundant system.

Let $S_{\mathcal{B},\delta,\alpha,n}$ be the set of numbers in base $\mathcal{B}$ with n digits in $\{\delta, \delta+1, \cdots, \delta+\alpha\}$ where δ is a positive or negative integer, and α a strictly positive integer. Such a system can be redundant (see [92, Sec. 2.6] for details). The next theorem explains how to determine all these parameters to have a redundant system.

In the theorem [92, p. 35], we know that if $\alpha > \mathcal{B}-1$, $k \in \left[\frac{\delta(\mathcal{B}^n-1)}{\mathcal{B}-1}, \frac{(\delta+\alpha)\cdot(\mathcal{B}^n-1)}{\mathcal{B}-1}\right] \forall k \in G_{\mathcal{B},\delta,\alpha,n}$ with $G_{\mathcal{B},\delta,\alpha,n}$ be the set of numbers that one can write with the system $S_{\mathcal{B},\delta,\alpha,n}$. The writing of an integer in the system $S_{\mathcal{B},\delta,\alpha,n}$ is not unique. It is a redundant system.

The next step is to determine which parameter values can be taken. In our case, we want $\mathcal{B} = 2$ and $n = \lfloor \log_2 p \rfloor + 1$ because we work in the binary base and all elements are in the finite field $\mathbb{F}_p$. In addition, we want to have a signed-digit representation. Therefore we take $\delta = -1$ and $\alpha = 2$. It means that all digits will be in the set $\{-1, 0, 1\}$. This system with these values allows the representation of negative integers because δ is negative, and is redundant because we have $\alpha = 2 > 1 = \mathcal{B} - 1$. In addition, the system $S_{\mathcal{B},\delta,\alpha,n}$ is symmetrical. Thus the set of digits used is in $\{\delta, \delta+1, \cdots, \delta+\alpha\}$ with $-\delta > 0$ and $\alpha = -2\delta$. With this system, we have

$$k = \sum_{0}^{n} k_i 2^i \text{ where } k_i \in \{-1, 0, 1\},$$

with k a strictly positive integer of length $n+1$ in 2-radix. In the following, we denote $\bar{1}$ for -1. When using this set of digits, one can consider the *binary signed-digit* (BSD) representation. We can note that this notation system is not conceptually different from the borrow-save notation [56]. Signed-digit representations can be used in the context of *canonic signed-digit* (CSD) representation [105], *Booth recoding* [13] and `NAF` representation [57, Sec. 3.3.1].
However, with the aforementioned systems, numbers are recoding in a subset of the considered redundant systems. In addition, there is no redundancy in these subsets. Recodings do not use the redundancy provided by these systems.

The system $S_{2,-1,2,n}$ does not allow us to compare integers represented with the set S [92, Sec. 2.6]. For that, we must have $-2\delta \geq \mathcal{B} + 1$. We do not know this property, and to add two integers represented in S is a difficult issue. In our case, we just want to represent the scalar k in a redundant representation to only use it for ECC scalar multiplication. The scalar is not added or compared with an integer. Curve-level operations are performed according to digits of k. Thus, we can neglect the fact that k cannot be compared when it is represented with the system $S_{2,-1,2,n}$.

Thus, $S_{2,-1,2,n}$ represents numbers in a redundant representation. For instance, for the integer $k = (11)_{10}$, one has:
$k = (01011)_{\text{BSD}} = 2^3 + 2^1 + 2^0$,
$k = (011\bar{1}1)_{\text{BSD}} = 2^3 + 2^2 - 2^1 + 2^0$,
$k = (10\bar{1}0\bar{1})_{\text{BSD}} = 2^4 - 2^2 - 2^0$,
$\vdots$

From [100, Sec. 14.2], we know that this digit set is maximally redundant because we have $\delta + \alpha = \mathcal{B} - 1 = 1$. It refers to a measure of the redundancy of a signed-digit representation. The range of numbers that can be represented by this signed-digit representation is $[2^{-n+1} - 2,\ 2 - 2^{-n+1}]$.

Now, we have a way to represent an integer in a redundant representation. The next question would be to know the total number of BSD for an integer (same thing applies for DBNS, see

section 1.2.1). Indeed, the goal is to randomly represent a given integer by using several times the redundancy of the employed representation. For cryptographic purpose, a number must have a great deal of representations.

2.3.2 Number of Binary Signed-Digit Representations

Let $\lambda(k,n)$ be the number of binary signed-digit representations for an integer $k \in [0, 2^n - 1]$ that is n bits long. It corresponds to the system $S_{2,-1,2,n}$. From [50], one has the following lemmas:

- $\lambda(0,n) = 1$,
- $\lambda(1,n) = n$,
- $\lambda(2^k,n) = n - k$,
- for k even, $\lambda(k,n) = \lambda(\frac{k}{2}, n-1)$,
- for k odd, $\lambda(k,n) = \lambda(\frac{k-1}{2}, n-1) + \lambda(\frac{k+1}{2}, n-1)$,
- for $2^{n-1} \leq k \leq 2^n - 1$, $\lambda(k,n) = \lambda(k - 2^{n-1}, n-1)$.

With these properties, one can compute recursively the number of binary signed-digit representations for an integer.

For instance, one has

$$\begin{aligned} \lambda(11,5) &= 8, \\ \lambda(149,9) &= 50, \\ \lambda(1\,365,12) &= 233, \\ \lambda(87\,381,17) &= 4\,181, \end{aligned}$$

with $n = 1 + \lfloor \log_2(k) \rfloor + 1$. These instances mean that there are 50 signed-digit representations of the number 149 using five digits. If one wants to have more representations, it is sufficient to add more digits to n. The following part deals with the random recoding of a number in signed-digit representation.

When k is in the range 160–500 for typical cryptographic sizes, the number of binary signed-digit representations is greater than in average 2^{100}. This average was produced by applying the λ formula for 10 000 values. Thus, we can consider that k has a very large number of possible representations. One can use such a representation to randomly recode a large integer for the scalar multiplication: it could be used as a countermeasure against some side-channel attacks.

2.3.3 Random Recoding

Considering a positive integer k, different signed-digit representations of this number can be obtained by replacing sequences by other sequences:
$0\bar{1} \Longleftrightarrow \bar{1}1$ and $01 \Longleftrightarrow 1\bar{1}$.

We denote $f(a,b)$ the function which applies to the previous two transformations:

$$f : \{\bar{1}, 0, 1\} \times \{\bar{1}, 0, 1\} \longrightarrow \{\bar{1}, 0, 1\} \times \{\bar{1}, 0, 1\}. \tag{2.1}$$

with
$f(0,\bar{1}) = (\bar{1}, 1)$,
$f(\bar{1}, 1) = (0, \bar{1})$,
$f(0,1) = (1, \bar{1})$,

$f(1,\bar{1}) = (0,1)$.
For other elements $(a,b) \in \{\bar{1},0,1\} \times \{\bar{1},0,1\}$, $f(a,b) = (a,b)$.

Let k be a number in binary representation where $k = (k_{n-1} \cdots k_1 k_0)_2$. One could think that it is sufficient to scan k two digits by two digits; change or not (k_{i+1}, k_i) according to a random bit. With this method, we change (k_{i+1}, k_i) to $f(k_{i+1}, k_i) = (k'_{i+1}, k'_i)$ if the random bit is 1. When the random bit is 0, $f(k_{i+1}, k_i) = (k_{i+1}, k_i)$; the function f is not applied. In the next loop iteration, we scan (k_{i+2}, k_{i+1}) or (k_{i+2}, k'_{i+1}) according to $f(k_{i+1}, k_i)$.

However this strategy is not complete because we do not have all possible representations. Such a strategy is equivalent to the Booth recoding. To have all possible representations, digits must interact with the previous digit and the next one. First of all, three digits by three digits of k are scanned in one direction (most or least significant digit first). According to a random value, the three considered digits are modified: the function f is applied to the two first (resp. the two last) digits; then the function f is applied to the two last (resp. the two first) digits.

If one first changes (k_{i+1}, k_i) to $f(k_{i+1}, k_i) = (k'_{i+1}, k'_i)$, the digits (k'_i, k_{i-1}) can then be changed by the function f. This method enables k_i to interact with all possible values: k_{i+1}, k_{i-1}, k'_{i+1} or k'_{i-1}. All digits of k can interact with the previous digit and the next one, before or after that the function f was applied. It enables to have all possible representations.

As an example, let $k = 11 = (1011)_2$. With the first method, we cannot obtain the last value (recoded digits are in bold font):
$k = 1\mathbf{01}1 \longrightarrow 11\mathbf{\bar{1}1} \longrightarrow 110\bar{1}$.

The algorithm in figure 2.4 presents a right–to–left algorithm which computes a random signed-digit representation of an integer k. The integer k is n bits long, and the output algorithm is k_{BSD}. As input, there is a random vector
$(rd^{(n-1)}, rd^{(n-2)}, \cdots, rd^{(2)}, rd^{(1)})$ where $rd^{(i)}$ is on 3 bits $rd^{(i)} = (rd_2^{(i)}, rd_1^{(i)}, rd_0^{(i)})$. This algorithm provides a random representation of an integer according to a random vector. The result of this algorithm is on $n+1$ bits.

input: a positive integer $k = (k_{n-1} \cdots k_0)_2$,
a random vector $(rd^{(n-1)}, \cdots, rd^{(1)})$ with $rd^{(i)} = \left(rd_2^{(i)} rd_1^{(i)} rd_0^{(i)}\right)_2$
output: $k = (k_n \cdots k_0)_{\mathrm{BSD}}$ where $k_i \in \{-1, 0, 1\}$

1: **for** i **from** 1 **to** $n-1$ **do**
2: **if** $rd_2^{(i)} = 1$ **then**
3: **if** $rd_1^{(i)} = 1$ **then**
4: $(k_{i+1}, k_i) \leftarrow f(k_{i+1}, k_i)$
5: **if** $rd_0^{(i)} = 1$ **then**
6: $(k_i, k_{i-1}) \leftarrow f(k_i, k_{i-1})$
7: **else**
8: **if** $rd_0^{(i)} = 1$ **then**
9: $(k_i, k_{i-1}) \leftarrow f(k_i, k_{i-1})$
10: **if** $rd_1^{(i)} = 1$ **then**
11: $(k_{i+1}, k_i) \leftarrow f(k_{i+1}, k_i)$
12: **return** k

Figure 2.4: Right–to–left random signed-digit representation recoding.

It is possible to have the algorithm in figure 2.4 in a left–to–right version. We present an instance of a random signed-digit recoding in figure 2.5 in a left–to–right version. We change the representation of the number $11 = (01011)_2$ by a random vector (above the arrows). This figure shows that one can change the representation of a number according to a random number. In addition, a number representation can be very different according to another representation. We can see that numbers can be recoded in different patterns.

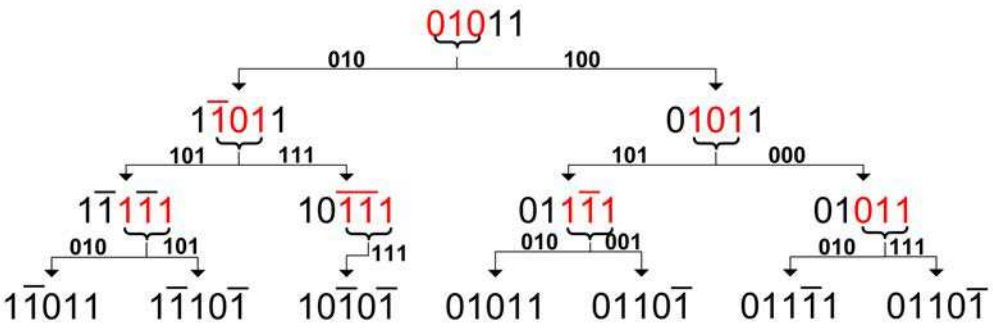


Figure 2.5: Example of recoding for $11 = (01011)_2$.

One must keep in mind that the purpose of this signed-digit representation is for ECC scalar multiplication. In this context, our approach is to randomize the scalar k, and thus the sequence which leads to the computation of $[k]P$. For that, k is represented is binary signed-digits and we randomize it. However this representation may have a too important Hamming weight, decreasing the numbers of 0s. It means that using a randomized signed-digit scalar increases the running time of the scalar multiplication algorithm. The following part is about increasing the numbers of 0s in a signed-digit representation.

2.3.4 Width–w Signed-Digit (wSD)

A width–w signed-digit representation of a strictly positive integer k is an expression

$$k = \sum_{i=0}^{n} 2^i k_i, \text{ where } k_i \in \{0, \pm 1, \pm 3, \ldots, \pm(2^w - 1)\},$$

and no w consecutive digits k_i are non-zero. The length of this representation is $n+1$. In binary representation, k is n bits long.

For instance, with $w = 2$ we have $(k)_{2SD}$ where k is a positive integer and $k_i \in \{\bar{3}, \bar{1}, 0, 1, 3\}$. $w = 1$ corresponds to the binary signed-digit representations (BSD), $(k)_{1SD} = (k)_{\text{BSD}}$.

Let k be a strictly positive integer. One has:
- $(k)_{2SD} = (k)_{SD}$,
- k does not have a unique width–w SD representation, denoted $(k)_{wSD}$ (contrary to wNAF),
- the length $(k)_{wSD}$ is one more digit than the length of the binary representation of k: $n + 1 = \lfloor \log_2 k \rfloor + 2$,
- the average density of non-zero digits among all width–w SDs of length n is $\frac{1}{w+1}$.

The wNAF representation is similar to wSD and has the same properties except that k does not have a unique representation $(k)_{wSD}$, contrary to $(k)_{w\text{NAF}}$. In addition the set of digits k_i in the wSD representation is higher than for wNAF. $k_i \in \{0, \pm 1, \pm 3, \ldots, \pm(2^{w-1} - 1)\}$ in a width-w

`NAF` representation. Thus more points must be pre-computed in wSD than in w`NAF`.

$(k)_{wSD}$ can be efficiently computed using the algorithm in figure 2.6. The scalar k is considered in binary signed-digit representation of length $n+1$. k is scanned w digits by w digits from right-to-left. We compute the decimal value of each word (line 6). This value can be negative. If the value is even, we perform a left shift by one, and we have a new width–w (line 4). If the value is odd, the least significant digit of each word takes the computed value (line 7) and the other digits take the value 0 (line 8).

input: window width w, and $k = (k_n \cdots k_0)_{\text{BSD}}$ where $k_i \in \{-1, 0, 1\}$
output: $(k)_{wSD}$

1: $i \longleftarrow 0$
2: **while** $i < n$ **do**
3: **if** $k_i = 0$ **then**
4: $i \longleftarrow i + 1$
5: **else**
6: $k_i \longleftarrow 2^{w-1}k_{i+w-1} + \cdots + 2^1 k_{i+1} + 2^0 k_i$
7: $(k_{i+w-1} \cdots k_{i+1}) \longleftarrow (0 \cdots 0)$
8: $i \longleftarrow i + w$
9: **return** k

Figure 2.6: Computing the width–w SD of a positive integer in BSD representation.

In the algorithm in figure 2.6, we use a positive integer k calculated by the algorithm in figure 2.4 in a random binary signed-digit representation. However it is not necessary to wait for the result of the new random representation of k in SD to perform $(k)_{wSD}$. The conversion of a positive integer in wSD can be managed on-the-fly. Both algorithms can be computed at the same time. It is sufficient to wait for the computation of w random digits in signed-digit representation, and start the computation in SD for these w digits. The algorithm in figure 2.7 computes a random $(k)_{wSD}$ using the two previous methods.

As an example, let $k = (1\,365)_{10} = (10101010101)_2$. Some width–$w$ SDs representations of k for $w = 2$ and $w = 3$ are:

$$(k)_{2SD} \begin{cases} (1\bar{1}01011\bar{1}10\bar{1}\bar{1})_{BSD} &= (01010030\bar{1}00\bar{3})_{2SD} \\ (10\bar{1}\bar{1}01011\bar{1}01)_{BSD} &= (100\bar{3}01010101)_{2SD} \\ (0101101\bar{1}0\bar{1}\bar{1}1\bar{1})_{BSD} &= (0100301\bar{1}00\bar{3}01)_{2SD} \\ (011\bar{1}10\bar{1}\bar{1}0101)_{BSD} &= (010030030\bar{3}0101)_{2SD} \\ & \vdots \end{cases}$$

input: window width w, and $k = (k_{n-1} \cdots k_0)_2$
output: $(k)_{wSD}$

```
1:  j ⟵ 0
2:  for i from 1 to n − 1 do
3:      change (k_{i+1}, k_i, k_{i−1}) according to a random vector
4:      if j ≠ w − 1 then
5:          j ⟵ j + 1
6:      else
7:          if k_{i−w−1} ≠ 0 then
8:              k_{i−w−1} ⟵ 2^{w−1} k_{i−2} + ··· + 2^0 k_{i−w−1}
9:              (k_{i−2} ··· k_{i−w}) ⟵ (0 ··· 0)
10:             j ⟵ 0
11:     if j ≠ 0 then
12:         j ⟵ 1
13:         while k_{n−w+j} do
14:             j ⟵ j + 1
15:         k_{n−w+j} ⟵ 2^{w−j} k_n + ··· + 2^0 k_{n−w+j}
16:         (k_n ··· k_{n−w+j+1}) ⟵ (0 ··· 0)
17: return k
```

Figure 2.7: Computing the width–w SD of a positive integer in binary representation.

$$(k)_{3SD} \begin{cases} (1\bar{1}01011\bar{1}1\bar{1}1\bar{1})_{BSD} &= (010005003003\bar{3})_{3SD} \\ (011\bar{1}1\bar{1}010101)_{BSD} &= (01003003\bar{3}0005)_{3SD} \\ (10\bar{1}\bar{1}011\bar{1}011\bar{1})_{BSD} &= (100\bar{3}00050005)_{3SD} \\ (1\bar{1}0110\bar{1}0\bar{1}\bar{1}01)_{BSD} &= (010030005\bar{5}003\bar{3})_{3SD} \\ & \vdots \end{cases}$$

It is not necessary to wait for the end of the computation of $(k)_{wSD}$ to use it. We can use the new random representation of k gradually. When digit k_i is used, the next digit can be already computed, or will be computed before we need to use it. Therefore there is no latency during the scalar multiplication. In addition, a new representation of k can be computed before the end of a scalar multiplication which can be calculated by the algorithm in figure 2.8. By the same token, the width-w `NAF` method is similar to the width-w SD method for scalar multiplication. One just must pre-compute more points (at line 2): 2^{w-2}.

2.3.5 Implementation

Figure 2.9 presents the architecture of the implemented recoding unit which starts from the least significant digit. It refers to the algorithm in figure 2.6. This unit recodes the scalar k on-the-fly using randomly chosen binary signed-digit representations. The scalar k is represented and stored in binary. The two-input multiplexers selects two consecutive digits of k to perform $f(k_j, k_{j-1})$, where f refers to the function (2.1). The two-output demultiplexers stores performed results by the function f in the appropriate digits. A control unit creates one signal which controls each multiplexer and demultiplexer according to a random vector.

input: $k = (k_{n-1} \cdots k_0)_2$, and $P \in E(\mathbb{F}_p)$
output: $Q = [k]P$

1: Use algorithm in figure 2.7 to compute $(k)_{wSD}$ according to a random vector
2: Compute $P_i = [i]P$ for $i \in \{1, 3, \ldots, (2^w - 1)\}$
3: $Q \longleftarrow \mathcal{O}$
4: **for** i **from** $n-1$ **downto** 0 **do**
5: $\quad Q \longleftarrow [2]Q$
6: $\quad$ **if** $(k_i \neq 0)$ **then**
7: $\quad\quad$ **if** $k_i > 0$ **then**
8: $\quad\quad\quad Q \longleftarrow Q + P_{k_i}$
9: $\quad\quad$ **else**
10: $\quad\quad\quad Q \longleftarrow Q - P_{-k_i}$
11: **return** Q

Figure 2.8: Width–w SD method for point multiplication.

The scalar k is always kept in binary representation. First, the register k sends the three first bits $(k_0k_1k_2)_2$. The function f randomly changes these three bits in $(k'_0k'_1k'_2)_{BSD} = (k'_0k'_1k'_2)_{SD}$. Next, the register k sends k_3, and f randomly changes $(k'_1k'_2k_3)_{SD}$. This operation is repeated until the k_{n-1} is sent. A w-bit register receives all randomly changed digits, waits for w digits, and computes the decimal value of the w-bit word (line 8 of algorithm in figure 2.7). Once the value is computed, the controller can launch the appropriate curve-level operation.

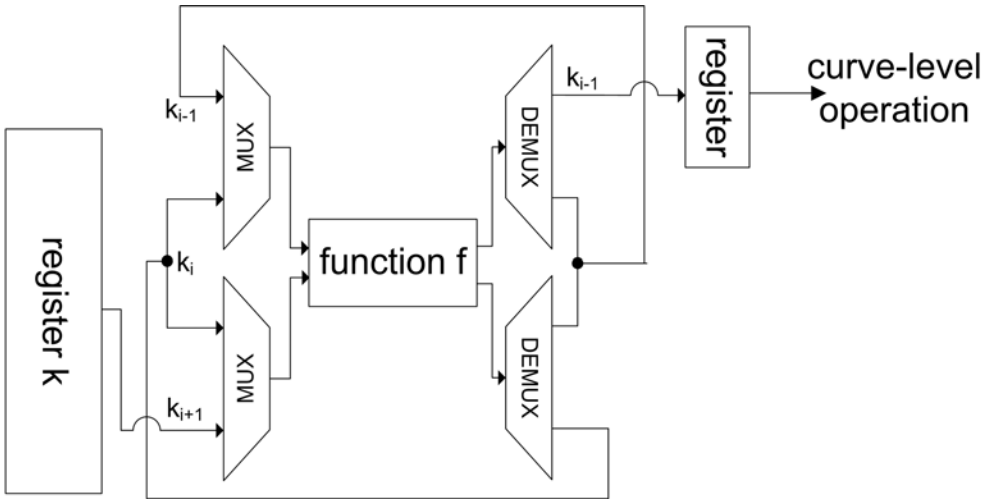
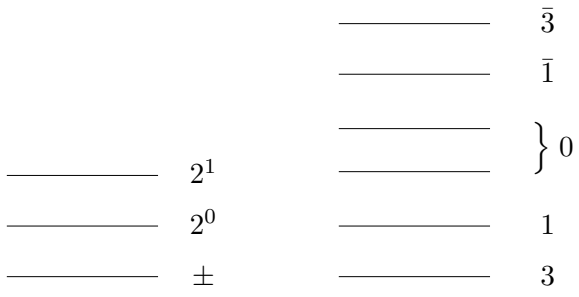


Figure 2.9: Architecture of the on-the-fly recoding unit with random window w signed-digit representations.

FPGA implementation results are reported in table 2.6 for the proposed recoding corresponding to the algorithm in figure 2.7. Two versions have been implemented: outputs are not encoded in the same way. The first presented version is implemented in sign magnitude and the second in one hot encoding. For instance with $w = 2$, the digit set is $\{\bar{3}, \bar{1}, 0, 1, 3\}$, and two circuit-level codings have been used:



For an index i, each digit of k may have $1 + 2^w$ values in sign magnitude (on the left). In this version, k_i is encoded on $w + 1$ bits. There is one bit for the sign, and the others for coding the value of $|k_i|$ in binary. The digit zero can be encoded in two ways: the sign bit can be 1 or 0. We have $k_i = (k_{i_w}, \cdots, k_{i_1}, k_{i_0})_2$. For instance, if $k_i = -3$ and $w = 3$, the output will be $k_i = (1, 0, 1, 1)_2$. If $k_i = 5$ and $w = 3$, the output will be $k_i = (0, 1, 0, 1)_2$.

In the one hot encoding version, there is one signal wire for each value. Outputs have always the same Hamming weight. In addition, this solution allows to have a constant Hamming distance. Indeed, the digit zero is the only one which can be consecutive due to the SD properties (section 2.3.4). Therefore two signals can code this value. If zero is coded with the first signal, the second zero will be coded by the second one. For instance, if $k_i = -3$ and $w = 2$, the output will be $k_i = (1, 0, 0, 0, 0, 0)_2$. In the same way, if $k_i = 0$ and $k_{i+1} = 0$, the outputs will be $k_i = (0, 0, 1, 0, 0, 0)_2$ and $k_{i+1} = (0, 0, 0, 1, 0, 0)_2$.

All hardware implementations reported in this section have been described in VHDL and implemented on a XC5VLX50T FPGA using ISE 12.4 from Xilinx with standard efforts for synthesis, place and route. FPGA implementation results are reported in table 2.6 for the wSD random recoding unit with $w \in \{1, 2, 3\}$ and $n \in \{160, 192, 224\}$. The scalar k is always kept in binary representation, and random recoding is computed on-the-fly.

n	w	area slices (FF/LUT)	freq. MHz
160	1	587 (850/1 779)	251
	2	784 (1 353/2 249)	245
	3	814 (1 516/2 301)	217
192	1	781 (1 009/2 595)	232
	2	851 (1 605/2 928)	240
	3	989 (1 803/3 220)	205
224	1	819 (1 169/2 707)	203
	2	1 054 (1 860/3 352)	205
	3	1 154 (2 091/3 607)	196

Table 2.6: FPGA implementation results of the wSD random recoding unit in one hot encoding.

In table 2.6, each digit of k is encoded on $2^w + 2$ bits (one hot encoding). The sign magnitude implementation has an equivalent speed and less than 0.5% circuit area compared to the one hot

encoding implementation.

Tables 2.8 and 2.7 report ASIC implementation results of the wSD random recoding in one hot encoding with $n \in \{160, 224\}$ and $w \in \{1, 2, 3\}$. The wSD recoding implementation is applied with a maximum path delay constraint of 10ns and 5ns from all inputs to all outputs (see section 2.2.4 for tools details).

n	w	delay	combinational	buf/inv	non combinational	total
160	1	10	20 174.0	2 805.9	28 107.6	48 281.6
		5	20 200.2	2 763.5	28 107.6	48 307.9
	2	10	29 556.0	3 130.6	28 420.3	57 976.3
		5	29 598.3	3 104.4	28 420.3	58 018.7
	3	10	40 255.2	3 171.0	47 281.1	87 536.3
		5	40 287.5	3 142.7	47 281.1	87 568.6
224	1	10	23 278.4	3 284.0	33 158.7	56 437.2
		5	23 381.3	3 249.7	33 158.7	56 540.1
	2	10	43 559.4	3 622.8	50 264.5	93 824.0
		5	43 368.8	3 596.6	50 264.5	93 953.1
	3	10	47 720.9	3 616.8	55 951.0	103 671.9
		5	46 216.0	3 544.2	55 951.0	102 167.1

Table 2.7: Area results (in μm^2) of ASIC implementation of the wSD random recoding unit in one hot encoding.

2.4 Comparison

In order to numerically validate our method, we compared our results to software results provided by PARI/GP which is a computer algebra system with many powerful number theory functions. We also used this software environment to experimentally evaluate the number of operations performed by the two proposed random recoding methods. All obtained results have been compared to standard recoding methods (`NAF`, w`NAF` see [57, p. 98]).

Below we compare our DBNS recoding and scalar multiplication algorithms. We report results for short Weierstrass curves ($y^2 = x^3 - ax + b$) over $\mathbb{F}_p$ with an unspecified curve parameter a and curves where the parameter $a = -3$, using Jacobian coordinates (denoted $\mathcal{J}$). We use best operation costs at curve level reported in table 1.3 in section 1.1.6 and table 1.5 in section 1.2.2.

The reported experiments have been realized using a prime field on 160 bits. We also used mixed point addition: Jacobian coordinates with affine ones (denoted $\mathcal{A}$). `mADD` is the cost of point addition in mixed coordinates ($\mathcal{J} + \mathcal{A} \to \mathcal{J}$).

We compared the $[k]P$ computation cost using our random DBNS recoding to the results based on wSD recoding and on standard methods. The average cost for computing $[k]P$ using 10 000 random DBNS recoding on 160 bits is

$$62\,\texttt{ADD} + 112\,\texttt{DBL} + 71\,\texttt{TPL}.$$

The corresponding results are reported in table 2.9 where `M` is the cost of one multiplication in

n	w	delay	cell internal	net switching	total dynamic	cell leakage
160	1	10	1 708.0	83.2	1 791.2	17.0
		5	3 416.6	167.7	3 584.4	17.0
	2	10	1 742.6	136.6	1 879.3	20.0
		5	3 483.8	274.8	3 758.6	20.0
	3	10	2 921.2	96.9	3 018.1	30.5
		5	5 845.1	195.1	6 040.2	30.5
224	1	10	2 041.9	100.5	2 142.4	19.9
		5	4 080.6	203.4	4 284.1	20.0
	2	10	3 126.8	116.0	3 242.8	32.7
		5	6 257.2	233.8	6 491.1	32.7
	3	10	3 483.6	116.2	3 599.9	37.5
		5	6 968.7	233.4	7 202.1	35.6

Table 2.8: Power results in (μW) of ASIC implementation of the wSD random recoding unit in one hot encoding.

$\mathbb{F}_p$. We use the standard cost approximation for square $\mathtt{S} \approx 0.8\,\mathtt{M}$.

method	performances with	
	$a = -3$	$a \neq -3$
double-and-add	1 922.0M	1 985.3M
NAF, SD	1 659.7M	1 723.0M
3NAF, 3SD	1 520.2M	1 583.7M
4NAF, 4SD	1 436.1M	1 499.1M
DBNS random recoding	1 659.2M	1 771.4M

Table 2.9: Comparison of scalar multiplication with $n = 160$.

The costs of wNAF and wSD are similar. wSD representation requires more pre-computed points than wNAF but this method has the advantage of having a randomized behaviour for several scalar multiplications. In addition, random recoding in DBNS needs more M for computing scalar multiplications than standard DBNS scalar multiplications. Indeed, the random recoding of a DBNS chain is less sparse than the initial one: several random recodings are performed to ensure that a recoding was not ever produced.

2.5 Conclusion

In this chapter two methods have been studied. In particular, we use two redundant representations for on-the-fly random recoding of the scalar k using the double-base number system and the signed-digit representation.

Thus, we have seen that the use of these representations can be considered realistic in hardware and implementable. The on-the-fly recoding units in DBNS and in wSD have been implemented in a FPGA and in an ASIC. All recoding units can work at a clock frequency greater

than the complete ECC processor (appendix A). In addition, the FPGA area required by the DBNS (resp. wSD) recoding unit represents less than 7% (resp. 28%) compared to the complete ECC processor.

Starting from a scalar k in binary representation, our method randomly provides different representations of the recoded scalar k. Then the number and the order of curve-level operations (point additions, doublings and triplings) is randomized during several scalar multiplications $[k]P$. Thus, it could be used as a countermeasure against some side-channel attacks. However, the security against side-channel attacks of such implementations must be evaluated by attacks. Only one attack is performed to evaluate arithmetic level protections against some side channel attacks. The attack results are reported in the next chapter 3.

A study between each protection system should be theoretically and practically evaluated and compared to know the tradeoffs between protection levels and hardware implementation results. In other words, what is the cost of each protection system, and how resistant is each system?

Chapter 3

Practical Security Evaluation Using Template Attacks

Very efficient side-channel attacks (SCA) have been proposed such as power analysis [80] or electromagnetic radiations analysis [3] [112]. Side-channel attacks allow the extraction of secret informations from running devices by measuring and analysing physical parameters such as power consumption, electromagnetic radiations or computation time. For instance, recording and analysing the instantaneous power consumption of a circuit may lead to very efficient power attacks (see side-channel attacks text book [80]). Simple power analysis (SPA, see [80, Chap. 5]) directly uses the fact that basic implementations of point addition and point doubling operations have different power traces. Thus, the recorded power traces will directly show where/when are the `ADD` operations and the sequence of `DBL` ones. With classical methods such as the double-and-add algorithm, the bits of k can be guessed. In addition, point additions (`ADD`) and subtractions (`SUB`) can be computed in such a way that there are indistinguishable in the traces. This motivates the use of signed digits. Thus an attacker cannot guess when there is an `ADD` or a `SUB`. It complicates the attack when there are a large number of non-zero digits.

Recently, a particularly interesting class of a probabilistic side-channel attack, called *templates attacks* was introduced in [27]. In a template attack, the attacker is assumed to know characteristics of a side channel over some processed data in a device. This characterization is stored and called template. The attacker matches the templates based on different key guess with the recorded traces.

In this chapter, two template attacks are performed on the point scalar multiplication, operation which dominates the computation time in ECC protocols. The goal of the performed attacks is to practically evaluate a DPA countermeasure based on scalar recoding in signed digits. The first (resp. second) one is on an implementation with a countermeasure against simple (resp. differential) side-channel attacks. The countermeasure against simple side-channel attacks uses a regular scalar multiplication algorithm [31]. For the second template attack, the scalar is randomly recoded. The scalar is recoded using a redundant representation studied and implemented in chapter 2: signed-digit representations can be used to be robust against differential SCAs.

Notations used below, which are the same ones as [80], are:

- $k = (k_{n-1} \cdots k_1 k_0)_2$, $k > 1$ is the n-bit scalar,
- $k^{(i)}$ is a possible key value for k,
- l is the number of collected traces,
- D is the number of points in each trace,

– $T = \begin{pmatrix} t_1^{(1)} & t_2^{(1)} & \cdots & t_D^{(1)} \\ t_1^{(2)} & \cdots & \cdots & t_D^{(2)} \\ \vdots & & & \vdots \\ t_1^{(l)} & \cdots & \cdots & t_D^{(l)} \end{pmatrix}$ is the matrix of power consumption values of size $l \times D$. Each row corresponds to one power trace t. T can be seen as a set of l traces t.

3.1 Template Attacks

Template attacks, introduced by Chari et al. [27], are a type of side-channel attack. This attack model used in [52, 53, 81, 98, 104, 117] is powerful: the adversary has full access to a device and is able to record side-channel data for chosen keys and plaintexts. These attacks characterize leakage information and obtain information on manipulated data. They are not about the execution regularity or the execution order of computed operations like in classical simple analysis. These attacks deal with the residual leakage according to the manipulated data. For example, one can consider the Hamming weight or the Hamming distance model.

Template attacks consist of two phases. The first one, the *profiling phase*, also named *template generation*, is the characterization where an attacker records side-channel data for chosen keys and plaintexts. A side channel of a device is characterized and templates are generated under attacker control. The second phase, the profiling phase, also named *template classification*, uses the characterization results to evaluate what is the probability that the secret key (i.e. the scalar)) k used by a cryptographic device is equal to $k^{(i)}$.

3.1.1 Template Generation

In this first phase, a large amount of traces are gathered. A trace is measured from a side channel such as power consumption or electromagnetic emanation. All traces are taken from the same operation or the same cryptographic algorithm. Templates are generated by traces with the properties of the probability distribution for all points. In other words, templates consist in estimating the distribution defined by the mean vector m and the noise covariance matrix C of the points.

Ideally, templates must be built for each possible operation, that is for all possible data and key hypothesis used by a cryptographic algorithm. In addition, each of these values is built for a large number of traces l. Thus, for each pair of possible data and key hypothesis, we have l traces and one corresponding template (m, C) [80, Sec. 4.2.1].

$$m = (m_1, \cdots, m_D) \qquad \text{with} \qquad m_i = \frac{1}{l} \sum_{j=1}^{l} t_i^{(j)},$$

$$C = \frac{1}{l-1} \sum_{j=1}^{l} (t^{(j)} - m)^\intercal (t^{(j)} - m).$$

where $t^{(j)}$ is the jth trace, and $t_i^{(j)}$ is the jth trace sample of the ith measure.

Determining the size of the covariance matrix can be a delicate issue in this attack. Direct computations by handling a huge amount of points in the trace can be inefficient. Using the entire trace will lead to large matrix inversion problems. Indeed, without reducing traces, a great deal of matrices of size D are going to be inverted. Experimentally, we observed that if D is 2 000 or longer, it will take too much time and can run into accuracy problems. Some errors

may occur in the second phase, the template classification (e.g. when inverting the covariance matrix).

The solution is to reduce the length of traces: some points leak more information than others and these are the points to use. The determination for a practical template attack of these points, so called *interesting points* or *points of interest*, is an important issue. The interesting points make for reducing the trace length because not all points of a trace are part of the template. They can be chosen by selecting points with maximal variance or with large differences between the average traces. Another technique for data analysis and processing called principal component analysis [63] can be applied.

In practice, templates can be generated in different ways. One can consider templates with power models, templates for intermediate values [58] or templates for pairs of data and keys [80, pp. 105–118]. The strategy varies from one device to another or the operations performed in the attacked device. In the case where one builds templates for each pair $(d^{(i)}, k^{(i)})$, the interesting points of a trace are all points or all instructions which involve $d^{(i)}$, $k^{(i)}$ and functions of $(d^{(i)}, k^{(i)})$.

3.1.2 Template Classification

This second phase consists in acquiring one or more traces t with the unknown secret key k, and in computing the probability that the actual device uses the secret key $k^{(i)}$. From this second phase, T is the matrix which corresponds to the set of traces t, with D points in each trace. For each generated template $(m, C)_{d^{(i)},k^{(i)}}$ and its corresponding traces, the probability density function pr of the multivariate normal distribution can be calculated as follows:

$$pr(t|k^{(i)}) = pr(t; (m, C)_{d^{(i)},k^{(i)}}) = \frac{1}{\sqrt{(2\pi)^D \det(C)}} \exp^{-\frac{1}{2}(t-m)C^{-1}(t-m)^\intercal}.$$

This probability must be computed with all templates generated during the first phase. Indeed, templates have been built for different pairs of $(d^{(i)}, k^{(i)})$. Assuming that the attacked subkey can take K values, one must calculate all probabilities

$$pr(t|k^{(1)}),\ pr(t|k^{(2)}),\ \cdots,\ pr(t|k^{(\mathsf{K})}),$$

with K as the total number of possible subkeys. When K is high, it can be difficult to generate associated templates. In practice, template attacks are performed to guess a part of the secret key. The highest probability should indicate the correct template. Indeed, each probability is associated with one template, and so with one key. To refine these results, one can calculate the probability of each possible template using Bayes' theorem recalled below. The probability $pr(k^{(i)}|t)$ which can be viewed as an update function of probabilities, is then computed on the prior probability $p(k^{(i)})$ and $pr(t|k^j)$ for $j = 1, \cdots, \mathsf{K}$:

$$pr(k^{(i)}|t) = \frac{pr(t|k^{(i)}) \times p(k^{(i)})}{\sum_{j=1}^{\mathsf{K}} \left(pr(t|k^{(j)}) \times p(k^{(j)})\right)}.$$

However this equation may not be sufficient because a single trace may not have enough information available to reveal the key. Indeed, some noise are present in every measurement in practice. The side-channel information related to the attacked sequences (the signal) needs to be larger than the side-channel information related to the unrelated sequences (the noise) [97, p. 74]. A perfect measurement setup would be a setup that only measures the power consumption of the part of the cryptographic device that is relevant for the attack. The power consumption of all other parts of the device is noise from the attacker's point of view [80, Sec. 3.5].

In order to enhance the efficiency of this attack, one can increase the number of measured traces. Considering a set of traces T allows to minimize the probability of false classification errors. The probability $pr(k^{(i)}|T)$ is computed like an extension from $pr(k^{(i)}|t)$, and is calculated by multiplying the probabilities and applying iteratively Bayes' theorem. Thus, one computes the probability $pr(k^{(i)}|T)$, that is a key corresponding to the measured traces:

$$pr(k^{(i)}|T) = \frac{\left(\prod_{j=1}^{l} pr(t^{(j)}|k^{(i)})\right) \times p(k^{(i)})}{\sum_{g=1}^{\mathsf{K}} \left(\left(\prod_{j=1}^{l} pr(t^{(j)}|k^{(g)})\right) \times p(k^{(g)})\right)},$$

where l is the number of traces in T. The higher the set of traces is, the more the success of the attack is increased.

3.2 Used Architecture for the Attacks

In this chapter, we use a generic architecture designed for ECC operations in the University College Cork (UCC, Ireland) by Byrne et al. [24]. The provided architecture and all works in this chapter were carried out by a doctoral exchange student, and was completed by using the funds from the European mobility grant from UEB (Université Européenne de Bretagne) to carry out research in UCC for three months (see appendix A).

3.2.1 Measurement Setups

Experimental template attacks were carried out on a Xilinx XC5VLX50 FPGA embedded on a Sasebo-GII. An oscilloscope measures the power consumption of the main power supply for the FPGAs internal logic. It is possible to measure the instantaneous power consumption of the board during the runtime of a cryptographic operation. The dynamic power consumption at the internal power nets is directly affected by the dynamic power consumption of cryptographic operation executed on the FPGA.

In order to implement the cryptoprocessor on the board, a wrapper was created to act as the input/output interface. The wrapper consists of a counter, a FSM (finite state machine) and a ROM instruction set containing the input points and a set of keys for testing. The counter counts through each instruction in the ROM. The FSM controls the loading of the input points and the scalar k and then triggers the processor to start the point scalar multiplication [24].

3.2.2 Proposed Architecture

An architecture for the scalar point multiplication can be generated for any characteristic p through the aforementioned reconfigurable architecture [24]. The architecture is described in appendix A.

An architecture was generated for the NIST recommended prime characteristic field $p_{192} = 2^{192}-2^{64}-1$ [57, Annex A.2.1]. The ECC processor (for curves over $\mathbb{F}_p$, $n = 192$ bits and Jacobian coordinates) presented in figure 3.1 includes one dedicated unit for each modular operation (modular addition, subtraction, multiplication and inversion).

The processor was generated for the Euclidean addition chains algorithm from [22]. This algorithm provides a regular behaviour for scalar multiplication. Given a point P on an elliptic curve E and an integer $k = (k_{n-1} \cdots k_0)_2$ transformed into an Euclidean addition chain (EAC), one can calculate the scalar multiplication $[k]P$ (algorithm in figure 3.2). Let the EAC computing the integer k be $v = (1, 2, 3, v_4, \cdots, v_s)$ with $v_i \in \mathbb{N}$. One can replace the v_is by 0 and 1, which gives the chain c. Thus, we use the notation $c = (c_4, \cdots, c_s)$ with $c_i \in \{0, 1\}$ to describe the

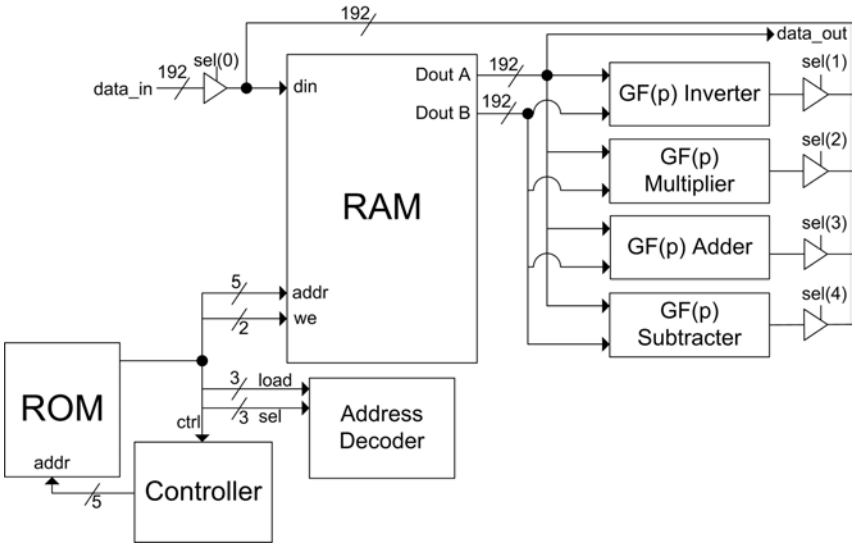


Figure 3.1: Specific cryptographic architecture for the NIST curve P-192 (from [24]).

