On the Cryptanalysis of Public-Key Cryptography

THÈSE N^{O} 5291 (2012)

PRÉSENTÉE LE 24 FÉVRIER 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE CRYPTOLOGIE ALGORITHMIQUE

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Joppe Willem BOS

acceptée sur proposition du jury:

Prof. B. Falsafi, président du jury
Prof. A. Lenstra, directeur de thèse
Dr P. C. Leyland, rapporteur
P. L. Montgomery, rapporteur
Prof. S. Vaudenay, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

Dit proefschrift is opgedragen aan mijn ouders
Jaap & Bettien

Acknowledgements

First and foremost I would like to thank my supervisor Arjen K. Lenstra for his guidance and advice during my PhD. I don't think there are many supervisors who give such detailed and constructive feedback to their PhD students. After the invitation to come and visit his laboratory for cryptologic algorithms (LACAL) at EPFL I quit my job in the Netherlands and came to Switzerland to start my PhD. My first year at EPFL I spent at the mathematics institute of geometry and applications at the chair of algebraic and geometric structures led by Eva Bayer-Fluckiger. I would like to thank all members of this laboratory for their help during my first year at EPFL.

I would like to thank all the post-doctoral researchers from LACAL who helped me during these years: Nicolas Gama, Dimitar P. Jetchev, Marcelo E. Kaihara, Thorsten Kleinjung, and Martijn Stam. Especially Thorsten, who was always patient and able to answer all my questions and acted as my second supervisor. Furthermore, I would also like to thank all the other PhD-students at LACAL: Maxime Augier, Alina Dudeanu, Andrea Miele, Seyyd Hasan Mirjalili, Alexandre Karlov, Shahram Khazaei, Dag Arne Osvik, Onur Özen, and Juraj Šarinay. Besides all the interesting discussions we also had a lot of fun during and after work and I discovered many great movies during the LACAL lunch-entertainment sessions. A special thanks goes out to the secretary of our lab: Monique Amhof. She was always available to help and assist if I had trouble with the local language or sort out any administrative troubles. Besides my colleagues I would like to thank Eline, which I married during my PhD, for all her support and love during this period. Especially when I was working from home and she noted that I was still looking at the "*boring screen*" (Linux terminal).

Part of this work was supported by the Swiss National Science Foundation under grant numbers 200021-119776, 206021-117409 and 206021-128727 and by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

Abstract

Nowadays, the most popular public-key cryptosystems are based on either the integer factorization or the discrete logarithm problem. The feasibility of solving these mathematical problems in practice are studied and techniques are presented to speed-up the underlying arithmetic on parallel architectures.

The fastest known approach to solve the discrete logarithm problem in groups of elliptic curves over finite fields is the Pollard rho method. The negation map can be used to speed up this calculation by a factor $\sqrt{2}$. It is well known that the random walks used by Pollard rho when combined with the negation map get trapped in fruitless cycles. We show that previously published approaches to deal with this problem are plagued by recurring cycles, and we propose effective alternative countermeasures. Furthermore, fast modular arithmetic is introduced which can take advantage of prime moduli of a special form using efficient "sloppy reduction." The effectiveness of these techniques is demonstrated by solving a 112-bit elliptic curve discrete logarithm problem using a cluster of PlayStation 3 game consoles: breaking a public-key standard and setting a new world record.

The elliptic curve method (ECM) for integer factorization is the asymptotically fastest method to find relatively small factors of large integers. From a cryptanalytic point of view the performance of ECM gives information about secure parameter choices of some cryptographic protocols. We optimize ECM by proposing carry-free arithmetic modulo Mersenne numbers (numbers of the form $2^M - 1$) especially suitable for parallel architectures. Our implementation of these techniques on a cluster of PlayStation 3 game consoles set a new record by finding a 241-bit prime factor of $2^{1181} - 1$.

A normal form for elliptic curves introduced by Edwards results in the fastest elliptic curve arithmetic in practice. Techniques to reduce the temporary storage and enhance the performance even further in the setting of ECM are presented. Our results enable one to run ECM efficiently on resource-constrained platforms such as graphics processing units.

Keywords: cryptanalysis, public-key cryptography, integer factorization, elliptic curve discrete logarithm problem, arithmetic

Résumé

De nos jours, les cryptosystèmes à clef publique les plus populaires sont basés soit sur le problème de la factorisation des entiers, soit sur celui du logarithme discret. La faisabilité de la résolution pratique de ces problèmes mathématiques est étudiée, et des techniques pour l'accélération de l'arithmétique sous-jacente sur des architectures parallèles sont présentées.