EAC generated for k [22]. The chain c is computed by the algorithm provided in [82] for each scalar multiplication.

input: $P \in E(\mathbb{F}_p)$, and the chain $c = (c_4 \cdots c_s)_2$ computing the scalar k
output: $[k]P$

1: $U_1 \leftarrow [2]P$
2: $U_2 \leftarrow P$
3: **for** i **from** 4 **to** s **do**
4: **if** $c_i = 1$ **then**
5: $U_1 \leftarrow U_1 + U_2$
6: $U_2 \leftarrow U_2$
7: **else**
8: $U_1 \leftarrow U_1 + U_2$
9: $U_2 \leftarrow U_1$
10: $U_1 \leftarrow U_1 + U_2$
11: **return** U_1

Figure 3.2: Scalar multiplication using Euclidean addition chains (from [31]).

The field operation count of the algorithm in figure 3.2 is $(s-2)$`ADD` + `1DBL`. It performs the scalar multiplication using one initial point doubling and then only point additions. This method allows to avoid simple power analysis attacks. Indeed, with only point additions, the analysis of side-channel information does not enable us to identify the bit pattern of the chain c which computes the scalar k.

In figure 3.3, a power trace of a part of a scalar multiplication using the algorithm in figure 3.2 is presented. The algorithm in figure 3.2 prevents simple power attacks. That is why there are only point additions in this power trace. One cannot guess key digits used with one single power trace.

The power consumption of the device is measured along a resistor using a digital oscilloscope.

The recorded voltage drop is proportional to the power consumption of the FPGA. In addition, the power supply (V_{DD}) is fixed. Thus, we refer to the voltage drop as power consumption and to the corresponding trace as power trace in this chapter.

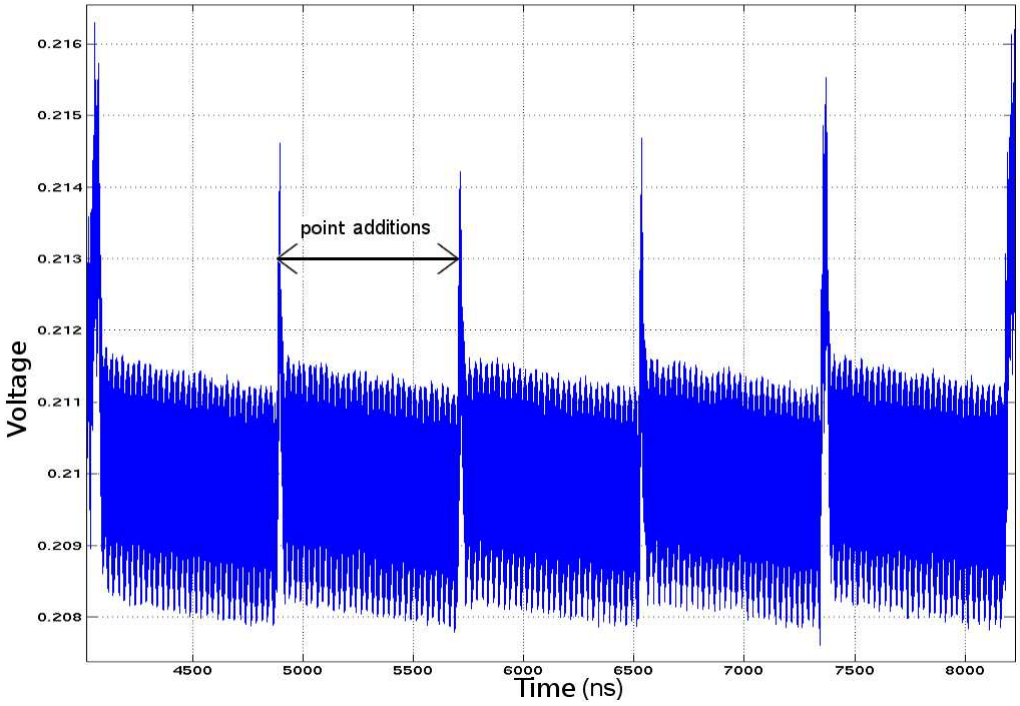


Figure 3.3: Power trace of the scalar multiplication during five iterations.

3.2.3 FPGA Implementation

Table 3.1 reports FPGA implementations of the ECC processor (for curves over $\mathbb{F}_p$, $n = 192$ bits and Jacobian coordinates). Two versions are implemented. The first one uses BRAMs while the second one does not. The implementation uses Euclidean addition chains method with one arithmetic unit per field operation. Hardware implementations reported in this chapter have been described in VHDL and implemented on a XC5VLX50 FPGA using ISE 9.2 from Xilinx with standard efforts for synthesis, place and route. We report best clock frequencies and numbers of occupied slices. We also report numbers of look-up tables (LUTs with 6 inputs in Virtex 5) and flip-flops (FFs) for area. A XC5VLX50 contains 7 200 slices with 4 LUT and 4 flip-flops per slice.

memory type	area		freq.
	slices (FF/LUT)	BRAM	MHz
distributed	2 455 (2 200/7 945)	0	154
BRAM	2 035 (1 933/7 022)	6	154

Table 3.1: FPGA implementation results for the complete ECC processor over $\mathbb{F}_p$ with $n = 192$ bits.

3.2.4 Power Model

Power consumption or electromagnetic emanations of a device depend on the operations they perform, on the data they process and is often proportional to the Hamming weight of the manipulated values. The bit switching activity consumes power in FPGAs. If we assume that the target device behaves accordingly to the Hamming weight model, the difference of scalar multiplications for different keys can be observed by measuring these side channels. Indeed, whereas the algorithm in figure 3.2 always performs point additions, the Hamming distance and weight is different for each key (see example below).

Example. *Let us see what happens with two different chains c and c':*

	$c = (100110)_2$		$c' = (110101)_2$
begin	$(U_1, U_2) = ([2]P, P)$	*begin*	$(U'_1, U'_2) = ([2]P, P)$
$c_4 = 1$	$(U_1, U_2) = ([3]P, P)$	$c'_4 = 1$	$(U'_1, U'_2) = ([3]P, P)$
$c_5 = 0$	$(U_1, U_2) = ([4]P, [3]P)$	$c'_5 = 1$	$(U'_1, U'_2) = ([4]P, P)$
$c_6 = 0$	$(U_1, U_2) = ([7]P, [4]P)$	$c'_6 = 0$	$(U'_1, U'_2) = ([5]P, [4]P)$
$c_7 = 1$	$(U_1, U_2) = ([11]P, [4]P)$	$c'_7 = 1$	$(U'_1, U'_2) = ([9]P, [4]P)$
$c_8 = 1$	$(U_1, U_2) = ([15]P, [4]P)$	$c'_8 = 0$	$(U'_1, U'_2) = ([13]P, [9]P)$
$c_9 = 0$	$(U_1, U_2) = ([19]P, [15]P)$	$c'_9 = 1$	$(U'_1, U'_2) = ([22]P, [9]P)$
end	$U_1 = [34]P$	*end*	$U'_1 = [31]P$

In the circuit, a 192-bit wide RAM is implemented. Outputs are registered and data is stored in LUTs. This distributed RAM is used for storing intermediate values. We can see in the example that for different values c_i of a chain, the RAM which contains all coordinates of U_1 and U_2, is different. In addition, all intermediate values computed during a point addition are stored in the RAM. In Jacobian coordinates, the running time for a point addition expressed in terms of field operations is 9 additions, 11 multiplications and 5 squares. When the FPGA computes a point addition, the power consumption of logic cells, the number of transitions and the Hamming weight of the RAM change following the bit pattern of the chain computing the scalar k. Indeed, input and output points, U_1 and U_2, are different depending on whether the key bit value is 1 or 0. The number of transitions ($0 \to 1$ and $1 \to 0$) that occur in the circuit during a certain time interval is different according the processed bit.

Power consumption of a circuit depends on the number of logic cells. What consumes is the Hamming distance change at the output of the RAM. As that change propagates through the circuit, other lines and buses change value and consume power until it reaches a register.

Figure 3.4 shows the difference in the mean power consumption for 1 000 scalar multiplications with a constant scalar k, and 1 000 scalar multiplications with another constant scalar k'. The number 1 000 is an arbitrarily choice. Below in section 3.3.2, we discuss on the number of acquired traces for each possible key. For a same operation when mean power traces are subtracted, no peaks are present. There are also smaller peaks at the beginning when computing the only point doubling $[2]P$. Figure 3.4 shows that the power consumption can be used to create templates to distinguish the difference between scalar multiplications performed with different secret keys. Indeed, there are distinct peaks when examining averages. These peaks indicate that power consumption only depends on some data at few instances, and that there is a difference

of leaking information with different keys.

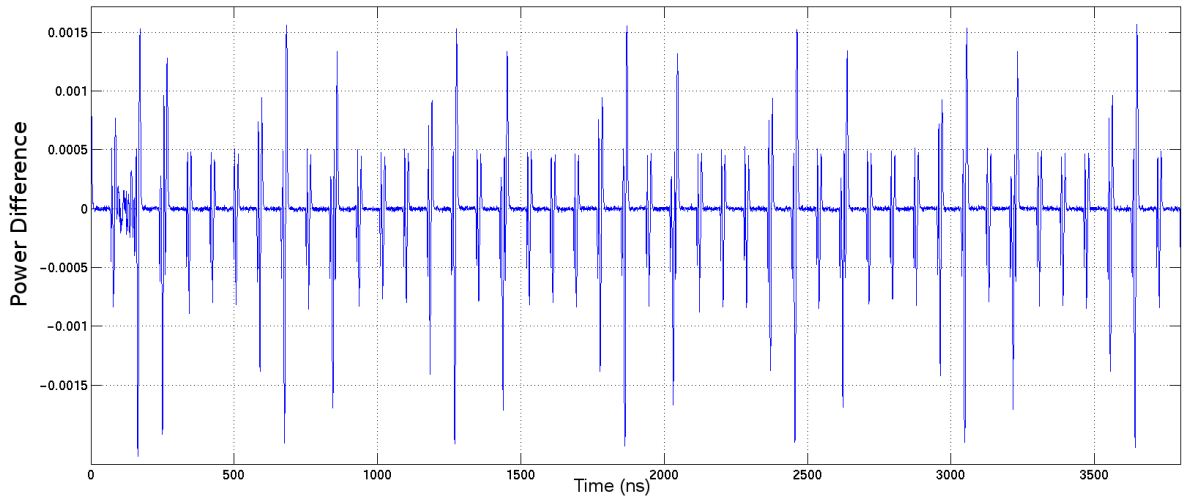


Figure 3.4: Difference between mean traces for 2 keys.

Attacks can be performed and the figure 3.4 has been generated because traces are aligned. Indeed, the oscilloscope is triggering: it is configured to start recording the considered side channel with the trigger signal. In the cryptoprocessor, the control signal triggers a delay loop in the finite state machine that waits until the scalar multiplication is complete.

3.3 Template Attacks Implementations

The goal of template attacks is to guess a part of the key k used by the cryptographic algorithm executed on the device under attack. Indeed, for template attacks, we are not attacking the entire scalar operation. We investigate a specific operation. One needs to decide which part of the algorithm to attack. For example, one can need some simple operations where the secret key k and the known point P are processed with an intermediate value, function of k and P.

In our attack, we only focus on several bits of the key. The set of these bits that we want to guess must be small enough to try all possible subkeys. The measured side channel in this attack is the power consumption of the cryptographic device. Communications are opened up and data are sent via the serial port to the FPGA which is set up for USB communications. A trigger signal notifies the oscilloscope when to start and when to finish trace acquisition. The result of the computation $[k]P$ is read back to ensure correct operation. The same point P is sent twenty times before reading back the power trace. The oscilloscope averages the traces to reduce noise by a twenty iterations loop. This loop improves trace quality. If one sends the same data many times, and takes the average, there will be less noise on the trace. It returns a "cleaner" trace. There is an average parameter which tells the oscilloscope how many times to average traces. Using this method, an attacker is powerful. Then, through a numerical computing environment, Matlab, the same data is also sent many times. The bigger the value for the average is, the cleaner the traces are, but the longer it takes to acquire them.

3.3.1 Template Attack with a SPA Countermeasure

Let the secret key $k = (k_{191} \cdots k_1 k_0)_2$. For this attack, we focus on the first three bits evaluated by the algorithm in figure 3.2. The goal of this attack is to guess three bits of the chain c which corresponds to the unknown secret key. That is why we now consider that we want to guess the three initial bits of the key instead of the three bits of the chain for the sake of simplicity. The number three is an arbitrarily choice. For that, the first strategy is to build templates for the $2^3 = 8$ possible values. We use this characterization for the attack. Here, templates are generated for all possible key values, because the input data P is supposed to be fixed.

Thus each possible key is sent via the serial port to the FPGA in order to measure the resulting power consumption. Building templates involves modelling the mean and covariance power consumption. One needs to be sure to measure enough traces to provide an accurate model, which is typically assumed in the region of 1 000 traces. On this device, we execute, read back and acquire 1 000 traces for each key, $(k^{(0)}, k^{(1)}, \cdots, k^{(7)}) = (000, 001, \cdots, 111)_2$. That is 1 000 scalar multiplications for each possible key. Thus, the set of traces T is a matrix of $8 \times 1\,000 = 8\,000$ rows.

The next step is to determine the interesting points. The strategy for providing the most information for the template is to take the vectors of means, to compute differences of each pair of mean vectors and sum them up. The highest peaks are the interesting points. These points allow firstly to reduce the traces length, and secondly to concentrate on points which leak more information than some others. Indeed, selecting interesting points allows to reduce redundant information. By this method, the information is reduced when the power consumption is the same, else the characteristics difference are discorded.

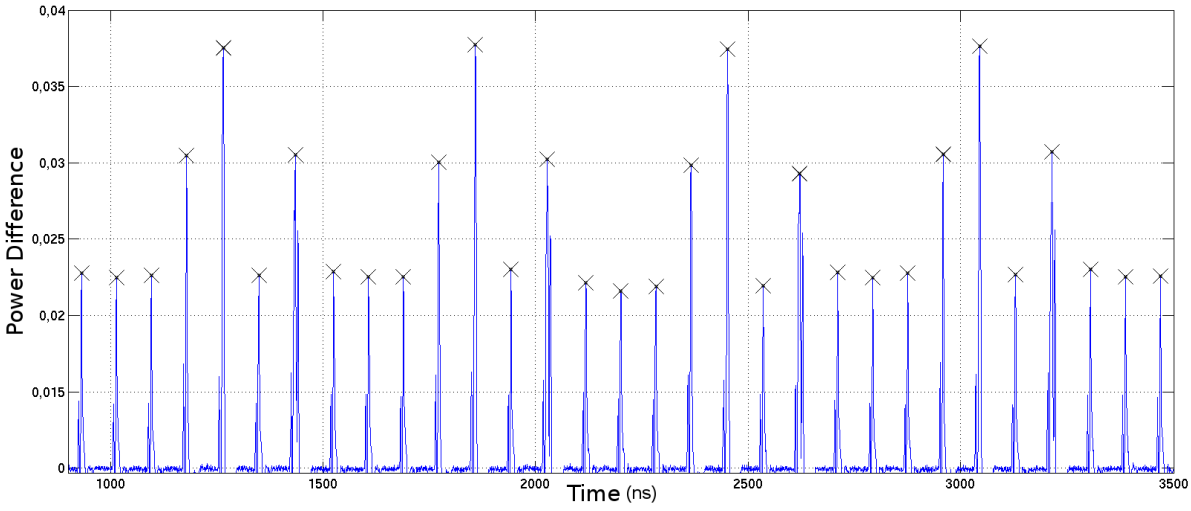


Figure 3.5: Interesting points (crosses) used to build templates.

Figure 3.5 shows the difference of means between different scalar multiplications. Crosses, which are the peaks of correlation traces, indicate the interesting points in time retained to generate templates. Only several points for each peak are necessary. One just needs to take the index of these points and then, to reduce the actual traces. These interesting points of power traces give the locations where the power transitions for the different operations occur.

Indeed, when traces have almost the same power consumption at a given time, the difference

of means is closed to 0 at this time. In this case, this point is not characteristic. When one mean of traces has not the same power consumption than the others at the same time, there will be a big peak due to the computation of the differences of each pair of mean vectors. It means that this point is characteristic to a trace. If there is a peak of power consumption at this time with a trace computed with an unknown key, one could guess the key used because only one template corresponds to this peak. A lower peak is present when several means of traces have almost the same power consumption, and the others do not. This peak is less characteristic, because it corresponds to a set of templates, i.e. a set of possible keys.

Reducing number of traces points allows to build templates for each possible operation. A vector of means and a covariance matrix, computed with the interesting points of the traces, characterize each power trace. This way, eight templates are built, that is eight vectors of means and eight matrices of covariance, because we focus on three bits of the key. These vectors and matrices correspond to the templates which represent the behaviour of the device when it processes a specific key. Indeed, the noise generated by the device varies according to the key used, and thus, created templates represent the distribution of probability $pr(k^{(i)}|t)$.

Now, the characterization is used with power traces from the device under attack to determine the secret key k. One must evaluate the probability density function of the multivariate normal distribution with the templates and the power traces of the device. In other words, we use these templates to know the probability that a key corresponds to the collected traces. For that, 100 traces are recorded with an unknown secret key. From this second phase, a trace t corresponds to a trace measured with the unknown secret key. T is the matrix of power consumption values which represents the set of 100 measured traces.

The probability $pr(k^{(i)}|t)$ is computed for each possible key and for each trace. These probabilities measure how well the generated templates fit to each trace t. Figure 3.6 shows the mean and the standard deviation of probabilities $pr(k^{(i)}|t)$ for the collected traces corresponding to a key value. The standard deviation is truncated when the value is out of the range $[0, 1]$. Intuitively, the correct key seems to be the key hypothesis $k^{(i)} = 5$. Indeed the majority of probabilities $pr(5|t)$ are much higher than for all others key guesses. For the key $k^{(i)} = 5$, we have a mean of probability 0.74 for the 100 traces, and a standard deviation of 0.26.

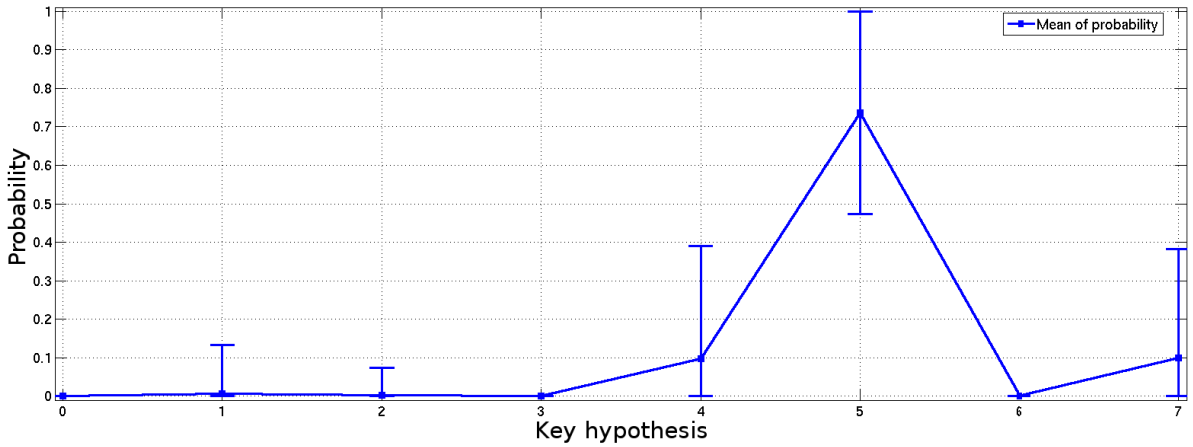


Figure 3.6: Mean and standard deviation of probabilities for each key hypothesis $k^{(i)}$.

Figure 3.7 shows the probability $pr(k^{(i)}|t)$ for the 100 collected traces, and for two keys possibilities. The key with the highest probability should indicate the correct template $k^{(i)} = 5$,

and another key guess $k^{(i)} = 7$. For example, the probability $pr(k^{(i)}|t)$ is 0.48 for the 75th trace for the key $k^{(i)} = 5$. Most of the time, the key guess $k^{(i)} = 5$ leads to the highest probability. That is why attacks based on a single trace do not always succeed. Indeed, with only one or few traces, the key $k^{(i)} = 7$ could lead to the highest probability. However with several traces, we can see that the key $k^{(i)} = 7$ does not correspond to the correct template.

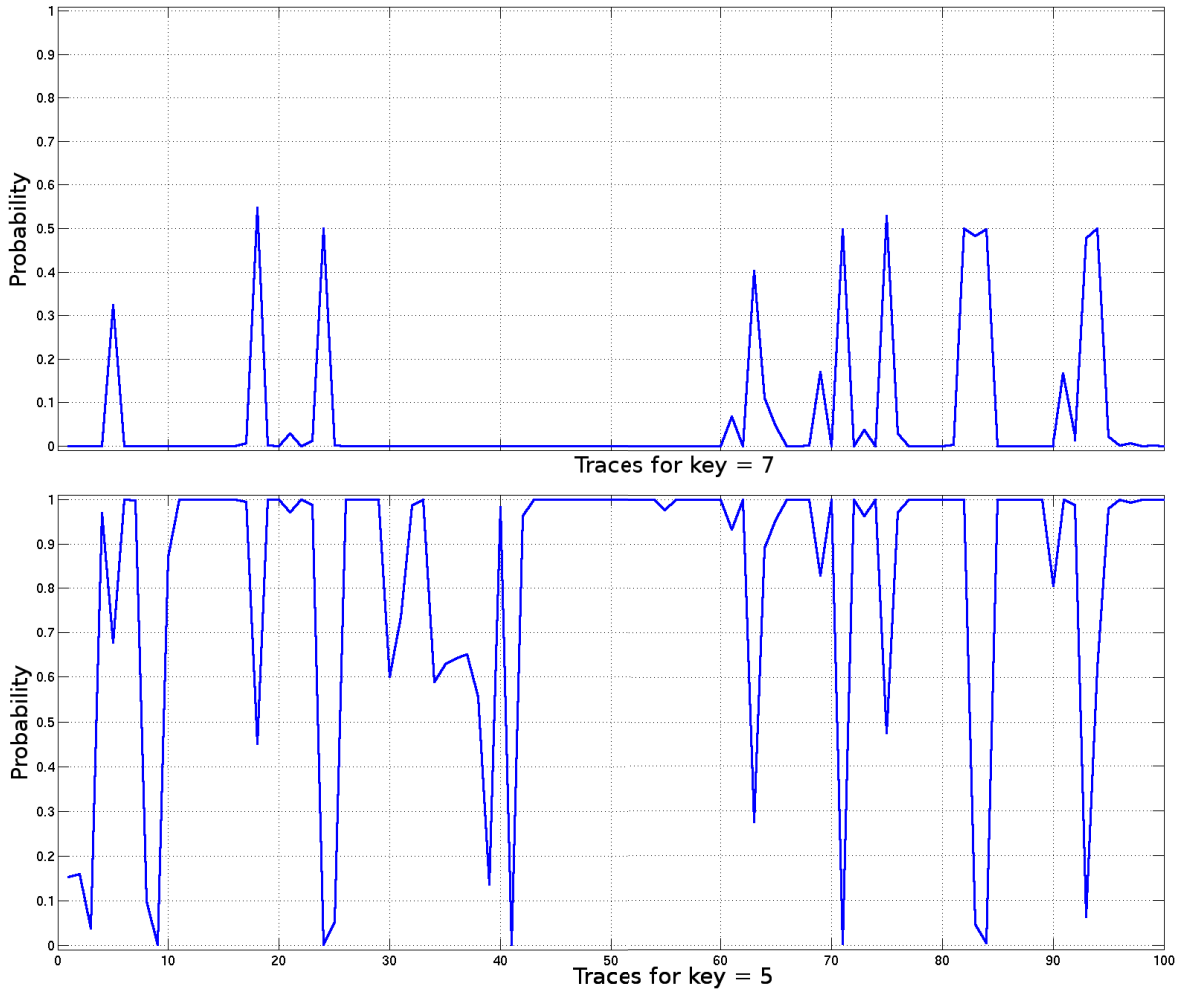


Figure 3.7: The probabilities $pr(k^{(i)}|t)$ for each trace t with $k^{(i)} = 7$ and $k^{(i)} = 5$.

100 traces have been recorded from the device under attack with an unknown secret key. For each trace, the probability density function was computed, and then, one can multiply the key probabilities for each trace by each other and scale again. This extension from $pr(k^{(i)}|t)$ allows to compute the probability $pr(k^{(i)}|T)$ where T is the matrix with 100 rows of power consumption values. Each row corresponds to one power trace. The result of this probability confirms the impression at the beginning, that is the key $k^{(i)} = 5$ is the correct one. Indeed, for all key possibilities, the probability is close to 0 except for the key $k^{(i)} = 5$ where this probability is almost 1. The figure 3.8 shows how the probability $pr(k^{(i)}|T)$ evolves as a function of the number of traces for $k^{(i)} = 5$. The correct key leads to a probability of almost 1 after 13 traces. This probability minimizes the probability of errors and reveals the correct key used by the device. Furthermore the traces order does not have an impact on this probability: the correct key leads

to a probability of almost 1 after a number of traces between 7 and 18 when one considers 1 000 experiments (each experiment corresponds to a different traces order).

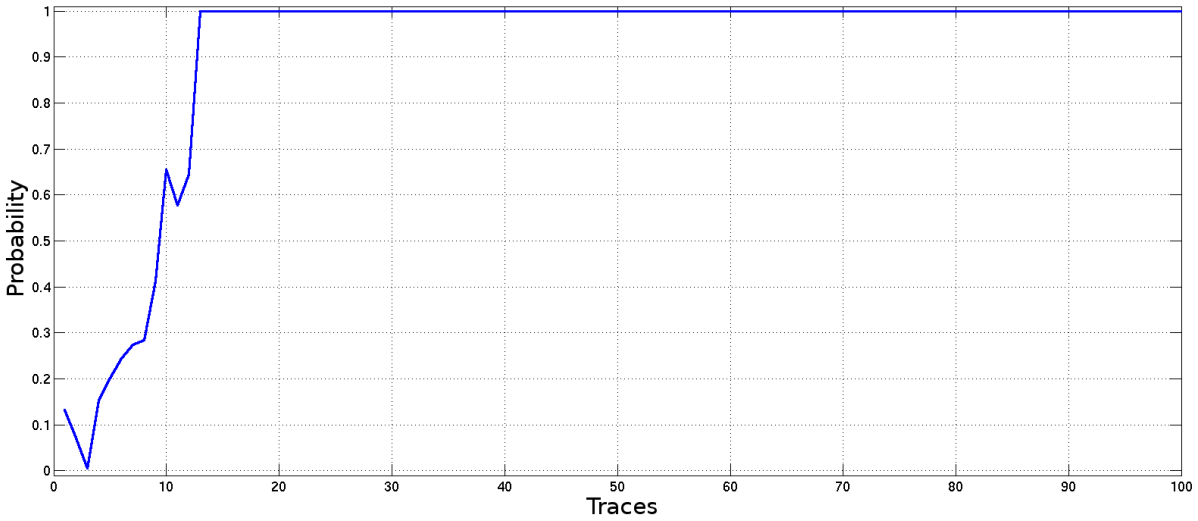


Figure 3.8: Evolution of the probabilities $pr(k^{(i)}|T)$ with an increasing number of traces ($k^{(i)} = 5$).

For a secure implementation against SPA, a template attack succeeds to find a piece of the secret key. The key k can be revealed by repeating this method until guessing the entire key.

This template attack has been performed with a fixed parameter. In particular, we have built the templates with 1 000 traces collected for each possible key. Afterwards, we will examine the number of the gathered traces and see the evolution for a practical template attack.

3.3.2 Evaluation of Traces Number During Templates Generation

The first phase of a template attack consists in the profiling phase. Templates are generated according to a mean vector and a noise covariance matrix which are calculated from a large amount of gathered traces. On the previous template attack, 1 000 traces were acquired for each possible key. For each one, the 1 000 traces are very similar because the same key and the same data have been processed. Whereas the 1 000 traces are collected with the same input point and the same key, there is a difference between every power trace due to noise. In order to view these differences, one can examine power consumption at a fixed time to see how the points of power traces are distributed. Differences are presented by the histogram in figure 3.9 with 1 000 traces at only one point at time 2 000 ns. For each trace at time 2 000 ns, we group traces which have the same power consumption. We can see that figure 3.9 is in the shape of a Gaussian.

Figure 3.9 shows that points in the power traces seem to follow a normal distribution. That is why one must measure a large amount of traces for a same key and the same data to obtain average power traces with the less electric noise. Afterwards, we are going to try to determine how many traces are necessary to generate good templates. By "good templates", we mean that the generated templates allow to succeed a template attack with a very high probability.

We consider that the obtained accuracy with 1 000 traces is always sufficient. Thus we choose this accuracy as a reference: 1 000 traces correspond to a 100 percent accuracy. Figure 3.10 presents the evolution of the accuracy of the obtained average in function of the number of

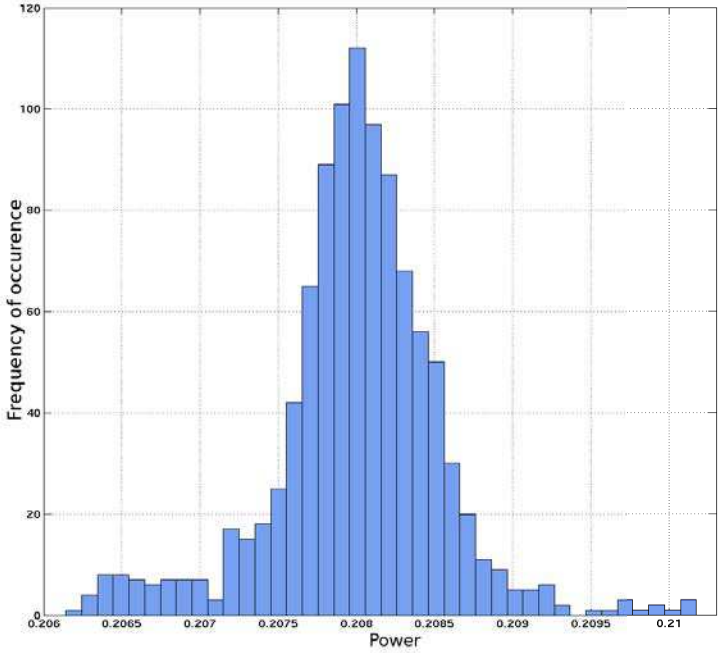


Figure 3.9: Histogram of the power consumption at 2 000 ns.

collected traces. For example, we can see that from 400 collected traces, the accuracy is greater than 0.995.

For each possible key which leads to an accuracy of 0.995, the attack template is always achieved with the secure implementation against SPA from 390 traces. We have the same results when we collect 1 000 traces. The probabilities $pr(k^{(i)}|T)$ always indicate the correct key used by the cryptographic device for a number of traces from 390 to 1 000.

Below, we will attempt a template attack with a secure implementation against DPA, based on a random binary signed-digit recoding of the scalar k. This countermeasure allows us to add randomness on the scalar for the scalar multiplication. To have a bigger probability to guess the correct key, we will build templates with 1 000 traces for each possible key.

3.3.3 Template attacks with a DPA Countermeasure

Randomizing the scalar k in the scalar multiplication $[k]P$ can be used as DPA countermeasure. The first countermeasure of Coron [35], which randomizes the scalar k in $\big(k + r\#E(\mathbb{F}_p)\big)$ where r is a random number, is not applied in this section. Template attacks can be efficient when one applies this countermeasure. Indeed, the lattice reduction method allows to guess the private key by using the property $k' \equiv k \bmod N$. The only effect of such a countermeasure is to increase the length of the system to solve [94].

In a non-redundant number system, all numbers can be represented in a unique way. A redundant number system allows multiple representations of some numbers, thus the name redundant. We propose an on-the-fly random recoding of the scalar digits k_i using a signed-digit (SD) representation. The redundancy of this representation allows to randomly choose among several representations of the key digits k_i. Then the number and order of operations at the curve level (point additions and point doublings) are randomized with a basic scalar multiplication algorithm. This may protect from some power or electromagnetic radiation attacks. Whereas redundancy of signed-digit number systems can be interesting in their carry-free addition property,

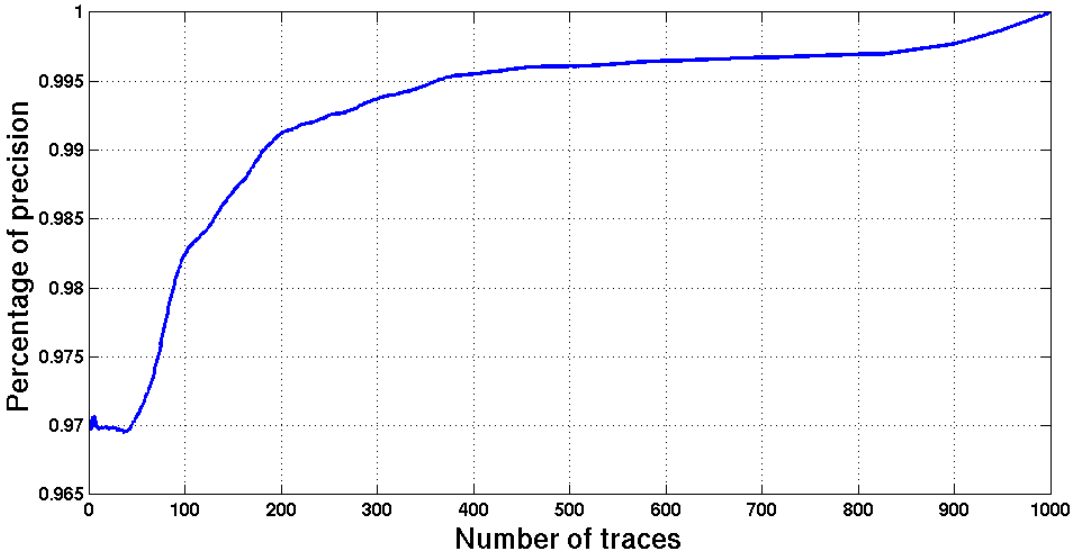


Figure 3.10: Evolution of the accuracy of the average obtained in function of the number of collected traces.

we just use this system for its redundancy, and change the number representation on-the-fly.

This countermeasure has been studied above in the section 2.3.3. The aforementioned section provides a unit which recodes on-the-fly an integer into a randomized signed-digit representation. The recoding unit was highly tested and integrated into the elliptic curve cryptosystem.

In this section, we perform a template attack on an implementation which includes the aforementioned recoding unit which provides a countermeasure against some side-channel attacks. The obstacle that comes into play in such an implementation is that the internal variables of the computation are randomizing.

input: $P \in E(\mathbb{F}_p)$ and $k = (k_{n-1} \cdots k_1 k_0)_{2SD}$ with $k_i \in \{\bar{1}, 0, 1\}$
output: $Q = [k]P$

1: $Q \leftarrow \mathcal{O}$
2: **for** i **from** $n-1$ **downto** 0 **do**
3: $\quad Q \leftarrow [2]Q$
4: $\quad$ **if** $k_i \neq 0$ **then**
5: $\quad\quad Q \leftarrow Q + k_i P$
6: $\quad$ **else**
7: $\quad\quad Q' \leftarrow Q + P$
8: **return** Q

Figure 3.11: Double-and-add-always applied with signed-digits for scalar multiplication.

Let the secret key $k = (k_{n-1} \cdots k_1 k_0)_2$. The scalar k, represented in binary representation, is randomly recoded on-the-fly in a signed-digit representation $k = (k_n \cdots k_1 k_0)_{2SD}$ with $k_i \in \{\bar{1}, 0, 1\}$. The recoding unit is based on the algorithm in figure 2.4, and is used for each scalar multiplication. The chosen scalar multiplication is presented in figure 3.11, and prevents against simple SCAs. Indeed, this algorithm always performs a point doubling and then a point addition whatever the processed bit. The computation cost is on average $n\,\texttt{ADD} + n\,\texttt{DBL}$. Whereas this

algorithm is resistant against simple side-channel attacks, fault attacks become possible. Indeed, it introduces dummy operations: an `ADD` is always performed but the output of the `ADD` may not be used by the algorithm.

Like in the previous template attack in section 3.3, we always focus on three bits evaluated by the scalar multiplication algorithm (in figure 3.11). The three bits starting most significant digit first are recoded according to the fourth digit 0, 1 or $\bar{1}$ (see algorithm in figure 2.4). Indeed, the fourth digit can be recoded before the third in 0, 1 or $\bar{1}$. Moreover, the third digit can change according to the fourth digit. That is why one needs to consider the fourth digit (knowing that we are most significant digit first).

the three initial bits of the key	the 4th digit is 0 $***0$	the 4th digit is 1 $***1$	the 4th digit is $\bar{1}$ $***\bar{1}$	number of possibilities
000		$001, 01\bar{1}, 1\bar{1}\bar{1}$	$00\bar{1}, 0\bar{1}1, \bar{1}11$	7
001	$01\bar{1}, 1\bar{1}\bar{1}$	$010, 1\bar{1}0$	000	6
010	$1\bar{1}0$	$011, 1\bar{1}1, 10\bar{1}$	$01\bar{1}, 001, 1\bar{1}\bar{1}$	8
011	$1\bar{1}1, 10\bar{1}$	100	$010, 1\bar{1}0$	6
100		$101, 11\bar{1}$	$10\bar{1}, 1\bar{1}1, 011$	6
101	$11\bar{1}$	110	100	4
110		111	$11\bar{1}, 101$	4
111			110	2

Table 3.2: Recoding possibilities for 3 initial bits.

Table 3.2 shows all possible recodings according to the three initial bits. For example, the key $(001)_2$ has six possible recodings according to the fourth digit which can be ever recoded by the recoding unit. In addition, some recodings are in several key possibilities. For example, the initial sequences $(010)_2$ and $(011)_2$ can be recoded in $(1\bar{1}1)$. The two recoded digits are bold font:
0101 $\longrightarrow$ $1\bar{1}$**01** $\longrightarrow$ $\underline{1\bar{1}}\underline{1\bar{1}}$,
0110 $\longrightarrow$ $\underline{1\bar{1}}$0.

The final recoding can be the same for different initial sequences. Therefore, for different key recodings, traces will match to themselves because the recoding will be the same for different keys. A template will have exactly the same behaviour for several keys. We can feel that a template attack will be very difficult to perform on such an implementation. One can think that it is sufficient to consider four bits instead of three. However in this case, the problem will be the same. Indeed, if we consider four bits, the number of possibilities will increase, and we will have to consider the fifth digit. Thus, some recodings will be in several key possibilities. In this case, the complexity of a template attack increases more and more. In this attack, we try to guess three digits of the secret key. Below in section 3.4.4, we discuss on the number of randomly recoded digits.

FPGA Implementation

The binary signed-digit recoding unit was added to the cryptoprocessor. This unit provides a random representation of the scalar k digit by digit according to a random number. Figure 3.12 presents the architecture of the cryptoprocessor (for curves over $\mathbb{F}_p$, $n = 192$ and Jacobian

coordinates) with the binary signed-digit unit. The controller launches the corresponding curve-level operation according to a digit of $k_i \in \{-1, 0, 1\}$ on 2 bits provided by the binary signed-digit unit.

The randomness was provided by a pseudo random sequence generator with *linear feedback shift registers* (LFSR). The period of the sequence was $2^{14} - 1 = 16\,383$. The period is large enough for the application. Indeed, templates are built with 1 000 traces. Thus, the provided randomness is sufficient for randomly recoding a number in signed digits. The primitive polynomial used was $x^{14} + x^{12} + x^{11} + x + 1$.

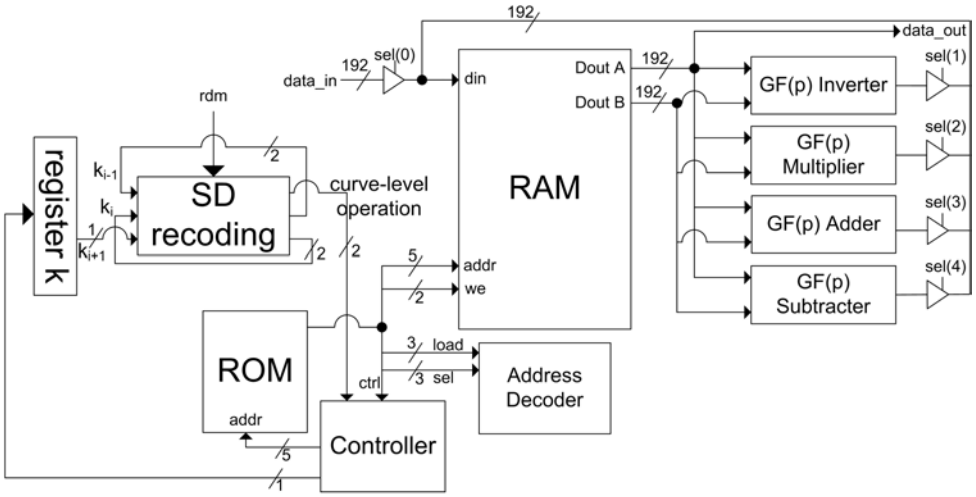


Figure 3.12: Architecture of the cryptoprocessor with the binary signed-digit recoding unit (with the NIST curve P-192).

Table 3.3 reports FPGA implementations of the ECC processor (for curves over $\mathbb{F}_p$, $n = 192$ bits and Jacobian coordinates). The implementation uses double-and-add-always method with one arithmetic unit per field operation. Table 3.3 presents implementation results with and without the key recoding unit.

SD random recoding unit	memory type	area		freq.
		slices (FF/LUT)	BRAM	MHz
with	distributed	2 808 (2 732/9 066)	0	150
	BRAM	2 313 (2 260/8 855)	6	150
without	distributed	2 550 (2 484/8 507)	0	150
	BRAM	2 084 (2 073/7 985)	6	150

Table 3.3: FPGA implementation results for the complete ECC processor over $\mathbb{F}_p$ with $n = 192$ bits.