La plus rapide approche connue pour la résolution du problème du logarithme discret sur les groupes des courbes elliptiques sur corps finis est la méthode du Rho de Pollard. L'application de négation permet d'accélérer le calcul par un facteur $\sqrt{2}$. Il est communément reconnu que les marches aléatoires utilisées par le Rho de Pollard, en combinaison avec l'application de négation, s'égarent dans des cycles infructueux. Nous montrons que les approches précédentes pour éviter cette difficulté sont pénalisées par des cycles récurrents, et nous proposons des contre-mesures efficaces. De plus, nous introduisons une arithmétique modulaire rapide, qui tire avantage de modules premiers de forme spéciale, en utilisant l'efficace "réduction hâtive". Nous montrons l'efficacité de ces techniques en résolvant un problème de logarithme discret sur une courbe elliptique de 112 bits, sur un cluster de consoles de jeu PlayStation 3, cassant ainsi un standard de chiffrement à clef publique, et réalisant un nouveau record mondial.

La méthode des courbes elliptiques (ECM) pour la factorisation des entiers est la méthode la plus rapide asymptotiquement pour identifier de relativement petits facteurs de grands entiers. D'un point de vue cryptanalytique, la performance d'ECM fournit des informations sur la sûreté du choix des paramètres de certains protocoles cryptographiques. Nous optimisons ECM en proposant une arithmétique modulo un nombre de Mersenne quelconque (nombres de la forme $2^M - 1$) sans retenues, particulièrement adaptée aux architectures parallèles. Notre implémentation de ces techniques sur un cluster de consoles de jeu PlayStation 3 réalise un nouveau record en identifiant un facteur premier de 241 bits de $2^{1181} - 1$.

Une forme normale pour les courbes elliptiques, introduite par Edwards, donne en pratique l'arithmétique la plus rapide pour les courbes elliptiques. Nous présentons des techniques pour réduire le stockage temporaire et améliorer encore plus la performance dans le contexte d'ECM. Nos résultats permettent d'utiliser ECM efficacement sur des plateformes aux ressources limitées comme les GPU (processeurs graphiques).

Mots-clefs: cryptanalyse, cryptographie à clef publique, factorisation des entiers, problème de logarithme discret sur une courbe elliptique, arithmétique

Zusammenfassung

Die gebräuchlichsten Public-key Kryptosysteme beruhen heutzutage entweder auf dem Faktorisierungsproblem oder dem diskreten Logarithmus-Problem. In dieser Arbeit wird zum einen untersucht, inwieweit es möglich ist, diese mathematischen Probleme zu lösen, und zum anderen werden Techniken zur Beschleunigung der zugrundeliegenden Arithmetik auf parallelen Architekturen vorgestellt.

Pollard's rho Verfahren ist der schnellste bekannte Ansatz, das diskrete Logarithmus-Problem in der Gruppe der Punkte einer elliptischen Kurve über einem endlichen Körper zu lösen. Dieses Verfahren kann mittels der Negationsabbildung um einen Faktor $\sqrt{2}$ beschleunigt werden. Bekanntlich können dabei die Zufallswege aus Pollard's rho Methode in fruchtlosen Zyklen enden. Wir zeigen, dass die bisherigen Ansätze, dieses Problem zu lösen, mit dem Problem der wiederkehrenden Zyklen zu kämpfen haben, und schlagen effektive Alternativen vor. Ausserdem stellen wir für Primmoduli einer speziellen Form eine schnelle modulare Arithmetik vor, die effiziente „saloppe Reduktion“ benutzt. Mit der Lösung eines 112-Bit elliptischen diskreten Logarithmus-Problems auf einem Verbund von PlayStation 3 Spielkonsolen, was einen Public-key Standard bricht und einen neuen Weltrekord aufstellt, wird die Effektivität dieser Techniken unter Beweis gestellt.

Die asymptotisch schnellste Methode, relativ kleine Faktoren grosser Zahlen zu finden, ist die elliptische Kurven Methode (ECM). Für die Kryptographie ist sie wichtig, um Informationen über sichere Parameter für einige kryptographische Protokolle zu erhalten. Wir haben ECM mit einer übertragsfreien Arithmetik optimiert, die für Arithmetik modulo Mersennezahlen (Zahlen der Form $2^M - 1$) auf parallelen Architekturen besonders geeignet ist. Mit unserer Implementierung dieser Techniken haben wir auf einem Verbund von PlayStation 3 Spielkonsolen mit einem 241-Bit Primfaktor von $2^{1181} - 1$ einen neuen Rekord aufgestellt.

Eine von Edwards eingeführte Normalform für elliptischen Kurven führt zur schnellsten Arithmetik auf elliptische Kurven in der Praxis. Im Falle der Anwendung auf ECM stellen wir Techniken vor, die den temporären Speicherbedarf reduzieren und die Laufzeit noch weiter verbessern. Dies erlaubt uns, ECM auf ressourcenbeschränkten Plattformen wie Graphikprozessoren laufen zu lassen.