We can see in table 3.3 that the version where the secret key is randomly recoded on-the-fly can work at the same frequency as the implementation without this countermeasure. However the protected version needs more registers and LUTs than the unprotected one (about 10% more slices). In addition, it is not necessary to wait for the end of the computation of the key recoding unit to use it. Recoding can be performed in parallel to point operations. The combination of recoding and scalar multiplication algorithms allows to overlap recoding steps by curve-level

operations. When the digit k_i is used, the next digit is already recoded, or will be recoded before the cryptosystem needs to use it. Therefore there is no latency during the scalar multiplication. Indeed, recoding a digit requires few clock cycles, whereas a curve-level operation requires hundreds to thousands of clock cycles.

Two side-channel attacks will be performed in the next section. Like for the previous template attack, templates are built on the same principle. A large number of traces are measured for all possible keys. Then, we estimate the mean vector and the covariance matrix of the multivariate normal distribution according to the interesting points. In the second phase, we use the characterization with power traces from the device under attack to determine the secret key.

We know that the number of possible recodings is different according to initial sequences (table 3.2). One can think that this difference could allow to find information with a sequence which has a small number of possible recodings. That is why the first template attack is on a key which has a small number of possible recodings. The second template attack is an extension of the first one: we focus on six bits of the key. The first three bits evaluated are randomly recoded, while the next three bits are not. For this attack, templates are built for the first three bits, and then for the next three bits evaluated by the scalar multiplication algorithm.

Template attack on the DPA secure implementation

In this template attack, we want to guess three bits of the key. To see if the countermeasure is efficient, we will perform a template attack with the key $(111)_2$ which has the less possible recodings. Indeed, table 3.2 shows that the subkey value $(111)_2$ can be recoded by itself or by the sequence $(110)_2$. A unit recodes randomly the three bits on-the-fly using randomly chosen binary signed-digit representations, whereas there are only two possibilities. This template attack is performed with the same method as the previous attack. In the training phase, 8 templates are built with 1 000 traces for each possible key by measuring the electrical activity. Templates are built according to the interesting points which contain the most information, by calculating the sum of the mean trace difference. In addition, it allows to reduce the number of points of each trace. The attack phase consists in measuring the power consumption of the cryptographic system with the secret key during 100 scalar multiplications. The attacker should be able to extract the key of the attacked device in this profiling phase.

The probabilities $pr(k^{(i)}|t)$ are computed. They are based on the result of the template matching using Bayes' theorem to answer the following question: given a trace t, what is the probability that the secret key of the device is equal to $k^{(i)}$? Figure 3.13 shows these probabilities for each possible key. The top figure shows the probabilities $pr(k^{(i)}|t)$ for each 100 traces t and keys $k^{(i)}$. The bottom figure shows the mean and the standard deviation of probabilities $pr(k^{(i)}|t)$ for the collected traces corresponding to a key value.

By the top figure, we can see that the probabilities are distributed and dispersed between each possible key. For the 100 traces, different templates fit to a given trace. The probabilities $pr(k^{(i)}|t)$ are the highest for too many different keys, and so they indicate too many different templates. No information can be deduced from this figure, and neither when one computes the mean and the standard deviation for each possible key. Indeed, by the bottom figure, the mean of probabilities for the 100 traces are about 0.2 for three key values, and their standard deviations are about 1.75. The means and the standard deviations are too low to deduce any information. These values are also distributed between each possible key.

With these results, we cannot deduce the best guess for the key used by the device. Ex-

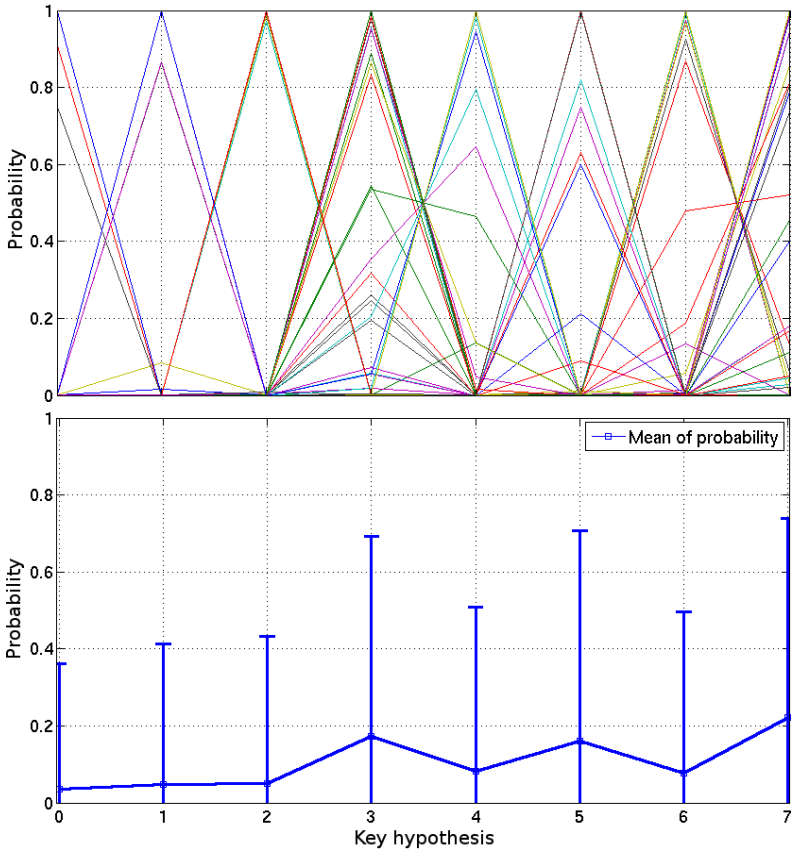


Figure 3.13: The probabilities $pr(k^{(i)}|t)$, mean and standard deviation of probabilities for each key hypothesis $k^{(i)}$.

tending this approach to determine the probability $pr(k^{(i)}|T)$ does not give either result on the key processed. Indeed, all probabilities $pr(k^{(i)}|T)$ are close to 0 for all key values. Thus, even with a key which has few possible recodings, this template model attack seems to be not practical.

Second template attack on the DPA secure implementation

In this template attack, we want to guess six bits of the key. The recoding unit recodes the three first bits on-the-fly using randomly chosen binary signed-digit representations, while the recoding unit is disabled for the next three bits. Two distinct template attacks are performed separately with the same approach. The fourth bit is read but is not modified by the recoding unit.

The first one deals with the first three recoded bits. In the training phase, 1 000 power traces are measured to characterize the device under attack. Therefore $2^3 = 8$ templates are built according to the three recoded bits and in particular according to their corresponding interesting points which contain the most information. After this profiling phase, the attacker should be able to extract the key of the attacked device.

The second one deals with the next three bits of the key. This sequence of bits does not change during the scalar multiplication, the random recoding unit being disabled. Therefore templates are built according to the eight possible keys, and seventeen possible input data, that is $8 \times 17 = 136$ templates. Indeed, the first three bits are randomly recoded and seventeen inter-

mediate values (see table 3.2) can be calculated by the scalar multiplication algorithm. The point Q used in the algorithm in figure 3.11 is not the same according to the first three bits, and so can have seventeen different values according to the first three recoded bits. Thus, templates are built with all possible keys ($2^3 = 8$) and the seventeen possibilities for the point Q. We wanted to know if an attacker could guess the correct key and to see the evolution of the probability density function of the multivariate normal distribution with a large number of templates. We can see the limitation of the attack for generating a large number of templates, i.e. 136. Indeed, weeks were necessary to measure the $136 \times 1\,000 = 136\,000$ traces. It would take too much time to measure so many traces to build templates for an attacker.

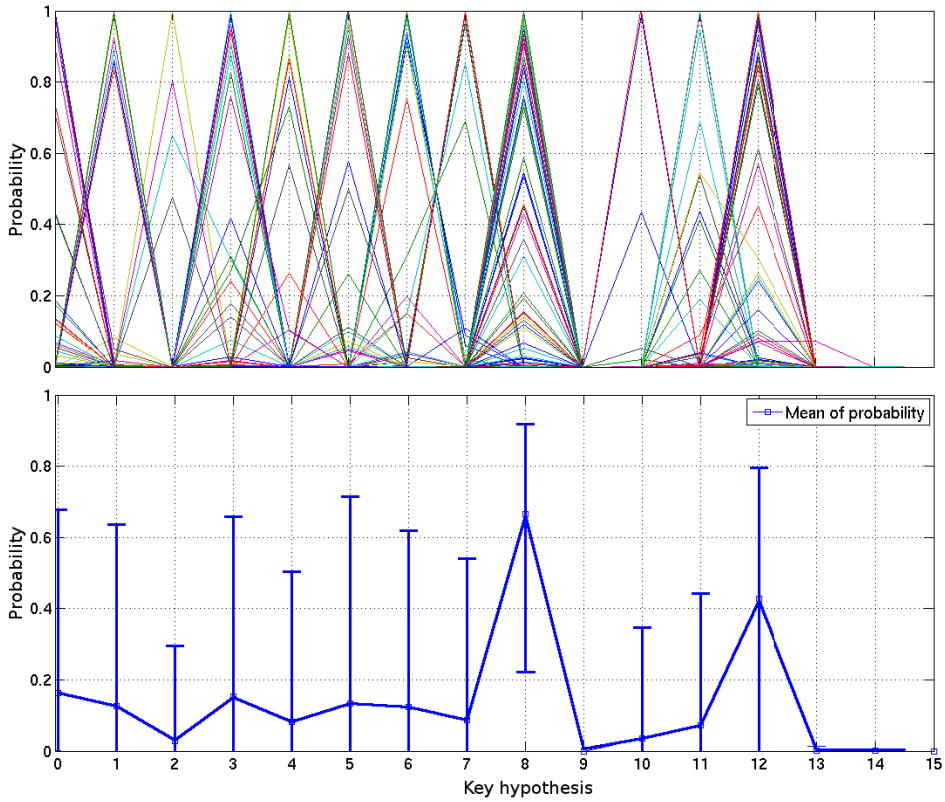


Figure 3.14: The probabilities $pr(k^{(i)}|t)$, mean and standard deviation of probabilities for each key hypothesis $k^{(i)}$ (the three first bits are randomly recoded on-the-fly).

The attack phase consists in measuring the power consumption of the device with a secret key. In this context, the probability distribution of the power consumption is computed for the 100 measured traces. Figure 3.14 shows the probability density function for the first three bits recoded on-the-fly and for the next three bits with templates built during the previous training phase. The top figure shows the probabilities $pr(k^{(i)}|t)$ for each trace t corresponding to each possible key. The bottom figure shows the mean and the standard deviation of probabilities $pr(k^{(i)}|t)$ for the collected traces corresponding to a key value.

For the first attack, keys are between $k^{(0)} = (000)_2 = 0$ and $k^{(7)} = (111)_2 = 7$. The key is recoded on-the-fly using randomly chosen binary signed-digit representations. Probabilities are distributed and dispersed between each possible key value. No templates seem to fit a given trace. In addition, the mean and standard deviation of probabilities $pr(k^{(i)}|t)$ for each 100 traces do not give any intuition about the correct key. There is not a majority of traces which a template

matches. The highest probabilities $pr(k^{(i)}|t)$ are shared between all possible keys. Thus, one cannot guess what key value is used for the first three bits.

The key hypothesis between 8 and 15 corresponds to the possible keys $(000, 001, \cdots, 111)$ for the next three bits. This piece of the key does not change for different scalar multiplications. Their probabilities $pr(k^{(i)}|t)$ are computed with each template. Figure 3.14 shows these probabilities for each key hypothesis and one input data. Indeed, there are 186 templates, and each key hypothesis corresponds to seventeen point inputs. Two key guesses seem to be the correct key. The majority of the 100 traces collected are higher for $k^{(i)} = 8$ and $k^{(i)} = 12$. In addition, the mean and the standard deviation of probabilities for these two keys confirm this impress. The key value $k^{(i)} = 8$ has a mean of 0.67 with a standard deviation about 1.6. The key value $k^{(i)} = 12$ has a mean of 0.42 with a standard deviation about 1.8. Thus two keys distinguish themselves, but the key $k^{(i)} = 8$ seems to be the correct one. For all other templates corresponding to the other input points, the probability $pr(k^{(i)}|t)$ is always between 0 and 0.3. The templates based on these input points and all key values do not match the traces.

Figure 3.15 shows how the probability $pr(k^{(i)}|t)$ evolves in the first attack as a function of the number of traces for each possible key. Each template is associated with a key, and the probabilities $pr(k^{(i)}|t)$ measure how well the templates fit to a given trace. With these keys recoded on-the-fly, one template fits to a given trace, and one fits to another. The probabilities $pr(k^{(i)}|t)$ are shared between each possible key.

In addition, the probability $pr(k^{(i)}|T)$ where T is the set of the 100 traces does not give either information. Indeed, for each key hypothesis, this probability is close to 0. This result is logical because for all possible keys and collected traces, the probability $pr(k^{(i)}|t)$ is 0 for several values of $k^{(i)}$ and traces t. This template attack performed on such an implementation does not succeed with a key recoded using randomly chosen binary signed-digit representations.

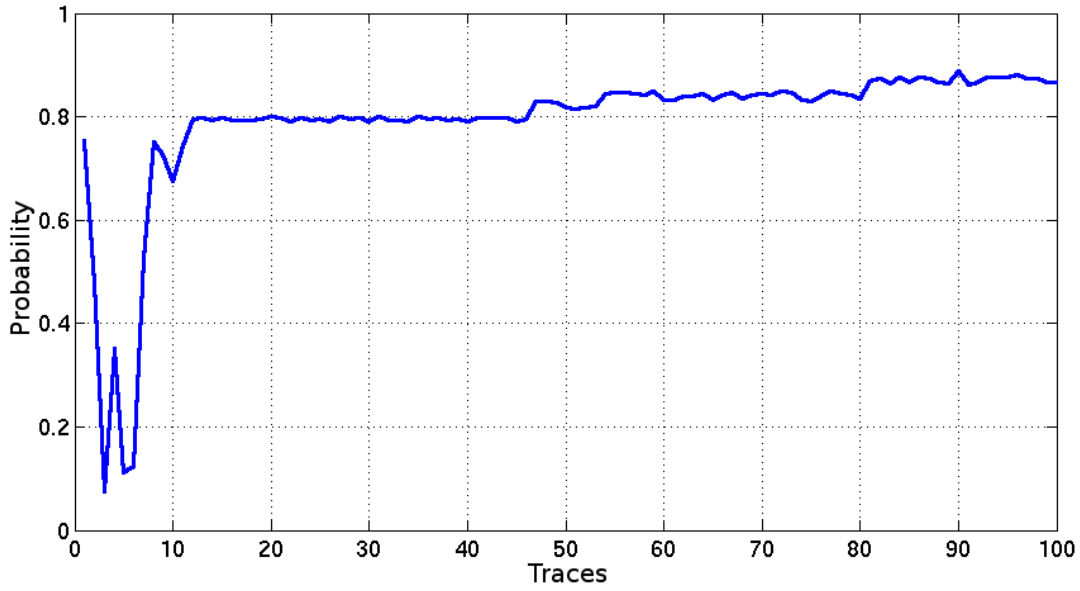


Figure 3.16: Evolution of the probabilities $pr(k^{(i)}|T)$ with an increasing number of traces for $k^{(i)} = 8$.

Figure 3.16 shows how the probability $pr(k^{(i)}|T)$ evolves as a function of the traces for $k^{(i)} = 8$ (i.e. bits $(000)_2$ for the next three bits) and one input data. This key guessed leads to almost

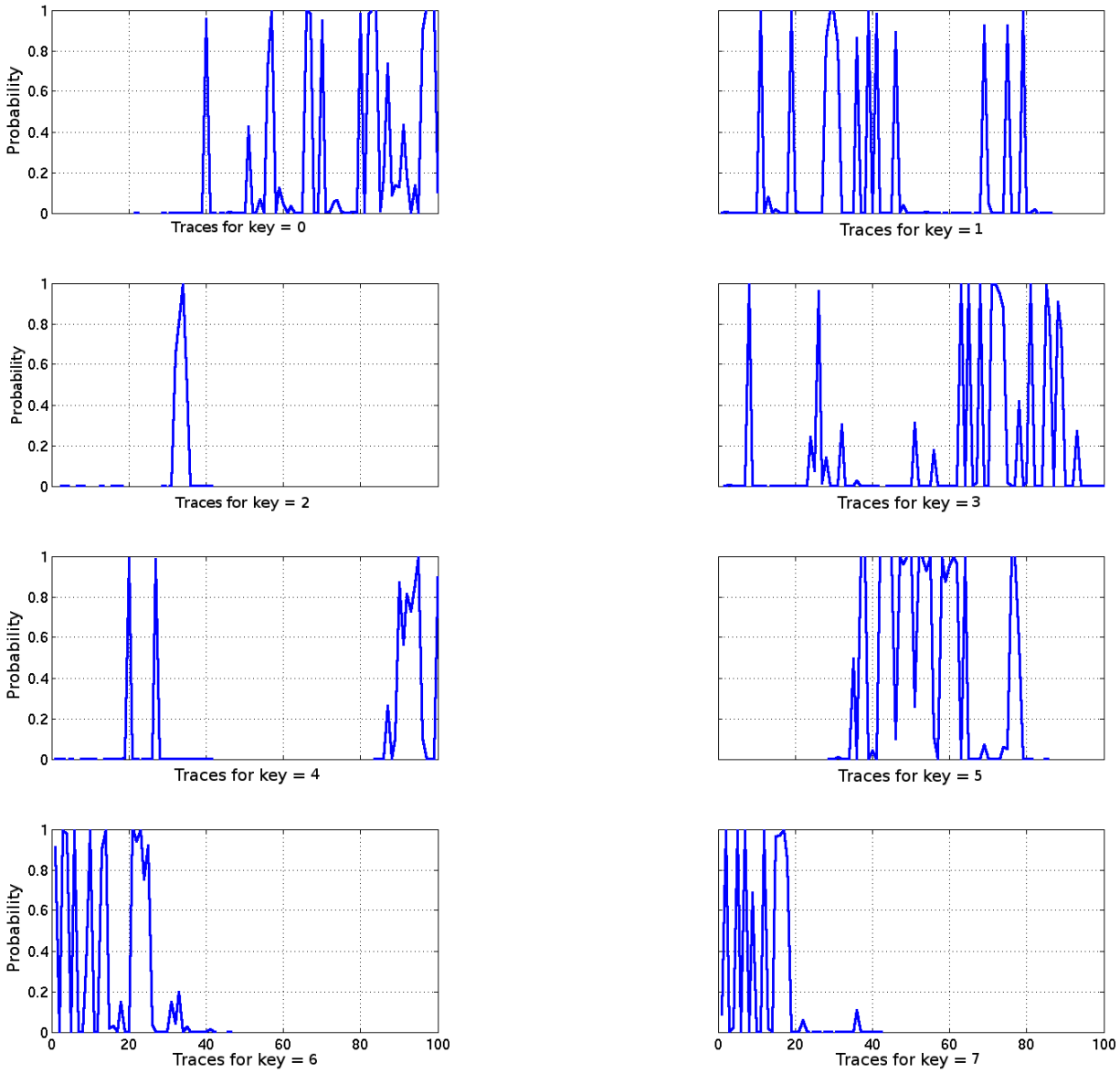


Figure 3.15: The probabilities $pr(k^{(i)}|t)$ for some traces t.

0.8 after 12 traces, and to almost 0.88 after 81 traces. The key $k^{(i)} = 8$ is the correct one. The second key, $k^{(i)} = 12$, which could be the correct key, has a probability $pr(k^{(i)}|T)$ close to 0 for all input data, like for all the other key possibilities. The mean and standard deviation of probabilities $pr(k^{(i)}|t)$ for $k^{(i)} = 12$ was lower than for the key $k^{(i)} = 8$. In addition, we can see in figure 3.14 that the number of higher probabilities $pr(k^{(i)}|t)$ is superior for $k^{(i)} = 8$ than for $k^{(i)} = 12$. These results confirm that the key $k^{(i)} = 12$ is not the correct key value.

This second template attack with a DPA countermeasure does not succeed to guess the first part of the randomly recoded key. For a key which is the same for all scalar multiplications, the template attack allows to find the correct key. In this second attack, templates have been built for different pairs of key guesses and input points. In particular, all possible keys have been associated with all possible input points, and one template fits to the given traces. It means that we guessed the three bits of the key $(k_{188}, k_{187}, k_{186})$ even if the number of templates to generate

increases consequently. In addition, we also know the corresponding point used by the three next bits, and so the key value which has generated this input point. In this attack, the template which matches the correct key guess corresponds to a certain input point Q. This input point fits to the computation of the key value $(1\bar{1}1)$. Thus, the first three bits of the key can be (010), (011), or (100). However we do not have any more information to deduce the correct first three bits among these three key possibilities.

3.4 Number of Recoded Digits for an Attack

Below, we consider the weight of a recoding, its number of digits different from zero. For instance, the recoding $(10\bar{1})$ has a weight of 2.

3.4.1 Weight of Recodings

In the previous section, table 3.2 presents all possible recodings for three bits. This table shows that a sequence of bits can have several recoding possibilities. In addition, recodings can have the same weight. Thus, a sequence which can be recoded as $(1\bar{1}1)$ and $(11\bar{1})$ has two possible recodings. However the digit $\bar{1}$ can be viewed like the digit 1. Indeed, most of scalar multiplication algorithms scan the key k digit by digit. When $k_i = \bar{1}$, a point subtraction is calculated. When $k_i = 1$, a point addition is calculated. An implementation of these computations can be calculated so that a point addition and a point subtraction are indistinguishable [57, p. 98].

Thus, the sequence of operations that leads to a point addition or a point subtraction can be similar. Indeed, the opposite of a point can be calculated in a straightforward way. For example, in affine coordinate, opposite of the point (x, y) is the point $(x, -y)$. In projective and Jacobian coordinates, the opposite point $(X : Y : Z)$ is $(X : -Y : Z)$. Thus, either one computes the opposite of a point and then computes a point addition, or one computes a point subtraction according to the formula of a point addition by integrating the opposite point.

In practice, one can consider two recodings with the same weight like one recoding. For instance, (101) and $(10\bar{1})$ can be considered equivalents, because `ADD` and `SUB` can be considered equivalents. However, the number of possible recodings is not the same as previously in the table 3.2. This property, combined with the use of a redundant system increases the security of the countermeasure: the number of possible recodings is closer to the total number of recodings, whereas the number of possible recodings decreases.

3.4.2 Recoding Possibilities of Initial Bits

Tables 3.4, 3.5 and 3.6 show all possible recodings for one, two and three initial bits, respectively. We can see their number of possibilities, denoted by "# poss", with and without the hypothesis $\bar{1} \Leftrightarrow 1$, that is the digit $\bar{1}$ is considered such the digit 1.

the first initial bit of the key	the 2nd digit is 0 $*0$	the 2nd digit is 1 $*1$	the 2nd digit is $\bar{1}$ $*\bar{1}$	# poss	# poss with $\bar{1} \Leftrightarrow 1$
0		1	$\bar{1}$	3	2
1			0	2	2

Table 3.4: Recoding possibilities for one bit.

the two initial bits of the key	the 3rd digit is 0 $**0$	the 3rd digit is 1 $**1$	the 3rd digit is $\bar{1}$ $**\bar{1}$	# poss	# poss with $\bar{1} \Leftrightarrow 1$
00		$01, 1\bar{1}$	$0\bar{1}, \bar{1}1$	5	3
01	$1\bar{1}$	10	00	4	4
10		11	$1\bar{1}, 01$	4	3
11			10	2	2

Table 3.5: Recoding possibilities for two bits.

the three initial bits of the key	the 4th digit is 0 $***0$	the 4th digit is 1 $***1$	the 4th digit is $\bar{1}$ $***\bar{1}$	# poss	# poss with $\bar{1} \Leftrightarrow 1$
000		$001, 01\bar{1}, 1\bar{1}\bar{1}$	$00\bar{1}, 0\bar{1}1, \bar{1}11$	7	4
001	$01\bar{1}, 1\bar{1}\bar{1}$	$010, 1\bar{1}0$	000	6	6
010	$1\bar{1}0$	$011, 1\bar{1}1, 10\bar{1}$	$01\bar{1}, 001, 1\bar{1}\bar{1}$	8	6
011	$1\bar{1}1, 10\bar{1}$	100	$010, 1\bar{1}0$	6	6
100		$101, 11\bar{1}$	$10\bar{1}, 1\bar{1}1, 011$	6	4
101	$11\bar{1}$	110	100	4	4
110		111	$11\bar{1}, 101$	4	3
111			110	2	2

Table 3.6: Recoding possibilities for three bits.

When one considers that the digit $\bar{1}$ is equivalent to the digit 1, the number of possible recodings decreases. Indeed, the sequences $(10\bar{1})$ and (101) are considered as one possible recoding for the initial bits (100) with the hypothesis $1 \Leftrightarrow \bar{1}$.

However according to the total number of possible recodings, the number of recodings for an initial sequence is higher when one considers $1 \Leftrightarrow \bar{1}$. Indeed, the total number of possible recodings for 1, 2 and 3 bits are 3, 9 and 27, respectively (see section 2.3.2 for the exact number of binary signed-digit representations). When one considers $1 \Leftrightarrow \bar{1}$, the total number of possible recodings for 1, 2 and 3 bits are 2, 4 and 8, respectively (for n bits, there are 2^n possible recodings). In this way, 0 and 1 can be recoded in each possible recoding (in table 3.4). Otherwise, $(1 \not\Leftrightarrow \bar{1})$, 1 cannot be recoded in $\bar{1}$.

In addition, the interval of the number of possibilities is shrunk. Indeed, for three initial bits, the interval of the number of possibilities is between 2 and 6 when one considers $\bar{1} \Leftrightarrow 1$. Otherwise, the interval is between 2 and 8. For two initial bits, the number of possibilities of recoding is between 2 and 5 over 9 possibilities when one considers that $\bar{1}$ and 1 are processed separately. Otherwise, the number of recoding possibilities is between 2 and 4 over 4 possibilities.

Thus, considering that digit $\bar{1}$ is equivalent to digit 1 increases the security of the countermeasure. Indeed, more initial sequences can be recoded in a same sequence. For example, the sequence $(1\bar{1}1)$ is one of the recodings for (010) and (011). However all initial sequences can be recoded in a sequence which has the same weight as $(1\bar{1}1)$. Therefore in this case, all values will match themselves because the recoding will be the same for all different keys. By extending this principle, a template will have exactly the same behaviour for several keys.

3.4.3 Antecedents of Recodings

Tables 3.7, 3.8 and 3.9 show all possible antecedents for each possible recoding. Recodings with a same weight are grouped to see the number of different antecedents.

all possible recodings	possible antecedents	number of antecedents	number of different antecedents
0	0, 1	2	} 2
1	0, 1	2	} 2
$\bar{1}$	0	1	

Table 3.7: Antecedents for all possible recodings for one initial bit.

all possible recodings	possible antecedents	number of antecedents	number of different antecedents
00	00, 01	2	} 2
01	00, 01, 10	3	} 3
$0\bar{1}$	00	1	
10	01, 10, 11	3	} 3
$\bar{1}0$		0	
11	10, 11	2	} 4
$1\bar{1}$	00, 01, 10	3	
$\bar{1}1$	00	1	
$\bar{1}\bar{1}$		0	

Table 3.8: Antecedents for all possible recodings for two initial bits.

According to table 3.6, one could think that the recoding possibilities for the three initial bits (111) is weak because only two recodings are possible: (110) and (111). However we can see that these recodings have 6 and 8 different antecedents, respectively. In other words, these recodings match 6 and 8 different templates. Thus, the security of this countermeasure is not only related to the number of possible recodings, but also to the number of possible antecedents for each possible recoding.

For example, let us consider the recoding possibilities for the two initial bits (10). This sequence can be recoded to (01), (10) and (11). It can be also recoded to ($1\bar{1}$), but this recoding is always considered by the sequence (11). The number of antecedents for each recoding is 2, 3 and 4, respectively. Therefore, when the sequence (10) does not change by the recoding, three traces out of four have the same behaviour. For each possible recoding, each sequence has the same behaviour as another.

3.4.4 Evaluation of the Number of Recoded Digits

The presented template attacks are on 3 bits of the key. Such an attack did not succeed with a protection based on a signed-digit representation of the key. A question could be, what

all possible recodings	possible antecedents	number of antecedents	number of different antecedents
000	000, 001	2	} 2
001	000, 001, 010	3	} 3
00$\bar{1}$	000	1	
010	001, 010, 011	3	} 3
0$\bar{1}$0		0	
011	010, 011, 100	3	} 5
01$\bar{1}$	000, 001	2	
0$\bar{1}$1	000, 010	2	
0$\bar{1}\bar{1}$		0	
100	011, 100, 101	3	} 3
$\bar{1}$00		0	
101	100, 101, 110	3	} 5
10$\bar{1}$	010, 011, 100	3	
$\bar{1}$01		0	
$\bar{1}$0$\bar{1}$		0	
110	101, 110, 111	3	} 6
1$\bar{1}$0	001, 010, 011	3	
$\bar{1}$10		0	
$\bar{1}\bar{1}$0		0	
111	110, 111	2	} 8
11$\bar{1}$	100, 101, 110	3	
1$\bar{1}$1	010, 011, 100	3	
1$\bar{1}\bar{1}$	000, 001, 010	3	
$\bar{1}$11	000	1	
$\bar{1}$1$\bar{1}$		0	
$\bar{1}\bar{1}$1		0	
$\bar{1}\bar{1}\bar{1}$		0	

Table 3.9: Antecedents for all possible recodings for three initial bits.

happens if one considers 1, 2, 4 or more bits of the key to build templates? Does the template attack work with the countermeasure?

First of all, when one considers 4 or more bits instead of 3, the attacker must build more templates: $2^4 = 16$ templates for four bits of the key, that is 2^n templates for n bits of the key.

Thus, the complexity of performing a template attack becomes more impractical when increasing the number of considered bits of the key. In addition, the number of recoding possibilities will increase consequently. Some values will be in several key possibilities, and the complexity of a template attack will increase more and more. In addition, the number of different antecedents for all possible recodings increases more and more when one considers 4 or more bits.

When one wants to consider 1 or 2 bits of the key to build templates, the number of possible recodings is of course less than when considering 3 bits of the key. However the number of possible recodings is closer to the total number of recodings. The number of different antecedents for each possible recoding is also closer to the total number of antecedents.

Particularly, when one considers 1 bit of the key to build templates, digits 0 and 1 can be recoded in each possible recoding when considering that $\bar{1}$ is equivalent to 1. In addition, digits 0 and 1 can be the antecedents of all possible recodings.

When one considers 2 bits of the key to build templates with the hypothesis that digit $\bar{1}$ is equivalent to digit 1, the number of possible recodings is between 2 and 4, over 4 possible recodings. When one considers 3 bits of the key, the interval is between 2 and 6 over 8 possible recodings. Therefore, there are more possible recodings for 2 bits than for 3 according to the total number of possible recodings. In the same way, there are more antecedents for all possible recodings for 2 bits than for 3 according to the total number of possible antecedents.

For building templates, an attacker cannot choose a too large number of bits if he/she wants to build templates for each possible key. From 7 bits, the attack can be very difficult to perform due to the too large number of templates to build. In addition, by choosing few bits of the key, traces match with more traces according to the number of possible recodings.

3.5 Conclusion

In this chapter, template attacks were performed to guess a part of the secret key. Templates were built to characterize a cryptographic device. The device computes a scalar multiplication $[k]P$ where k represents the secret key (i.e. the scalar) and P a fixed point of an elliptic curve. A template consists of a mean vector and a covariance matrix. These values were estimated by the power consumption measured for different keys. The goal of template attacks is to guess a part of the key k used.

This strategy for building templates was used to perform template attacks. First of all, a template attack was performed on a secure implementation against simple SCAs using a typical countermeasure. The template attack succeeds to guess three bits of the key used by the cryptographic algorithm executed by the device. Then, a template attack was performed on a secure implementation against differential SCAs. The countermeasure uses a redundant number system which allows to randomly choose among several representations of the key digits. Thus, the key k was randomly recoded on-the-fly and had different representations for different scalar multiplications. The performed template attack does not succeed to guess a part of the key with such a countermeasure.

We know that the number of key recodings depends on the key value. Thus the recoding of a key is not equiprobable according to the other key recodings. An attacker could use this observation to deduce some information about the key. An attacker could know that a key has for instance 6 possible recodings, and so deduces that this key belongs to a key subset.

Another solution to perform and succeed in doing a template attack might be to build templates of templates. Indeed, for each key value several recodings are possible, and one needs to separate each possible recoding of the templates built for one key. Thus templates would be not built for each possible key, but for each possible recoding of the key. For example, 4 templates must be built for the key value (110) when one wants to guess 3 bits of the key. One would compute the probability that a key matches a subset of templates. Thus, instead of building 2^n templates, one can build 3^n templates for each digit possibility if k_i can take three different digits. In other words, one can build templates for all key hypothesis.

The complexity of building these templates comes from the fact that an attacker must choose and block the recoding of each key. In addition, if an attacker can choose any recoding, the number of traces acquisitions increases consequently because the number of recodings is much higher than a system without random recoding. For example, for templates built for 3 digits of the key, the attacker must build 43 templates instead of $2^3 = 8$ which is the number of recodings. This new number of templates is much higher according to the initial number of templates to build, and can be an obstacle for an attacker. Indeed, it may take too much time to measure so many traces to build the required templates.

Chapter 4

On-the-Fly Multi-Base Recoding

In ECC, many works concern security and speed of the main operation, the scalar multiplication. Scalar representation significantly impacts the number of point operations to be executed and the overall computation time. Consequently scalar recoding methods are very popular: non-adjacent forms (NAF and `wNAF`), double- or multi-base number systems (DBNS/MBNS), etc. Previous fast recoding methods require a pre-computation step prior to scalar multiplication. For `wNAF`, several multiples of P must be precomputed and stored. For DBNS/MBNS, the scalar is generally recoded off-line. In addition, previous DBNS and MBNS recoding methods do not provide any hardware implementation.

In this chapter, we provide an on-the-fly hardware implementation of a multi-base recoding method for ECC scalar multiplication. The three first sections are made up of two conference articles published in ARITH [26] and SympA [11] 2013. However this chapter provides more explanations, methods and hardware implementations.

In section 4.1/4.2, we present a method and its FPGA implementation to recode, on-the-fly, the scalar using MBNS without pre-computation. Our recoding is performed in parallel to curve-level operations (addition, doubling, tripling, etc). The recoding method uses very cheap divisibility tests for each base element (e.g. $2, 3, 5, 7$) and an efficient implementation of exact division algorithms used for multiple-precision arithmetic. Exact division refers to division where the remainder is known to be zero.

We only deal here with elliptic curves defined over $\mathbb{F}_p$ but our method can be easily applied in the $\mathbb{F}_{2^m}$ case (fine tuning is slightly different due to different cost ratios `I/M` and `S/M`). Section 4.1 presents a simple unsigned version of our method while section 4.2 presents optimizations using signed-digit multi-base representation. Section 4.3 compares our results to state-of-art ones. Section 4.4 evaluates our method with pre-computed points.

4.1 Proposed Muti-Base Recoding and Scalar Multiplication in ECC

The notations used in this chapter are:

- $k = (k_{n-1}k_{n-2}\dots k_1k_0)_2$ is the n-bit *scalar* ($k > 1$) stored into a t words by w bits memory with $w(t-1) < n \leq wt$ (i.e. last word may be padded using 0s). $k^{(i)}$ denotes the ith word of k starting from least significant for $0 \leq i < t$,
- $\mathcal{B}$ is the *multi-base* with l *elements* (co-prime integers), $\mathcal{B} = (b_1, b_2, \dots, b_l)$ under conditions $b_1 < b_2 < \dots < b_l$,

- predicate divisible$(x, \mathcal{B})$ returns true if x is divisible by at least one base element in $\mathcal{B}$ (false for the other case),
- number x is represented as the *sum of terms* $x = \sum_{i=1}^{n'} \left(d_i \prod_{j=1}^{l} b_j^{e_{j,i}} \right)$ with $d_i = \pm 1$,
- *term* $(d_i, e_{1,i}, e_{2,i}, \ldots, e_{l,i})$ is defined by $d_i \times \prod_{j=1}^{l} b_j^{e_{j,i}}$ in the multi-base $\mathcal{B}$ (index i may be omitted when context is clear),
- Q, P are points on the curve, the scalar multiplication computes $Q = [k]P$.

In practice in the circuit, a large integer is represented and stored in multi-precision into a t words vector by w bits. The storage of k in multi-precision is illustrated at the architecture level in figure 4.1. The input of the memory block is the address of the jth word of k. The output on the read port is the content of the word $k^{(j)}$. In our hardware implementations, this memory block will be implemented in LUT (*look-up table*) to measure the impact of n and w in relation to the circuit area. Obviously, the presented method can be applied to implementations with dedicated memory blocks (e.g. BRAM in Xilinx FPGAs).

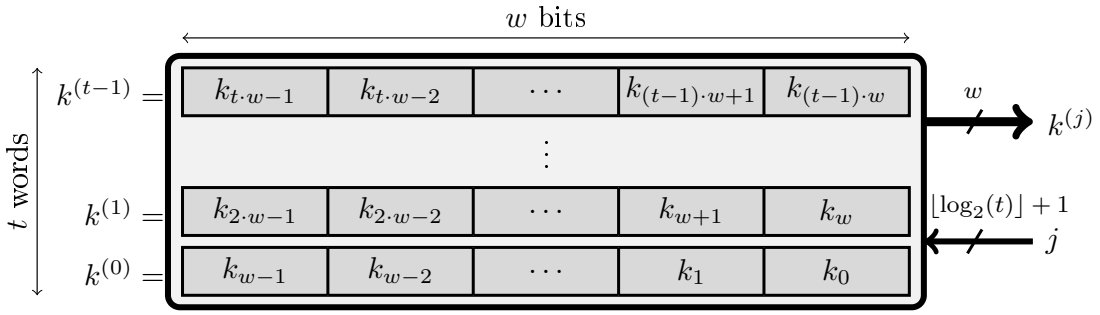


Figure 4.1: Storage of k into t words by w bits.

In this section, we present a simple unsigned version ($d_i = 1$) of the method for the sake of simplicity. This allows us to provide simple explanations of architecture units and scheduling aspects. Section 4.2 details more efficient versions based on signed representation ($d_i = \pm 1$). All architecture units described in this section can be directly used or slightly adapted for signed representation.

4.1.1 Unsigned Algorithms

The MBNS unsigned recoding algorithm is presented in figure 4.2. Its principle is simple. Divisibility of k by $\mathcal{B}$ elements is tested. When k is not divisible, 1 is subtracted to k. In practice, $b_1 = 2$ will be selected for efficiency purpose (divisibility is ensured 50% of time). This means that $k-1$ will be divisible by 2 (if $b_1 > 2$, it still works but with more iterations). For lines 8–12, the scalar k is divided by all base elements b_j in $\mathcal{B}$ as much as possible. To this end, we use cheap divisibility tests and exact divisions when divisibility is ensured. This division step provides the term exponents $e_1, e_2, \ldots, e_l$. LT denotes the list of terms which stores the MBNS recoding of k, LT $= \big((d_1, e_{1,1}, e_{2,1}, \ldots, e_{l,1}), (d_2, e_{1,2}, e_{2,2}, \ldots, e_{l,2}), \ldots\big)$ with $d_i \in \{0, 1\}$. Only the first term may have $d_1 = 0$ (if the initial k is immediately divisible by one base element). Divisibility tests at line 3 and 10 can be shared. The algorithm stops when $k \leq 1$ due to Horner form such as $2^a 3^b 5^c \big(1 + 2^{a'} 3^{b'} 5^{c'} (1)\big)$.

Such a strategy is not a new one: similar methods are discussed in [1], [2], [29], [38], [39], [76], [77] and [78]. In particular, these references deal with the conversion of an integer in MBNS, and some of them extend to windowed multi-base chains (MBNS is a generalization of DBNS).

input: a positive integer k, and the multi-base $\mathcal{B} = (b_1, b_2, \ldots, b_l)$
output: list of terms LT which stores the MBNS recoding of k
$\mathrm{LT} = \big((d_1, e_{1,1}, e_{2,1}, \ldots, e_{l,1}), (d_2, e_{1,2}, e_{2,2}, \ldots, e_{l,2}), \ldots\big)$

1: $\mathrm{LT} \leftarrow \emptyset$
2: **while** $k > 1$ **do**
3: **if** not$\big($ divisible$(k, \mathcal{B})\big)$ **then** (*divisibility test*)
4: $d \leftarrow 1$
5: $k \leftarrow k - 1$
6: **else**
7: $d \leftarrow 0$
8: **for** j **from** 1 **to** l **do**
9: $e_j \leftarrow 0$
10: **while** $k \equiv 0 \bmod b_j$ **do** (*divisibility test*)
11: $e_j \leftarrow e_j + 1$
12: $k \leftarrow k/b_j$ (*exact division*)
13: $\mathrm{LT} \leftarrow \mathrm{LT} \cup (d, e_1, e_2, \ldots, e_l)$
14: **return** LT

Figure 4.2: MBNS recoding algorithm (unsigned version).

Divisibility tests are detailed in section 4.1.2. They are implemented using $t + \varepsilon$ clock cycles (ε is a small constant) for all $b_j \neq 2$ and only one for $b_j = 2^s$ with $s \leq w$ (parameter s will be explained latter). Once k is divisible by b_j, we use fast exact division algorithms to perform $k \leftarrow k/b_j$ with one dedicated optimized algorithm for each base element. Exact divisions are detailed in section 4.1.3. The computation time for each exact division is $t + \varepsilon'$ clock cycles for all $b_j \neq 2$. In case of division by 2^s we only use shifts.

MBNS recoding algorithm in figure 4.2 works in a *serial way*: one multi-base term at a time and starting with the least significant one of k. Each term can be immediately used in the scalar multiplication algorithm in figure 4.3. This algorithm computes the scalar multiplication $Q = [k]P$ using LT (the MBNS representation of k). This algorithm is a multi-base adaptation of the standard left-to-right scalar multiplication algorithm (see for instance [57, Sec. 3.3.1]). Operation $Q + d \times P$ at line 5 is `NOP` (no operation) or `ADD` since $d \in \{0, 1\}$.

input: $P \in E(\mathbb{F}_p)$, and the list of terms LT which stores the MBNS recoding of the scalar k
$\mathrm{LT} = \big((d_1, e_{1,1}, e_{2,1}, \ldots, e_{l,1}), (d_2, e_{1,2}, e_{2,2}, \ldots, e_{l,2}), \ldots\big)$
output: $Q = [k]P$

1: $Q \leftarrow \mathcal{O}$
2: **foreach** t **in** LT **do** $(t = (d, e_1, e_2, \ldots, e_l))$
3: $Q \leftarrow Q + d \times P$ $(d \in \{0, 1\} \Rightarrow$ `NOP`/`ADD`$)$
4: **for** j **from** 1 **to** l **do**
5: $P \leftarrow \big[b_j^{e_j}\big]P$ (`DBL`, `TPL`, `QPL`, ...)
6: $Q \leftarrow Q + P$
7: **return** Q

Figure 4.3: MBNS scalar multiplication algorithm $Q = [k]P$.

The combination of recoding algorithm in figure 4.2 and scalar multiplication algorithm in figure 4.3, allows to overlap recoding steps and curve-level operations. For instance, when divisibility by 3 is detected, exact division by 3 and TPL operations can be launched in parallel. The same approach applies for other base elements (division by 2 and DBL, division by 5 and QPL, etc.). The recoding algorithm produces a MBNS representation with a recursive factorization similar to Horner scheme. Figure 4.11 illustrates a complete example. Unlike previous DBNS and MBNS methods, our recoding can be fully embedded in hardware and operates on-the-fly. Firstly, we do not need costly tables or computations such as the approximation of k by multi-base terms. This strategy is closed to the ideas of Longa [75]. Secondly, as soon as a divisibility is detected, we can launch the corresponding curve-level operation. There is no need to wait for a complete term before starting corresponding curve-level operations.

However as we start with least significant terms first, we cannot use point addition with mixed coordinates (mADD). We are obliged to use standard point addition which is a little slower. Clearly our method is not competitive compared to the fastest state-of-art ones when costly off-line recoding is possible. However it provides the first full on-the-fly hardware implementation. Off-line recodings limit practical applications in secure embedded systems.

Overlapping recoding and curve-level operations is possible due to the very fast divisibility tests and exact divisions. For instance, with $n = 160$, $w = 12$ and $t = 14$, divisibility tests by all $b_j \neq 2$ and exact division by one $b_j \neq 2$ require $t + 3 = 17$ and $t + 4 = 18$ clock cycles, respectively. These small durations must be compared to the duration of one DBL, TPL, QPL, etc. These curve-level operations are significantly slower than the recoding steps required to produce one new term (at least one order of magnitude).

There is a short latency at the very beginning (less than 0.01% of total $[k]P$ computation time for $n = 160$ and even less for larger fields). The first curve-level operation is determined using the first divisibility test results. After $t+\varepsilon$ clock cycles there are two cases: i) k is divisible by one multi-base element b_j then a DBL, TPL, QPL, etc. can be launched depending on which b_j divides k, ii) k is not divisible by $\mathcal{B}$ elements and an ADD can be launched.

Let $C(b_j)$ be the curve-level operation related to multi-base element b_j, e.g. $C(2) =$ DBL, $C(3) =$ TPL. Let TIME(op) be the computation time of curve-level operation op. The number of terms in LT is denoted n'. The total computation time of scalar multiplication $[k]P$ where k is recoded by LT is: $n' \cdot \text{TIME}(\texttt{ADD}) + \sum_{j=1}^{l} \left(\left(\sum_{t \in \text{LT}} e_{j,t} \right) \cdot \text{TIME}\big(C(b_j)\big) \right)$.

For instance, if $b_i \in (2, 3, 5, 7)$, n' point additions (ADD), $\sum_{j=1}^{n'} e_{2,j}$ point doublings (DBL), $\sum_{j=1}^{n'} e_{3,j}$ point triplings (TPL), $\sum_{j=1}^{n'} e_{5,j}$ point quintuplings (QPL) and $\sum_{j=1}^{n'} e_{7,j}$ point septuplings (SPL) will be launch. The number of curve operations depends on the MBNS representation of k.

Selection of multi-base elements in $\mathcal{B}$ requires an experimental evaluation. Figure 4.4 reports statistical $[k]P$ results on the total computation time (in M) for 100 000 random 160-bit values recoded using the unsigned MBNS algorithm and for various multi-bases. These results show that the most efficient multi-base is $\mathcal{B} = (2, 3, 5, 7)$ with our parameters [1]. Adding $b_j = 11$ does not improve the performance while it makes the architecture larger and slower. This will also be confirmed in the signed case in section 4.2. Base b_1 is always selected as 2 due to simple divisibility and exact division by 2 for standard binary representation of k.

1. curves over $\mathbb{F}_p$, $n = 160$, simplified Weierstrass equation, Jacobian coordinates, costs of curve-level in table 4.16

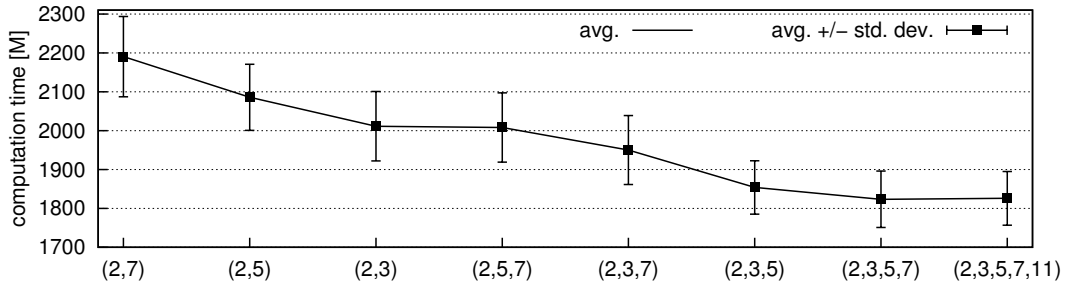


Figure 4.4: Statistical performance evaluation of unsigned MBNS recoding and scalar multiplication.

Recoding algorithm presented in figure 4.2 is a simple unsigned version. In section 4.2, we will present extensions to signed MBNS with fewer terms leading to faster computations. Scalar multiplication algorithm in figure 4.3 is not secure against SCAs. Simple power analysis attacks can be used due to the different behaviour of curve-level operations in lines 3 and 5. In such an attack, whereas one cannot directly "read" the bits of the key, an attacker could distinguish the curve-level operations, and thus the attacker could reconstruct the scalar k used. We will see in section 4.2 how signed MBNS recoding can be used as a protection against some attacks.

4.1.2 Implementation of the Divisibility Tests

At each recoding step, the scalar remainder must be tested for divisibility by all b_j for $1 \leq j \leq l$. Testing divisibility by 2^s with $s \leq w$ in a radix-2 representation is straightforward and is implemented in a very small module of the recoding unit (see section 4.1.4). One has just to check if least significant bit(s) is(/are) zero(s) or not. For $b_j \neq 2$, we use a very old method based on specific properties of the sum of argument digits modulo b_j. This method for divisibility test by b_j in radix-r representation is reported in a Blaise Pascal's post-mortem publication [101] (in Latin, see [109] for comments in English). This method is often called Pascal's tape or ribbon. Latter in this chapter, we will see that the most efficient multi-bases are $(2, 3, 5)$ and $(2, 3, 5, 7)$. Then, we first provide details for divisibility tests by $b_j \in \{3, 5, 7\}$ as they lead to the most efficient $\mathcal{B}$. Table 4.1 reports the remainders $2^i \bmod b_j$ for $b_j \in \{3, 5, 7\}$ and $i \leq 16$. They form a periodic sequence.

	i																
b_j	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	**2**	**1**
5	1	3	4	2	1	3	4	2	1	3	4	2	1	**3**	**4**	**2**	**1**
7	2	1	4	2	1	4	2	1	4	2	1	4	2	1	**4**	**2**	**1**

Table 4.1: Pascal's tapes for divisibility tests by $b_j \in \{3, 5, 7\}$ (values are $2^i \bmod b_j$).

Using table 4.1 with Pascal's tape in case $b_j = 3$ (the periodic sequence is $(2\,1)^*$), one has:

$$\begin{aligned} k \bmod 3 &= (\ldots + 2^3 k_3 + 2^2 k_2 + 2^1 k_1 + k_0) \bmod 3 \\ &= (\ldots + 2k_3 + k_2 + 2k_1 + k_0) \bmod 3 \\ &= \Big(\underbrace{\sum (2k_{2i+1} + k_{2i})}_{\alpha}\Big) \bmod 3 \end{aligned}$$

Computation of α requires the sum of many 2-bit words ($n \gg 100$, where k is the n-bit scalar). α is a multi-bit integer, then it must be recursively reduced using the same method. There is a trade-off between the size of the intermediate accumulators and the reduction completion. Architecture presented on figure 4.5 decomposes this large operation into partial sums accumulated and partial reduction on a limited number of bits for each word of the scalar. This is the purpose of the light blocks denoted "$\sum$ `for` $b_j = 3$" and connected registers on figure 4.5. Then t clock cycles are required for the accumulation and partial reduction. The very last reduction steps and comparison to b_j is denoted "$\mathcal{R}$ `for` b_j" in the second type of light blocks. Clock, reset and enable signals are not represented on the figures.

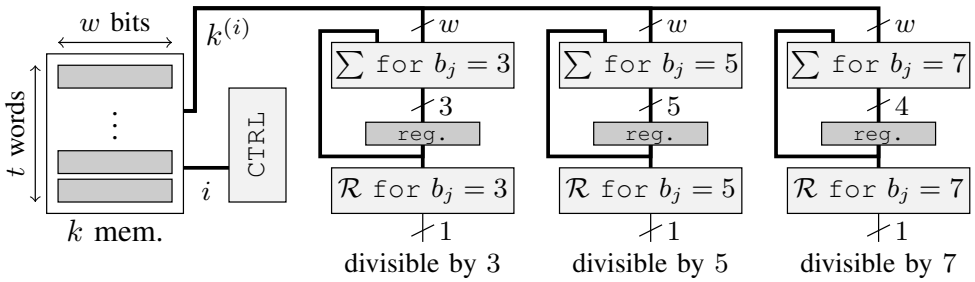


Figure 4.5: Divisibility test architecture.

The same kind of computation is performed for all $b_j \neq 2$. For $b_j = 7$, the sum uses 3-bit words with the sequence $(4\,2\,1)^*$. For $b_j = 5$, the sequence is $(3\,4\,2\,1)^*$ where the digit 3 requires a specific treatment. A first solution uses $3 = 1 + 2$ and an unsigned sum with additional inputs. A second solution considers $3 \equiv -2 \bmod 5$ and a signed sum with less operands (but with sign extension). Architecture on figure 4.5 allows parallel divisibility test by several b_j in only one computation. For instance, if k is divisible by 15, both divisibility tests by $b_j = 3$ and $b_j = 5$ return true after $t + 3$ (resp. $t + 4$) clock cycles with $w = 12$ (resp. $w = 24$). The memory element for k storage on the left of figure 4.5 will be shared with other units as we will see in section 4.1.3 and section 4.1.4.

The parameter w significantly impacts performances. The lengths of remainders sequences for $b_j = 3, 5, 7$ are $2, 4, 3$, respectively (see table 4.1). To avoid complex decoding schemes, we use multiples of the least common multiple of the lengths. As $\mathrm{lcm}(2, 4, 3) = 12$, we tried implementations for $w = 12$ and $w = 24$ (larger multiples of 12 would slow down the recoding).

All hardware implementations reported in this chapter have been described in VHDL and implemented on a XC5VLX50T FPGA using ISE 12.4 from Xilinx with standard efforts for synthesis, place and route. We report numbers of clock cycles, best clock frequencies and numbers of occupied slices. We also report numbers of look-up tables (LUTs with 6 inputs in Virtex 5) and flip-flops (FFs) for area. A XC5VLX50T contains 7200 slices with 4 LUT and 4 flip-flops per slice. We use flip-flops for all storage elements. FPGA implementation results are reported

in table 4.2 for divisibility tests by $b_j \in \{3, 5, 7\}$. Implementation results are the same whatever the value t, and so whatever n. Indeed, all block outputs do not depend on t. Thus table 4.2 reports FPGA implementation results only for $w \in \{12, 24\}$, without the memory element for k storage. Two versions have been implemented, where only the divisibility test unit by 5 changes. Indeed, the periodic sequence for 5 in table 4.1 is $(3421)^*$. The first version (i) considers $3 = 1+2$ (unsigned version), while the second (ii) considers $3 \equiv -2 \bmod 5$ (signed version). The second version requires the use of 2's complement representation to perform all intermediate sums.

Below, all unit implementations of the divisibility test unit are done with the first version (i). Indeed, the unsigned version of the divisibility test unit for $b_j = 5$ is more efficient in clock frequency and circuit area. This may be related to the extra cost of the sign extension for additions/subtractions in 2's complement.

w	divisibility version for 5	area slices (FF/LUT)	freq. MHz	clock cycles
12	i	25 (40/81)	543	$t+3$
	ii	41 (38/111)	451	
24	i	67 (53/152)	549	$t+4$
	ii	86 (48/187)	549	

Table 4.2: FPGA implementation results for divisibility tests by $b_j \in \{3, 5, 7\}$.

FPGA implementation results are reported in table 4.3 for $n = 160$, 256 and 521 bits, and for divisibility tests by $b_j \in \{3, 5, 7\}$. These implementation results include the memory element for k storage.

n	w	t	area slices (FF/LUT)	freq. MHz	clock cycles
160	12	14	30 (56/98)	423	$t+3$
	24	7	65 (74/189)	410	$t+4$
256	12	22	32 (57/99)	422	$t+3$
	24	11	64 (80/196)	395	$t+4$
521	12	44	34 (58/100)	420	$t+3$
	24	22	69 (82/214)	388	$t+4$

Table 4.3: FPGA implementation results for divisibility tests by $b_j \in \{3, 5, 7\}$ and $n \in \{160, 256, 521\}$ bits.

Directly apply the Pascal's tapes method for more divisibility tests, like $b_j \in \{3, 5, 7, 9, 11, 13\}$, complicates the control and increases the number of internal registers used during the accumulation loop. Table 4.4 reports the remainder $2^i \bmod b_j$ for $b_j \in \{9, 11, 13\}$ and $i \leq 16$. Periodic sequences for $b_j \in \{9, 11, 13\}$ increases significantly, and their coefficients in the Pascal's tapes become more complex. It leads to a more complex control.

Arithmetic parameters of the architecture have been modified to have more complex divisibility tests and to study the links between the length and the form of the periodic sequence in the

	i																
b_j	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
9	7	8	4	2	1	5	7	8	4	2	1	**5**	**7**	**8**	**4**	**2**	**1**
11	9	10	5	8	4	2	1	**6**	**3**	**7**	**9**	**10**	**5**	**8**	**4**	**2**	**1**
13	3	8	4	2	1	**7**	**10**	**5**	**9**	**11**	**12**	**6**	**3**	**8**	**4**	**2**	**1**

Table 4.4: Pascal's tapes for divisibility tests by $b_j \in \{9, 11, 13\}$ (values are $2^i \bmod b_j$).

Pascal's tapes. Instead of considering bit remainders $2^i \bmod b_j$, we considered sub-words remainders $\left(2^{12}\right)^i \bmod b_j$. The Pascal's tape is reported in table 4.5 for $b_j \in \{3, 5, 7, 9, 11, 13, 17, 25\}$. In this table, values are remainders $\left(2^{12}\right)^i \bmod b_j$ for $i \leq 9$.

	i									
b_j	9	8	7	6	5	4	3	2	1	0
3	1	1	1	1	1	1	1	1	1	**1**
5	1	1	1	1	1	1	1	1	1	**1**
7	1	1	1	1	1	1	1	1	1	**1**
9	1	1	1	1	1	1	1	1	1	**1**
11	3	9	5	4	1	**3**	**9**	**5**	**4**	**1**
13	1	1	1	1	1	1	1	1	1	**1**
17	16	1	16	1	16	1	16	1	**16**	**1**
25	6	11	16	21	1	**6**	**11**	**16**	**21**	**1**

Table 4.5: Pascal's tapes for divisibility tests by $b_j \in \{3, 5, 7, 9, 11, 13, 17, 25\}$ (values are $\left(2^{12}\right)^i \bmod b_j$).

Lines corresponding to the divisors 3, 5, 7, 9 and 13 in table 4.5 report the same periodic sequence $(1)^*$: a unique accumulation register can be used. The same accumulator is shared during the t clock cycles, and then a final reduction "$\mathcal{R}'$" for each divisor $b_j \in \{3, 5, 7, 3^2\}$ is computed in the architecture in figure 4.6. In this chapter, we are interested in only $b_j \in \{3, 5, 7\}$ and small powers of these values. Indeed, several curve-level operations can be launched if the scalar is divisible by a power of b_j. For instance, when the scalar is divisible by a 3^2, two TPLs can be launched. In this version, the accumulation register output is larger ($w + \lceil \log_2(t) \rceil$ bits) than registers output in figure 4.5.

FPGA implementation results for divisibility tests are reported in table 4.6 for $n = 160$, 256 and 521 bits, and $b_j \in \{3, 5, 7, 9\}$.

4.1.3 Implementation of the Exact Division by Multiple-Base Elements

Here, *exact division* k/b_j means that we know that the dividend k is divisible by the divisor b_j (using divisibility tests presented above). Divisibility property allows to significantly optimize the division algorithm. [62] provides an efficient algorithm when the radix is prime or power of 2. This algorithm is often used in multiple-precision software libraries such as GMP. Division by 2^s is not considered in this section. It is performed using shifts in the global recoding unit presented in section 4.1.4.

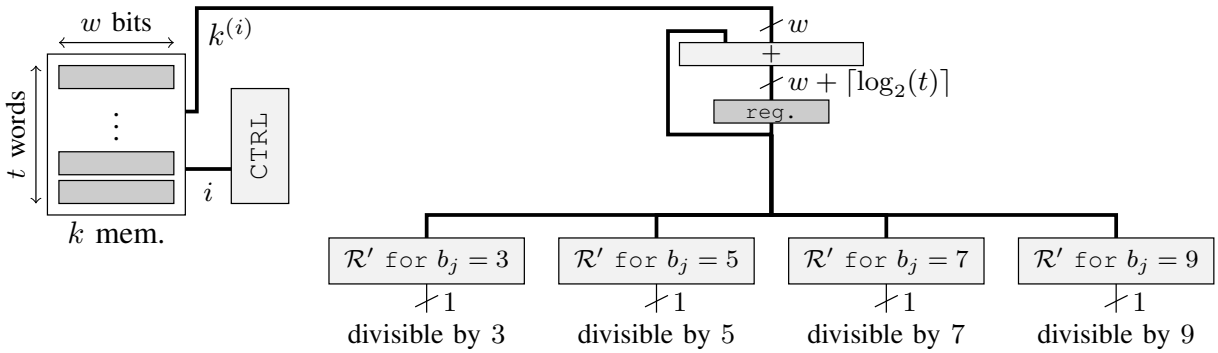


Figure 4.6: Divisibility test architecture by $b_j \in \{3, 5, 7, 9\}$ with the Pascal's tapes improvement.

n	w	t	area slices (FF/LUT)	freq. MHz	clock cycles
160	12	14	32 (86/98)	454	$t+3$
	24	7	50 (136/142)	490	$t+4$
256	12	22	37 (88/99)	460	$t+3$
	24	11	52 (148/149)	495	$t+4$
521	12	44	35 (90/105)	424	$t+3$
	24	22	55 (152/152)	485	$t+4$

Table 4.6: FPGA implementation results for divisibility tests by $b_j \in \{3, 5, 7, 9\}$ and $n \in \{160, 256, 521\}$ bits, with the Pascal's tapes improvement.

We use a dedicated version of algorithm presented in [62] for $b_j \in \{3, 5, 7\}$ and optimized for hardware implementation. The algorithm, presented in figure 4.7, operates in t iterations in a word-serial way starting with least significant (w is the word size). Iteration number i deals with $k^{(i)}$ the ith word of k. The inverse of divisor b_j modulo 2^w is a constant and always exists since multi-base elements are co-prime and include 2.

The main differences for the 3 operations ($b_j \in \{3, 5, 7\}$) are the multiplication by modular inverse line 4 and the comparison to constants line 7. All other elements are shared in the architecture (operators, control, registers).

Table 4.7 reports binary representations of modular inverses for exact division by 3, 5, 7 and 12/24-bit words. Multiplication $r \times (b_j^{-1} \bmod 2^w)$ at line 4 in figure 4.7 is implemented as a sequence of additions/subtractions and shifts using an in-house multiplication by constant algorithm [16]. Subtraction at line 3 is inserted in the sequence. Some adders in the 3 sequences are shared to reduce area. Table 4.7 reports the number of additions/subtractions γ required to perform $r \times (b_j^{-1} \bmod 2^w)$.

The architecture of our exact division by $b_j \in \{3, 5, 7\}$ unit is presented on figure 4.8 (without clock, reset and enable signals). At iteration i, word $k^{(i)}$, read (`R` port) from scalar memory, is added to $-c$ and used in the addition sequence (block denoted "$\times(b_j \bmod 2^w)$" and "$\pm$ `seq.`") corresponding to b_j. The correct value is selected by `MUX1` and written in scalar memory (`W` port). We use an in-place version of the algorithm $k \leftarrow k/b_j$ to keep memory footprint as small as possible.

input: two integers $k = (k^{(t-1)} \cdots k^{(0)})_2$ and b_j, where $k \bmod b_j = 0$
output: k/b_j

```
1:  c ← 0
2:  for i from 0 to t − 1 do
3:      r ← k^(i) − c
4:      r ← r × (b_j^{-1} mod 2^w)
5:      c ← 0
6:      for h from 1 to b_j − 1 do
7:          if r ≥ h × ⌈(2^w − 1)/b_j⌉ then
8:                  c ← c + 1
9:      k^(i) ← (r_{w−1} ··· r_0)
10: return k
```

Figure 4.7: Exact division k/b_j algorithm (from [62]).

b_j	$b_j^{-1} \bmod 2^{12}$, γ	$b_j^{-1} \bmod 2^{24}$, γ
3	$(101010101011)_2$, 3	$(101010101010101010101011)_2$, 4
5	$(110011001101)_2$, 3	$(110011001100110011001101)_2$, 4
7	$(110110110111)_2$, 3	$(110110110110110110110111)_2$, 4

Table 4.7: Modular inverses used in exact divisions.

Operation sequences leading to the computations $y = x_1 \times 3^{-1} \bmod 2^w$, $y' = x'_1 \times 5^{-1} \bmod 2^w$ and $y'' = x''_1 \times 7^{-1} \bmod 2^w$ for a length word $w = 12$ (resp. $w = 24$) are reported in table 4.8 (resp. in table 4.9). Multiplication $r \times (b_j^{-1} \bmod 2^w)$ at line 4 in figure 4.7 is partially used: only the first w bits are used (line 9). Thus operation sequences in tables 4.8 and 4.9 can be simplified. For instance, the line $y' = x'_4 + (x'_1 \times 2^{12})$ in table 4.8 is unnecessary.

$2\,731 \equiv 3^{-1} \bmod 2^{12}$	$3\,277 \equiv 5^{-1} \bmod 2^{12}$	$3\,511 \equiv 7^{-1} \bmod 2^{12}$
$x_2 = x_1 + (x_1 \times 2^2)$	$x'_2 = x'_1 + (x'_1 \times 2^4)$	$x''_2 = x''_1 + (x''_1 \times 2^6)$
$x_3 = x_2 + (x_2 \times 2^4)$	$x'_3 = x'_2 + (x'_1 \times 2^8)$	$x''_3 = x''_2 + (x''_2 \times 2^3)$
$x_4 = -x_2 + (x_1 \times 2^4)$	$x'_4 = x'_3 - (x'_3 \times 2^2)$	$y'' = -x''_3 + (x''_1 \times 2^{12})$
$y = -x_3 + (x_4 \times 2^8)$	$y' = x'_4 + (x'_1 \times 2^{12})$	

Table 4.8: Operation sequences for the multiplication by constants with $w = 12$.

Comparisons in loop lines 6–8 of the algorithm in figure 4.7 are unrolled and implemented as combinatorial logic (block denoted "`cmp.` b_j") for each specific $b_j \in \{3, 5, 7\}$. The correct value c is selected by `MUX2` and is sent to the addition sequences.

The exact division unit has been implemented least significant word (LSW) first. A second version has been studied and implemented most significant word (MSW) first. The exact division starting from MSW (most significant word) algorithm is presented in figure 4.9 and its proof is proposed in appendix B. The proof of division exact algorithm starting from LSW (least significant word) is proved in [62]. The strategy is the same for the computation of $r \times (b_j^{-1} \bmod 2^w)$ in both exact division versions. The only difference between the two algorithms is the

$11\,184\,811 \equiv 3^{-1} \bmod 2^{24}$	$13\,421\,773 \equiv 5^{-1} \bmod 2^{24}$	$14\,380\,471 \equiv 7^{-1} \bmod 2^{24}$
$x_2 = x_1 + (x_1 \times 2^4)$	$x'_2 = x'_1 - (x'_1 \times 2^2)$	$x''_2 = x''_1 + (x''_1 \times 2^3)$
$x_3 = x_2 + (x_2 \times 2^2)$	$x'_3 = x'_2 + (x'_1 \times 2^4)$	$x''_3 = x''_2 + (x''_1 \times 2^6)$
$x_4 = x_3 + (x_3 \times 2^8)$	$x'_4 = x'_3 + (x'_3 \times 2^{12})$	$x''_4 = x''_1 + (x''_1 \times 2^{21})$
$x_5 = -x_3 + (x_1 \times 2^8)$	$x'_5 = x'_1 - (x'_4 \times 2^2)$	$x''_5 = -x''_4 + (x''_1 \times 2^{24})$
$y = -x_4 + (x_5 \times 2^{16})$	$y' = x'_5 + (x'_4 \times 2^8)$	$x''_6 = x''_3 + (x''_3 \times 2^9)$
		$y'' = x''_5 - (x''_6 \times 2^3)$

Table 4.9: Operation sequences for the multiplication by constants with $w = 24$.

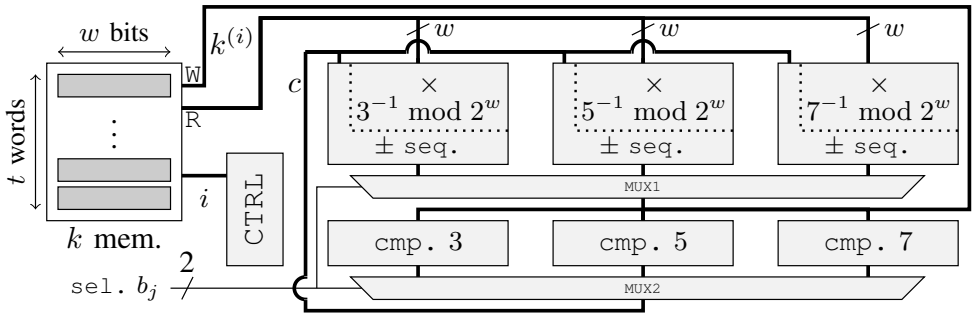


Figure 4.8: Exact division unit architecture.

computation of the value c at line 8 in figure 4.7 and at line 3 in figure 4.9.

input: two integers $k = (k^{(t-1)} \cdots k^{(0)})_2$ and b_j, where $k \bmod b_j = 0$
output: k/b_j

1: $c \leftarrow 0$
2: **for** i **from** $t-1$ **to** 0 **do**
3: $\quad c \leftarrow (c + k^{(i)}) \bmod b$
4: $\quad r \leftarrow k^{(i)} - c$
5: $\quad r \leftarrow r \times (b^{-1} \bmod 2^w)$
6: $\quad k^{(i)} \leftarrow (r_{w-1} \cdots r_0)$
7: **return** k

Figure 4.9: Exact division k/b_j algorithm starting most significant words first.

Here is an example in hexadecimal of an exact division by $b_j = 3$ starting from MSW for a word length $w = 12$ with a dividend $k = (1\,170\,000\,261)_{10} = (045\,BCC\,985)_{16}$ and the value $3^{-1} \bmod 2^w \equiv (AAB)_{16}$:

$$(045 - (045 \bmod 3)) \times AAB = (045 - 0) \times AAB = 02E\,\mathbf{017}$$
$$(BCC - ((0 + BCC) \bmod 3)) \times AAB = (BCC - 2) \times AAB = 7DC\,\mathbf{3EE}$$
$$(985 - ((2 + 985) \bmod 3)) \times AAB = (985 - 0) \times AAB = 658\,\mathbf{DD7}$$

$$k/3 = (017\,3EE\,DD7)_{16}$$
$$= (390\,000\,087)_{10}$$

In practice, the value c in the second exact division algorithm in figure 4.9 comes directly from the divisibility test unit. Indeed, exact division starting from MSW unit accumulates $k^{(i)}$ modulo b_j for each word. For instance, the divisibility test unit returns $\left(\sum_{i=t-3}^{t-1}(k^{(i)} \bmod b_j)\right) \bmod b_j$ for the three last words of k. For the first word $k^{(0)}$, the divisibility test unit returns 0 when the scalar k is divisible by b_j, i.e. $k \equiv \left(\sum_{i=0}^{t-1}(k^{(i)} \bmod b_j)\right) \bmod b_j \equiv 0 \bmod b_j$.

FPGA implementation results for the exact division unit are reported in table 4.10 for $n = 160$. For $w = 24$, speed decreases due to the complexity of the comparison blocks. For instance, in case $b_j = 7$, six comparisons are required. The version starting from LSW is slower than the MSW version because comparisons are required. The number of clock cycles is $t + O(1)$ where the constant depends on w, and is the same for the two versions.

starting from	w	area slices (FF/LUT)	freq. MHz	clock cycles
LSW	12	59 (138/171)	291	$t+4$
	24	152 (441/448)	202	$t+5$
MSW	12	73 (178/206)	442	$t+4$
	24	178 (494/540)	346	$t+5$

Table 4.10: FPGA implementation results for exact division by $b_j \in \{3, 5, 7\}$ and $n = 160$ bits.

4.1.4 Unsigned Multiple-Base Recoding Unit

The complete recoding unit architecture is presented in figure 4.10 with the exact division starting from LSW. The scalar memory stores the scalar k (t words by w bits). The small subtraction block is in charge of line 5 in the recoding algorithm figure 4.2. The `DTD-2` block is in charge of divisibility test (1-bit result) and exact division (w-bit bus) by 2 or small powers of 2 (2^s with $s \leq w$, if $s > w$ several iterations are used). Divisibility test unit for other base elements is described in section 4.1.2 (3-bit output) while the exact division unit is described in section 4.1.3 (w-bit output). `MUX` selects which unit output should be written in the scalar memory. The global controller (`CTRL`) generates all high-level control signals for the architecture units (these signals and clock are not represented on the figure). It also provides the global control with informations on which curve-level operations must be launched.

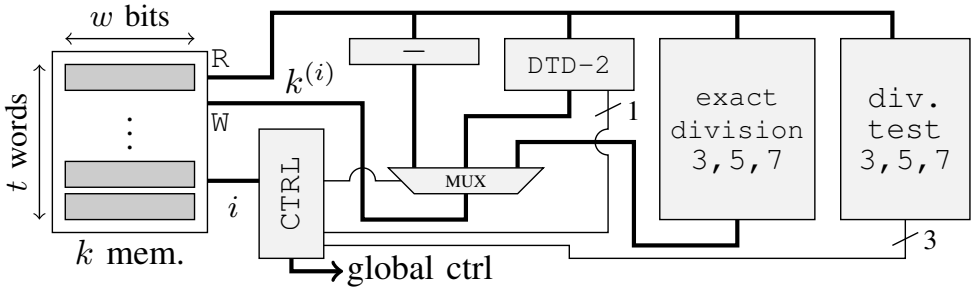


Figure 4.10: Complete recoding unit architecture.

The proposed architecture has been implemented with exact division starting from LSW and MSW. The exact division unit is the one which is implemented with these two versions. Indeed,

only divisibility tests and exact divisions can start from the right or the left, and the implementation of divisibility tests is the same when one begins from both sides. All other components must start from LSW. For the exact division starting from MSW, a second counter has been implemented in order to send the scalar k by the other way according to all other components. Thus, when results are stored from MSW, one needs to wait $t+4$ clock cycles for $w = 12$ ($t+5$ for $w = 24$) to perform operations which starts from LSW. Indeed, a component starting from LSW must wait the end of operations performed from MSW. For instance, when one computes an exact division starting from MSW and wants to add one to the scalar after the division, he/she must wait the result of the division. However when one computes an exact division starting from LSW and wants to subtract one to the scalar after the division, he/she can wait the result of the first word of the division and launch the subtraction just after.

The FPGA implementation results for the complete recoding unit are reported in table 4.11 for $\mathcal{B} = (2, 3, 5, 7)$ and the two versions of the exact division unit (see section 4.2.4 for comparison to a ECC processor). The MBNS recoding unit is smaller than the total area of a complete ECC processor and it can operate at 200 MHz. For two different sizes of n, hardware implementations have a little more area for a same clock frequency. We plan to investigate additional pipeline stages in the recoding unit to increase the clock frequency (but with more complex control).

n	exact division starting from	w	area slices (FF/LUT)	freq. MHz
160	LSW	12	145 (301/412)	237
		24	316 (682/908)	202
	MSW	12	149 (326/422)	262
		24	313 (732/895)	206
256	LSW	12	155 (302/415)	237
		24	321 (683/897)	202
	MSW	12	179 (334/468)	262
		24	377 (733/1 002)	262

Table 4.11: FPGA implementation results for complete recoding unit with $\mathcal{B} = (2, 3, 5, 7)$.

The implementation requires an initial step which consists to load the scalar k into scalar memory word by word. It is not necessary to wait that the computed scalar be loaded to launch divisibility tests. It is sufficient to wait one clock cycle to ensure that the first word is loaded. Thus one clock cycle is necessary to know if the scalar is divisible by a small power of 2, and $t + 3$ clock cycles to know if the scalar can be divided by one of the others multi-base elements for $w = 12$ ($t + 4$ clock cycles for $w = 24$). If the scalar is not divisible, the controller launches the computation $k - 1$. Thus, the multi-base recoding unit needs to wait $t + 3$ clock cycles in the worst case to launch the first curve-level operation for $w = 12$ ($t + 4$ clock cycles for $w = 24$).

When the first curve-level operation is launched, the proposed multi-base recoding unit finds sequences of addition, doubling, tripling, quintupling and septupling points to launch. Finding a new recoding sequence of k such as $k = (2^{e_1}3^{e_2}5^{e_3}7^{e_4} + 1) \times k'$ takes $\lceil e_1/12 \rceil$ clock cycles for e_1 point doublings whatever w. For $w = 12$, it takes $e_2(t + 4) + e_3(t + 4) + e_4(t + 4)$ clock cycles for e_2 point triplings, e_3 point quintuplings and e_4 point septuplings, with $(e_2 + e_3 + e_4)(t + 3)$ clock cycles for the divisibility tests. In addition, one needs to wait $t + 3$ clock cycles again to

know if k is not any more divisible by a multi-base element (one must replace $t+4$ by $t+5$ and $t+3$ by $t+4$ when $w=24$). After having found this sequence, a point addition can be launched, and one needs to wait $t+1$ clock cycles to compute the new number k in $k-1$ whatever w.

Thus, recoding x' computing to $(2^{e_1} \cdot \prod_{i=2}^{l} b_i^{e_i}) \times x'$ takes

$$t\Big(2(\sum_{i=2}^{l} e_i)+1\Big) + \Big(\lceil e_1/12 \rceil + 7(\sum_{i=2}^{l} e_i) + 3\Big)$$

clock cycles (replace 7 by 9 and 3 by 4 for $w=24$).

For $n=160$, $w=12$ and $t=14$, recoding a scalar corresponding to $2^8 3^4 5^2 7^1$ requires 262 clock cycles. This is less than one `DBL` operation (even for a parallel architecture). It reduces to 157 clock cycles for $w=24$ and $t=7$. The same term recoding requires 712 clocks cycles for $n=521$ and $w=12$ (382 clock cycles for $w=24$).

The recoding unit operates significantly faster than curve-level operations and in parallel to them. It provides curve-level operations to be launched on-the-fly without interruptions for curve-level schedule as illustrated on figure 4.11. On this figure, "`CLO`" denotes curve-level operations, `DT` denotes divisibility test, "`res.`" their results and "$/b_j$" exact division by b_j. For $k=87$, the recoding algorithm produces $87 = 0 + 3^1 \times (1 + 2^2 7^1)$.

The first term $(0+3)$ recoding is performed as fast as possible while for the second one $(1+2^2 7^1)$ it is spread over the computations but without interrupting, curve-level operations. This provides us options when designing the control. Divisibility by 2^2 is detected using only one `DT` ($2 \leq w$), this would not be the case for $b_j \neq 2$. For instance, divisibility by 3^2 requires our `DT` and exact division blocks to be used twice.

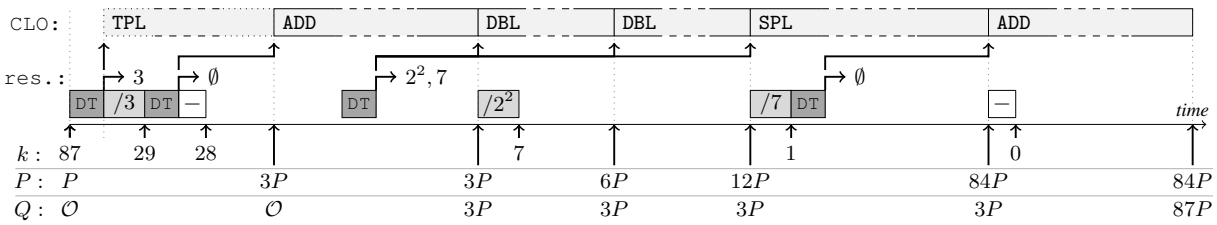


Figure 4.11: MBNS recoding and scalar multiplication illustration for $k=87$.

For $\mathbb{F}_p$ implementations, a limited w is sufficient to ensure a fast recoding compare to curve-level operations. In case of curves defined over $\mathbb{F}_{2^m}$, it may be necessary to use larger w. However even in that case, the recoding unit is still small in a complete processor.

4.1.5 Validation

Both recoding and scalar multiplication algorithms have been implemented in PARI/GP (`http://pari.math.u-bordeaux.fr/`) and SAGE (`http://www.sagemath.org/`) mathematical softwares (two people were in charge of one version GP or SAGE). The results of the two versions have been compared to the mathematical values and cross-checked for very large random data sets (millions of scalars for many sizes $n \in [160, 600]$ bits).

Functional validation of the architecture was done using some VHDL simulations on limited sets of random data and compared to the mathematical values. For performance validation, a

great deal of random tests and comparisons to state-of-art results have been performed. The corresponding results are reported in section 4.3.

4.2 Signed-Digit Optimizations

The unsigned recoding algorithm in figure 4.2 of section 4.1.1, only performs $k \leftarrow k-1$ when k is not divisible by $\mathcal{B}$ elements (this corresponds to a point addition at the curve level). We recall that $d = 0$ only for the very first term when the initial k is divisible by, at least, one base element in $\mathcal{B}$. For all other terms $d = 1$. As with other number systems (e.g. Booth recoding [13], `wNAF`, Avizienis representations [5], DBNS, etc.), using *signed digits* may help us to reduce the number of terms.

4.2.1 Signed-Digit MBNS Recoding

A simple modification of the MBNS recoding algorithm figure 4.2 is required to support signed digits as illustrated on figure 4.12. A *selection function* S has been introduced to select the digit $d = \pm 1$ to be used for each term $(d, e_1, e_2, \ldots, e_l)$ when k is not divisible by $\mathcal{B}$ elements. Depending on d, updating the scalar requires a subtraction ($d = 1$ similarly to the unsigned version) or an addition ($d = -1$). The scalar multiplication algorithm figure 4.3 is unchanged. At the curve level, digit values correspond to: a point addition when $d = 1$, a point subtraction when $d = -1$ and no operation when $d = 0$ (for the very first term only).

unsigned version			*signed version*	
4:	$d \leftarrow 1$	$\longrightarrow$	4:	$d \leftarrow \mathrm{S}(k)$
5:	$k \leftarrow k-1$		5:	$k \leftarrow k-d$

Figure 4.12: Modifications between the unsigned and signed versions of the recoding algorithm.

Determining S such that the recoding algorithm always produces the shortest list of terms is a very hard problem. We experimented with several heuristic selection functions and trade-offs between recoding performances (LT length) and implementation complexity (i.e. silicon area and recoding speed). Below, we present and compare 4 of these selection functions.

Minimum value selection function (`min`)

`min` is illustrated on figure 4.13. When k is not divisible, then $k-1$ and $k+1$ will be divisible by, at least, 2 (we always use $b_1 = 2$ in practice) and potentially other b_j. For each value $k-1$ and $k+1$, divisibility tests and exact division units are used to produce k' and k''. k' (resp. k'') corresponds to $k-1$ (resp. $k+1$) divided as much as possible by $\mathcal{B}$ elements. S returns $d = 1$ if $k' < k''$ else it returns $d = -1$ (test $k' \leq k''$ leads to similar performances). The `min` selection function only provides a local minimum for the total number of terms.

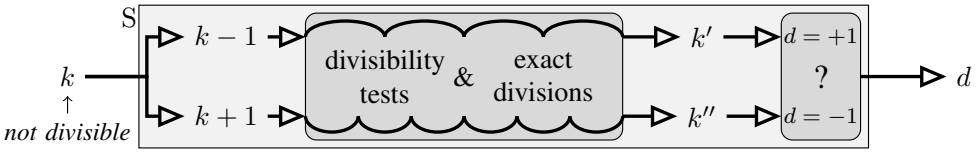


Figure 4.13: Principle of the `min` selection function.

A second scalar memory (t words $\times$ w bits) has been added in the recoding unit to store both k' and k''. The exponents corresponding to both $k-1$ and $k+1$ have been computed and stored during the exploration. The controller is adapted such that the correct set of exponents is selected.

The authors from [45] use a similar strategy. However they claim that tree based approach conversion is too costly for hardware implementation of systems using integers in the cryptographic range (p. 437, Sec. 3). In this chapter, we show that such a strategy can be easily implemented in hardware, without reducing the clock frequency and with a weak additional area (see sections 4.2.4and 4.2.5).

Maximum number of divisors selection function (`max_nb_div`)

In `min`, half of divisibility tests and exact divisions are discarded. Indeed, either k or k' is unused at each step by S. In `max_nb_div`, the number of base elements b_j which divide $k-1$ and $k+1$ is counted. S returns d according to the maximum number of divisors among $k-1$ and $k+1$. Only divisibility unit for $k-1$ and $k+1$ is used, not the exact division unit. `max_nb_div` is a cheap optimization but with low efficiency (see section 4.2.2).