Schlagwörter: Kryptanalyse, Public-key Kryptographie, Primfaktorzerlegung, diskretes Logarithmus-Problem für elliptische Kurven, Arithmetik

Riassunto

Al giorno d'oggi, i sistemi crittografici a chiave pubblica più popolari, sono basati sul problema della fattorizzazione di numeri interi o su quello del logaritmo discreto. Verrà presentato lo studio relativo alla risoluzione di tali problemi matematici nella pratica e saranno presentate tecniche per accelerare l'aritmetica utilizzata su architetture parallele.

L'approccio più veloce per risolvere il problema del logaritmo discreto in un un gruppo di punti su una curva ellittica è il metodo rho di Pollard. L'utilizzo della "mappa di negazione" può essere adottato per velocizzare l'elaborazione di un fattore $\sqrt{2}$. E' ben noto che le passeggiate aleatorie usate da Pollard, combinate con la mappa di negazione, possono entrare in cicli infruttuosi. Mostreremo che, gli approcci pubblicati precedentemente in letteratura per affrontare questo problema, sono affetti da cicli ricorrenti e proporremo contromisure alternative efficaci. Inoltre, verrà introdotta un'aritmetica modulare veloce, per moduli dalla forma speciale, basata su una tecnica di riduzione efficiente denominata "riduzione pigra". L'efficacia di tali tecniche è stata dimostrata risolvendo un'instanza del problema del logaritmo discreto su una curva ellittica a 112-bit usando un cluster di console PlayStation 3: uno standard crittografico a chiave pubblica è stato attaccato con successo ed un nuovo record mondiale è stato stabilito.

Il metodo delle curve ellittiche (ECM) per la fattorizzazione di interi è asintoticamente il metodo più veloce per trovare fattori piccoli (relativamente) di interi molto grandi. Dal punto di vista della crittanalisi le prestazioni di ECM influiscono sulla scelta dei parametri di sicurezza di alcuni protocolli crittografici. Noi abbiamo ottimizzato ECM, proponendo un'aritmetica senza resti particolarmente adatta ad architetture parallele e moduli definiti da numeri di Mersenne: numeri della forma $2^M - 1$. La nostra implementazione di queste tecniche, su un cluster di console PlayStation 3, ha stabilito un nuovo record: è stato trovato un fattore di 241-bit del numero $2^{1181} - 1$.

Una forma normale per le curve ellittiche introdotta da Edwards consente di lavorare, nella pratica, con l'aritmetica delle curve ellittiche più veloce in assoluto. Verrano presentate tecniche pratiche per ridurre l'occupazione di memoria e per migliorare le prestazioni di tale aritmetica. I nostri risultati consentono di eseguire ECM in maniera efficiente su piattaforme dalle risorse limitate come i processori grafici.

Termini di indicizzazione: crittanalisi, crittografia a chiave pubblica, fattorizzazione di numeri interi, problema del logaritmo discreto su una curva ellittica, aritmetica

Samenvatting

Tegenwoordig zijn de populairste asymmetrische cryptosystemen gebaseerd op het probleem van de ontbinding van een geheel samengesteld getal in priemfactoren of op het discrete logaritme probleem. De praktische mogelijkheden om deze wiskundige problemen op te lossen worden bestudeerd en technieken worden gepresenteerd om de berekeningen te versnellen op parallelle computerarchitecturen.

De snelste manier om het discrete logaritme probleem in groepen van elliptische krommen over een eindig lichaam op te lossen is het Pollard rho algoritme. Spiegelbeelden kunnen worden gebruikt om de berekening met een factor $\sqrt{2}$ te versnellen, tenzij de toevalsbewegingen erdoor in nutteloze cycli terecht komen. We tonen aan dat eerder gepubliceerde methoden om dit probleem op te lossen door terugkerende cycli niet werken en we laten zien hoe ook dit probleem kan worden opgelost. Verder introduceren we "slordige reductie" om modulair rekenen met getallen van een speciale vorm te versnellen. We laten zien dat dit in de praktijk werkt door een 112-bit elliptische kromme asymmetrische standaard te kraken. De berekening werd gedaan op een cluster bestaande uit PlayStation 3 spelcomputers en zette een nieuw wereldrecord.

De elliptische kromme methode (ECM) is de asymptotisch snelste methode om kleine priemfactoren te vinden. De grootte van de factoren die ermee gevonden kunnen worden geeft aan hoe de parameters van sommige cryptosystemen gekozen moeten worden. Voor toepassing van ECM op Mersenne getallen (getallen van de vorm 2^M-1) hebben we een snelle overdrachtsvrije rekenmethode ontwikkeld die zeer geschikt is voor parallelle computerarchitecturen. Op het spelcomputercluster hebben we er een nieuw ECM record mee gevestigd door een 241-bit priemfactor te vinden van $2^{1181} - 1$.