Approximated minimum value selection function (`approx`)

To provide a cheap optimization but with higher performances, the `approx` selection function compares approximations of k' and k'' from `min` (instead of computing them exactly like in `min`). Exponents e'_j correspond to the divisibility test results for $k-1$: $k-1 = k' \prod_{j=1}^{l} b_j^{e'j}$. Exponents e''_j are those for $k+1$. The approximations of k' and k'' are respectively defined by

$$\delta' = \lfloor \log_2(k-1) \rfloor + 1 - \sum_{j=1}^{l} e'_j \log_2(b_j),$$

and

$$\delta'' = \lfloor \log_2(k+1) \rfloor + 1 - \sum_{j=1}^{l} e''_j \log_2(b_j),$$

where $\lfloor \log_2(k-1) \rfloor + 1$ is the position of the most significant bit (MSB) of $k-1$ (idem for $k+1$). MSB positions can be easily detected using the divisibility test unit (during the t iteration loop for each word). Approximation of weight $\prod_{j=1}^{l} b_j^{e'j}$ (or with e''_j exponents) uses the sum of multiplications by $\log_2(b_j)$ constants. The divisibility unit returns limited exponents: $e'_j \leq 1$ for $b_j \neq 2$ and $e'_j \leq w$ for $b_j = 2$ (see section 4.1.2). Then, there is no need for multiplications.

As an example, for $\mathcal{B} = (2, 3, 5, 7)$, $\delta' = \mathrm{MSB(k-1)} - e_{b_1=2} - 1.5e_{b_2=3} - 2.25e_{b_3=5} - 2.75e_{b_4=7}$ where $e_{b_1=2} \leq w$ and $e_{b_2=3}, e_{b_3=5}, e_{b_4=7} \leq 1$. The constants come from: $\log_2 3 \approx 1.59$, $\log_2 5 \approx 2.32$, and $\log_2 7 \approx 2.81$. The approximation for both k' and k'', as well as their comparison, can be easily implemented using a very small circuit which uses several adders in practice (see [16]).

2 steps minimum value selection function (`min2`)

The last selection function uses the time margin illustrated on figure 4.4 using a recursion limited to the next term. The first step uses `min` with $(k-1, k+1)$ to produce (k', k''). The second step uses `min` with $(k'-1, k'+1, k''-1, k''+1)$ to produce $(\zeta', \zeta'', \zeta''', \zeta'''')$. S returns d

according to the minimum value among $\zeta', \zeta'', \zeta''', \zeta''''$.

Below, we compare all selection functions.

Authors of [45] proposed a tree-based approach for recoding k in DBNS ($\mathcal{B} = (2, 3)$). Their approach is to have a bound, a number of minimum selection function `min`. It is a generalization of `min2`. Fixing a bound, they can use `min4` function, for 4 steps minimum value selection function. The last step uses `min`, and S returns d according to the minimum value among with all computed terms.

We use a similar approach where the bound is limited to 2 (i.e. it corresponds to our `min2` optimization) due to implementation limitations in hardware. We plan to study hardware implementations of this type of idea for larger bounds (e.g. 3 or 4). For their experiments, they chose a bound equals to 4 which "is a good compromise between the length of the chain and the time necessary to find it".

4.2.2 Experimental Analysis

Figure 4.14 presents statistical analysis results for average computation time (in multiplication over $\mathbb{F}_p$: `M`) of 100 000 scalar multiplications with 160-bit random scalars recoded. Figure 4.14 uses our signed MBNS algorithm for various multi-bases and the 4 selection functions presented in section 4.2.1. Most efficient multi-bases are $\mathcal{B} = (2, 3, 5)$ and $\mathcal{B} = (2, 3, 5, 7)$ with very close performance. Selection function `max_nb_div` is not efficient. Selection functions `min` and `approx` have very close performance. The best performance is obtained by `min2` with slightly better computation time than `approx`. Selection function `approx` is the best trade-off between performance and recoding cost (`min` and `min2` require longer computations and more energy).

4.2.3 Randomized Selection Function

We experimented with a simple randomized selection function (`rnd`) as a protection against some side channel attacks. When k is not divisible by $\mathcal{B}$ elements, S returns $d = 1$ or $d = -1$ randomly. Obviously this leads to larger number of terms (and point additions/subtractions) in the recoding as reported in table 4.12 (for simplified Weierstrass curves with $a \neq -3$ and 100 000 random scalars). This table provides the differences according to the multiplication number. Proposed randomization scheme allows a scalar sub-string to be represented using totally different recodings for several $[k]P$ with the same k. This is a direct protection against some differential attacks due to the very huge number of different representations using signed digits, see [87, Tab. 2, p. 394]. For protection against simple attacks (based on only one trace or a very few traces), real robustness of the randomized selection function relies on the fact that point addition and point subtraction cannot be distinguished in traces. Indeed, protecting the sign change when using point subtraction is supposed to be simple in the circuit. However we still must perform a more complete security evaluation at hardware level using a very advanced attack system and to compare to other protection schemes (e.g. addition chains [23]).

4.2.4 FPGA Implementation

The signed MBNS recoding algorithm has been implemented on FPGA (see end of section 4.1.2 for target and tools details). Table 4.13 reports corresponding results for `approx` selection function, $\mathcal{B} = (2, 3, 5, 7)$, $n = 160$ and $n = 256$ bits and the two versions of exact division, when starting from LSW and MSW. The signed recoding unit includes: a second scalar memory (t words by w bits), the specific circuit for `approx` selection function and a complete unit control. Compared to the unsigned version table 4.11, area overhead for signed version is very

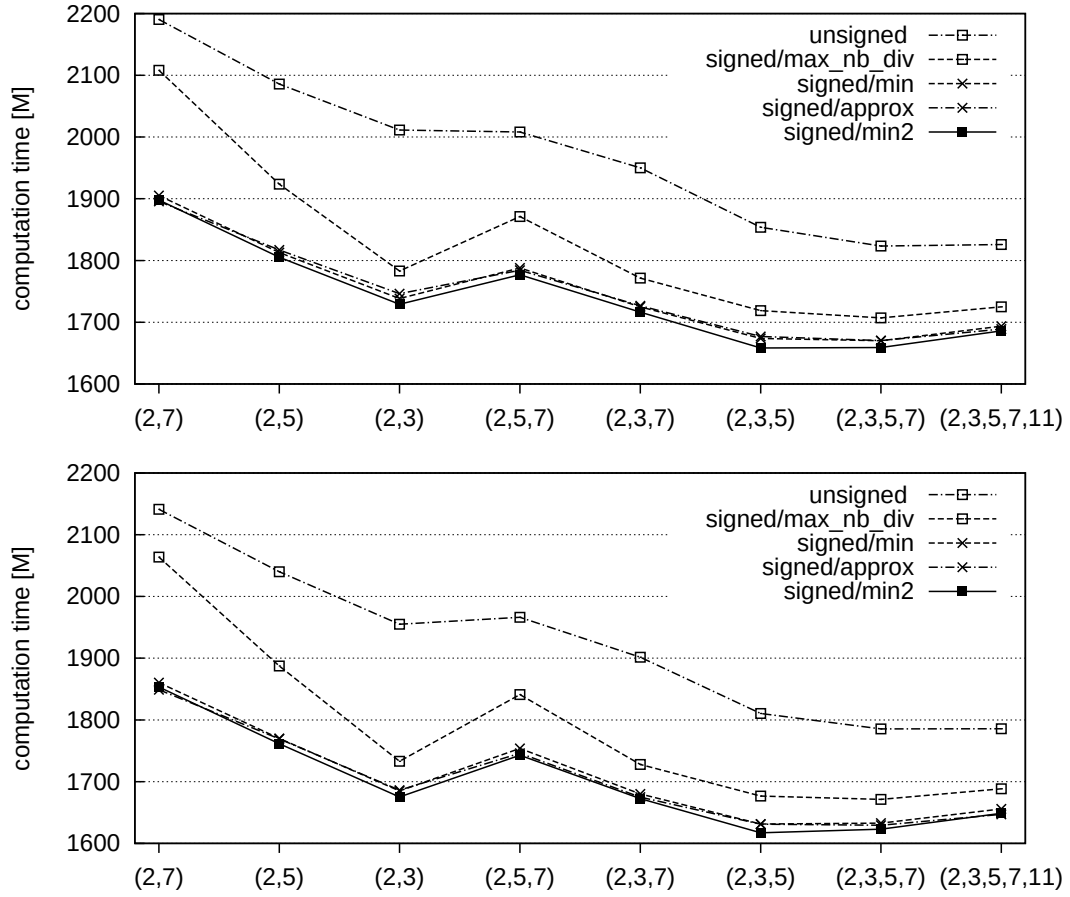


Figure 4.14: Statistical performance evaluation of signed MBNS recoding and scalar multiplication (top: $a \neq -3$, bottom: $a = -3$).

small: 13% more slices (8% FF and 13% LUTs). In Virtex 5 FPGAs, very small memories, such scalar ones, can be efficiently implemented using distributed RAMs in the LUTs of SLICEMs. This explains why only 25 additional flip-flops are required for the signed version while there is a 168-bit memory ($t = 14$ and $w = 12$) for the second scalar memory. The same clock frequency is achieved for both signed and unsigned versions (the critical path is the exact division unit).

In order to compare the MBNS recoding unit to a complete ECC processor, table A.1 in appendix A reports two FPGA implementations (a small version and a large one) of an ECC processor provided by the authors of [24] (for curves over $\mathbb{F}_p$, $n = 160$ bits and Jacobian coordinates). Our MBNS signed recoding unit works at higher frequency than the ECC processor. In addition its area represents less than 10% (resp. 7%) for $n = 160$ and $w = 12$ compared to the complete small (resp. large) version of the ECC processor.

4.2.5 ASIC Implementation

Tables 4.14 and 4.15 report ASIC implementation results of the signed MBNS recoding implementation starting from LSW. The signed MBNS recoding implementation is applied with a maximum path delay constraint of 10ns and 5ns from all inputs to all outputs.

ASIC results reported have been synthesized into gate-level netlists using standard V_{th} (SVT) cells of an industrial 130nm bulk CMOS technology library using Synopsys Design Compiler G-

$\mathcal{B}$	rnd		min		diff.
	M	#ADD	M	#ADD	[%]
(2,3)	1 960.5	49.3	1 738.5	34.0	12.8
(2,3,5)	1 843.0	39.8	1 673.7	28.0	10.1
(2,3,5,7)	1 811.4	34.8	1 670.0	24.8	8.5
(2,3,5,7,11)	1 816.7	32.1	1 693.5	22.9	7.3

Table 4.12: Average computation time (M) and point addition number (#ADD) for $n = 160$ bits scalar multiplications with the randomized selection function.

n	exact division starting from	w	area slices (FF/LUT)	freq. MHz
160	LSW	12	161 (326/466)	236
		24	338 (724/1 005)	202
	MSW	12	167 (350/465)	261
		24	339 (781/1 015)	206
256	LSW	12	168 (329/469)	236
		24	343 (727/1 008)	202
	MSW	12	189 (369/502)	261
		24	401 (795/1 112)	206

Table 4.13: FPGA implementation results for complete signed recoding unit with $\mathcal{B} = (2, 3, 5, 7)$.

2012.06-SP5. The standard cells used were restricted to a set {nand2, nor2, xor2, inv} of logic gates without loss of generality.

Below, figure 4.15 reports a pie chart which illustrates the area in ASIC for each unit (figure 4.10) of the MBNS recoding, with $n \in \{160, 256\}$ and $w = 12$, and with a maximum path delay constraint of 10ns.

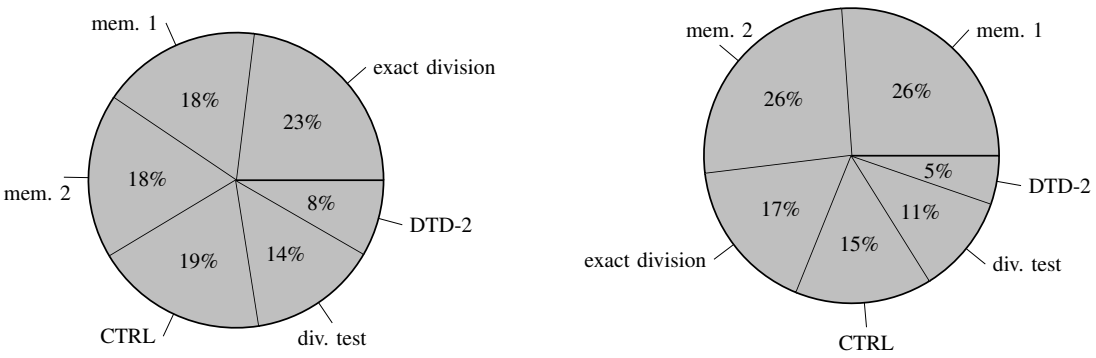


Figure 4.15: Pie chart of the ASIC area with $w = 12$ (right: $n = 160$, left: $n = 256$).

n	delay	w	combinational	buf/inv	non combinational	total
160	10	12	12 072.9	919.8	22 643.0	34 716.0
		24	26 364.8	2 469.0	35 478.5	61 843.3
	5	12	10 902.9	912.8	12 594.0	28 496.9
		24	26 756.1	2 202.7	35 478.5	62 234.6
256	10	12	16 026.6	1 341.4	30 334.6	46 361.3
		24	25 336.0	2 628.4	44 277.5	69 613.5
	5	12	13 959.0	1 218.3	29 753.7	29 753.7
		24	26 672.9	2 366.1	44 277.5	69 950.4

Table 4.14: Area results (in μm^2) of ASIC implementation for complete signed recoding unit with $\mathcal{B} = (2, 3, 5, 7)$.

n	delay	w	cell internal	net switching	total dynamic	cell leakage
160	10	12	1 425.3	166.9	1 592.3	12.2
		24	2 445.6	307.0	2 752.7	21.0
	5	12	2 255.6	267.6	2 523.3	10.1
		24	5 044.9	656.4	5 701.4	21.6
256	10	12	1 854.2	183.2	2 037.4	16.5
		24	2 842.0	310.4	3 152.5	24.2
	5	12	3 579.6	366.0	3 945.7	15.6
		24	5 823.4	661.6	6 485.1	24.7

Table 4.15: Power results in (μW) of ASIC implementation for complete signed recoding unit with $\mathcal{B} = (2, 3, 5, 7)$.

4.3 Comparison to State-of-Art

Below we compare our MBNS recoding and scalar multiplication algorithms for various multi-bases to state-of-art methods. We report results over $\mathbb{F}_p$ for simplified Weierstrass curves ($y^2 = x^3 - ax + b$) with unspecified parameter a and $a = -3$, using Jacobian coordinates, similarly to most of DBNS/MBNS references.

4.3.1 Costs of Curve-Level Operations

Table 4.16 reports computation costs, given in field-level operations (`M`, `S`) for various curve-level operations over $\mathbb{F}_p$ from literature. For multiple publications from a group of authors, we only report the best results. Some costs of curve-level operations have already been presented in chapter 1. EFD is the web site Explicit-Formulas Database `http://hyperelliptic.org/EFD`. λ`DBL` (resp. λ`TPL`) denotes a sequence of λ successive `DBL` (resp. `TPL`) operations (e.g. $k = 2^\lambda$ or $k = 3^\lambda$).

4.3.2 Performance Comparisons

Some previous scalar multiplication algorithms require additional points to speed up computations (see [57, Sec. 3.3.1] for `wNAF`, [7] for DBNS and [77] for MBNS). These additional

curves	references	curve-level operations						
		ADD	mADD	DBL	TPL	QPL	SPL	EPL
$a \neq -3$	EFD	**11M + 5S**	**7M + 4S**	**1M + 8S**	**5M + 10S**	n/a	n/a	n/a
	[40], [41]	12M + 4S	8M + 3S	4M + 6S	10M + 6S	n/a	n/a	n/a
	[87]	12M + 4S	8M + 3S	4M + 6S	10M + 6S	15M + 10S	n/a	n/a
	[78]	**11M + 5S**	**7M + 4S**	2M + 8S	6M + 10S	10M + 14S	17M + 14S	27M + 18S
	[54]	n/a	n/a	**1M + 8S**	**5M + 10S**	**7M + 16S**	15M + 24S	**17M + 30S**
	[76]	**11M + 5S**	**7M + 4S**	2M + 8S	6M + 11S	9M + 15S	**13M + 18S**	n/a
	[75]	12M + 4S	8M + 3S	4M + 6S	9M + 7S	14M + 10S	19M + 12S	29M + 16S
$a = -3$	EFD	**11M + 5S**	**7M + 4S**	**3M + 5S**	**7M + 7S**	n/a	n/a	n/a
	[40], [41]	12M + 4S	8M + 3S	4M + 4S	10M + 6S	n/a	n/a	n/a
	[87]	n/a	n/a	n/a	n/a	15M + 8S	n/a	n/a
	[78]	**11M + 5S**	**7M + 4S**	**3M + 5S**	**7M + 7S**	11M + 11S	18M + 11S	**28M + 15S**
	[77], [76]	**11M + 5S**	**7M + 4S**	**3M + 5S**	7M + 8S	**10M + 12S**	**14M + 15S**	n/a
	[75]	**11M + 5S**	**7M + 4S**	**3M + 5S**	9M + 5S	14M + 8S	19M + 10S	29M + 14S

curves	references	λDBL	λTPL
$a \neq -3$	[40], [41], [61]	4λM + $(4\lambda + 2)$S	$(11\lambda - 1)$M + $(4\lambda + 2)$S
	[87]	4λM + $(4\lambda + 2)$S	10λM + $(6\lambda - 5)$S

curves	references	λTPL / λ'DBL
$a \neq -3$	[40], [41]	$(11\lambda + 4\lambda' - 1)$M + $(4\lambda + 4\lambda' + 3)$S

Table 4.16: Costs of curve-level operations from literature and curves over $\mathbb{F}_p$.

points are multiples of the initial point P and are stored in the cryptoprocessor during the complete scalar multiplication (2 n-bit registers per additional point). For instance wNAF requires the set of additional points $\{[3]P, [5]P, [7]P, \ldots, [2^{w-1} - 1]P\}$ to be pre-computed prior to $[k]P$ operation. Most of methods assume pre-computed points represented using affine coordinates to benefit from fast mixed coordinates addition mADD. Table 4.17 reports some costs of typical pre-computations. Costs at field level include a conversion to affine coordinates which requires field inversions (usually 1M + 1S + 2I per addition point). We assume 1I = 15M for $\mathbb{F}_p$ inversion, and we apply the typical cost assumption used in many references: 1S = 0.8 M.

pre-computations	methods	computation time		
			$a \neq -3$	$a = -3$
$[3]P$	3NAF	1mADD + 1DBL	49.4M	49.0M
	DBNS	1TPL	44.8M	44.4M
$[3]P$, $[5]P$, $[7]P$	4NAF	3mADD + 2DBL	140.8M	140.0M
	DBNS	2mADD + 1DBL + 1TPL	136.2M	135.4M

Table 4.17: Typical costs of pre-computations for additional points.

These costs can be neglected for multiple successive $[k]P$ operations with the same point P, but it is not the case if the point P changes before each scalar multiplication (e.g. support of various protocols/sizes, base point randomization method, etc.).

Table 4.18 compares scalar multiplication methods from literature to our signed MBNS in

terms of performances and pre-computations for curves over $\mathbb{F}_p$, Jacobian coordinates, $a \neq -3$, $n = 160$ bits and 100 000 random scalars. Table 4.19 reports similar comparisons for $a = -3$. Our signed MBNS scalar multiplication method has been evaluated using the `approx` selection function and for the multi-bases $(2,3)$, $(2,3,5)$ and $(2,3,5,7)$. In these tables, DBNS results have been computed using the PARI/GP program kindly provided by the author of [46]. This program generates a signed DBNS chain using pre-computations and where approximations are obtained by a search table.

references	methods	performances	pre-computations		recoding
			storage	operations	
	double-and-add	1 985.3M	$\emptyset$	$\emptyset$	$\emptyset$
	NAF	1 723.0M	$\emptyset$	$\emptyset$	on-the-fly & very cheap
	3NAF	1 583.7M	1 point	49.4M	on-the-fly & very cheap
	4NAF	1 499.1M	3 points	157.8M	on-the-fly & very cheap
[40]	DBNS	1 863.0M	$\emptyset$	$\emptyset$	off-line & costly
[41]	DBNS	1 722.3M	$\emptyset$	$\emptyset$	off-line & costly
[7]	DBNS	1 558.4M	7 points	>150M	off-line & costly
[46]	DBNS	1 615.3M	$\emptyset$	$\emptyset$	off-line & costly
[2]	6HBTF_A	1 803.8	$\emptyset$	$\emptyset$	on-the-fly & small
	6HBTF_B	1 627.2	$\emptyset$	$\emptyset$	off-line & small
	12HBTF_A	1 701.4	1 point	25M	on-the-fly & small
	12HBTF_B	1 555.5	1 point	56.8M	off-line & small
this work	$(2,3)$ MBNS	1 746.2M	$\emptyset$	$\emptyset$	on-the-fly & small
	$(2,3,5)$ MBNS	1 679.9M	$\emptyset$	$\emptyset$	on-the-fly & small
	$(2,3,5,7)$ MBNS	1 670.4M	$\emptyset$	$\emptyset$	on-the-fly & small

Table 4.18: Comparison of scalar multiplication methods (curves with $a \neq -3$ and $n = 160$).

Pre-computation of 7 points for reference [7] is more costly than 4NAF pre-computations (table 4.17). For 3NAF and 4NAF methods, when adding the overhead cost for pre-computing additional points, scalar multiplication computation time raises 1633.1M and 1656.9M, respectively (for $a \neq -3$). Thus, 3NAF method can be considered faster than 4NAF when adding costs of pre-computed points.

Results of the table 4.19 which refer to [77] and [78] are based on a left-to-right MBNS scalar multiplication algorithm to benefit from fast mixed coordinates addition mADD while the scalar is recoded using a right-to-left algorithm (this strategy prevents them from providing an on-the-fly computation). If we use a similar strategy, the computation cost reduction is estimated to $4.8 = \big((11 + 5 \times 0.8) - (7 + 4 \times 0.8)\big)$ times the number of ADD operations. In case $\mathcal{B} = (2,3,5)$ and $n = 160$, this leads to a reduction about 134M. Hence, references [77] and [78] are still faster than our method but with a much smaller difference, but without hardware implementations.

The hybrid binary-ternary number system (HBTNS) [39], a special case of double-base representation, can be used to find a short and sparse representation for a single scalar. The window hybrid binary-ternary form (wHBTF), introduced in [2] is based on HBTNS. Their method could be fully implemented in hardware and thus could work on-the-fly. They give the number of

references	methods	performances	pre-computations		recoding
			storage	operations	
	double-and-add	1 922.0M	∅	∅	∅
	NAF	1 659.7M	∅	∅	on-the-fly & very cheap
	3NAF	1 520.2M	1 point	49.0M	on-the-fly & very cheap
	4NAF	1 436.1M	3 points	156.6M	on-the-fly & very cheap
[46]	DBNS	1 563.2M	∅	∅	off-line & costly
[7]	DBNS	1 504.3M	7 points	>150M	off-line & costly
[87]	(2, 3, 5)MBNS	1 645.4M	∅	∅	off-line & costly
		1 606.4M	1 point	≈40M	off-line & costly
		1 566.4M	3 points	≈150M	off-line & costly
		1 552.3M	7 points	>150M	off-line & costly
		1 486.4M	5 points	>150M	off-line & costly
[78]	(2, 3)NAF	1 514.0M	∅	∅	small
	(2, 3, 5)NAF	1 490.0M	∅	∅	small
	(2, 3, 5, 7)NAF	1 491.0M	∅	∅	small
	(2, 3)NAF_3	1 460.0M	1 point	≈40M	small
	(2, 3, 5)NAF_3	1 444.0M	1 point	≈40M	small
	(2, 3, 5, 7)NAF_3	1 449.0M	1 point	≈40M	small
	(2, 3)NAF_4	1 384.0M	3 points	>150M	small
	(2, 3, 5)NAF_4	1 383.0M	3 points	>150M	small
	(2, 3, 5, 7)NAF_4	1 394.0M	3 points	>150M	small
[77]	(2, 3, 5)NAF	1 460.0M	∅	∅	costly
	(2, 3, 5)NAF	1 426.0M	6 points	>150M	costly
[2]	6HBTF_A	1 752.6	∅	∅	on-the-fly & small
	6HBTF_B	1 576.0	∅	∅	off-line & small
	12HBTF_A	1 647.7	1 point	24.2M	on-the-fly & small
	12HBTF_B	1 501.7	1 point	56M	off-line & small
this work	(2, 3)MBNS	1 686.2M	∅	∅	on-the-fly & small
	(2, 3, 5)MBNS	1 631.0M	∅	∅	on-the-fly & small
	(2, 3, 5, 7)MBNS	1 625.2M	∅	∅	on-the-fly & small

Table 4.19: Comparison of scalar multiplication methods (curves with $a = -3$ and $n = 160$).

curve-level operations, and we have computed the cost of scalar multiplication according to the costs of ADD and mADD, i.e. the recoding is considered on-the-fly or not. That is why we give two results for the methods 6HBTF and 12HBTF according to the recoding: when it is considered to be on-the-fly & small (wHBTF_A) or off-line & small (wHBTF_B).

Figure 4.16 presents the number of curve-level operations for various mutli-bases and DBNS recodings computed using the PARI/GP program from reference [46]. Results correspond to $n = 160$, curve with $a \neq -3$, and the approx selection function.

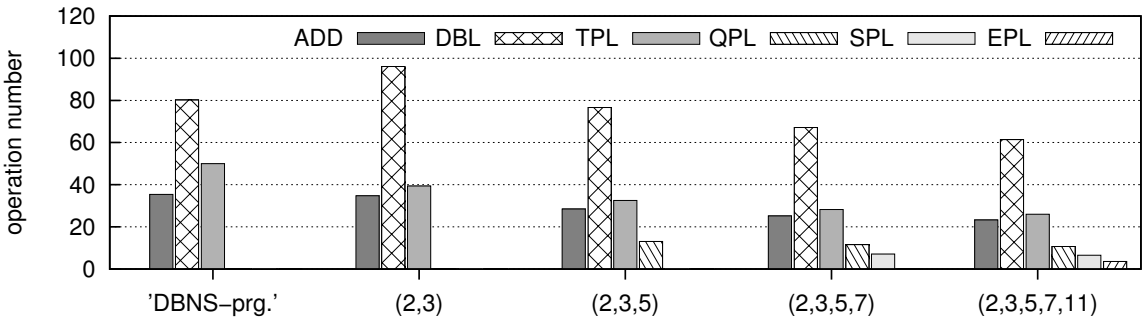


Figure 4.16: Comparison of curve-level operations count.

We obtain similar results with curves provided by the NIST (FIPS 186-2 see [57, appendix A.2.1] for details). Table 4.20 presents the number of curve-level operations of 100 000 scalar multiplications using the `approx` selection function and our signed MBNS algorithm with the different prime fields and bases. The reported experiments have been realized for different curves, with $a = -3$ and $a \neq -3$.

$\mathcal{B}$	NIST curves	curve-level operations	performance with $a \neq -3$	performance with $a = -3$
(2,3)	P-192	41.2ADD + 107.1DBL + 53.7TPL	2 109.9M	2 045.5M
(2,3,5)		33.9ADD + 85.8DBL + 42.1TPL + 17.1QPL	2 030.9M	1 976.5M
(2,3,5,7)		30.1ADD + 75.6DBL + 35.9TPL + 14.6QPL + 9.1SPL	2 020.0M	1 959.7M
(2,3)	P-224	47.9ADD + 124.4DBL + 62.4TPL	2 452.8M	2 378.0M
(2,3,5)		39.5ADD + 99.7DBL + 49.0TPL + 19.9QPL	2 361.3M	2 298.1M
(2,3,5,7)		35.1ADD + 88.0DBL + 41.7TPL + 17.1QPL + 10.6SPL	2 348.9M	2 2278.8M
(2,3)	P-256	54.7ADD + 142.5DBL + 71.4TPL	2 803.3M	2 717.8M
(2,3,5)		45.2ADD + 113.8DBL + 56.1TPL + 22.8QPL	2 700.2M	2 627.7M
(2,3,5,7)		32.4ADD + 95.5DBL + 45.8TPL + 18.6QPL + 15.8SPL	2 575.4M	2 506.4M
(2,3)	P-384	82.3ADD + 214.0DBL + 104.2TPL	4 209.6M	4 081.2M
(2,3,5)		68.0ADD + 170.7DBL + 84.3TPL + 34.0QPL	4 055.4M	3 953.3M
(2,3,5,7)		60.4ADD + 151.0DBL + 71.6TPL + 29.0QPL + 18.0SPL	4 037.7M	3 917.1M
(2,3)	P-521	111.8ADD + 290.1DBL + 145.5TPL	5 714.0M	5 539.8M
(2,3,5)		92.3ADD + 231.8DBL + 114.3TPL + 46.4QPL	5 503.6M	5 363.6M
(2,3,5,7)		82.1ADD + 205.0DBL + 97.2TPL + 39.7QPL + 24.6SPL	5 482.2M	5 318.6M

Table 4.20: Curve-level operations count (NIST curves).

Authors of [47] proposed a similar recoding approach. They use a 4 steps minimum selection function `min4` presented above in section 4.2.1. They recode two scalars in DBNS ($\mathcal{B} = (2, 3)$) to perform two scalar multiplications with joint sparse form method. With inverted Edwards coordinates and $n = 192$, they obtain a performance of 72ADD+107DBL+53TPL curve-level operations. We have similar results for the recoding with $n = 192$. However they use mixed addition when

they produce multiplication costs. It implies that they recode a scalar into DBNS, then compute the result as a sum of terms, and finally perform an adaptation of the standard right-to-left scalar multiplication algorithm. Using this adaptation allow them to have less multiplications (1 952M) by the use of mixed addition. Their limitation is that they must store all terms and wait for the scalar totally recoded to begin scalar multiplication. This strategy prevents them from providing an on-the-fly computation.

4.4 Extended Signed-Digit MBNS Recoding

Above, we proposed a multi-base recoding for ECC scalar multiplication without pre-computation. This section is not a part of the ARITH paper. Several scalar multiplication methods use pre-computations. In this part, we allow to have pre-computations. We want to see what happens at theoretical and practical levels. We want to know if the MBNS recoding method is more efficient with pre-computed points, and how choose them.

A simple modification of the signed-digit MBNS recoding algorithm figure 4.2 is required to others digits than -1 and 1. We denote extended signed-digit MBNS recoding when d can have other digits than -1 and 1. The selection function S on figure 4.12 selects the digit d to be used. For instance, one can choose d such that $d = \pm 3$. At curve level, digit values always correspond to: the point addition $Q + [3]P$ when $d = 3$, the point subtraction $Q - [3]P$ when $d = -3$, and no operation when $d = 0$.

Thus points must be pre-computed when S selects the digit d such that $d \neq \pm 1$ and $d \neq 0$. For instance $P_{|-3|} = P_3 = [3]P$ when $d = \pm 3$. Line 3 of the scalar multiplication algorithm on figure 4.3 becomes $Q \leftarrow Q + sign(d) \times P_{|d|}$. It is the same strategy used by the `wNAF` method. The scalar multiplication algorithm performs point addition in Jacobian, and thus pre-computed points do not require field inversions. For instance, pre-computing the point $[3]P$ costs 13M with $a = -3$, and 12.6M with $a \neq -3$, instead of 49.4M and 49.0M with `3NAF` method.

The selection function can select a large set of numbers. Number of pre-computed points depends on the set of d. Let the digit set of d be denoted $\mathcal{D}$. For instance, $\mathcal{D}_{\pm 1}$ denote that $d \in \{\pm 1\}$, $\mathcal{D}_{\pm 1,\pm 3}$ denote that $d \in \{\pm 1, \pm 3\}$, etc. The number of pre-computed points is equal to $\#\{|d_i|\} \backslash \{\pm 1, 0\}$. We recall that $d = 0$ only for the very first term when the initial k is divisible by, at least, one base element in $\mathcal{B}$. The value ± 1 is not considered because it corresponds to the point P. For instance, one precomputed point must be calculated for $\mathcal{D}_{\pm 3}$, $\mathcal{D}_{\pm 5}$, ..., $\mathcal{D}_{\pm 1,\pm 3}$, $\mathcal{D}_{\pm 1,\pm 5}$, etc. Similarly to the way, two precomputed points must be calculated when $\mathcal{D}_{\pm 3,\pm 5}$, $\mathcal{D}_{\pm 1,3,\pm 5}$, $\mathcal{D}_{\pm 1,\pm 5,-7}$, etc.

From now, we consider that $\mathcal{D}_{d_i}$ corresponds to $\mathcal{D}_{\pm d_i}$. Indeed, when one considers the pre-computed point $[3]P$, having $d_i = \pm 3$ can enable to have a better approximation of k, with only one pre-computed point. In addition, the elements of d must be chosen with attention.

First, one always should have odd elements in d. Indeed, when k is not divisible by the multi-base elements, k plus or minus an even element will be not divisible by at least $b_1 = 2$.

Second, one should have all $|d_i| \notin \mathcal{B}$. The reasons for this choice are the same than previously: if k is not divisible by the multi-base elements, the computation of $k \pm b_j$ cannot be divisible by b_j. For instance, if $d_i = 3$ and k is not divisible by the multi-base elements, one has $k \bmod 3 \neq 0$. Thus the number $k \pm 3$ wont be divisible by the base 3. Having $d_i = \pm 3$ prevents the possibility from being divisible by 3 with $b_2 = 3$. Therefore $\mathcal{D}$ should be a set of prime numbers different

from the multi-base. For instance, when $\mathcal{B} = (2, 3)$, it may be not interesting to have $\pm 3 \in \mathcal{D}$.

Third, the values -1, 1, the prime number immediately greater than b_l or his opposite must be in d (b_l is considered to be the maximum value of the multi-base $\mathcal{B}$). Else, k may never be 1 when one computes $k \leftarrow k \pm d_i$: in this case, the recoding algorithm never stops. For instance, when $\mathcal{B} = (2, 3)$, d_i can take values in $\mathcal{D}_1$, $\mathcal{D}_{1,5}$, $\mathcal{D}_{1,7}$, ..., $\mathcal{D}_5$, $\mathcal{D}_{5,7}$, $\mathcal{D}_{5,7,11}$, etc. For instance, if the digit set is $\mathcal{D}_7$ or $\mathcal{D}_{7,11}$, the recoding algorithm can never stop if k reaches the value 5.

Once one chose the appropriate set of $\mathcal{D}$, the selection function S selects between all values $k - d_i$ when k is not divisible by any multi-base elements. When the set $\mathcal{D}$ is big, one could think that the selection function becomes too complex to evaluate. For instance, S can choose the values between $k - 5$, $k - 1$, $k + 1$ and $k + 5$ with $\mathcal{D}_{1,5}$. However the controller does not become more complex and unrealistic in hardware.

Indeed, the best value $k - d_i$ can be defined during a curve-level operation (`ADD`, `DBL`, `TPL`, etc.). These operations require a sequence of 8–12 field-level operations, and each field requires tens (for large operators) to hundreds (for small iterative operators) of clock cycles (see figure 1.3 in section 1.1.7 for the typical number of operations required at each level).

Thus, S just needed to choose between the first two value of $\mathcal{D}$, and selecting one. Then, S selects between the selected value and the third value of $\mathcal{D}$ to select the best one. This process is realized iteratively until S scanned all values of $\mathcal{D}$. That is why S is not more complex with a bigger set $\mathcal{D}$. In addition, there is no need to have more than the two initial registers with this strategy. Only two registers are necessary to compute and choose between the two values $k - d_i$ and $k - d_j$.

The selection function S works during the curve-level operations. Thus, the more S is used, the more scalar multiplication is fast. Indeed, S will have more time to select the best value. Moreover, the fact that S works during the curve-level operations enables to add noise in the power traces. Thus, it could be more difficult for an attacker to have clean traces and thus to guess the scalar.

4.4.1 Implementation Results

The Signed MBNS recoding algorithm with several pre-computed points has been implemented on FPGA (see end of section 4.1.2 for target and tools details). In theory, a ROM is implemented to store the pre-computed points.

Our implementation launches the curve-level operations with the corresponding point (e.g. P, $[3]P$, etc.) to be computed. Each pre-computed point does not change during the scalar multiplication. Thus in practice, FPGA compilation tools consider each pre-computed point like constants during circuit operation. Registers are replaced by logic. Therefore FPGA implementations are practically the same than in table 4.13, which reports the signed MBNS recoding implementation results for `approx` selection function, $\mathcal{B} = (2, 3, 5, 7)$, $n \in \{160, 256\}$ bits and the two versions of exact division.

The version with 1 (resp. 5, 10) pre-computed point requires 1% more area (resp. 3%, 4%) compared to area results of ASIC implementation without pre-computations (table 4.14). The version with 1 (resp. 5, 10) pre-computed point has 10% more total dynamic power (resp. 18%, 23%) compared to power results of ASIC implementation without pre-computations (table 4.15).

4.4.2 Performance

Table 4.21 reports some costs of point pre-computations. Costs at field level do not include a conversion to affine coordinates which requires field inversions. Indeed, the considered scalar multiplication does not use mixed additions. There are two costs for the pre-computation of $[11]P$, because when $5 \in \mathcal{B}$, it can be more interesting to use the operation sequence of `QPL` (only when $a \neq -3$). Else, only the `mADD`, `ADD`, `DBL` and `TPL` formulas are necessary to pre-compute the points in table 4.21. For these points, it is sufficient that 2 and 3 are in the multi-base. In practice, it is always considered.

pre-computations	computation time		
		$a \neq -3$	$a = -3$
$[5]P$	1mADD + 2DBL	25.0M	24.2M
$[7]P$	1mADD + 3DBL	32.4M	31.2M
$[11]P$	1mADD + 2DBL + 1TPL	38.0M	34.4M
$[11]P$	1mADD + 1DBL + 1QPL	37.4M	36.8M
$[13]P$	1mADD + 2DBL + 1TPL	38.0M	34.4M
$[17]P$	1mADD + 2DBL	39.8M	38.2M
$[5]P$, $[7]P$	2mADD + 1DBL + 1TPL	40.8M	40.0M
$[5]P$, $[11]P$	2mADD + 2DBL + 1TPL	48.2M	44.6M
$[5]P$, $[13]P$	2mADD + 2DBL + 1TPL	48.2M	44.6M
$[5]P$, $[17]P$	2mADD + 4DBL	50.0M	48.4M
$[7]P$, $[11]P$	1mADD + 1ADD + 3DBL + 1TPL	60.4M	56.4M
$[7]P$, $[13]P$	2mADD + 3DBL + 1TPL	55.6M	51.6M
$[7]P$, $[17]P$	2mADD + 4DBL	50.0M	48.4M
$[11]P$, $[13]P$	2mADD + 2DBL + 1TPL	48.2M	47M
$[11]P$, $[17]P$	1mADD + 1ADD + 4DBL + 1TPL	67.8M	63.4M

Table 4.21: Typical costs of pre-computations for additional points.

Table 4.23 and table 4.24 compares scalar multiplication methods with the extended MBNS method with different pre-computed points for curves over $\mathbb{F}_p$, Jacobian coordinates, $n = 160$ bits, 100 000 random scalars, $a \neq -3$ and $a = -3$. The extended MBNS scalar multiplication method has been evaluated using `approx` selection function and for multi-bases $(2, 3)$, $(2, 3, 5)$ and $(2, 3, 5, 7)$. These two tables use a method which can be fully embedded in hardware, work on-the-fly. In addition, this method requires a small area.

We can see in table 4.23 and table 4.24 that the more pre-computed points there are, the more interesting it is. In addition, results are quite similar for a same number of the set $\mathcal{D}$. Thus, it can be not interesting to have pre-computed points which have a bigger cost of pre-computations. One has just to select pre-computed points with the lower cost.

We obtain similar results with curves provided by the NIST. Table 4.22 presents the number of curve-level operations of 100 000 scalar multiplications using the signed MBNS algorithm with the different provided prime fields and bases, and with one pre-computed point. The reported experiments have been realized for different curves, with the `approx` selection function, $a = -3$

and $a \neq -3$.

$\mathcal{B}$	NIST curves	$\mathcal{D}$	curve-level operations	performance with $a = -3$	performance with $a \neq -3$
(2,3)	P-192	$\mathcal{D}_{1,5}$	32.4ADD + 106.1DBL + 54.5TPL	1 915.1 M	1 979.3 M
(2,3,5)		$\mathcal{D}_{1,7}$	27.1ADD + 78.1DBL + 41.5TPL + 20.9QPL	1 886.0 M	1 938.0 M
(2,3,5,7)		$\mathcal{D}_{1,11}$	24.4ADD + 71.9DBL + 34.5TPL + 14.0QPL + 11.9SPL	1 887.6 M	1 949.6 M
(2,3)	P-224	$\mathcal{D}_{1,5}$	37.6ADD + 123.3DBL + 63.4TPL	2 226.1 M	2 300.8 M
(2,3,5)		$\mathcal{D}_{1,7}$	31.5ADD + 90.7DBL + 48.3TPL + 24.3QPL	2 192.5 M	2 253.0 M
(2,3,5,7)		$\mathcal{D}_{1,11}$	28.3ADD + 83.6DBL + 40.0TPL + 16.3QPL + 13.8SPL	2 194.2 M	2 266.3 M
(2,3)	P-256	$\mathcal{D}_{1,5}$	43.0ADD + 141.0DBL + 72.4TPL	2 544.5M	2 629.9M
(2,3,5)		$\mathcal{D}_{1,7}$	36.0ADD + 103.7DBL + 55.TPL + 27.8QPL	2 506.2M	2 575.4M
(2,3,5,7)		$\mathcal{D}_{1,11}$	32.4ADD + 95.5DBL + 45.8TPL + 18.6QPL + 15.8SPL	2 508.4M	2 590.8M
(2,3)	P-384	$\mathcal{D}_{1,5}$	65.1ADD + 214.3DBL + 107.2TPL	3 827.1 M	3 955.7 M
(2,3,5)		$\mathcal{D}_{1,7}$	54.3ADD + 154.8DBL + 83.4TPL + 41.9QPL	3768.7 M	3 872.3 M
(2,3,5,7)		$\mathcal{D}_{1,11}$	48.7ADD + 142.8DBL + 69.0TPL + 28.0QPL + 23.9SPL	3 769.5 M	3 893.3 M
(2,3)	P-521	$\mathcal{D}_{1,5}$	87.8ADD + 287.7DBL + 147.5TPL	5 189.2 M	5 363.3 M
(2,3,5)		$\mathcal{D}_{1,7}$	73.5ADD + 211.2DBL + 112.7TPL + 56.7QPL	5 112.8 M	5 253.6 M
(2,3,5,7)		$\mathcal{D}_{1,11}$	66.3ADD + 194.3DBL + 93.4TPL + 38.1QPL + 32.4SPL	5 118.0 M	5 285.9 M

Table 4.22: Curve-level operations count with one pre-computed point (NIST curves).

$\mathcal{B}$	$\mathcal{D}$	curve-level operations	performance with $a = -3$	performance with $a \neq -3$
$(2,3)$	$\mathcal{D}_1$	34.1ADD + 89.0DBL + 44.6TPL	1 686.2 M	1 746.2 M
	$\mathcal{D}_5$	34.2ADD + 88.9DBL + 44.4TPL	1 694.1 M	1 747.4 M
	$\mathcal{D}_{1,5}$	26.8ADD + 88.0DBL + 45.2TPL	1 588.9 M	1 642.2 M
	$\mathcal{D}_{1,7}$	27.6ADD + 83.4DBL + 48.3TPL	1 605.5 M	1 658.2 M
	$\mathcal{D}_{1,11}$	27.5ADD + 90.3DBL + 44.0TPL	1 598.9 M	1 652.6 M
	$\mathcal{D}_{1,13}$	27.5ADD + 88.3DBL + 45.2TPL	1 599.5 M	1 652.9 M
	$\mathcal{D}_{1,17}$	28.9ADD + 88.1DBL + 45.3TPL	1 621.1 M	1 674.5 M
	$\mathcal{D}_{5,7}$	27.6ADD + 88.5DBL + 44.7TPL	1 597.3 M	1 650.6 M
	$\mathcal{D}_{5,11}$	27.7ADD + 80.7DBL + 49.6TPL	1 606.5 M	1 658.6 M
	$\mathcal{D}_{5,13}$	28.2ADD + 90.3DBL + 43.6TPL	1 604.4 M	1 657.9 M
	$\mathcal{D}_{5,17}$	27.3ADD + 88.7DBL + 44.6TPL	1 592.6 M	1 645.9 M
	$\mathcal{D}_{1,5,7}$	24.3ADD + 84.2DBL + 47.8TPL	1 556.4 M	1 609.2 M
	$\mathcal{D}_{1,5,11}$	24.3ADD + 84.1DBL + 47.9TPL	1 556.8 M	1 609.6 M
	$\mathcal{D}_{1,5,13}$	24.2ADD + 89.1DBL + 44.7TPL	1 550.9 M	1 604.4 M
	$\mathcal{D}_{1,5,17}$	24.8ADD + 87.5DBL + 45.7TPL	1 560.5 M	1 613.8 M
	$\mathcal{D}_{1,7,11}$	24.8ADD + 91.1DBL + 43.5TPL	1 557.7 M	1 611.5 M
	$\mathcal{D}_{1,7,13}$	24.3ADD + 83.3DBL + 48.3TPL	1 556.5 M	1 609.2 M
	$\mathcal{D}_{1,7,17}$	25.8ADD + 83.1DBL + 48.6TPL	1 579.9 M	1 632.6 M
	$\mathcal{D}_{1,11,13}$	24.7ADD + 81.2DBL + 49.8TPL	1 566.4 M	1 618.8 M
	$\mathcal{D}_{1,11,17}$	24.8ADD + 88.6DBL + 45.1TPL	1 560.9 M	1 614.4 M
	$\mathcal{D}_{1,13,17}$	24.8ADD + 87.3DBL + 46.0TPL	1 561.5 M	1 614.8 M
	$\mathcal{D}_{5,7,11}$	25.0ADD + 87.4DBL + 45.5TPL	1 559.7 M	1 612.9 M
	$\mathcal{D}_{5,7,13}$	24.6ADD + 88.8DBL + 44.6TPL	1 552.5 M	1 605.8 M
	$\mathcal{D}_{5,7,17}$	24.6ADD + 82.1DBL + 48.9TPL	1 559.4 M	1 611.8 M
	$\mathcal{D}_{5,11,13}$	25.4ADD + 82.5DBL + 48.6TPL	1 570.7 M	1 623.2 M
	$\mathcal{D}_{5,11,17}$	24.3ADD + 82.7DBL + 48.4TPL	1 553.5 M	1 605.9 M
	$\mathcal{D}_{5,13,17}$	24.5ADD + 89.0DBL + 44.5TPL	1 550.8 M	1 604.2 M

Table 4.23: Comparison of scalar multiplication methods ($\mathcal{B} = (2,3)$, curves with $n = 160$).

$\mathcal{B}$	$\mathcal{D}$	curve-level operations	performance with $a=-3$	performance with $a\neq -3$
$(2,3,5)$	$\mathcal{D}_1$	$28.1\mathtt{ADD}+71.2\mathtt{DBL}+35.0\mathtt{TPL}+14.2\mathtt{QPL}$	1 631.0 M	1 679.9 M
	$\mathcal{D}_7$	$28.1\mathtt{ADD}+70.7\mathtt{DBL}+35.0\mathtt{TPL}+14.3\mathtt{QPL}$	1 637.4 M	1 682.5 M
	$\mathcal{D}_{1,7}$	$22.4\mathtt{ADD}+64.9\mathtt{DBL}+34.5\mathtt{TPL}+17.3\mathtt{QPL}$	1 564.7 M	1 607.9 M
	$\mathcal{D}_{1,11}$	$23.0\mathtt{ADD}+71.1\mathtt{DBL}+35.4\mathtt{TPL}+14.1\mathtt{QPL}$	1 565.2 M	1 610.6 M
	$\mathcal{D}_{1,13}$	$22.5\mathtt{ADD}+67.5\mathtt{DBL}+31.2\mathtt{TPL}+18.5\mathtt{QPL}$	1 565.2 M	1 608.4 M
	$\mathcal{D}_{1,17}$	$23.3\mathtt{ADD}+64.2\mathtt{DBL}+31.1\mathtt{TPL}+20.0\mathtt{QPL}$	1 582.8 M	1 624.9 M
	$\mathcal{D}_{7,11}$	$22.6\mathtt{ADD}+70.2\mathtt{DBL}+30.4\mathtt{TPL}+17.7\mathtt{QPL}$	1 560.0 M	1 603.8 M
	$\mathcal{D}_{7,13}$	$22.8\mathtt{ADD}+70.1\mathtt{DBL}+36.2\mathtt{TPL}+13.7\mathtt{QPL}$	1 559.1 M	1 604.4 M
	$\mathcal{D}_{7,17}$	$23.3\mathtt{ADD}+65.9\mathtt{DBL}+37.7\mathtt{TPL}+14.6\mathtt{QPL}$	1 571.2 M	1 615.6 M
	$\mathcal{D}_{1,7,11}$	$20.3\mathtt{ADD}+67.1\mathtt{DBL}+32.9\mathtt{TPL}+17.5\mathtt{QPL}$	1 531.4 M	1 575.0 M
	$\mathcal{D}_{1,7,13}$	$20.2\mathtt{ADD}+65.2\mathtt{DBL}+34.8\mathtt{TPL}+17.0\mathtt{QPL}$	1 532.2 M	1 575.7 M
	$\mathcal{D}_{1,7,17}$	$20.6\mathtt{ADD}+62.5\mathtt{DBL}+34.7\mathtt{TPL}+18.3\mathtt{QPL}$	1 542.3 M	1 584.9 M
	$\mathcal{D}_{1,11,13}$	$20.3\mathtt{ADD}+64.9\mathtt{DBL}+33.0\mathtt{TPL}+18.4\mathtt{QPL}$	1 535.5 M	1 578.4 M
	$\mathcal{D}_{1,11,17}$	$20.4\mathtt{ADD}+65.6\mathtt{DBL}+33.4\mathtt{TPL}+17.8\mathtt{QPL}$	1 535.9 M	1 579.1 M
	$\mathcal{D}_{1,13,17}$	$20.7\mathtt{ADD}+67.4\mathtt{DBL}+32.0\mathtt{TPL}+18.0\mathtt{QPL}$	1 539.3 M	1 582.7 M
	$\mathcal{D}_{7,11,13}$	$20.4\mathtt{ADD}+66.8\mathtt{DBL}+32.7\mathtt{TPL}+17.6\mathtt{QPL}$	1 531.0 M	1 574.3 M
	$\mathcal{D}_{7,11,17}$	$20.3\mathtt{ADD}+66.7\mathtt{DBL}+32.1\mathtt{TPL}+17.9\mathtt{QPL}$	1 529.7 M	1 573.0 M
	$\mathcal{D}_{7,13,17}$	$20.7\mathtt{ADD}+67.4\mathtt{DBL}+37.9\mathtt{TPL}+13.8\mathtt{QPL}$	1 530.6 M	1 575.4 M
$(2,3,5,7)$	$\mathcal{D}_1$	$25.0\mathtt{ADD}+62.8\mathtt{DBL}+29.8\mathtt{TPL}+12.2\mathtt{QPL}+7.6\mathtt{SPL}$	1 625.2 M	1 670.4 M
	$\mathcal{D}_{11}$	$25.0\mathtt{ADD}+62.7\mathtt{DBL}+29.7\mathtt{TPL}+12.2\mathtt{QPL}+7.6\mathtt{SPL}$	1 624.0 M	1 674.1 M
	$\mathcal{D}_{1,11}$	$20.2\mathtt{ADD}+59.7\mathtt{DBL}+28.6\mathtt{TPL}+11.6\mathtt{QPL}+9.9\mathtt{SPL}$	1 565.8 M	1 617.3 M
	$\mathcal{D}_{1,13}$	$20.2\mathtt{ADD}+59.5\mathtt{DBL}+28.0\mathtt{TPL}+15.6\mathtt{QPL}+7.0\mathtt{SPL}$	1 560.9 M	1 608.9 M
	$\mathcal{D}_{1,17}$	$20.2\mathtt{ADD}+54.4\mathtt{DBL}+25.9\mathtt{TPL}+15.8\mathtt{QPL}+9.9\mathtt{SPL}$	1 577.7 M	1 626.9 M
	$\mathcal{D}_{11,13}$	$19.9\mathtt{ADD}+53.9\mathtt{DBL}+27.6\mathtt{TPL}+15.0\mathtt{QPL}+9.7\mathtt{SPL}$	1 568.0 M	1 617.1 M
	$\mathcal{D}_{11,17}$	$20.0\mathtt{ADD}+59.2\mathtt{DBL}+29.1\mathtt{TPL}+14.8\mathtt{QPL}+7.0\mathtt{SPL}$	1 555.4 M	1 603.6 M
	$\mathcal{D}_{1,11,13}$	$18.1\mathtt{ADD}+55.4\mathtt{DBL}+28.3\mathtt{TPL}+14.7\mathtt{QPL}+9.1\mathtt{SPL}$	1 539.4 M	1 588.5 M
	$\mathcal{D}_{1,11,17}$	$18.1\mathtt{ADD}+55.6\mathtt{DBL}+27.7\mathtt{TPL}+14.8\mathtt{QPL}+9.2\mathtt{SPL}$	1 541.3 M	1 590.6 M
	$\mathcal{D}_{1,13,17}$	$18.3\mathtt{ADD}+56.7\mathtt{DBL}+27.0\mathtt{TPL}+14.7\mathtt{QPL}+9.3\mathtt{SPL}$	1 542.6 M	1 592.1 M
	$\mathcal{D}_{11,13,17}$	$18.0\mathtt{ADD}+56.0\mathtt{DBL}+28.6\mathtt{TPL}+14.2\mathtt{QPL}+9.0\mathtt{SPL}$	1 534.7 M	1 583.9 M

Table 4.24: Comparison of scalar multiplication methods ($\mathcal{B}\in\{(2,3,5),(2,3,5,7)\}$, curves with $n=160$).

4.5 Conclusion

In this chapter, we proposed a simple multi-base recoding algorithm which can be fully implemented in hardware without any pre-computations for ECC scalar multiplication. The scalar recoding is performed on-the-fly and in parallel to curve-level operations without additional latency. The proposed recoding circuit uses cheap divisibility test by multi-base elements (e.g. $(2,3,5,7)$) and exact division using very small dedicated hardware units. Whereas we proposed and used only "classic" algorithms, we provide the first complete hardware implementation. Thus, DBNS/MBNS methods can be used in hardware.

Our MBNS recoding and scalar multiplication method is a little less competitive compared to other DBNS/MBNS methods when off-line recoding can be used. When we allow us to have pre-computations, our MBNS recoding can always be a little less competitive. In addition, pre-computation costs can be neglected for multiple successive $[k]P$ operations with the same P. Thus, MBNS recoding with pre-computations is clearly justified.

Whereas our method is a little less competitive, our method leads to more efficient solutions in embedded applications fully integrated in hardware without resources for costly recoding and limited storage.

As future work, we plan to deal with more advanced recoding schemes to reduce the number of produced terms, and with more statistical results for a better set of the output of S. Indeed, most of the time is spent in curve-level operations. Thus, all units can be used until that the current curve-level operation is finished to find a better decomposition of the scalar. For instance, why not adding ± 1 to k even if k is divisible by a multi-base element? All these possibilities can improved more randomization schemes to increase robustness against side channel attacks.

Chapter 5

Atomic Blocks through Regular Algorithms

In this chapter, we use existing scalar multiplication algorithms in a different way. These algorithms can have a regular behaviour such that they can provide a countermeasure against some simple side-channel attacks. In addition, we work at the field level to produce a regular sequence of operations, that is atomic blocks. Randomization can be added in the sequences. Again, it can be used as a countermeasure against some differential attacks. Hardware implementations of the proposed solutions are provided in this chapter.

When scalar multiplication algorithms are considered with least significant bits first, i.e. right-to-left algorithms, some curve-level operations can be computed in parallel. We use existing algorithms and perform them in other way. Figure 5.1 presents two right-to-left methods to compute $[k]P$: double-and-add (on the left) and double-and-add-always (on the right) algorithms. The point P is updated at the end of the computation of Q' on the right algorithm.

However as the below algorithms start with least significant bits first, point additions with mixed coordinates (`mADD`) cannot be used. Standard point addition are obliged to be used, which is a little slower.

input: $k = (k_{n-1}k_{n-2}\dots k_1k_0)_2$, $P \in E(\mathbb{F}_p)$

output: $Q = [k]P$

1:	$Q \longleftarrow \mathcal{O}$	$Q \longleftarrow \mathcal{O}$
2:	**for** i **from** 0 **to** $n-1$ **do**	**for** i **from** 0 **to** $n-1$ **do**
3:	**if** $k_i = 1$ **then** $Q \longleftarrow Q + P$	$Q' \leftarrow Q + P$ // $P \leftarrow [2]P$
4:	$P \longleftarrow [2]P$	**if** $k_i = 1$ **then** $Q \leftarrow Q'$
5:	**return** Q	**return** Q

Figure 5.1: Right-to-left scalar multiplication algorithms.

The number of curve-level operations of the double-and-add-always algorithm is n `ADD` + n `DBL`. Indeed, one `ADD` and one `DBL` are always performed at each loop iteration. The two curve-level operations are performed at line 3.

However point additions and point doublings can be performed at the same time, with a multiplexer controlling the output of `ADD`. In this case, when one considers real execution time

of the right algorithm in figure 5.1 the cost is reduced to n `ADD` if `ADD` and `DBL` (at line 3) are computed in parallel, since a `DBL` is usually faster than an `ADD`.

Other scalar multiplication algorithms can use such a parallelization scheme. Thus, real execution time is smaller than their corresponding sequential costs. For instance, the Montgomery ladder [90] algorithm can use such a parallelization scheme. The idea of this method is that the difference between the 2 points P and Q at each loop iteration is constant and corresponds to the initial point P. The real execution time of the Montgomery ladder algorithm is reduced to $(n-1)$ `ADD` when `ADD` and `DBL` are performed in parallel.

Other scalar multiplication algorithms can use this parallelisation scheme. `NAF` has a representation where at most one digit $k_i \in \{-1, 0, 1\}$ is non-zero over two consecutive digits. So, an `ADD` can be performed in parallel to 2 `DBL`s. The algorithm in figure 5.2 presents the `NAF` method for scalar multiplication starting least significant digit first, also called the right-to-left scalar multiplication using `NAF` method. The `NAF` length of k is at most one more digit than the length of the binary representation of k: $k = (k_n k_{n-1} \dots k_1 k_0)_2$ where $k_i \in \{0, \pm 1\}$. Usually, `NAF` method for scalar multiplication starts most significant digit first (i.e. left-to-right).

input: $P \in E(\mathbb{F}_p)$, and $k = (k_n k_{n-1} \dots k_1 k_0)_2$ where $k_i \in \{0, \pm 1\}$ and $k_{i+1} k_i = 0 \quad \forall i$
output: $Q = [k]P$

1: $Q \leftarrow \mathcal{O}$
2: **for** i **from** 0 **to** n **do**
3: **if** $k_i \neq 0$ **then**
4: $Q \leftarrow Q + k_i P$
5: $P \leftarrow [2]P$
6: **return** Q

Figure 5.2: Right-to-left `NAF` method for scalar multiplication.

When one starts least significant digit first, lines 4 and 5 of the algorithm in figure 5.2 can be performed in parallel. Obviously, reading and writing the point P cannot be an option. Performing in parallel the two aforementioned lines does not involve reading and writing at the same time. Indeed, the first step of the lines 4 and 5 is to read the point P. So, when the point doubling at line 5 is calculated, the new point P can be stored without any problem. At each loop iteration the register which contains the point P is sent to the two curve-level operations `ADD` and `DBL`. At the end of `DBL`, the register P is updated.

In addition, one can use the inherent property of the `NAF` method, that is no two consecutive signed digits $k_i \in \{-1, 0, 1\}$ are non-zero. Both algorithms in figure 5.2 and 5.3 compute scalar multiplication using `NAF` method. The algorithm in figure 5.3 presents another way to use this `NAF` property. Each computation at line 3 can be performed in parallel.

We can remark that the scalar k can be negative at the loop iteration n' by beginning least significant digit first. When one considers the first n' bits of k, $(k_{n'-1} \cdots k_0)$, then one can have: $\left(\sum_{i=0}^{n'-1} 2^i k_i \ < \ 0\right)$ with $k_i \in \{-1, 0, 1\}$ and with $n' < n$.

In the algorithm in figure 5.3, the three following operations are computed in parallel:
$P \leftarrow [4]\mathbf{P}$
$Q' \leftarrow Q + k_{2i}\mathbf{P}$
$Q'' \leftarrow Q + k_{2i+1}[2]\mathbf{P}$

input: $P \in E(\mathbb{F}_p)$, and $k = (k_n k_{n-1} \dots k_1 k_0)_2$ where $k_i \in \{0, \pm 1\}$ and $k_{i+1} k_i = 0 \quad \forall i$
output: $Q = [k]P$

1: $Q \leftarrow \mathcal{O}$
2: **for** i **from** 0 **to** $\lfloor n/2 \rfloor$ **do**
3: $\quad P \leftarrow [4]P \quad // \quad Q' \leftarrow Q + k_{2i}P \quad // \quad Q'' \leftarrow Q + k_{2i+1}[2]P$
4: $\quad$ **if** $k_{2i} \neq 0$ **then**
5: $\quad\quad Q \leftarrow Q'$
6: $\quad$ **elsif** $k_{2i+1} \neq 0$ **then**
7: $\quad\quad Q \leftarrow Q''$
8: **return** Q

Figure 5.3: Extended right-to-left `NAF` method for scalar multiplication.

with k is in `NAF` representation ($k_i \in \{\bar{1}, 0, 1\}$). Points $\mathbf{P}$ in bold font are the same value, i.e. the same point. This point is read simultaneously for the three calculations. The output of the calculation $P \leftarrow [4]P$ is updated only at the end of the iteration.

An example with $k = (100\bar{1}0\bar{1}000100000\bar{1}01001000000\bar{1})_2 = (28\,375\,615)_{10}$ is carried out as below. If one denotes P_0 the value of P, the input point of the algorithm in figure 5.3, the columns P, Q, Q' and Q'' represent $[x]P_0$. For instance, for $i = 2$, $P = 64$ is $P = [64]P_0$.

i	P	Q	k_{2i}	Q'	k_{2i+1}	Q''	k'
init.	1	0					intermediate key
0	4	-1	$\bar{1}$	-1	0	0	$(0\bar{1})$
1	16	–	0	–	0	-1	$(000\bar{1})$
2	64	–	0	–	0	–	$(00000\bar{1})$
3	256	63	1	63	0	–	$(01000000\bar{1})$
4	1 024	575	0	–	1	575	$(1001000000\bar{1})$
5	4 096	-1 473	0	575	$\bar{1}$	-1 473	$(\bar{1}01001000000\bar{1})$
6	16 384	–	0	-1 473	0	–	$(00\bar{1}01001000000\bar{1})$
7	65 536	–	0	–	0	–	$(0000\bar{1}01001000000\bar{1})$
8	262 144	64 063	1	64 063	0	–	$(010000\bar{1}01001000000\bar{1})$
9	1 048 576	–	0	–	0	64 063	$(0001000\bar{1}01001000000\bar{1})$
10	4 194 304	-984 513	$\bar{1}$	-984 513	0	–	$(0\bar{1}0001000\bar{1}01001000000\bar{1})$
11	16 777 216	-5 178 817	$\bar{1}$	-5 178 817	0	-984 513	$(0\bar{1}0\bar{1}0001000\bar{1}01001000000\bar{1})$
12	67 108 864	28 375 615	0	–	1	28 375 615	$(100\bar{1}0\bar{1}0001000\bar{1}01001000000\bar{1})$
end		28 375 615					k

Above, all intermediate states are made explicit. We can see that the value of the point Q returned by the algorithm in figure 5.3 is always equal to $[k']P$ (k' is the intermediate key of k).

The sequential equivalent of the algorithm in figure 5.3 for the three operations at line 3 is:
$Q' \leftarrow Q + k_{2i}P$
$P \leftarrow [2]P$
$Q'' \leftarrow Q + k_{2i+1}P$

$P \leftarrow [2]P$

The proposed algorithm is an adaptation of the scalar multiplication with k in `NAF` representation starting from least significant bits (main loop):

for i **from** 0 **to** n **do**
  if $k_i \neq 0$ **then** $Q \leftarrow Q + k_i P$
  $P \leftarrow [2]P$

which can be seen like below (by taking the advantage of the `NAF` representation, i.e. $k_i k_{i+1} = 0 \quad \forall i$):

$i \leftarrow 0$
while $i \leq n$ **do**
  if $k_i \neq 0$ **then**
    $Q \leftarrow Q + k_i P$
    $P \leftarrow [4]P$
    $i \leftarrow i + 2$
  else
    $P \leftarrow [2]P$
    $i \leftarrow i + 1$

The algorithm in figure 5.4 is a new representation of the algorithm in figure 5.3.

input: $P \in E(\mathbb{F}_p)$, and $k = (k_{n-1} \dots k_1 k_0)_2$ where $k_i \in \{0, \pm 1\}$ and $k_{i+1} k_i = 0 \quad \forall i$
output: $Q = [k]P$

1: $Q \leftarrow \mathcal{O}$
2: **for** i **from** 0 **to** $\lfloor n/2 \rfloor$ **do**
3: $P' \leftarrow [2]P$
4: $P'' \leftarrow [2]P' \quad // \quad Q' \leftarrow Q + k_{2i} P \quad // \quad Q'' \leftarrow Q + k_{2i+1} P'$
5: **if** $k_{2i} \neq 0$ **then**
6: $Q \leftarrow Q'$
7: **elsif** $k_{2i+1} \neq 0$ **then**
8: $Q \leftarrow Q''$
9: $P \leftarrow P''$
10: **return** Q

Figure 5.4: Extended right-to-left `NAF` method for scalar multiplication.

In practice, it is not necessary to have registers for the points P', P'', Q' et Q'' (lines 3 and 4 of the algorithm in figure 5.4). Indeed, the algorithm in figure 5.3 shows that the points P' and P'' are not necessary. Besides, either Q' or Q'' must be computed; then the point Q stores the computation according to k_{2i} and k_{2i+1}. Whereas there are two `ADD`s at line 4, only one point addition is computed.

When one computes $Q'' \leftarrow Q + k_{2i+1}[2]P$, the point $[2]P$ is computed during the calculation of $P \leftarrow [4]P$: indeed, $[4]P = [2]([2]P)$. Thus, the calculation of Q'' does not require one `DBL`. This can be seen in figure 5.5.

In this way, whereas the algorithm in figure 5.3 and 5.4 perform 2 `ADD`s, it is possible to

compute only 1 ADD in parallel to 2 DBLs. Thus, it allows to reduce the real execution time to $(\lfloor n/2 \rfloor + 1)$ ADD when the ADD and DBLs are performed in parallel. Figure 5.5 presents the parallelization between two curve-level operations for 2NAF. The very first ADD must begin after 1 DBL. Then, one ADD is performed during 2 DBLs, and an input point is selected among the outputs of the 2 previous DBLs.

This method cannot be extended to vNAF for $v \geq 3$. Indeed, if $v = 3$, a doubling is performed to both the point P and the pre-computed point $P_3 = [3]P$. Thus, we only consider this method with $v = 2$. Whereas wNAF is the classical notation, the term w has been already defined in the previous chapters: let $k = (k_{n-1}k_{n-2}\dots k_1k_0)_2$ be the n-bit *scalar* stored into a t words by w bits memory with $w(t-1) < n \leq wt$ (i.e. last word may be padded using 0s). For $0 \leq i < t$, $k^{(i)}$ denotes the ith word of k starting from least significant.

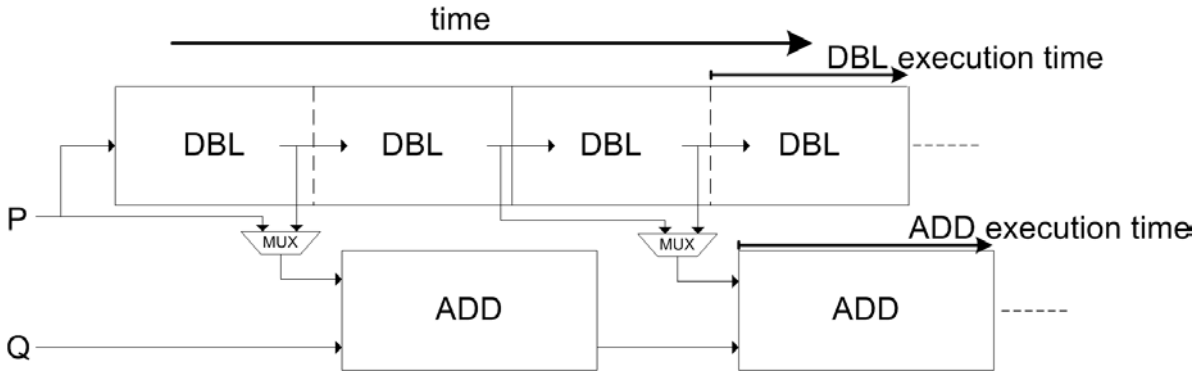


Figure 5.5: Parallelization for the right-to-left scalar multiplication using NAF representation.

When one considers NAF method for scalar multiplication, the real execution time is $(\lfloor n/2 \rfloor + 1)$ ADD if all computations at line 3 in figure 5.3 are computed in parallel. Indeed, an ADD is always performed during 2 DBLs. In addition, the computation of 2 DBLs is generally faster in field-level operations than one ADD (see table 1.3 in section 1.1.6 for typical curve-level costs in Jacobian). If it is not the case, the real execution time is n DBL.

In practice, performing the two curve-level operations ADD and DBL in parallel does not mean that there is a component for an ADD and another for a DBL. Generally, some arithmetic units are implemented, and curve-level operations use and share all arithmetic units. The two curve-level operations ADD and DBL are a sequence of additions, subtractions, multiplications and squares over the field $\mathbb{F}_p$. Parallelizing ADD and DBL implies that several multipliers and adders must be implemented. Thus, the number of implemented arithmetic units is an important issue. Indeed, there is a trade-off between speed and area which is affected by the number of implemented arithmetic units. More arithmetic units will reduce execution time but will increase circuit area. Of course, if a circuit only contains one multiplier and one adder, our method is not relevant. Scheduling methods can be used to schedule resources, i.e. select the order of execution of arithmetic units for the operation sequences ADDs and DBLs.

5.1 Scheduling Sequences

A simple methodology, named *as-soon-as-possible* (ASAP) [120], schedules operations either at the earliest time step possible (latest time step possible is referred to the ALAP methodology). More advanced scheduling algorithms, such as list-based scheduling (LBS) [10], can efficiently

schedule a set of instructions given a set of resource constraints. However here we just focus on the ASAP method. Indeed, we just want to show that sequences of curve-level operations can be considered as a graph, and so that there are several possibilities for scheduling curve-level operations.

ASAP assumes limitless resources. At each time step a number of operations can be scheduled. Results for these operations are not available until the next time step. In an ASAP scheme, the operations are allocated to the earliest possible time step. Figure 5.6 presents the schedule for a point doubling in Jacobian coordinates for curve parameter $a = -3$ (from EFD). The corresponding formula is given below in table 5.1. Each line of figure 5.6 corresponds to one time step, i.e. to the operations which can be performed in parallel. An ASAP approach identifies any operations whose operands are available, and schedules them to the next possible time step. Operations in a same line (same time step) can be computed in parallel. But all operations are considered to be scheduled in a same time step. There is no difference between multiplications and additions. For example, figure 5.6 shows that at least 9 time steps are necessary to perform a `DBL`. All field-level operations are assumed to have a length 1, i.e. to take one time step. In practice, one can consider that it is not the case: for instance, a multiplication can run during several additions.

We can see in figure 5.6 dependencies of each modular operation for a `DBL`. The operation sequences which lead to a `DBL`, can be different according the number of arithmetic units in the circuit. So, there are more possible operation sequences when one considers several `DBL`s with one `ADD`.

5.2 Atomic Scalar Multiplication

Chevallier-Mames et al. [28] introduced the concept of atomicity. Atomic algorithms produce a regular sequence of operations. So `ADD` and `DBL` are expressed as a repetition of instruction blocks which appear equivalent for SPA. Such a block is called *side-channel atomic block*. Atomicity consists then of expressing `ADD` and `DBL` as sequences of atomic patterns.

For example, [28] proposed a pattern constituted by one multiplication, two additions and one negation. When operations are not used, it is possible to compute dummy operations to always have the same operation sequences. From the execution of the scalar multiplication algorithm, one may observe the number of performed atomic patterns but cannot distinguish the process scalar bits. In [75, App. B] and [55], other atomic pattern improvement methods have been proposed. They minimize the number of required field-level operations by the use of Jacobian coordinates and by introducing the use of squaring, which is generally cheaper than multiplication: a square is often considered to be equivalent to 0.8 multiplication over $\mathbb{F}_p$.

Unified formulas [51] [18] are a type of atomicity by providing a same formula for both `ADD` and `DBL` sequences. They render an `ADD` indistinguishable from a `DBL`. Using these formulas, one may observe the number of operations but cannot differentiate between the processed scalar bits, assuming that an `ADD` and a `DBL` are indistinguishable. Thus, it can be used as a countermeasure against SPA.

Our solution is to use parallelization of right-to-left algorithms, and thus to provide a sequence of field-level operations for the computation of one `ADD` and v `DBL`s with the v`NAF` method. In figure 5.6, we can see that, when one implements less than 2 multipliers, and less than 4 adders, ASAP scheduling for point doubling shows that there are several operation sequences for the `DBL` computation. We consider that there is at least one multiplier and one adder implemented. If not,

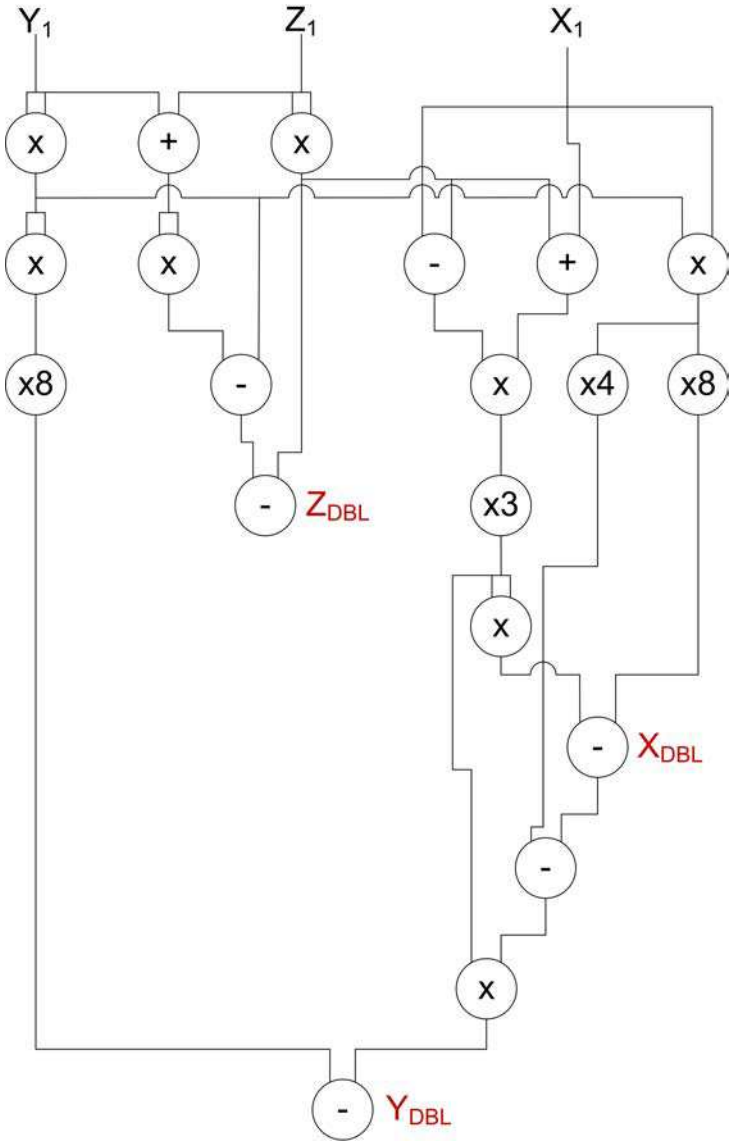


Figure 5.6: ASAP schedule for point doubling (Jacobian coordinates and $a = -3$).

no parallelization can be performed, and our solution is not relevant. So, we can already guess that, when one computes 2 DBLs and one ADD in parallel, there are several operation sequences corresponding to a number of multipliers and adders.

A Python script has been written which provides all possible operation sequences for a generic number of arithmetic units. The Python script takes in input formulas of DBL and ADD, the number of multipliers and adders implemented, and the number of DBLs to perform in parallel to one ADD. For instance, the right-to-left NAF algorithm in figure 5.3 computes in parallel 2 DBLs and 1 ADD.

Thus, NAF method for scalar multiplication provides first a DBL, and then sequences of identical patterns. These patterns, which perform 2 DBLs and 1 ADD, are a succession of field-level operations which can contain for example 3 multiplications and 2 additions. Multiplications by constants are considered as additions. Curve-level operations are a succession of field-level operations. For example, the DBL formula in Jacobian with $a = -3$ is presented below in table 5.1 (from EFD): $[2]P = [2](X_1 : Y_1 : Z_1) = (X_{\mathtt{DBL}} : Y_{\mathtt{DBL}} : Z_{\mathtt{DBL}})$.

$$
\begin{aligned}
R1 &= Z_1 \times Z_1 \\
R2 &= Y_1 \times Y_1 \\
R3 &= X_1 \times R2 \\
R4 &= X_1 - R1 \\
R5 &= X_1 + R1 \\
R6 &= R4 \times R5 \\
R7 &= 3 \times R6 \\
R8 &= R7 \times R7 \\
R9 &= 8 \times R3 \\
X_{\texttt{DBL}} &= R8 - R9 \\
R11 &= Y_1 + Z_1 \\
R12 &= R11 \times R11 \\
R13 &= R12 - R2 \\
Z_{\texttt{DBL}} &= R13 - R1 \\
R15 &= 4 \times R3 \\
R16 &= R15 - X_{\texttt{DBL}} \\
R17 &= R2 \times R2 \\
R18 &= 8 \times R17 \\
R19 &= R7 \times R16 \\
Y_{\texttt{DBL}} &= R19 - R18
\end{aligned}
$$

Table 5.1: Point doubling formula in Jacobian, with $a = -3$ (from EFD).

It is obvious that several operation sequences can lead to the `DBL` computation can be different. For instance, the register $R2$ (line 2) can be computed before the register $R1$ (line 1). One can visualize this property in figure 5.6, the ASAP scheduling for `DBL`. Table 5.2 presents two possible sequences for 2 `DBL`s and 1 `ADD` in Jacobian with $a = -3$ (from EFD). Each operation corresponds to a modular operation over $\mathbb{F}_p$. These two sequences compute the two points

$[4](X_1 : Y_1 : Z_1) = (X_{\texttt{QDL}} : Y_{\texttt{QDL}} : Z_{\texttt{QDL}})$,

and

$(X_2 : Y_2 : Z_2) + (X_3 : Y_3 : Z_3) = (X_{\texttt{ADD}} : Y_{\texttt{ADD}} : Z_{\texttt{ADD}})$,

with 3 multipliers and 3 adders. `QDL` refers to the coordinates after computing a point quadrupling. Each line of figure 5.6 is computed in parallel, one after the other. The two possible sequences 2 `DBL`s and 1 `ADD` in table 5.2 correspond to one iteration loop of the scalar multiplication algorithm using the `NAF` method in figure 5.3 and 5.4.

Our Python script renames all temporary variables, and re-uses variables which are not used anymore. Indeed, we want to reduce the number of registers as much as possible. When there is no operation (symbol $\emptyset$), dummy operations can be performed.

An atomic block should be a sequence of operations which cannot be directly linked to the key digits. There, atomicity consists in always computing a sequence of field-level operations in parallel (for instance, 3 multiplications and 3 additions in table 5.2). Furthermore, these operation sequences are computed to have a regular scalar multiplication algorithm by performing

multiplier 1	multiplier 2	multiplier 3	adder 1	adder 2	adder 3
$R1 = Z_1 \times Z_1$	$R2 = Y_1 \times Y_1$	$R41 = Z_2 \times Z_2$	R11=$Y_1 + Z_1$	$R65 = Z_2 + Z_3$	$(Y_3 = -Y_3)$
$R3 = X_1 \times R2$	$R11 = R11 \times R11$	$R17 = R2 \times R2$	$R4 = X_1 - R1$	$R5 = X_1 + R1$	$\emptyset$
$R5 = R4 \times R5$	$R2 = Z_3 \times Z_3$	$R44 = X_3 \times R41$	$R4 = 8 \times R3$	$R11 = R11 - R2$	$R3 = 4 \times R3$
$R1 = X_2 \times R2$	$R45 = Z_3 \times R2$	$R65 = R65 \times R65$	$R5 = 3 \times R5$	$R11 = R11 - R1$	$R17 = 8 \times R17$
$R8 = R5 \times R5$	$R21 = R11 \times R11$	$R45 = Y_2 \times R45$	$R1 = R44 - R1$	$R65 = R65 - R41$	$\emptyset$
$R41 = Z_2 \times R41$	$\emptyset$	$\emptyset$	$R8 = R8 - R4$	$R4 = 2 \times R1$	$R65 = R65 - R2$
$R41 = Y_3 \times R41$	$R4 = R4 \times R4$	$Z_{\texttt{ADD}} = R65 \times R1$	$R3 = R3 - R8$	$R25 = R8 + R21$	$R44 = R8 - R21$
$R3 = R5 \times R3$	$R25 = R44 \times R25$	$R2 = R2 \times R4$	$R41 = R41 - R45$	$\emptyset$	$\emptyset$
$R4 = R1 \times R4$	$\emptyset$	$\emptyset$	$R3 = R3 - R17$	$R25 = 3 \times R25$	$R41 = 2 \times R41$
$R65 = R25 \times R25$	$R1 = R3 \times R3$	$R11 = R41 \times R41$	$R3 = R3 + R11$	$R5 = 2 \times R2$	$\emptyset$
$R8 = R8 \times R1$	$R3 = R3 \times R3$	$R17 = R1 \times R1$	$R11 = R11 - R4$	$\emptyset$	$\emptyset$
$R4 = R45 \times R4$	$\emptyset$	$\emptyset$	$R44 = 8 \times R8$	$R8 = 4 \times R8$	$R17 = 8 \times R17$
$\emptyset$	$\emptyset$	$\emptyset$	$X_{\texttt{QDL}} = R65 - R44$	$R3 = R3 - R1$	$X_{\texttt{ADD}} = R11 - R5$
$\emptyset$	$\emptyset$	$\emptyset$	$Z_{\texttt{QDL}} = R3 - R21$	$R8 = R8 - X_{\texttt{QDL}}$	$R2 = R2 - X_{\texttt{ADD}}$
$R8 = R25 \times R8$	$R2 = R41 \times R2$	$\emptyset$	$R4 = 2 \times R4$	$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$	$Y_{\texttt{QDL}} = R8 - R17$	$Y_{\texttt{ADD}} = R2 - R4$	$\emptyset$
$R1 = Z_1 \times Z_1$	$R2 = Y_1 \times Y_1$	$R41 = Z_2 \times Z_2$	$R11 = Y_1 + Z_1$	$R65 = Z_2 + Z_3$	$(Y_3 = -Y_3)$
$R3 = X_1 \times R2$	$R11 = R11 \times R11$	$R17 = R2 \times R2$	$R4 = X_1 - R1$	$R5 = X_1 + R1$	$\emptyset$
$R5 = R4 \times R5$	$R42 = Z_3 \times Z_3$	$R65 = R65 \times R65$	$R11 = R11 - R2$	$R17 = 8 \times R17$	$\emptyset$
$R2 = X_2 \times R42$	$R44 = X_3 \times R41$	$R47 = Z_2 \times R41$	$R5 = 3 \times R5$	$R3 = 4 \times R3$	$R11 = R11 - R1$
$R8 = R5 \times R5$	$R1 = R11 \times R11$	$R45 = Z2 \times R42$	$R2 = R44 - R2$	$R65 = R65 - R41$	$R4 = 8 \times R3$
$R45 = Y_2 \times R45$	$R47 = Y_3 \times R47$	$\emptyset$	$R4 = 2 \times R2$	$R8 = R8 - R4$	$R65 = R65 - R42$
$R4 = R4 \times R4$	$Z_{\texttt{ADD}} = R65 \times R2$	$\emptyset$	$R3 = R3 - R8$	$R44 = R8 - R1$	$R25 = R8 + R1$
$R5 = R42 \times R4$	$R3 = R5 \times R3$	$R25 = R44 \times R25$	$R47 = R47 - R45$	$\emptyset$	$\emptyset$
$R4 = R2 \times R4$	$\emptyset$	$\emptyset$	$R3 = R3 - R17$	$R25 = 3 \times R25$	$R47 = 2 \times R47$
$R41 = R47 \times R47$	$R17 = R3 \times R3$	$R42 = R25 \times R25$	$R3 = R3 + R11$	$R11 = 2 \times R5$	$\emptyset$
$R3 = R3 \times R3$	$R2 = R17 \times R17$	$R8 = R8 \times R17$	$R41 = R41 - R4$	$\emptyset$	$\emptyset$
$R4 = R45 \times R4$	$\emptyset$	$\emptyset$	$R41 = 8 \times R8$	$R8 = 4 \times R8$	$R2 = 8 \times R2$
$\emptyset$	$\emptyset$	$\emptyset$	$X_{\texttt{QDL}} = R42 - R41$	$R3 = R3 - R17$	$X_{\texttt{ADD}} = R41 - R11$
$\emptyset$	$\emptyset$	$\emptyset$	$Z_{\texttt{QDL}} = R3 - R1$	$R8 = R8 - X_{\texttt{QDL}}$	$R5 = R5 - X_{\texttt{ADD}}$
$R5 = R47 \times R5$	$R8 = R25 \times R8$	$\emptyset$	$\emptyset$	$\emptyset$	$R4 = 2 \times R4$
$\emptyset$	$\emptyset$	$\emptyset$	$Y_{\texttt{ADD}} = R5 - R4$	$Y_{\texttt{QDL}} = R8 - R2$	$\emptyset$

sequence 1

sequence 2

Table 5.2: Two possible sequences for 2 DBLs and 1 ADD in parallel.

sequences of curve-level operations in parallel (2 DBLs and 1 ADD in parallel for instance).