Een paar jaar geleden heeft Edwards de tot nu toe snelste manier bedacht om met elliptische krommen te rekenen. We laten zien hoe de voor toepassing op ECM vereiste hoeveelheid geheugen drastisch kan worden verminderd. Dit maakt het mogelijk ECM te versnellen op architecturen met beperkt geheugen zoals grafische kernen (GPUs).

Sleutelwoorden: cryptanalyse, asymmetrische cryptografie, ontbinden in factoren, rekenkunde, discrete logaritme probleem voor elliptische krommen

Contents

Chapter 1

Introduction

Obtaining the original meaning of encrypted data without using the corresponding secret material is part of the research area known as *cryptanalysis*. Cryptanalysis, often referred to as the practice of code breaking, together with cryptography, the science of hiding information, are the two branches of the larger research area known as cryptology. Within cryptology one can (roughly) distinguish three different research fields, each fulfilling a different practical need: cryptographic hash functions, symmetric-key and asymmetric-key cryptography. The latter is also known as *public-key* cryptography, here the methods used to hide the information use different keys, a public and a private, for hiding and revealing the message respectively. This thesis is concerned with both the theoretical and practical aspects of *public-key cryptanalysis*.

In the late 1970s, Rivest, Shamir and Adleman proposed an approach to realize public-key cryptography in practice which is known as the RSA algorithm [174]. The core idea described in their paper is still valid today and resisted many years of cryptanalysis [28]. The RSA algorithm is, without doubt, currently the most widely used public-key cryptosystem and has been standardized in the public-key cryptography standard [112]. The mathematical foundation of the RSA scheme is the *integer factorization problem*, this problem can be defined as follows [201, (Integer Factoring, p. 290)].

Definition 1.1 (The Integer Factorization Problem)**.** Integer factoring *is the following problem: given a positive integer* n*, find positive integers* v *and* w*, both greater than* 1*, such that* $n = v \cdot w$*.*

Another approach to realize public-key cryptography is based on the algebraic structure of elliptic curves over finite fields. Elliptic curve cryptography (ECC) [124,143] enjoys increasing popularity since its invention in the mid 1980s. The attractiveness of small key-sizes [131,135] has placed this public-key cryptosystem as the preferred alternative to RSA. This is emphasized by the current migration away from 80-bit to 112-bit security where, for instance, the United States' National Security Agency restricts the use of public key cryptography in "Suite B" [151] to ECC. Popular ECC based schemes are based on the ElGamal cryptosystem [75]

and the digital signature algorithm [199]. The mathematical problem used as the theoretical foundation in these systems is known as the *discrete logarithm problem* and can be defined as follows [201, (Discrete Logarithm Problem, p. 164)].

Definition 1.2 (The Discrete Logarithm Problem)**.** *Let g be a generator for a cyclic group $\mathbf{G}$. Given an element $y \in \mathbf{G}$, the* discrete logarithm problem *is to find an integer x such that $g^x = y$.*

Note that not all public-key schemes are based on these two problems; examples of other mathematical problems used are the hardness of decoding a general linear code (used in the McEliece cryptosystem [141]) and lattice based problems (used in the Goldreich-Goldwasser-Halevi encryption [90] and NTRU [106]) but the use of such schemes in practice is limited.

Although the integer factorization and discrete logarithm problems are not proven to be hard, many people believe that this is the case; e.g. there exists no polynomial time integer factorization method (or (sub)exponential but feasible in practice) on a classical computer (polynomial in the number of bits of the number to be factored). On a quantum computer, however, one *can* factor (and compute discrete logarithms) in polynomial time due to Shor's algorithm [183]. This thesis is only concerned with methods and algorithms running on classical (non-quantum) computers. To make the situation even worse, it is not even known if breaking RSA is equivalent to factoring; there are results pointing in different directions [1, 29].

This thesis studies how efficiently one can solve the mathematical problems stated in Definition 1.1 and Definition 1.2. Obtaining the secret information by other means than solving these problems is not considered. Examples of such other methods can be found in the research area related to *side channel attacks* [126, 127]: attacks which use information gained from the physical implementation of a certain scheme to break its security; e.g. the elapsed time or power consumption.

A common approach to study to what extent these mathematical problems can be solved in practice is combining the state-of-the-art algorithms and resources. It might be necessary to adopt the algorithms to a specific architecture or to build a machine specifically designed for such tasks. As an example, the world's first programmable, digital, electronic, computing device known as the Colossus [79] was designed for cryptanalytic purposes[1]. A current shift in architecture design is to move towards many-core processors [159]. From a practical point of view this thesis aims to present and optimize algorithms which are specifically suitable to run on such parallel architectures (just as in the early 1990s, e.g. [70, 71]). The prime candidates considered are the heterogeneous, multi-core, single-instruction, multiple data Cell broadband engine (Cell) architecture and the single-instruction, multiple thread graphics processing unit (GPU) architecture families. We think that the techniques described in this thesis, and the implementation of these algorithms on parallel architectures, can be used to better understand what practical parameters should be used to provide a sufficient level of confidence in the security used in modern public-key cryptosystems. These fast (parallel)

[1]The Colossus machine was used to break the codes produced by the Lorenz SZ40/42 cipher machine in the second world war.

arithmetic routines also find their application in cryptography by enhancing the performance of asymmetric cryptographic primitives.

From a theoretical point of view, we adopt and optimize arithmetic procedures to these architectures to lower the required run-time. We also study some problems when using the negation map optimization, an approach which results in a constant factor speedup when solving the elliptic curve discrete logarithm problem, and give solutions to circumvent them. In the factorization setting we study methods to reduce the runtime and space (memory) requirement when using Edwards curves to accelerate the elliptic curve factorization method.

1.1 Publications and Thesis Outline

During my time as a PhD student I had the opportunity to work together and learn from many talented people. Not all the publications resulting from these fruitful collaborations have made it into this thesis, still they deserve to be mentioned here.

A project performed together with Onur Özen when following the PhD course *security and cooperation in wireless networks* by Prof. J.-P. Hubaux resulted in the publication:

- [41] J. W. Bos, O. Özen, and J.-P. Hubaux. Analysis and optimization of cryptographically generated addresses. In P. Samarati, M. Yung, F. Martinelli, and C. A. Ardagna, editors, *Information Security Conference - ISC 2009*, volume 5735 of *Lecture Notes in Computer Science*, pages 17-32. Springer, Heidelberg, 2009.

Different projects regarding techniques to implement and optimize symmetric schemes resulted in publications. These papers do not fit the general topics discussed in this thesis since they are mainly concerned with symmetric cryptography.

- [32] J. W. Bos, N. Casati, and D. A. Osvik. Multi-stream hashing on the PlayStation 3. In *Applied Parallel Computing - PARA 2008*, volume 6126 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 2008.

- [157] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In S. Hong and T. Iwata, editors, *Fast Software Encryption - FSE 2010*, volume 6147 of *Lecture Notes in Computer Science*, pages 75-93. Springer, Heidelberg, 2010.

- [43] J. W. Bos and D. Stefan. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems - CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 279-293. Springer, Heidelberg, 2010.

- [42] J. W. Bos, O. Özen, and M. Stam. Efficient hashing using the AES instruction set. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011*, *Lecture Notes in Computer Science*. pages 507-522, Springer, Heidelberg, 2011.

A risk assessment concerning the higher security standards, is published as a technical paper:

- [34] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. On the security of 1024-bit RSA and 160-bit elliptic curve cryptography. *Cryptology ePrint Archive, Report 2009/389*, 2009. http://eprint.iacr.org/

Improved arithmetic techniques for the Cell architecture are described in:

- [33] J. W. Bos and M. E. Kaihara. Montgomery multiplication on the Cell. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics - PPAM 2009*, volume 6067 of *Lecture Notes in Computer Science*, pages 477-485. Springer, Heidelberg, 2010.

I was also part of the international team which factored a 768-bit RSA integer, this new integer factorization world record is described in:

- [120] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In T. Rabin, editor, *Crypto 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, Heidelberg, 2010.

- [121] T. Kleinjung, J. W. Bos, A. K. Lenstra, D. A. Osvik, K. Aoki, S. Contini, J. Franke, E. Thomé, P. Jermini, M. Thiémard, P. Leyland, P. L. Montgomery, A. Timofeev, and H. Stockinger. A heterogeneous computing environment to solve the 768-bit RSA challenge. *Cluster Computing*, pages 1-16, 2010.

The latter is a journal version which describes the heterogeneous computing details.

The chapters in this thesis are based on the following papers (in reversed chronological order):

- [37] J. W. Bos and T. Kleinjung. ECM at work, 2012. Work in progress.

- [31] J. W. Bos. Low-latency elliptic curve scalar multiplication, 2012. Submitted for publication.

- [39] J. W. Bos, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Efficient SIMD arithmetic modulo a Mersenne number. In *IEEE Symposium on Computer Arithmetic - ARITH-20*, pages 213-221, IEEE Computer Society, 2011.

- [35] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. In International Journal of Applied Cryptography, volume 2, number 3, pages 212–228, 2012.

- [38] J. W. Bos, T. Kleinjung, and A. K. Lenstra. On the use of the negation map in the Pollard rho method. In G. Hanrot, F. Morain, and E. Thomé, editors, *Algorithmic Number Theory - ANTS-IX*, volume 6197 of *Lecture Notes in Computer Science*, pages 67-83. Springer, Heidelberg, 2010.