Thus, atomicity consists in re-writing the sequence of field-level operations into a sequence of

identical atomic patterns. Table 5.2 proposes two different atomic patterns. In addition, a scalar multiplication performing one of these patterns always performs 2 DBLs and 1 ADD in a same way, by using 3 multipliers and 3 adders. Once again, each line of table 5.2 must be performed in parallel.

The two sequences use 20 temporary variables, in 16 time steps. The execution time of one time step corresponds to the execution time for a multiplication. These sequences perform 2 DBLs and 1 ADD, but with less field-level operations than when one computes the three curve-level operations one after the other: first 1DBL, then 1DBL and finally 1ADD. Indeed, at the very beginning, ADD and DBL can compute a same modular operation. In addition, simplifications can be done when several doublings are performed one after the other. Our Python script simplifies all non-necessary computations.

Sequences in table 5.2 are provided for the NAF method scalar multiplication algorithm in figures 5.3 and 5.4. This algorithm performs point additions $Q \pm P$ or $Q \pm [2]P$. Input points of ADD and the 2 DBLs can be different. That is why the provided sequences consider that the input point of ADD is different to the 2 DBLs. In addition, a point subtraction can be computed. Sequences provide this case by the expression $Y_3 = -Y_3$ in brackets. This operation is always performed. The controller stores the result when a point subtraction is performed.

Atomic sequences in table 5.2 provides regular behaviour of the NAF method for scalar multiplication. Thus, it can provide a countermeasure against some simple side channel attacks. Thus, our Python script provides several sequences for a number of multipliers and adders. One sequence can be performed $\lfloor n/2 \rfloor$ times to have the result of the scalar multiplication $[k]P$ (with a DBL at the first loop iteration).

In addition, our Python script produces several possible sequences for a given number of multipliers and adders. The produced sequences are in the same time step which is as small as possible. Thus, when several sequences are implemented, one can randomly perform a sequence. For instance, when 2 sequences are implemented, scalar multiplications use randomly the two sequences during the $\lfloor n/2 \rfloor$ iterations. In the same way, if one implements $\lfloor n/2 \rfloor$ sequences, the scalar multiplication algorithm can use all sequences once. Of course, sequences must be based on the same number of adders and multipliers with the same time step. For several scalar multiplications, the circuit will not have the same behaviour when one has such a strategy. Thus, this may lead to a countermeasure against some differential side channel attacks. Below, we provide the architecture with the storage of the sequences. In hardware, a ROM is implemented and stores sequences.

5.3 Experiment Results and Implementation

The experiments reported below have been performed using our Python script. Table 5.3 presents the number of all possible sequences, with the minimum time step for different numbers of multipliers (denoted mult.) and adders (denoted add.). We give the number of required registers to perform 2 DBLs and 1 ADD in parallel. It corresponds to NAF right-to-left method (algorithms in figure 5.3 and 5.4) for scalar multiplication.

Let *Seq* be the set of all possible sequences for (2/3) DBLs and 1 ADD, for a specific number of multipliers and adders, and "$addByMult$" the number of re-used adders during only one multiplication time step. Indeed, a modular addition is faster than a modular multiplication over the finite field $\mathbb{F}_p$. Therefore multiple calls can be done to the adders. For example, if three adders are implemented with the parameter $addByMult = 2$, six modular additions can

be performed during multiplications: first 3 modular additions are computed in parallel, and then 3 other modular additions can be computed in parallel. These six modular additions are performed during one multiplication time step.

We give the occupation rate for the multipliers and the adders. The occupation rate is divided into two columns for the adders when $addByMult = 2$. That is why there is the symbol $\emptyset$ in the second column of "add. 2" when $addByMult = 1$. Indeed when $addByMult = 1$, the adders can be used during one multiplication time step. Thus, adders and multipliers are considered to be computed in the same execution time.

When there are less multipliers and adders, the possible sequences for curve-level operations and the time step increases. In addition, having the parameter $addByMult > 1$ reduces the execution time for several `DBL`s and one `ADD`, but the occupation rate for the second adder is between 20 % and 40 %. Thus, if one wants to have all arithmetical units occupied, a large number of dummy operations must be computed. If dummy operations are performed, each atomic block comprises the same sequence of field-level operations. For instance, 2 `DBL`s and 1 `ADD` can always be performed in parallel with 3 multiplications and 3 additions. With dummy operations, the device will always use all arithmetic units, and thus it could make more uniform the power consumption. Thus, it could be more difficult for an attacker to guess the secret key.

number of				# *Seq*	time	occupation rate		
mult.	add.	*addByMult*	registers		step	mult. [%]	add. 1 [%]	add. 2 [%]
2	2	1	18	28 231	21	80	88	$\emptyset$
		2	19	45 543	18	89	48	20
	3	1	21	932	20	80	88	$\emptyset$
		2	20	1 991	18	89	48	20
	4	1	21	895	20	80	88	$\emptyset$
		2	20	1 771	18	89	36	15
3	2	1	19	17 474	20	67	92	$\emptyset$
		2	19	23 615	13	82	62	40
	3	1	20	375	16	67	77	$\emptyset$
		2	20	635	13	82	59	23
	4	1	20	151	16	67	58	$\emptyset$
		2	20	436	13	82	46	19

Table 5.3: Experiment results for different number of multipliers and adders with `NAF` right-to-left atomic methods.

5.4 Implementation

Our Python script gives several sequences for several `DBL`s and 1 `ADD`. Each sequence corresponds to a sequence of field-level operations. Below, we present an architecture of the `NAF` right-to-left atomic methods. In our architecture, each field-level operation is written by 17 bits: 5 bits for the first operand address, 5 bits for the second one, 2 bits for the modular operation $(+, -, \times)$, and 5 bits for the address of the result. 5 bits are sufficient because less than $2^5 = 32$ temporary variables are necessary to perform 2 `DBL`s with 1 `ADD`.

Thus a sequence is written in our architecture as a succession of instructions. Each address corresponds to a memory address where values are stored. Figure 5.7 illustrates this method. A global controller generates all high-level control signals for the architecture units. If several

sequences of DBLs with one ADD are implemented, a random number is sent to the controller, and the controller selects which sequence will be performed.

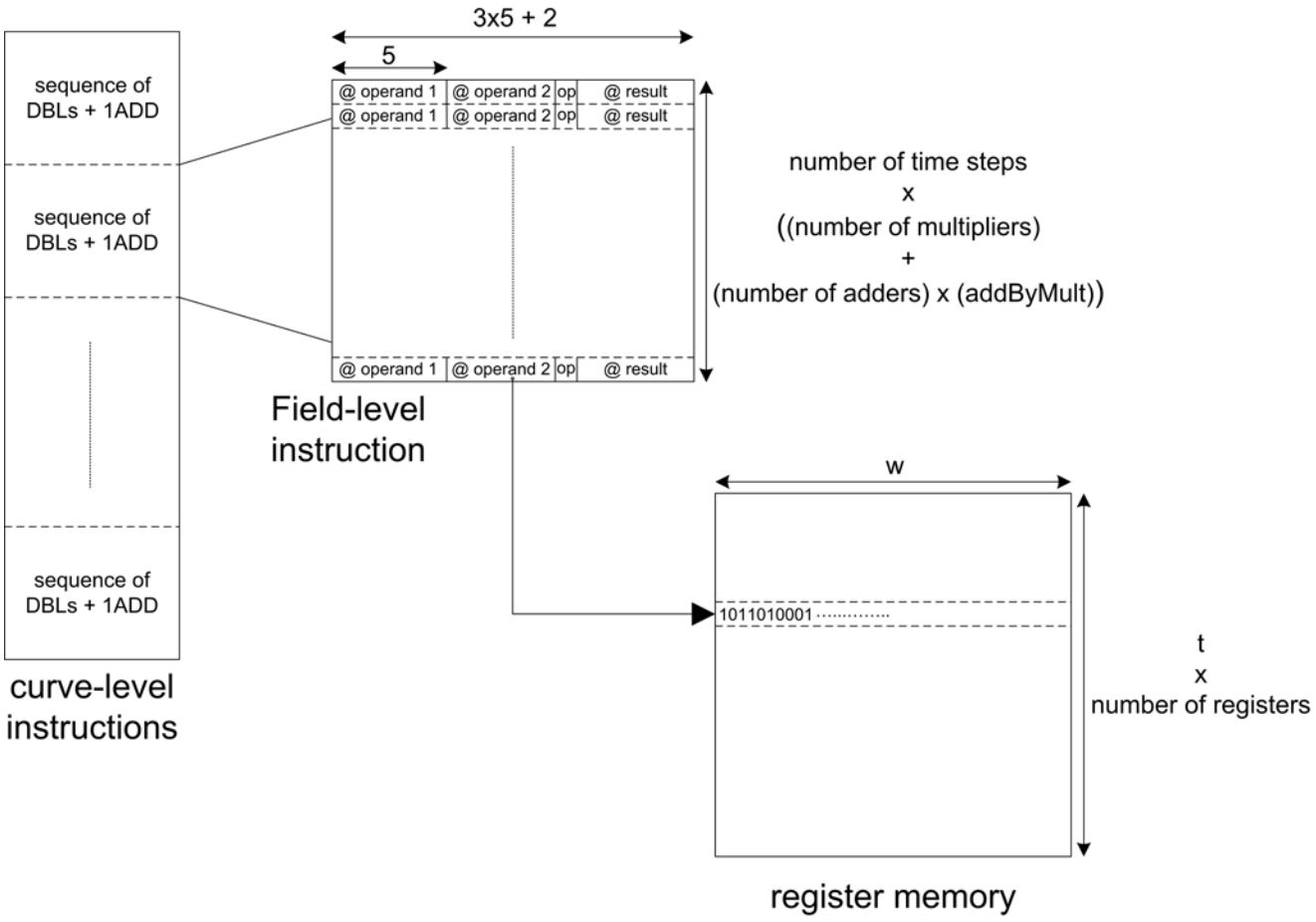


Figure 5.7: Illustration of the sequences.

In practice, registers are stored into t words by w bits. In our architecture, we consider to have a register file with a read and a write port. Thus an address refers to the first word. At each clock cycle, a word is sent to an arithmetic unit (modular addition/subtraction and multiplication). Results of the computed operation are sent word by word t times to the corresponding address register. Figure 5.8 presents the unit architecture for m arithmetic units (multipliers and adders over the finite field $\mathbb{F}_p$). The global controller controls the registers memory and all arithmetical units. Once again, if several sequences of DBLs with one ADD are implemented, the controller selects a sequence according to a random number.

In our hardware architecture, a RAM is implemented to store points and all intermediate values ("mem." block). The RAM output depends on the implemented number m of arithmetic units: the RAM sends $2 \times w$ bits for each arithmetic unit, that is a $2wm$ bit output. Indeed, all arithmetic units have two input operands on w bits. For instance, when 3 multipliers and 2 adders are implemented, the RAM sends $(3 + 2) \times 2 \times w = 10w$ bits. One a same way, each arithmetic unit has a w-bit output. Thus, the RAM input is wm bits.

The two bits of the sequences (op in figure 5.7) correspond to the field-level operation to be computed. One bit is necessary for the adders to know if a modular addition or a modular

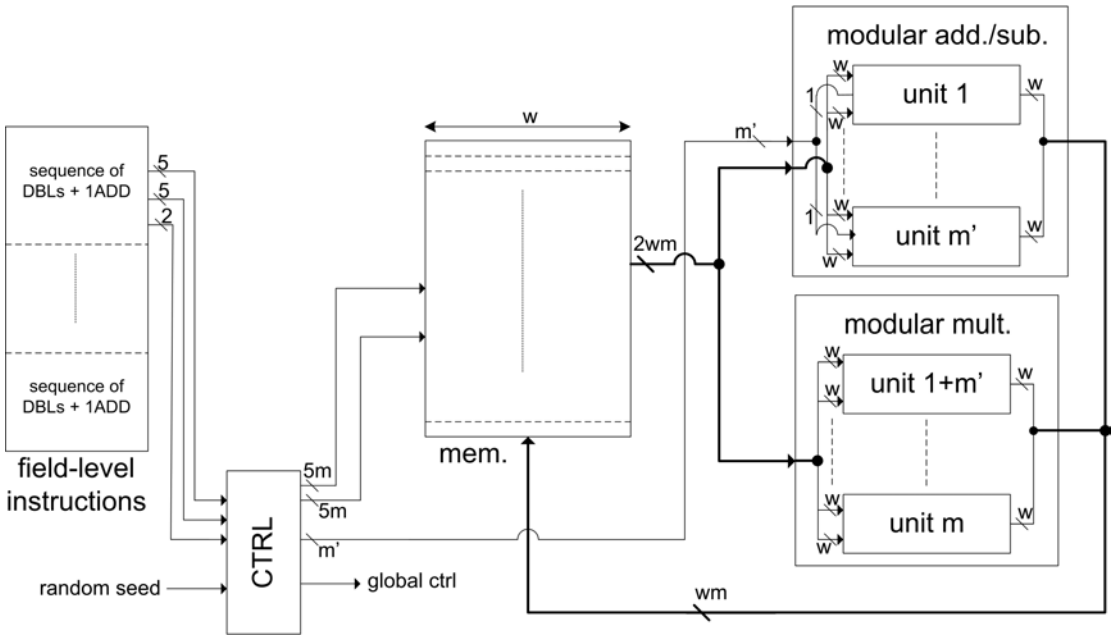


Figure 5.8: Architecture overview.

subtraction must be computed. In the next section, we provide implementation results for arithmetic units: addition/subtraction unit and multiplication unit over the prime field $\mathbb{F}_p$.

All hardware implementations reported in this chapter have been described in VHDL and implemented on a XC5VLX50T FPGA using ISE 14.1 from Xilinx with standard efforts for synthesis, place and route (except in table 5.7 where different FPGAs are used). We report numbers of clock cycles, best clock frequencies and numbers of occupied slices. We also report numbers of look-up tables (LUTs with 6 inputs in Virtex 5) and flip-flops (FFs) for area. A XC5VLX50T contains 7200 slices with 4 LUTs and 4 flip-flops per slice. We use flip-flops for all storage elements.

5.4.1 Arithmetic Hardware Implementation

In this section, we present implementations of modular arithmetic over the prime field $\mathbb{F}_p$. All elements of $\mathbb{F}_p$ are n-bit long in the range 160–600 bits for typical cryptographic sizes. Modular addition/subtraction and multiplication receive w bits at each clock cycle (during t clock cycles). Thus inputs and outputs of the two units are w-bit long, and each unit is performed in a constant execution time.

Modular Addition/Subtraction

A same unit has been implemented for both modular addition and subtraction with the method presented in [37, Sec. 3.2]. The bit op allows to perform modular addition or subtraction: $\texttt{op} = 0$ refers to a modular addition, while $\texttt{op} = 1$ refers to a modular subtraction.

$$a \ \texttt{op}\ b \bmod p = \begin{cases} a + (\texttt{op}\ \texttt{XOR}\ b) & \textbf{if } 0 \leq (a \ \texttt{op}\ b) < p \\ a + (\texttt{op}\ \texttt{XOR}\ b) + (\texttt{NOT}(\texttt{op})\ \texttt{XOR}\ p) & \textbf{else} \end{cases}$$

The FPGA implementation results for the modular addition/subtraction are reported in Table 5.4 for $n \in \{160, 224\}$ and for different parameters of w and t.

n	t	w	area slices (FF/LUT)	freq. MHz	clock cycles
160	20	8	277 (333/403)	259	t+1
	14	12	294 (346/410)	257	
	10	16	286 (330/375)	259	
	7	24	287 (330/415)	255	
	5	32	289 (328/462)	254	
256	32	8	374 (481/516)	257	
	16	16	361 (478/471)	248	
	8	32	397 (476/553)	250	

Table 5.4: FPGA implementation results for modular addition/subtraction.

Modular Multiplication

Modular multiplication has been implemented using the Montgomery multiplication (MM) method. The Montgomery multiplication algorithm was presented in section 1.3 figure 1.11, and computes $\text{MM}(a, b) \equiv abR^{-1} \equiv TR^{-1} \bmod p$ where R is a power of the base ($R = 2^n$). The architecture of the Montgomery multiplication is presented below in figure 5.9.

Inputs are w-bit long, but the first step of the algorithm is to wait for all t words of the operands a and b. Then, the multiplicand is scanned bit by bit, while the multiplier is scanned word by word. The FPGA implementation results for the Montgomery multiplication are reported in Table 5.5 for $n \in \{160, 224\}$ and for different parameters of w and t.

n	t	w	area slices (FF/LUT)	freq. MHz	clock cycles
160	20	8	521 (694/865)	212	$tw(t+1)+2t+1$
	14	12	451 (733/791)	205	
	10	16	493 (708/842)	207	
	7	24	387 (755/700)	195	
	5	32	452 (740/763)	192	
256	32	8	632 (967/984)	202	
	16	16	699 (987/1 091)	206	
	8	32	636 (1 005/859)	187	

Table 5.5: FPGA implementation results for Montgomery multiplication.

Modular addition/subtraction and multiplication units have been highly tested and verified by a C program which uses the library GMP (*GNU Multiple Precision*). Each unit computed a great deal of values provided by a file, and results have been verified for different parameters t

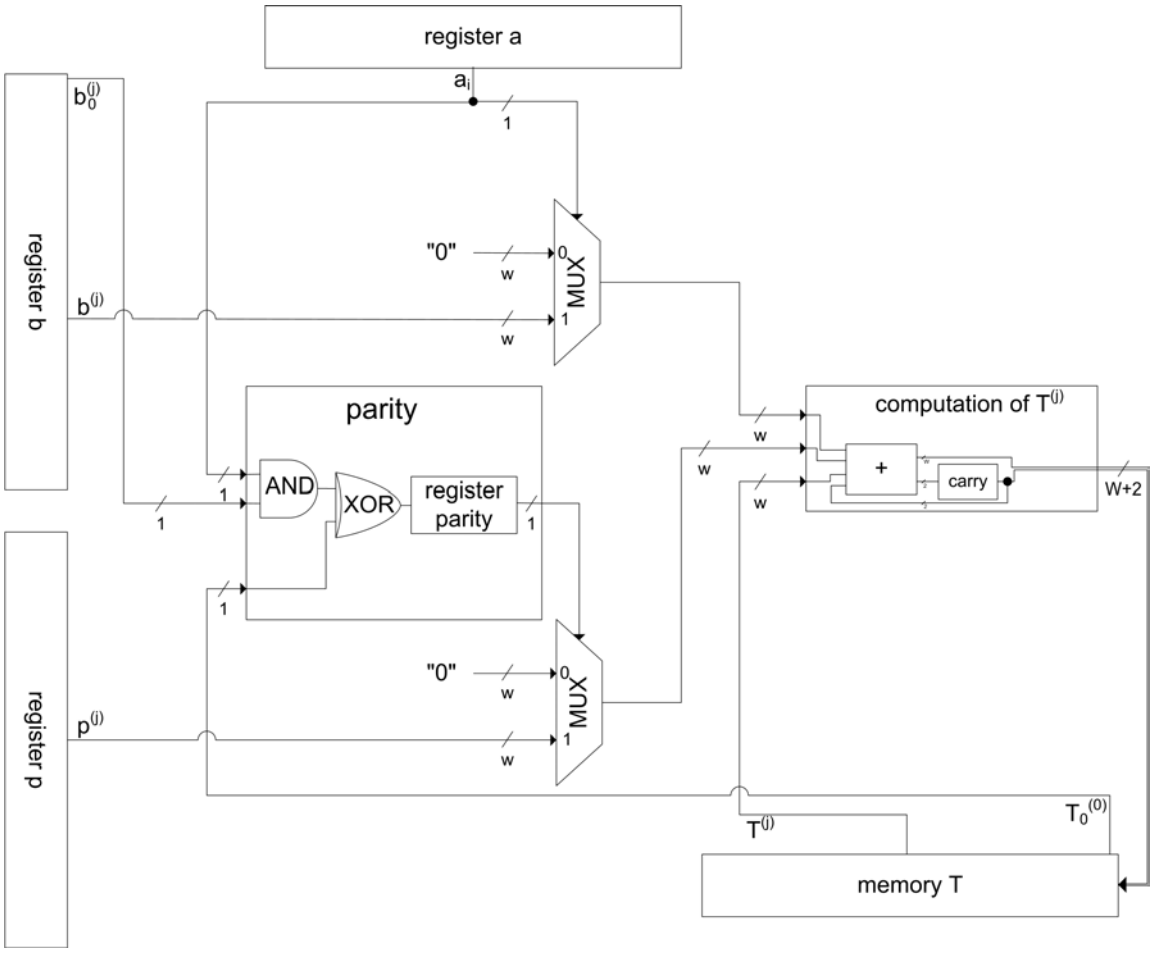


Figure 5.9: Architecture for Montgomery multiplication.

and w.

Whereas the Montgomery inversion is not used in this chapter, this unit has been implemented and presented in appendix C.

5.4.2 Global FPGA Implementation Results

The FPGA implementation results are reported in table 5.6 for $n \in \{160, 192, 256\}$ and $w \in \{16, 32\}$ with different numbers of multipliers and adders. The FPGA implementation performs an ECC scalar multiplication which follows the architecture presented in figure 5.8 for the `NAF` method, and one sequence of 2 `DBL`s with 1`ADD` (see end of section 5.4 for target and tools details). Our implementation works at higher clock frequency than the ECC processor (in appendix A) provided by the authors of [24] (for curves over $\mathbb{F}_p$, $n = 160$ and Jacobian coordinates). In addition its area can be considered equivalent for $n = 160$ and $w = 32$ compared to the complete large version of the ECC processor which uses two arithmetic units per field operation. When five sequences are implemented, hardware implementations have the same clock frequency and about 7% more area. When ten sequences are implemented, hardware implementations have about 4% smaller clock frequency and about 10% more area.

n	number of multipliers	number of adders	w	area slices (FF/LUT)	freq. MHz
160	2	3	16	2 188 (2 407/5 866)	219
			32	2 273 (2 425/5 826)	193
	3	3	16	2 726 (3 140/6 720)	219
			32	2 796 (3 202/6 592)	191
	3	4	16	2 911 (3 489/7 094)	215
			32	2 938 (3 547/7 053)	187
192	2	3	16	2 589 (2 889/6 980)	195
			32	3 912 (4 536/9 215)	189
	3	3	16	3 233 (3 748/8 026)	193
			32	3 310 (3 811/8 026)	188
	3	4	16	3 452 (3977 /8 433)	193
			32	3 478 (3 989/8 434)	184
256	2	3	16	3 478 (4 003/8 943)	186
			32	3 530 (4 061/8 656)	185
	3	3	16	4 279 (4 964/10 918)	183
			32	4 357 (5 027/10 492)	186
	3	4	16	4 563 (5 249/11 648)	183
			32	4 578 (5 282/11 009)	181

Table 5.6: FPGA implementation results.

For $n = 160$ bits, the implementation computes 80 times 2 `DBL`s with 1 `ADD` in parallel with the right-to-left `2NAF` algorithm. When the number of sequences is equal to one, the implementation always performs the same sequence of field-level operations. When the number of sequences is equal to ten, each sequence is performed in average between eight times.

Table 5.7 reports FPGA implementation results for various FPGAs. For each FPGA result, one sequence of `2DBL`s with `1ADD` is implemented for $n = 192$, $w = 32$ and three units of multiplier and adder. In our implementation, multipliers take the same number of cycles for each FPGAs. On a same way, adders terminate in a constant time for each FPGAs. In addition, our implementation always compute a sequence whatever the digit of k, and all sequence have the same depth: each sequence has the same execution time. Thus, we can use the clock frequency to compare devices between each other.

In this thesis, we only focus on Xilinx FPGAs. Xilinx provide several FPGA families, and each family fits for many different markets. Provided informations come from the Xilinx web site `http://www.xilinx.com/products/silicon-devices/fpga/index.htm`. Thus, each family targets a specific implementation :

- spartan family for low-power footprint,
- virtex family for high performances,
- artix family for lower power,
- kintex family for intermediate solutions,
- zynq family for high-end embedded-system.

family	nb. per slice of LUTs	nb. per slice of FFs	device	area slices (FF/LUT)	freq. MHz
spartan 3E	2	2	XC3 S500E	10 568 (5 692/3 818)	80
spartan 3	2	2	XC3 S1000	10 568 (5 733/3 818)	71
spartan 3	2	2	XC3 S1000L	10 569 (5 733/3 818)	61
spartan 3A	2	2	XC3 SD3400A	10 572 (5 692/3 824)	75
virtex 4	4	2	XC4 VSX55	3 793 (3 810/8 979)	139
virtex 4	4	2	XC4 VFX60	10 575 (5 716/3 818)	139
virtex 4	4	2	XC4 VLX60	10 575 (5 716/3 818)	139
virtex 5	4	4	XC5 VLX50	3 792 (3 811/8 991)	168
virtex 5	4	4	XC5 VLX50T	3 310 (3 811/8 026)	188
virtex 5	4	4	XC5 VSX50T	3 792 (3 810/8 991)	188
virtex 5	4	4	XC5 VFX70T	3 310 (3 811/8 026)	188
spartan 6	4	8	XC6 SLX75L	3 788 (3 912/9 093)	69
spartan 6	4	8	XC6 SLX75	3 789 (3 912/9 093)	108
spartan 6	4	8	XC6 SLX75T	3 789 (3 912/9 093)	108
virtex 6	4	8	XV6 VLX75T	10 575 (5 716/3 818)	166
virtex 6	4	8	XV6 VCX75T	10 575 (3 811/8 991)	166
virtex 6	4	8	XC6 VLX75TL	3 792 (3 811/8 991)	163
virtex 6	4	8	XC6 VSX315TL	3 310 (3 811/8 026)	163
virtex 6	4	8	XC6 VLX760L	3 792 (3 810/8 979)	163
artix 7	4	8	XC7 A200T	3 783 (3 810/8 995)	136
artix 7	4	8	XC7 A200TL	3 791 (3 811/8 991)	129
kintex 7	4	8	XC7 K410T	3 793 (3 810/8 990)	196
kintex 7	4	8	XC7 K410TL	3 791 (3 811/8 990)	140
virtex 7	4	8	XC7 VX330T	3 793 (3 810/8 990)	196
virtex 7	4	8	XC7 VH580T	3 793 (3 810/8 979)	178
virtex 7	4	8	XC7 V585T	3 793 (3 810/8 979)	178
zynq	4	8	XC7 Z020	3 783 (3 910/8 978)	198

Table 5.7: FPGA implementation results ($n = 192$ and $w = 32$).

For each FPGA family, there are several devices. The first number is for the FPGA version. For instance, XC6 stands for the 6th FPGA series. The next number is for the FPGA sizes. For instance, the XC6VLX75T is the smallest, the XC6VLX240T is the mid-range and the XC6VLX720 is the largest Xilinx Virtex 6 FPGA. The number of logic cells, embedded memory, DSP modules, user IOs, etc. increases according to the size of the considered device. In addition, there is a specific application for each FPGA family:

- LX for high-performance logic,
- LXT for high-performance logic with power serial connectivity,

- LXTL for lower power,
- SX for optimized for DSP and memory-intensive applications,
- SXT for DSP and memory-intensive applications,
- FX for optimized for embedded processing and data communications,
- FXT for embedded processing with highest speed serial connectivity.

Once again, table 5.7 reports FPGA results for a same implementation and for various FPGAs. Each line is coloured in grey. The more dark the colour is, the more area the circuit requires, and the lower clock frequency is.

Direct comparison in term of area is difficult when the number of LUTs and flip-flops per slice is different between FPGAs. However, several FPGAs require about 10 570 slices, and the majority of FPGAs requires about 3 700 slices.

5.4.3 Global ASIC Implementation Results

ASIC results have been synthesized into gate-level netlists using standard V_{th} (SVT) cells of an industrial 130nm bulk CMOS technology library using Synopsys Design Compiler G-2012.06-SP5. The standard cells used were restricted to a set {nand2, nor2, xor2, inv} of logic gates without loss of generality. The scalar multiplication implementation is applied with a maximum path delay constraint of 10ns from all inputs to all outputs.

ASIC results follows the architecture presented in figure 5.8 for the 3`NAF` method, one sequence of 2`DBL`s with 1`ADD` and two multiplier and adder units. Results have been synthesized for $n \in \{160, 192, 256\}$ and $w \in \{16, 32\}$.

n	w	combinational	buf/inv	non combinational	total
160	16	238 698.9	10 118.2	164 398.5	403 098.2
	32	218 409.0	12 260.7	153 512.2	371 921.2
192	16	293 257.4	12 329.6	201 928.5	495 186.0
	32	291 586.4	17 172.2	203 555.5	495 205.5
256	16	330 537.9	14 163.6	226 975.1	557 513.1
	32	327 086.6	18 757.0	228 604.0	555 690.5

Table 5.8: Area results (in μm^2) of ASIC implementation of the ($n = 160$).

n	w	cell internal	net switching	total dynamic	cell leakage
160	16	9 975.3	496.5	10 172.0	134.9
	32	9 065.7	503.5	9 569.4	124.9
192	16	11 848.4	589.8	12 438.4	165.8
	32	11 985.6	625.7	12 611.6	166.2
256	16	13 321.5	654.1	13 976.1	186.7
	32	13 443.9	695.9	14 140.0	185.3

Table 5.9: Power results in (μW) of ASIC implementation of the ($n = 160$).

Area and power results of ASIC implementation are higher than ASIC results of the large

version of the ECC processor (appendix A): 20% (resp. 13%) more total area for $n = 160$ and $w = 16$ (resp. $w = 32$). In addition, there is 2% for $n = 160$ and $w = 16$ more total dynamic power for our power results in ASIC; and 4% less total dynamic power for $w = 32$.

When we add one multiplier and adder unit to the previous ASIC implementation (that is three multipliers and three adders are implemented), there are about 10% more total area and 7% more total dynamic power in ASIC.

5.5 Conclusion

In this chapter, we use existing scalar multiplication algorithms in a different way. These algorithms can have a regular behaviour such that they can provide a countermeasure against some side-channel attacks. Atomic sequences have been provided for some of the right-to-left scalar multiplication algorithms. With the supposition that some field-level operations can be computed in parallel, we provide several atomic sequences. A Python script provides all possible sequences for a generic number of multipliers and adders and for the computation of 2 `DBL`s with one `ADD` in parallel. This script can have in input any `ADD` and `DBL` formulas. In addition, the number of temporary variables is reduced as much as possible, and the Python script simplifies computations when some values can be re-used.

When one chooses to have several different sequences, the implementation randomly selects one sequence for each computation of `DBL`s with one `ADD`. Once again, it may lead to a countermeasure against some differential attacks. Indeed, for several scalar multiplications, the sequence of operations which leads to the computation of $[k]P$ can be different. A full hardware implementation has been implemented according to such a strategy. However one must be cautious when several sequences are implemented. Indeed, one needs to know how many differences there are between sequences: two sequences can be provided for 2 `DBL`s and 1 `ADD` with only very few differences. In this case, the implementation behaviour would be very similar between the two sequences, because the majority of used registers will be the same.

In this study, multiplications and squares are performed without differences, i.e. `1M` = `1S`. The typical cost assumption used in many references is that one square corresponds to 0.8 multiplication. Thus, instead of having only multipliers and adders, one could have some arithmetical units which take into account the difference between squares and multiplications. In this case our Python script must be extended. It would provide all possible operation sequences for a generic number of adders, multipliers and squares. In addition, the same reasoning can be done between additions and multiplications by one constant. By keeping this reasoning, only one type of arithmetic unit can be implemented: it may be interesting to only perform operations in the form of $ab \pm c$. Operations at the curve level can be expressed with only this type of operation.

All these improvements could lead to a more efficient countermeasure against simple SCA. Indeed, it could protect an ECC cryptosystem both at a high level (curve-level operations) and at a lower level (field-level operations). However, the security against SCAs of such an implementation must be evaluated by attacks.

In our hardware architecture, a RAM is implemented to store points and all intermediate values. The RAM can be implemented in three different ways. With the first one, the RAM output depends on the implemented number of arithmetic units: the RAM sends $2 \times w$ bits for each arithmetic unit. Indeed, all arithmetic units have two input operands on w bits. For instance, when 3 multipliers and 2 adders are implemented, the RAM sends $(3+2) \times 2 \times w = 10w$

bits. The first strategy has been implemented in our architecture.

Secondly, the RAM output could only depend on one type of arithmetic unit. If adder delays were shorter than multiplier delays, the RAM could send all data to the multipliers, and then to the adders without delay penalty. In this case, the output of the RAM is reduced to $6w$ bits for the implementation of 3 multipliers and 2 adders. This second strategy could reduce the circuit area according to the first one.

Thirdly, the RAM could have only a $2w$-bit output by extending this reasoning. The RAM could send data to each arithmetic unit, one after the other: the RAM could send data to the first multiplier, then to the second multiplier, etc. This third strategy could reduce the circuit area according to the second one. However, there is a time penalty with this strategy. Indeed, the last arithmetic unit calculates and sends the computed value latter according to the first one. Thus, the RAM needs more clock cycle to store all computed values.

Conclusion

The theoretical security of cryptosystems based on elliptic curves usually relies on the ECDLP problems. However the security of the cryptosystems against physical attacks, which are considered as very strong threats in embedded security applications, should be provided by cryptosystem designers. Recently, very efficient side-channel attacks have been proposed such as power consumption or electromagnetic emanation analysis.

In this Ph.D. thesis, protections at the arithmetic level against some side-channel attacks, and more particularly against some attacks by observation, are provided for ECC scalar multiplication. We implement them in hardware. Each method and each implementation has been validated (at theoretical and practical levels), designed in hardware and practically evaluated on FPGAs and ASICs. We wanted to show that most of solutions are realistic in hardware and implementable. In addition, we wanted to know what is the costs of the proposed solutions in term of area or clock frequency. We compared our results to a complete ECC processor in appendix A provided by the authors of [24] (for curves over $\mathbb{F}_p$, $n = 160$ bits and Jacobian coordinates).

In this work, countermeasures used for protecting scalar multiplications are based on a specific type of blinding: randomizing the scalar. Some redundant representations are studied and used to randomly recode the scalar k. This countermeasure consists in computing scalar multiplication with a randomly recoded k for each scalar multiplication. Thus, the sequence of operations which leads to the computation $[k]P$ will be different for all recodings of k. Such a method can be used as a countermeasure against some differential SCAs.

For each studied redundant representation, the scalar k can be randomly recoded on-the-fly. We considered signed-digit representation, which can be a redundant representation. A scalar was randomly recoded in signed-digit representation, and we studied a width–w signed-digit representation for both accelerating the scalar multiplication and having a random recoding of k. A complete hardware implementation provided an on-the-fly random recoding of the scalar k for several w. The FPGA area represented less than 28% than the complete ECC cryptosystem, and worked at a greater clock frequency.

We considered that the use of such a representation can be a countermeasure against some SCAs. To validate this countermeasure, a practical SCA has been performed on a scalar multiplication where the scalar k was randomly recoded on-the-fly using a signed-digit representation. The considered SCA was performed in collaboration with the cryptographic group from UCC. A generic architecture was designed for ECC operations at UCC, and the countermeasure implementation has been integrated to the ECC cryptoprocessor. The attack, called template attack, is a powerful one: the attacker is assumed to know characteristics of a side channel over some processed data of a device. This attack enabled to evaluate the robustness of the countermeasure. Such an attack failed in finding a piece of the scalar k. However evaluating how much this countermeasure is efficient against SCAs is still an open and a hard problem.

Other redundant representations have been studied. The double-base number system (DBNS)

simultaneously uses two bases for representing numbers. We used the DBNS natural redundancy to on-the-fly randomly recode the scalar k. First, we considered to have k in a DBNS chain to randomly recode the scalar into another DBNS chain. This method was implemented in hardware and practically evaluated on FPGAs and ASICs. The FPGA area required by the DBNS recoding unit represented less than 7% compared to the complete ECC processor, and worked at a greater clock frequency.

Then, conversion from an integer into multi-base number system (MBNS) has been studied (DBNS is a particular case of MBNS). Thus, we proposed a multi-base recoding algorithm which can be fully implemented in hardware for ECC scalar multiplications. The scalar recoding was performed on-the-fly and in parallel to curve-level operations without additional latency. Two versions were provided, one without any pre-computations, and the other with pre-computed points to accelerate scalar multiplications. To our knowledge, it seems that we provided the first on-the-fly hardware implementation of a multi-base recoding method for ECC scalar multiplication. The FPGA area represented less than 10% than the ECC processor, and our MBNS signed recoding unit worked at higher frequency.

Lots of parameters during the recoding of an integer into MBNS can be changed to accelerate the scalar multiplication. For example, we plan to reduce the number of produced terms and improve randomization schemes to increase robustness against SCAs. Indeed, more randomizations can be made on the scalar k during the recoding into MBNS. However this MBNS recoding countermeasure has not been practically evaluated yet. Studying such countermeasures is an important issue and perspective. Randomizations do not have a uniform distribution: two recodings can have some identical patterns. Then, the question would be: how can we evaluate the information leakage, i.e. the robustness of randomizations?

Another kind of countermeasure against SCAs has been studied. We used existing scalar multiplication algorithms to perform them in such a way that they have a regular behaviour. In addition, we worked at the field level to produce a regular sequence of operations, that is in performing a succession of curve-level operations regardless on the scanned bits. Such methods can be used as countermeasures against simple SCAs. Thus, each curve-level operation was performed with a sequence of identical operations at the field level. We developed a program which gives all possible sequences according to generic parameters. In addition, randomization can be added in the sequences. Thus, we provided a method which can be a countermeasure against some both simple and differential SCAs, but an evaluation of randomizations is necessary to know if it does not affect the natural protection of this method against simple SCAs.

Hardware implementations with the proposed solutions have been provided. Using such methods, a complete ECC scalar multiplication has been implemented in hardware. Compared to the ECC processor provided by the authors of [24], ours worked at an equivalent clock frequency, but with about 7% more area. Thus, a future work could consist in providing a pipelined architecture. In addition, our hardware implementation has been designed for a generic numbers of arithmetic units: multiplication and addition/subtraction over the finite field $\mathbb{F}_p$. Future works could consist in implementing other arithmetic units to accelerate the field-level operations and to improve the security of the cryptoprocessor.

Eventually, the hardware implementation can randomly choose among several implemented sequences. One could add more randomizations in several levels: randomization on registers, on operations, on memory addresses, ... to provide more countermeasures against differential SCAs. However, the security against SCAs of such an implementation must be evaluated by attacks.

Appendix A

Complete ECC Processor

During the thesis, i spent three months at *University College Cork* (UCC, Ireland) through a doctoral exchange student which was completed by using the funds from the European mobility grant from UEB (*Université Européenne de Bretagne*). I worked under the supervision of Dr. Liam Marnane and with members of the *Coding and Cryptography Research group*: Dr. Brian Baldwin, Dr. Andrew Byrne and Neil Hanley.

A generic architecture was designed for ECC operations (figure A.1) at UCC. The architecture incorporates a RAM block, a ROM controller and some arithmetic units for a given field. Everything can be configured from the size of the RAM block, to arithmetic units operations (number of modular additions and multiplications) and generating the ROM instructions for a given algorithm. A set of sequences and a simple state machine controls the processor. The ECC processor is configurable for some scalar multiplication algorithms and for any characteristic and field but here the focus is on the prime field implementations [21, Chap. 3].

A customized processor has been generated using this architecture. VHDL files are generated according to a scalar multiplication algorithm, a prime field, and a specific number of arithmetic units. Thus, we have the source code of the generated ECC processor. It has been adapted to our needs.

The initial task was to perform a side-channel attack with the generated processor (chapter 3). Then, a unit which recodes on-the fly the scalar k (section 3.3.3) was added to the processor. In addition, more scalar multiplication algorithms with their appropriate parameters have been added in order to compare some proposed methods to a complete ECC processor.

FPGA implementations reported in this appendix have been described in VHDL and implemented on a XC5VLX50 FPGA using ISE 9.2 from Xilinx with standard efforts for synthesis, place and route. We report best clock frequencies and numbers of occupied slices. We also report numbers of look-up tables (LUTs with 6 inputs in Virtex 5) and flip-flops (FFs) for area. A XC5VLX50 contains 7200 slices with 4 LUT and 4 flip-flops per slice.

Table A.1 reports FPGA implementations of an ECC processor (for curves over $\mathbb{F}_p$, $n \in \{160, 192\}$ bits and Jacobian coordinates). The first one (small version) uses the `NAF` method with one arithmetic unit per field operation, while the second (large version) uses the 4`NAF` method and two arithmetic units per field operation but one modular inversion. The implementation with the Euclidean addition chains method was provided and generated by the cryptographic processor generator provided by the author of [24].