- [30] J. W. Bos. High-performance modular multiplication on the Cell processor. In M. A. Hasan and T. Helleseth, editors, *Arithmetic of Finite Fields - WAIFI 2010*, volume 6087 of *Lecture Notes in Computer Science*, pages 7-24. Springer, Heidelberg, 2010.
- [40] J. W. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe. ECC2K-130 on Cell CPUs. In D. J. Bernstein and T. Lange, editors, *Africacrypt 2010*, volume 6055 of *Lecture Notes in Computer Science*, pages 225-242. Springer, Heidelberg, 2010.
- [36] J. W. Bos, M. E. Kaihara, and P. L. Montgomery. Pollard rho on the PlayStation 3. In *Special-purpose Hardware for Attacking Cryptographic Systems - SHARCS 2009*, pages 35-50, 2009.

This thesis is organized as follows. Chapter 2 recalls most of the preliminaries required for the subsequent chapters. Chapter 3 deals with fast arithmetic on parallel architectures, presenting both low-latency and high-throughput algorithms and compares the performance of modular arithmetic when using generic or special moduli. Chapter 4 comments on the use of the negation map optimization technique and shows that this optimization, theoretically resulting in a factor $\sqrt{2}$ speedup when solving the elliptic curve discrete logarithm problem, is in practice plagued by recurring cycles. Methods to reduce and avoid these events are presented. Chapter 5 presents the details behind two approaches to solve the elliptic curve discrete logarithm problem on the Cell broadband engine: the current 112-bit prime field record and an ongoing attempt which aims to solve a logarithm problem using a specific family of elliptic curves over binary extension fields: the so-called anomalous binary or Koblitz curves. Chapter 6 discusses the arithmetic designed for parallel architectures behind our implementation of the elliptic curve method (ECM) for integer factorization. Using these methods we have set the current ECM record by factoring Mersenne numbers. Chapter 7 presents an approach to lower the number of required elliptic curve additions and to lower the storage requirement in ECM when using Edwards curves.

Chapter 2

Preliminaries

In this chapter some of the techniques and methods are recalled which are used in the remainder of this thesis. Some chapters contain their own preliminary section if these techniques, ideas or theories are used in that chapter only. An overview of most of the work described here can be found in volume 2 of *The Art of Computer Programming* by Knuth [122] or, concerned with the more arithmetic aspects, the book by Brent and Zimmermann [50].

2.1 Radix Representation and Bit Lengths

Throughout this thesis we use different ways to represent integers. Let us define the notation.

- (k-bit integer). For $k \in \mathbf{Z}_{>0}$ a *k-bit integer* is an integer w such that $0 \leq w < 2^k$.
- (signed k-bit integer). A *signed k-bit integer* is an integer w such that $-2^{k-1} \leq w < 2^{k-1}$.
- (radix-r representation). For $r \in \mathbf{Z}_{>1}$ a *radix-r representation* of an integer z with $0 \leq z < r^s$ is a sequence of s *radix-r digits* $(w_j)_{j=0}^{s-1}$ such that $z = \sum_{j=0}^{s-1} w_j r^j$ and $w_j \in \mathbf{Z}_{\geq 0}$. Note that this representation is unique if $0 \leq w_j < r$ for $0 \leq j < s$.
- (signed k-bit radix-r representation). If $2^k \geq r$, a *signed k-bit radix-r representation* of z is a sequence $(w_j)_{j=0}^{s}$ of signed k-bit integers such that $z = \sum_{j=0}^{s} w_j r^j$. We denote *signed radix-2^k representation* for signed k-bit radix-2^k representation.

2.2 Parallel Architectures

Most of the algorithms presented in this thesis have been implemented. The target platforms are parallel architectures, the main focus is on the Cell broadband engine architecture and

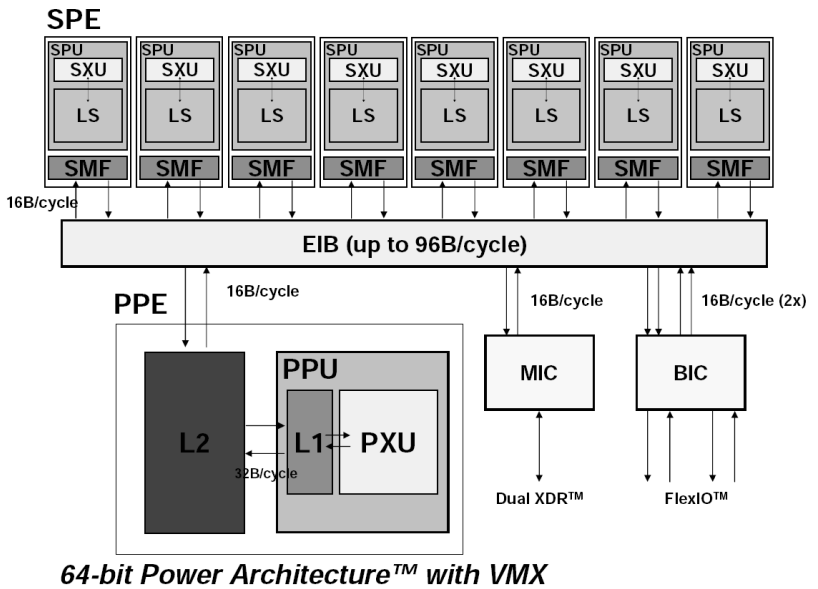

Figure 2.1: Overview of the Cell Broadband Engine Architecture. Figure taken from [94].

the graphics processing unit platforms. Previously published GPU implementations cover asymmetric cryptography, such as RSA [104, 150, 193], and ECC [4, 193], and symmetric cryptography [43, 102, 103, 140, 157, 205]. The GPU has also been considered to enhance the performance of cryptanalytic computations in the settings of finding hash collisions [25] and smoothness testing [17, 18]. The GPU has shown to be useful when accelerating software in routers [97] and offloading the cryptographic workload when using SSL [111]. Besides the Cell broadband engine implementations discussed in this thesis the Cell has been considered for fast arithmetic and cryptography [33,43,56,62,63,157] as well as for cryptanalysis [17,191,192].

Below we briefly recall some characteristics of these platforms.

2.2.1 The Cell Broadband Engine

The Cell processor [94, 107], jointly developed by Sony, Toshiba, and IBM, is a powerful heterogeneous multiprocessor. The Cell has a *Power Processing Element* (PPE), a dual-threaded Power architecture-based 64-bit processor with access to a 128-bit AltiVec/VMX single instruction, multiple data (SIMD) unit. Its main processing power, however, comes from eight *Synergistic Processing Elements* (SPEs) [194]. Each SPE consists of a Synergistic Processing Unit (SPU), 256 KB of private memory called Local Store (LS), and a Memory Flow Controller (MFC). To avoid the complexity of sending explicit direct memory access requests to the MFC, all code and data must fit within the LS. An overview of the Cell is given in Figure 2.1.

Each SPU runs independently from the others at 3.192GHz and is equipped with a large register file containing 128 registers of 128 bits each. Most SPU instructions work on 128-

bit operands denoted as *quadwords*. The instruction set is partitioned into two sets: one set consists of (mainly) 4- and 8-way SIMD arithmetic instructions on 32-bit and 16-bit operands respectively, while the other set consists of instructions operating on the whole quadword (including the load and store instructions) in a single instruction, single data (SISD) manner. The SPU is an asymmetric processor; each of these two sets of instructions is executed in a separate pipeline, denoted by the *even* and *odd* pipeline for the SIMD and SISD instructions, respectively. For instance, the {4, 8}-way SIMD left-rotate instruction is an even instruction, while the instruction left-rotating the full quadword is dispatched into the odd pipeline. When dependencies are avoided, a single pair consisting of one odd and one even instruction can be dispatched every clock cycle.

One of the first applications of the Cell processor was to serve as the heart of Sony's PS3 game console. Although the Cell contains 8 SPEs, in the PS3 one is disabled and a second is reserved by Sony. Thus, with the first generation PS3s the programmer has access to six SPEs. Access to the SPEs has been entirely disabled in the current version of the game console. For independent applications serving the supercomputing community, the Cell has been placed in blade servers, with newer variants containing the PowerXCell 8i, a derivative of the Cell that offers enhanced double-precision floating-point capabilities. The SPEs are particularly useful as (cryptographic) accelerators. For this purpose, PCIe[1] cards are available (either equipped with a complete Cell processor or a stripped-down version containing 4 SPEs) so that workstations can benefit from the computational power of the SPEs.

2.2.2 Integer and Bit Arithmetic on the SPU

We interpret each 128-bit SPU register v as a four-tuple of 32-bit values (v_1, v_2, v_3, v_4), where v_i is the i^{th} *word* of v which may be interpreted as signed or unsigned 32-bit integer. Below, a, b, c, d are 128-bit registers and all operations are for $i = 1, 2, 3, 4$ simultaneously.