ASIC implementations reported in this appendix have been synthesized into gate-level netlists using standard V_{th} (SVT) cells of an industrial 130nm bulk CMOS technology library using

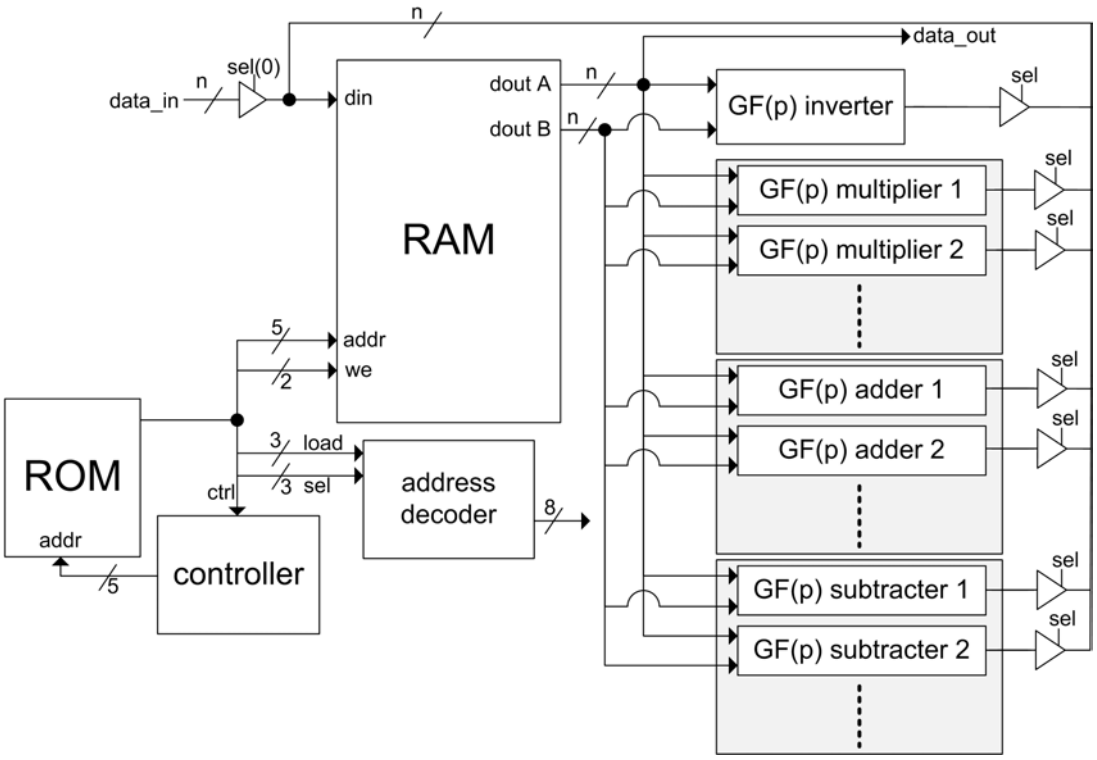


Figure A.1: Reconfigurable cryptographic processor (from [24]).

Synopsys Design Compiler G-2012.06-SP5 applying a maximum path delay constraint of 10ns from all inputs to all outputs. The standard cells used in both cases were restricted to a set {nand2, nor2, xor2, inv} of logic gates without loss of generality.

Tables A.2 and A.3 report ASIC implementations of an ECC processor (for curves over $\mathbb{F}_p$, $n = 160$ bits and Jacobian coordinates) using two methods. The first one (small version) uses the `NAF` method with one arithmetic unit per field operation, while the second (large version) uses the `4NAF` method and two arithmetic units per field operation but one modular inversion.

n	method	version	memory type	area		freq.
				slices (FF/LUT)	BRAM	MHz
160	NAF	small	distributed	2 204 (3 971/6 816)	0	155
			BRAM	1 793 (3 641/6 182)	6	155
	4NAF	large	distributed	3 182 (4 668/7 361)	0	142
			BRAM	2 427 (4 297/6 981)	6	142
192	double-and-add-always		distributed	2 550 (2 484/8 507)	0	150
			BRAM	2 084 (2 073/7 985)	6	150
	Euclidean addition chains		distributed	2 455 (2 200/7 945)	0	154
			BRAM	2 035 (1 933/7 022)	6	154

Table A.1: FPGA implementation results for a complete ECC processor over $\mathbb{F}_p$.

method	combinational	buf/inv	non combinational	total
NAF (small)	95 980.8	10 853.2	87 458.9	185 174.9
4NAF (large)	170 099.8	16 740.7	142 379.4	321 412.7

Table A.2: Area results (in μm^2) of ASIC implementation for a complete ECC processor over $\mathbb{F}_p$ with $n = 160$.

method	cell internal	net switching	total dynamic	cell leakage
NAF (small)	7 179.3	300.9	7 480.2	82.7
4NAF (large)	9 585.6	385.9	9 971.5	111.7

Table A.3: Power results in (μW) of ASIC implementation for a complete ECC processor over $\mathbb{F}_p$ with $n = 160$.

Appendix B

Proof of Exact Division Algorithm Starting from MSW

In this appendix, we give a proof of exact division algorithm starting from MSW presented in section 4.1.3. The algorithm is written below. $k = (k_{n-1}k_{n-2}\dots k_1k_0)_2$ is the n-bit *scalar* ($k > 1$) stored into a t words by m bits memory with $m(t-1) < n \leq mt$ (i.e. last word may be padded using 0s). $k^{(i)}$ denotes the ith word of k starting from least significant for $1 \leq i \leq t$.

input: two integers $k = (k^{(t)} \cdots k^{(1)})$ and b_j, where $k \bmod b_j = 0$
output: k/b_j

1: $c \leftarrow 0$
2: **for** i **from** t **to** 1 **do**
3: $\quad c \leftarrow (c + k^{(i)}) \bmod b$
4: $\quad r \leftarrow k^{(i)} - c$
5: $\quad r \leftarrow r \times (b^{-1} \bmod 2^m)$
6: $\quad k^{(i)} \leftarrow (r_{m-1} \cdots r_0)$
7: **return** k

Figure B.1: Exact division k/b_j algorithm starting most significant words first.

Proof. When one unrolls algorithm in figure 4.9 with the integer k and the divisor b, the exact division corresponds to:

$$
\begin{aligned}
k' = k/b = & \left(\left(\left(k^{(t)} - (k^{(t)} \bmod b) \right) \times (b^{-1} \bmod 2^m) \right) \bmod 2^m \right) \times 2^{m(t-1)} + \\
& \left(\left(\left(k^{(t-1)} - (k^{(t)} + k^{(t-1)} \bmod b) \right) \times (b^{-1} \bmod 2^m) \right) \bmod 2^m \right) \times 2^{m(t-2)} + \\
& \cdots + \\
& \left(\left(k^{(1)} - \left(\sum_{j=1}^{t} k^{(j)} \bmod b \right) \right) \times (b^{-1} \bmod 2^m) \right) \bmod 2^m.
\end{aligned}
$$

When $k'^{(i)} = k^{(i)}/b$ is the ith word of the exact division k/b, we have the following equation:

$$k'^{(i)} = k^{(i)}/b = \Big(\Big(k^{(i)} - (\sum_{j=i}^{t} k^{(j)} \bmod b)\Big) \times (b^{-1} \bmod 2^m)\Big) \bmod 2^m.$$

If one writes $k^i = bk_i + c_i$ with $k_i \equiv c_i \bmod b$, then the previous equation becomes:

$$k'^{(i)} = k^{(i)}/b = \Big(\Big(k^{(i)} - (\sum_{j=i}^{t} c_j \bmod b)\Big) \times (b^{-1} \bmod 2^m)\Big) \bmod 2^m.$$

In order to prove algorithm figure 4.9, we calculate $b \times k'$ and we will verify that:

$$b \times k' \;=\; k \;=\; \sum_{i=1}^{m} k^{(i)} 2^{i-1} \;=\; \sum_{i=1}^{m} (k_i b + c_i) 2^{i-1}.$$

That is $b \times k'^{(i)} = k^{(i)} = k_i b + c_i$. Below, we will write b^{-1} instead of $(b^{-1} \bmod 2^m)$. In addition, m has be chosen such that $2^m \equiv 1 \bmod b$. Thus $2^m - 1$ is multiple of b, and $\gamma = \frac{2^m-1}{b} \in]0, 2^m[$. In addition, $-\gamma \equiv b^{-1} \bmod 2^m$ because $b\gamma \equiv -1 \bmod 2^m$, and so $\gamma \equiv (-b)^{-1} \bmod 2^m$.

$$t^{(i)}/b \;=\; k_i + c_i b^{-1} \;=\; k_i + c_i \times -\frac{2^m - 1}{b} \;=\; k_i + c_i \times \frac{2^{m+1} + 1}{b},$$

because $\frac{2^{m+1}+1}{b} \equiv -\frac{2^m-1}{b} \equiv b^{-1} \bmod b$. When $2c_i + 1 \leq b$, $t^{(i)}/b < \frac{2^m-1}{b}$ because $\frac{2^m-1}{b} + \frac{2^{m+1}}{b} = \frac{3\times 2^m}{b}$ with the maximum value of k_i is $\frac{2^m-1}{b}$. Else the value $t^{(i)}/b$ is larger than 2^m and one needs to add the carry 2^m to the next computation $b \times t'^{(i+1)} = b \times (t^{(i+1)}/b)$ which corresponds to $b \times 2^m$. For simplification, we add the value αb with $\alpha \in \{0, 1\}$ to all computations $b \times k'^{(i)}$ $\forall i \in \{2, \cdots, m\}$ instead of having two cases.

Computation for $i = 1$, that is $b \times k'^{(1)}$:

$$b \times k'^{(1)} \;=\; b \times \Big(k_1 + c_1 b^{-1} - \big((\sum_{i=1}^{m} c_i) \bmod b\big) b^{-1}\Big) \;=\; b \times (k_1 + c_1 b^{-1}),$$

because k is divisible by b, and so $(\sum_{i=1}^{m} k^{(i)} \bmod b) \equiv (\sum_{i=1}^{m} c_i \bmod b) \equiv 0 \bmod b$. By replacing b^{-1} by $-\gamma$, we have:

$$\begin{aligned}
b \times k'^{(1)} &= b \times (k_1 + c_1 b^{-1}) \\
&= bk_1 - b\gamma c_1 \\
&= bk_1 - (2^m - 1)c_1 \\
&= bk_1 + c_1 - 2^m(c_1 + \alpha b) \\
&= bk_1 + c_1 + 2^m\big(- (\sum_{i=2}^{m} -c_i) \bmod b + \alpha b\big) \\
&= k^{(1)} + 2^m\Big(- c_1 + \big((\sum_{i=1}^{m} c_i) \bmod b\big) + \alpha b\Big).
\end{aligned}$$

The final expression contains two terms, $bk_1 + c_1 = bk'^{(1)} = k^{(1)}$ and $2^m\big(- (\sum_{i=2}^{m} -c_i) \bmod b\big)$. The second term is obviously greater than 2^m, and so will be added to the next operation $b \times k'^{(2)}$. We have the first expression expected, that is $k^{(1)} = bk'^{(1)} = bk_1 + c_1$. Below, we directly replace

bb^{-1} by $-3\gamma = (-2^m + 1)$.

Computation for $i = 2$, that is $b \times k'^{(2)}$:

$b \times k'^{(2)} = b \times \Big(k_2 + c_2 b^{-1} - \big((\sum_{i=2}^{m} c_i) \bmod b\big) b^{-1}\Big)$. However one needs to add the previous carry computed during $b \times k'^{(1)}$.

$$
\begin{aligned}
b \times k'^{(2)} &= b \times \Big(k_2 + c_2 b^{-1} - \big((\sum_{i=2}^{m} c_i) \bmod b\big) b^{-1}\Big) - \big((\sum_{i=2}^{m} -c_i) \bmod b\big) + \alpha b \\
&= bk_2 + c_2 - \big((\sum_{i=2}^{m} c_i) \bmod b\big) - \big((\sum_{i=2}^{m} -c_i) \bmod b\big) + \alpha b \\
&\qquad + 2^m \Big(- c_2 + \big((\sum_{i=2}^{m} c_i) \bmod b\big) + \alpha b\Big) \\
&= bk_2 + c_2 + 2^m \Big(- c_2 + \big((\sum_{i=2}^{m} c_i) \bmod b\big) + \alpha b\Big) \\
&= k^{(2)} + 2^m \Big(- c_2 + \big((\sum_{i=2}^{m} c_i) \bmod b\big) + \alpha b\Big)
\end{aligned}
$$

In a same way, the final expression of $b \times k'^{(2)}$ contains two terms, $bk_2 + c_2 = bk'^{(2)} = k^{(2)}$ and $2^m \Big(- c_2 + \big((\sum_{i=2}^{m} c_i) \bmod b\big)\Big)$. This second term, greater than 2^m, will be added to the next operation $b \times k'^{(3)}$. In addition,

$$\big((\sum_{i=2}^{m} c_i) \bmod b\big) - \big((\sum_{i=2}^{m} -c_i) \bmod b\big) + \alpha b = 0,$$

because

$$\sum_{i=2}^{m} c_i - b = (\sum_{i=2}^{m} c_i) \bmod b,$$

if $\sum_{i=2}^{m} c_i \geq b$, that is when $\alpha = 1$. That is why

$$c_2 - \big((\sum_{i=2}^{m} c_i) \bmod b\big) - \big((\sum_{i=2}^{m} -c_i) \bmod b\big) + \alpha b = c_2.$$

Computation for $i = 3$, that is $b \times k'^{(3)}$:

$$b \times k'^{(3)} = b \times \Big(k_3 + c_3 b^{-1} - \big((\sum_{i=3}^{m} c_i) \bmod b\big) b^{-1}\Big).$$

However one needs to add the previous carry computed during $b \times k'^{(2)}$.

$$\begin{aligned} b \times k'^{(3)} &= b \times \Big(k_3 + c_3 b^{-1} - \big((\sum_{i=3}^{m} c_i) \bmod b\big) b^{-1}\Big) + \Big(- c_2 + \big((\sum_{i=2}^{m} c_i) \bmod b\big)\Big) + \alpha b \\ &= bk_3 + c_3 - \big((\sum_{i=3}^{m} c_i) \bmod b\big) - c_2 + \big((\sum_{i=2}^{m} c_i) \bmod b\big) + \alpha b \\ &\quad + 2^m \Big(- c_3 + \big((\sum_{i=3}^{m} c_i) \bmod b\big) + \alpha b\Big) \\ &= bk_3 + c_3 + 2^m \Big(- c_3 + \big((\sum_{i=3}^{m} c_i) \bmod b\big) + \alpha b\Big) \\ &= k^{(3)} + 2^m \Big(- c_3 + \big((\sum_{i=3}^{m} c_i) \bmod b\big) + \alpha b\Big). \end{aligned}$$

This computation enables to have the expected result $bk_3 + c_3 = bk'^{(3)} = k^{(3)}$ plus a carry which will be added to the next $b \times k'^{(4)}$. We can see that the result of $b \times k'^{(3)}$ is similar to the previous calculation of $b \times k'^{(2)}$. Thus all computations $b \times k'^{(j)}$ are in the form for $1 < j < m$.

Computation all values $j \in \{2, \cdots, m-1\}$, that is $b \times k'^{(j)}$:

$$\begin{aligned} b \times k'^{(j)} &= b \times \Big(k_j + c_j b^{-1} - \big((\sum_{i=j}^{m} c_i) \bmod b\big) b^{-1}\Big) + \Big(- c_{j-1} + \big((\sum_{i=j-1}^{m} c_i) \bmod b\big)\Big) + \alpha b \\ &= bk_j + c_j - \big((\sum_{i=j}^{m} c_i) \bmod b\big) - c_{j-1} + \big((\sum_{i=j-1}^{m} c_i) \bmod b\big) + \alpha b \\ &\quad + 2^m \Big(- c_j + \big((\sum_{i=j}^{m} c_i) \bmod b\big) + \alpha b\Big) \\ &= bk_j + c_j + 2^m \Big(- c_j + \big((\sum_{i=j}^{m} c_i) \bmod b\big) + \alpha b\Big) \\ &= k^{(j)} + 2^m \Big(- c_j + \big((\sum_{i=j}^{m} c_i) \bmod b\big) + \alpha b\Big). \end{aligned}$$

We can see that we have the expected result, corrected by the carry obtained during the computation for $b \times k'^{(j-1)}$. In addition,

$$\big((\sum_{i=j}^{m} c_i) \bmod b\big) - \big((\sum_{i=j}^{m} -c_i) \bmod b\big) + \alpha b = 0,$$

because

$$\sum_{i=j}^{m} c_i - b = (\sum_{i=j}^{m} c_i) \bmod b,$$

if $\sum_{i=j}^{m} c_i \geq b$, that is when $\alpha = 1$. That is why:

$$c_j - \big((\sum_{i=j}^{m} c_i) \bmod b\big) - \big((\sum_{i=j}^{m} -c_i) \bmod b\big) + \alpha b = c_j.$$

Now, we must compute $b \times k'^{(m)}$ knowing that:

$$\begin{aligned} k^{(m)}/b &= k^{(m)} - (k^{(m)} \bmod b) \\ &= k_m + c_m - c_m \\ &= k_m, \end{aligned}$$

in adding the carry obtained during the computation $b \times k'^{(m-1)}$, that is

$$-c_{m-1} + \big((\sum_{i=m-1}^{m} c_i) \bmod b\big) :$$

$$\begin{aligned} b \times k'^{(m)} &= b \times \Big(k_m - c_{m-1} + \big((\sum_{i=m-1}^{m} c_i) \bmod b\big)\Big) + \alpha b \\ &= bk_m - bc_{m-1} + b\big((c_{m-1} + c_m) \bmod b)\big) + \alpha b \\ &= k_m + c_m \\ &= k^{(m)}. \end{aligned}$$

Thus, we can see that:

$$b \times k'^{(i)} \;=\; k_i b + c_i \;=\; k^{(i)},$$

for all $i \in [0, m]$. Thus:

$$b \times k' \;=\; \sum_{i=1}^{m} (k_i b + c_i) 2^{i-1} \;=\; \sum_{i=1}^{m} k^{(i)} 2^{i-1} \;=\; k,$$

which is the expected result.

□

Appendix C

Montgomery Inversion

Montgomery inversion [72] has been implemented following algorithms in figure 1.13 and 1.14 presented in section 1.3. Contrary to the modular addition and multiplication implementations presented in section 5.4.1, execution time is not constant. It changes in function of the number to be inverted. Figure C.1 presents the architecture of the Montgomery inversion which performs $a^{-1} \bmod p$ where p is a large prime number.

The global controller generates high-level control signals for the architecture unit. The controller calls for right shifts registers u and v, and according to this shift, it calls for left shifts registers r and v (algorithm in figure 1.13). During the second phase (algorithm in figure 1.14), the controller selects between a shift or a sum according the register value r.

The FPGA implementation results for the Montgomery inversion are reported in table C.1 for different parameters of w and t. Hardware implementation results have been described in VHDL and implemented on a XC5VLX50T FPGA using ISE 14.1 from Xilinx with standard efforts for synthesis, place and route. We report numbers of clock cycles, best clock frequencies and numbers of occupied slices. We also report numbers of look-up tables (LUTs with 6 inputs in Virtex 5) and flip-flops (FFs) for area. A XC5VLX50T contains 7200 slices with 4 LUTs and 4 flip-flops per slice. We use flip-flops for all storage elements. The implementation has been tested on 100 000 random number a. We give the minimum and the maximum clock cycles which has been necessary to perform a Montgomery inversion for the 100 000 random numbers.

			area	freq.	clock
n	t	w	slices (FF/LUT)	MHz	cycles
160	20	8	1 132 (1 975/3 054)	130	9 632 – 13 028
	14	12	1 198 (2 071/3 388)	130	8 359 – 11 284
	10	16	1 094 (1 974/2 984)	130	5 117 – 6 894
	7	24	1 134 (2 069/3 257)	130	4 133 – 5 562
	5	32	1 124 (1 973/3 301)	132	2 270 – 3 038
224	28	8	1 473 (2 743/4 248)	130	11 027 – 14 901
	14	16	1 541 (2 742/4 188)	130	6 155 – 8 293
	7	32	1 496 (2 743/4 253)	132	2 675 – 3 582

Table C.1: FPGA implementation results for Montgomery inversion.

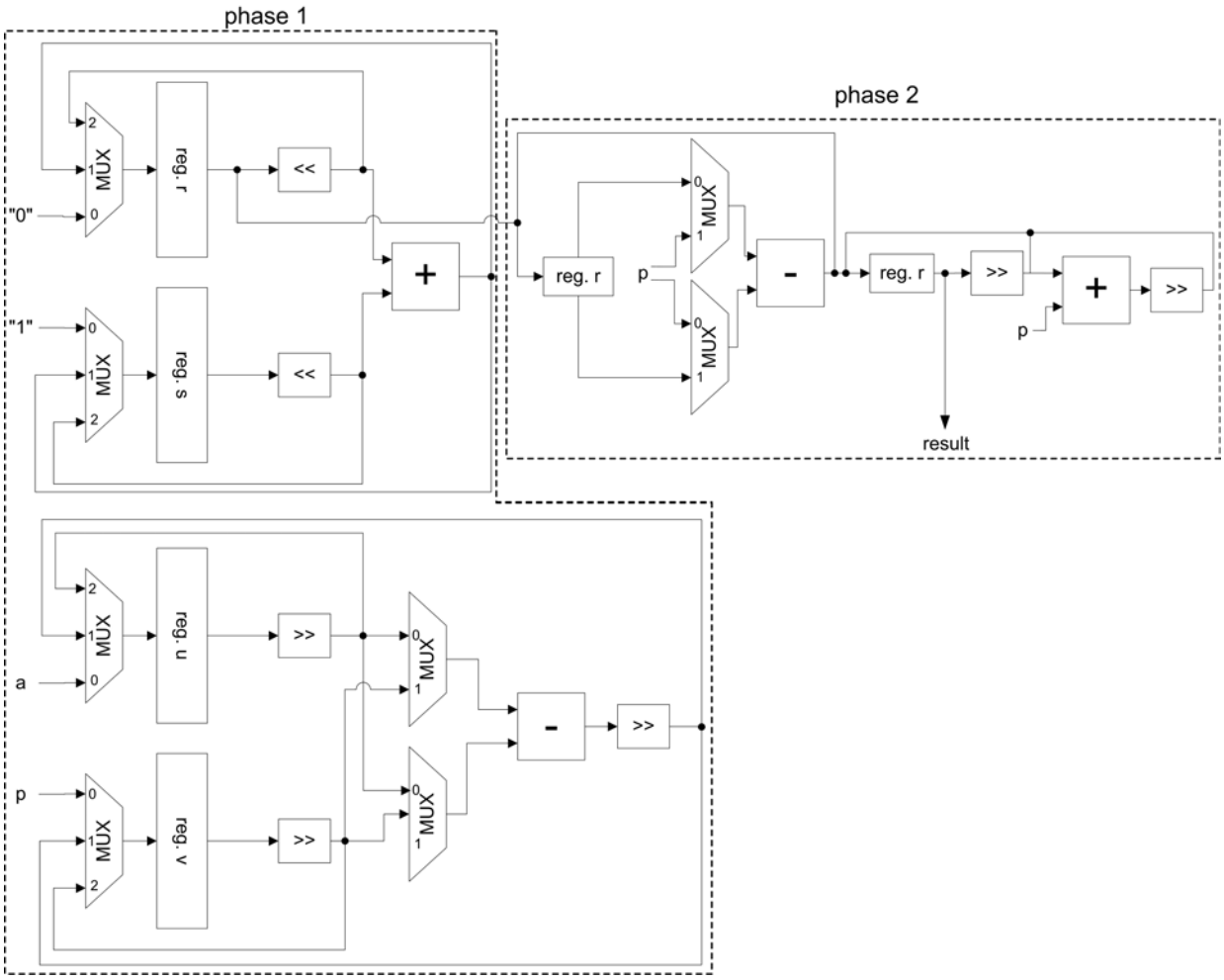


Figure C.1: Architecture for Montgomery inversion.

Bibliography

[1] J. Adikari, V. Dimitrov, and L. Imbert. Hybrid binary-ternary joint form and its application in elliptic curve cryptography. In *Proc. 19th Symposium on Computer Arithmetic (ARITH)*, pages 76–83. IEEE Computer Society, 2009.

[2] J. Adikari, V. S. Dimitrov, and L. Imbert. Hybrid binary-ternary number system for elliptic curve cryptosystems. *IEEE Transaction Computers*, 60(2):254–265, 2011.

[3] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM side-channel(s). In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *LNCS*, pages 29–45. Springer, 2003.

[4] T. Akishita and T. Takagi. Zero-value point attacks on elliptic curve cryptosystem. In *Proc. 6th Information Security Conference (ISC)*, volume 2851 of *LNCS*, pages 218–233. Springer, 2003.

[5] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10(3):389–400, 1961.

[6] R. Barua, S. K. Pandey, and R. Pankaj. Efficient window-based scalar multiplication on elliptic curves using double-base number system. In *Proc. 8th International Conference on Progress in Cryptology (INDOCRYPT)*, volume 4859 of *LNCS*, pages 351–360. Springer, 2007.

[7] D. J. Bernstein, P. Birkner, T. Lange, and C. Peters. Optimizing double-base elliptic-curve single-scalar multiplication. In *Proc. 8th International Conference on Progress in Cryptology (INDOCRYPT)*, volume 4859 of *LNCS*, pages 167–182. Springer, 2007.

[8] V. Berthé and L. Imbert. On converting numbers to the double-base number system. In *Advanced Signal Processing Algorithms Architectures and Implementations XIV (SPIE)*, volume 5559, pages 70–78, 2004.

[9] V. Berthé and L. Imbert. Diophantine approximation, Ostrowski numeration and the double-base number system. *Discrete Mathematics and Theoretical Computer Science*, 11(1):153–172, 2009.

[10] G. M. Bertoni, L. Breveglieri, P. Fragneto, G. Pelosi, and P. Di Milano. Parallel hardware architectures for the cryptographic tate pairing. In *Proc. 3rd International Conference on Information Technology: New Generations (ITNG)*, pages 186–191. IEEE Computer Society, 2006.

[11] K. Bigou, T. Chabrier, and A. Tisserand. Opérateur matériel de tests de divisibilité par des petites constantes sur de très grands entiers. In *15ème Symposium en Architectures nouvelles de machines (SympA)*, 2013.

[12] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic curves in cryptography*, volume 265. Cambridge University, 1999.

[13] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[14] W. Bosma. Signed bits and fast exponentiation. *Journal de théorie des nombres de Bordeaux*, 13:27–41, 2001.

[15] A. Bosselaers, R. Govaerts, and J. Vandewalle. Comparison of three modular reduction functions. In *Proc. 13th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, volume 773 of *LNCS*, pages 175–186. Springer, 1994.

[16] N. Boullis and A. Tisserand. Some optimizations of hardware multiplication by constant matrices. *IEEE Transactions on Computers*, 54(10):1271–1282, 2005.

[17] E. Brier and M. Joye. Fast point multiplication on elliptic curves through isogenies. In *Proc. 15th International Symposium Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC)*, volume 2643 of *LNCS*, pages 43–50. Springer, 2003.

[18] E. Brier, M. Joye, and T. E. De Win. Weierstrass elliptic curves and side-channel attacks. In *Proc. 6th International Conference on Theory and Practice in Public Key Cryptography (PKC)*, volume 2274 of *LNCS*, pages 335–345. Springer, 2002.

[19] M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the NIST elliptic curves over prime fields. In *Proc. 1st Conference on Topics in Cryptology: The Cryptographer's Track at RSA*, volume 2020 of *LNCS*, pages 250–265. Springer, 2001.

[20] J. A. Buchmann. *Introduction to cryptography*. Springer, second edition, 2004.

[21] A. Byrne. *Reconfigurable Architectures for Elliptic Curve and Pairing Based Cryptography*. PhD thesis, University College Cork, Ireland, 2010.

[22] A. Byrne, N. Méloni, F. Crowe, W. P. Marnane, A. Tisserand, and E. M. Popovici. SPA resistant elliptic curve cryptosystem using addition chains. *International Journal of High Performance Systems Architecture*, 1(2):133–142, 2007.

[23] A. Byrne, N. Méloni, A. Tisserand, E. M. Popovici, and W. P. Marnane. Comparison of simple power analysis attack resistant algorithms for an elliptic curve cryptosystem. *Journal of Computers*, 2(10):52–62, 2007.

[24] A. Byrne, E. M. Popovici, and W. P. Marnane. Versatile processor for GF(p^m) arithmetic for use in cryptographic applications. *IET Computers & Digital Techniques*, 2:253–264, 2008.

[25] T. Chabrier, D. Pamula, and A. Tisserand. Hardware implementation of DBNS recoding for ECC processor. In *Asilomar Conference on Signals, Systems and Computers*, pages 1129–1133. IEEE, 2010.

[26] T. Chabrier and A. Tisserand. On-the-fly multi-base recoding for ECC scalar multiplication without pre-computations. In *Proc. 21th Symposium on Computer Arithmetic (ARITH)*. IEEE Computer Society, 2013.

[27] S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *LNCS*, pages 13–28. Springer, 2002.

[28] B. Chevallier-Mames, M. Ciet, and M. Joye. Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.

[29] M. Ciet, M. Joye, K. Lauter, and P. L. Montgomery. Trading inversions for multiplications in elliptic curve cryptography. *Designs, Codes and Cryptography*, 39(2):189–206, 2006.

[30] C. Clavier, J.-S. Coron, and N. Dabbous. Differential power analysis in the presence of hardware countermeasures. In *Proc. 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 1965 of *LNCS*, pages 252–263. Springer, 2000.

[31] C. Clavier and M. Joye. Universal exponentiation algorithm - a first step towards provable SPA-resistance. In *Proc. 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *LNCS*, pages 300–308. Springer, 2001.

[32] H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, first edition, 2000.

[33] H. Cohen, M. Atsuko, and O. Takatoshi. Efficient elliptic curve exponentiation using mixed coordinates. In *Proc. 8th International Conference on the Theory and Applications of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT)*, volume 1514 of *LNCS*, pages 51–65. Springer, 1998.

[34] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Mathematics and its Applications. Chapman and Hall/CRC, 2005.

[35] J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 1717 of *LNCS*, pages 292–302. Springer, 1999.

[36] R. Crandall and C. Pomerance. *Prime numbers: a computational perspective*, volume 182. Springer, 2001.

[37] J.-P. Deschamps, J. L. Imana, and G. D. Sutter. *Hardware implementation of finite-field arithmetic*. McGrawHill-Hill, 2009.

[38] V. Dimitrov and T. Cooklev. Hybrid algorithm for the computation of the matrix polynomial $I + A + \ldots + A^{N-1}$. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 42(7):377–380, 1995.

[39] V. Dimitrov and T. Cooklev. Two algorithms for modular exponentiation using nonstandard arithmetics. *IEICE Transaction on Fundamentals of Electronics, Communications and Computer Sciences*, 78(1):82–87, 1995.

[40] V. Dimitrov, L. Imbert, and P. K. Mishra. Efficient and secure elliptic curve point multiplication using double-base chains. In *Proc. 11th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, volume 3788 of *LNCS*, pages 59–78. Springer, 2005.

[41] V. Dimitrov, L. Imbert, and P. K. Mishra. The double-base number system and its application to elliptic curve cryptography. *Mathematics of Computation*, 77(262):1075–1104, 2008.

[42] V. Dimitrov, G. Jullien, and R. Muscedere. *Multiple-base Number System: Theory and Applications*, volume 2. CRC Press, 2012.

[43] V. S. Dimitrov, G. A. Jullien, and W. C. Miller. An algorithm for modular exponentiation. *Information Processing Letters*, 66(3):155–159, 1998.

[44] V. S. Dimitrov, G. A. Jullien, and W. C. Miller. Theory and applications of the double-base number system. *IEEE Transactions on Computers*, 48(10):1098–1106, 1999.

[45] C. Doche and L. Habsieger. A tree-based approach for computing double-base chains. In *Proc. 13th Australasian Conference on Information Security and Privacy (ACISP)*, volume 5107 of *LNCS*, pages 433–446. Springer, 2008.

[46] C. Doche and L. Imbert. Extended double-base number system with applications to elliptic curve cryptography. In *Proc. 7th International Conference on Cryptology (INDOCRYPT)*, volume 4329 of *LNCS*, pages 335–348. Springer, 2006.

[47] C. Doche, D. R. Kohel, and F. Sica. Double-base number system for multi-scalar multiplications. In *Proc. 28th Annual International Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, volume 5479 of *LNCS*, pages 502–517. Springer, 2009.

[48] V. Dupaquis and A. Venelli. Redundant modular reduction algorithms. In *Smart Card Research and Advanced Applications*, volume 7079 of *LNCS*, pages 102–114. Springer, 2011.

[49] S. R. Dussé and B. S. Kaliski. A cryptographic library for the motorola DSP56000. In *Proc. 9th Annual International Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, volume 473 of *LNCS*, pages 230–244. Springer, 1991.

[50] N. Ebeid and M. A. Hasan. On binary signed digit representations of integers. *Designs, Codes and Cryptography*, 42(1):43–65, 2007.

[51] H. M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, 2007.

[52] P. Fouque, G. Leurent, D. Réal, and F. Valette. Practical electromagnetic template attack on HMAC. In *Proc. 11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 5747 of *LNCS*, pages 66–80. Springer, 2009.

[53] B. Gierlichs, K. Lemke-Rust, and C. Paar. Templates vs. stochastic methods. In *Proc. 8th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 4249 of *LNCS*, pages 15–29. Springer, 2006.

[54] P. Giorgi, L. Imbert, and T. Izard. Optimizing elliptic curve scalar multiplication for small scalars. In *Proc. Mathematics for Signal and Information Processing*, volume 7444, page 74440N. SPIE, 2009.

[55] C. Giraud and V. Verneuil. Atomicity improvement for elliptic curve scalar multiplication. In *Smart Card Research and Advanced Application*, volume 6035 of *LNCS*, pages 80–101. Springer, 2010.

[56] A. Guyot, Y. Herreros, and J.-M. Muller. JANUS, an on-line multiplier/divider for manipulating large numbers. In *Proc. 9th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 106–111. IEEE Computer Society, 1989.

[57] D. Hankerson, S. Vanstone, and A. Menezes. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[58] N. Hanley, M. Tunstall, and W. P. Marnane. Using templates to distinguish multiplications from squaring operations. *International Journal of Information Security*, 10(4):255–266, 2011.

[59] K. Hensel. *Theorie der algebraischen Zahlen*, volume 1. BG Teubner, 1908.

[60] J. Hoffstein, J. Pipher, and J. H. Silverman. *An Introduction to Mathematical Cryptography*. Springer, first edition, 2008.

[61] K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara. Fast implementation of public-key cryptography on a DSP TMS320C6201. In *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 1717 of *LNCS*, pages 61–72. Springer, 1999.

[62] T. Jebelean. An algorithm for exact division. *Journal of Symbolic Computation*, 15(2):169–180, 1993.

[63] I. T. Jolliffe. *Principal Component Analysis*. Springer, second edition, 2002.

[64] M. Joye. Introduction élémentaire à la théorie des courbes elliptiques. Technical report, Université Catholique de Louvain UCL, 1995.

[65] M. Joye. *Advances in Elliptic Curve Cryptography*, volume 317 of *London Mathematical Society Lecture Note Series*, chapter Defenses Against Side-Channel Analysis, pages 87–100. Cambridge University Press, 2005.

[66] M. Joye, Ç. K. Koç, C. Paar, and S.-M. Yen. The Montgomery powering ladder. In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *LNCS*, pages 291–302. Springer, 2003.

[67] M. Joye and C. Tymen. Protections against differential analysis for elliptic curve cryptography - an algebraic approach. In *Proc. 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *LNCS*, pages 377–390. Springer, 2001.

[68] D. Kahn. *The Codebreakers: The Story of Secret Writing*. Scribner Book Company, second edition, 1996.

[69] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, P. Gaudry, P. L. Montgomery, D. A. Osvik, H. T. Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In *Proc. 30th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, volume 6223 of *LNCS*, pages 333–350. Springer, 2010.

[70] D. E. Knuth. *The art of computer programming: seminumerical algorithms*, volume 2. Addison-Wesley, third edition, 1997.

[71] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[72] Ç. K. Koç. The Montgomery modular inverse - revisited. *IEEE Transactions on Computers*, 49(7):763–766, 2000.

[73] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proc. 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, LNCS, pages 388–397. Springer, 1999.

[74] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, second edition, 1994.

[75] P. Longa. Accelerating the scalar multiplication on elliptic curve cryptosystems over prime fields. Master's thesis, University of Ottawa, 2007.

[76] P. Longa and C. Gebotys. Setting speed records with the (fractional) multibase non-adjacent form method for efficient elliptic curve scalar multiplication. Technical Report 118, Cryptology ePrint Archive (IACR), 2008.

[77] P. Longa and C. Gebotys. Fast multibase methods and other several optimizations for elliptic curve scalar multiplication. In *Proc. 12th International Conference on Theory and Practice in Public Key Cryptography (PKC)*, volume 5443 of *LNCS*, pages 443–462. Springer, 2009.

[78] P. Longa and A. Miri. New multibase non-adjacent form scalar multiplication and its application to elliptic curve cryptosystems. Technical Report 52, Cryptology ePrint Archive (IACR), 2008.

[79] H. Mamiya, A. Miyaji, and H. Morimoto. Efficient countermeasures against RPA, DPA, and SPA. In *Proc. 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 3156 of *LNCS*, pages 343–356. Springer, 2004.

[80] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer, 2007.

[81] M. Medwed and E. Oswald. Template attacks on ECDSA. In *Proc. 9th International Workshop on Information Security Applications (WISA)*, volume 5379 of *LNCS*, pages 14–27. Springer, 2008.

[82] N. Méloni. New point addition formulae for ECC applications. In *Proc. 1st International Workshop on the Arithmetic of Finite Fields (WAIFI)*, volume 4547 of *LNCS*, pages 189–201. Springer, 2007.

[83] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[84] T. S. Messerges. Using second-order power analysis to attack DPA resistant software. In *Proc. 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 1965 of *LNCS*, pages 238–251. Springer, 2000.

[85] S. D. Miller and R. Venkatesan. Spectral analysis of Pollard rho collisions. In *Proc. 7th International Conference on Algorithmic Number Theory (ANTS)*, volume 4076 of *LNCS*, pages 573–581. Springer, 2006.

[86] V. S. Miller. Use of elliptic curves in cryptography. In *Proc. 5th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, volume 218 of *LNCS*, pages 417–426. Springer, 1986.

[87] P. K. Mishra and V. Dimitrov. Efficient quintuple formulas for elliptic curves and efficient scalar multiplication using multibase number representation. In *Proc. 10th International Conference on Information Security (ISC)*, volume 4779 of *LNCS*, pages 390–406. Springer, 2007.

[88] P. K. Mishra and V. Dimitrov. A graph theoretic analysis of double base number systems. In *Proc. 8th International Conference on Progress in Cryptology (INDOCRYPT)*, volume 4859 of *LNCS*, pages 152–166, 2007.

[89] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[90] P. L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[91] G.E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, International*, volume 21, pages 11–13. IEEE, 1975.

[92] J.-M. Muller. *Arithmétique des Ordinateurs*. Masson, 1989.

[93] R. Muscedere, V. S. Dimitrov, G. A. Jullien, and W. C. Miller. Efficient conversion from binary to multi-digit multi-dimensional logarithmic number systems using arrays of range addressable look-up tables. *Proc. 21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 130–138, 2002.

[94] P. Q. Nguyen and J. Stern. Lattice reduction in cryptology: An update. In *Proc. 4th International Symposium on Algorithmic Number Theory*, volume 1838 of *LNCS*, pages 85–112. Springer, 2000.

[95] National Institute of Standards and Technology (NIST). Recommended elliptic curves for federal government use, 1999. Available at `http://csrc.nist.gov/encryption`.

[96] J. Omura. A public key cell design for smart card chips. In *Proc. 1st International Symposium on Information Theory and its Applications (ISITA)*, pages 983–985, 1990.

[97] E. Oswald. *Advances in Elliptic Curve Cryptography*, volume 317 of *London Mathematical Society Lecture Note Series*, chapter Side Channel Analysis, pages 69–86. Cambridge University Press, 2005.

[98] E. Oswald and S. Mangard. Template attacks on masking - resistance is futile. In *Proc. 7th The Cryptographer's Track at RSA Conference (CT-RSA)*, volume 4377 of *LNCS*, pages 243–256. Springer, 2007.

[99] D. Pamula. *Arithmetic operators on $GF(2^m)$ for cryptographic applications: performance - power consumption - security tradeoffs.* PhD thesis, Silesian University of Technology (PL) and University of Rennes 1 (FR), 2012.

[100] K. K. Parhi. *VLSI digital signal processing systems: design and implementation.* John Wiley & Sons, 1999.

[101] B. Pascal. *Œuvres complètes*, volume 5, chapter De Numeribus Multiplicibus, pages 117–128. Librarie Lefèvre, 1819.

[102] G. N. Purohit and A. S. Rawat. Fast scalar multiplication in ECC using the multi base number system. *International Journal of Computer Science Issues*, 8(1):131–137, 2011.

[103] G. N. Purohit, A. S. Rawat, and M. Kumar. Elliptic curve point multiplication using MBNR and point halving. *International Journal of Advanced Networking and Applications*, 3(5):1329–1337, 2012.

[104] C. Rechberger and E. Oswald. Practical template attacks. In *Proc. 5th International Workshop on Information Security Applications (WISA)*, volume 3325 of *LNCS*, pages 440–456. Springer, 2004.

[105] G. W. Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:231–308, 1960.

[106] Certicom Research. Standards for efficient cryptography, SEC 1: Elliptic curve cryptography, 2000.

[107] M.-S. Rita. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *Proc. 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 1965 of *LNCS*, pages 78–92. Springer, 2000.

[108] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 21(2):120–126, 1978.

[109] J. Sakarovitch. *Elements of Automata Theory*, chapter Prologue: M. Pascal's Division Machine, pages 1–6. Cambridge, 2009.

[110] R. Santoro, O. Sentieys, and S. Roy. On-the-fly evaluation of FPGA-based true random number generator. In *Proc. 8th International Symposium on VLSI (ISVLSI)*, pages 55–60. IEEE Computer Society, 2009.

[111] R. Santoro, A. Tisserand, O. Sentieys, and S. Roy. Arithmetic operators for on-the-fly evaluation of TRNGs. In *Proc. Advanced Signal Processing Algorithms, Architectures and Implementations XVIII*, volume 7444, pages 1–12. SPIE, 2009.

[112] L. Sauvage, S. Guilley, and Y. Mathieu. Electromagnetic radiations of FPGAs: High spatial resolution cartography and attack on a cryptographic module. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(1):4–28, 2009.

[113] R. Schoof. Elliptic curves over finite fields and the computation of square roots mod p. *Mathematics of Computation*, 44(170):483–494, 1985.

[114] J. H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, second edition, 1985.

[115] J. H. Silverman. *Advanced Topics in the Arithmetic of Elliptic Curves*, volume 151 of *Graduate Texts in Mathematics*. Springer, 1994.

[116] J. H. Silverman and J. Suzuki. Elliptic curve discrete logarithms and the index calculus. In *Proc. 6th of the International Conference on the Theory and Applications of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT)*, LNCS, pages 110–125. Springer, 1998.

[117] F.-X. Standaert and C. Archambeau. Using subspace-based template attacks to compare and combine power and electromagnetic information leakages. In *Proc. 10st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 5154 of *LNCS*, pages 411–425. Springer, 2008.

[118] J. Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397–405, 1967.

[119] A. F. Tenca and Ç. K. Koç. A scalable architecture for Montgomery multiplication. In *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 1717 of *LNCS*, pages 94–108. Springer, 1999.

[120] R. A. Walker and S. Chaudhuri. Introduction to the scheduling problem. *IEEE Design & Test of Computers*, 12(2):60–69, 1995.