The call $d = \texttt{spu\_add}(a, b)$ does 4-way SIMD 32-bit integer addition, calculating $d_i = (a_i + b_i) \bmod 2^{32}$. Other instructions generate the corresponding carries ($c = \texttt{spu\_genc}(a, b)$: $c_i = \lfloor (a_i + b_i)/2^{32} \rfloor$), include existing carries in additions ($d = \texttt{spu\_addx}(a, b, c)$: $d_i = (a_i + b_i + c_i) \bmod 2^{32}$), or generate the carries of the latter additions ($e = \texttt{spu\_gencx}(a, b, c)$: $e_i = \lfloor (a_i + b_i + c_i)/2^{32} \rfloor$). The corresponding integer subtraction instructions are `spu_sub`, `spu_genb`, `spu_subx`, and `spu_genbx`, where the '`b`' in '`genb`' indicates *borrow*: no borrow occurs if the borrow-bit is set to 1 (one), and a borrow occurs if the borrow-bit is set to 0 (zero). These are all even pipeline instructions that take two cycles.

The call $c = \texttt{spu\_mulo}(a, b)$ does 4-way SIMD $16 \times 16 \to 32$-bit unsigned integer multiplication, calculating $c_i = (a_i \bmod 2^{16}) \cdot (b_i \bmod 2^{16})$. There are two signed and one unsigned 4-way SIMD $(16 \times 16) + 32 \to 32$-bit multiply-and-add instructions. One of the signed ones calculates $c_i = (a_i \cdot b_i + d_i) \bmod 2^{32}$, where a_i and b_i are interpreted as signed 16-bit integers (i.e., their 16 most significant bits are ignored), and d_i and c_i are signed 32-bit integers.

The other two multiply-and-add instructions (the other signed one, and the unsigned one) work instead on the 16 most significant bits of a_i and b_i, ignoring the $2 \times 4 \times 16$

[1] Peripheral component interconnect express (PCIe) is a computer expansion card bus standard for attaching hardware devices in a computer.

least significant bits. The unsigned instruction is used for modular multiplication: the call $c = \texttt{spu\_mhhadd}(a, b, d)$ calculates $c_i = (\lfloor a_i/2^{16} \rfloor \cdot \lfloor b_i/2^{16} \rfloor + d_i) \bmod 2^{32}$, where d_i and c_i are unsigned 32-bit integers. All these multiplications are even pipeline instructions, one of them can be dispatched per cycle, taking seven cycles.

The call $c = \texttt{spu\_and}(a, b)$ calculates the 128-bit value $a \wedge b$, i.e., the bitwise-and of its inputs. The word-wise comparison call $c = \texttt{spu\_cmpeq}(a, b)$ results in $c_i = 2^{32} - 1$ (i.e., all one bits across c's i^{th} word) if a_i and b_i have the same value and $c_i = 0$ (i.e., all zero bits) otherwise. The $d = \texttt{spu\_sel}(a, b, p)$ instruction acts as a 2-way multiplexer; depending on the input pattern p the corresponding bit from either a or b is selected as the output bit in d. All three are even pipeline instructions with a two cycle latency.

The or-across instruction call $\texttt{spu\_orx}(a)$ returns the 32-bit value $a_1 \vee a_2 \vee a_3 \vee a_4$, i.e., the bitwise inclusive or across the words of a. Using $d = \texttt{spu\_shuffle}(a, b, c)$ any 16 entries of a 32-byte table (a and b) can be looked up simultaneously: the pattern in c shuffles 16 of the 32 bytes of a and b to the output d, in such a way that the j^{th} byte of c determines the j^{th} byte of d, as a copy of a byte of a or b or as one of the constants $\{\texttt{0x00, 0xFF, 0x80}\}$. It allows duplicate copies. Both are odd four cycle latency instructions.

The positioning of bits in the top-half-words as in `spu_mhhadd` requires byte-rearranging shifts and shuffles. These are odd pipeline instructions that can be dispatched almost for free if they are interleaved with the even pipeline arithmetic ones. The split call $(b, c) = \texttt{spu\_split}(a)$ re-arranges bytes: $b_i = \lfloor a_i/2^{16} \rfloor$ and $c_i = a_i \bmod 2^{16} \in \{0, 1, 2, \ldots, 2^{16} - 1\}$, i.e., b_i gets a_i's top-half shifted right over two bytes and c_i gets a_i's bottom-half. This can be implemented in a variety of ways using a combination of two SPU instructions: using two even pipeline instructions, or two odd ones, or one of each. The opposite effect is achieved by $a = \texttt{spu\_merge}(b, c)$: $a_i = 2^{16} b_i + c_i$, implemented using a single shuffle instruction. For $0 \leq k < 32$, the shift instruction call $b = \texttt{spu\_sl}(a, k)$ left-shifts a_i over k bits: $b_i = a_i 2^k \bmod 2^{32} \in \{0, 1, 2, \ldots, 2^{32} - 1\}$.

2.2.3 Compute Unified Device Architecture

Graphics Processing Units (GPUs) have mainly been game- and video-centric devices. Due to the increasing computational requirements of graphics-processing applications, GPUs have become very powerful parallel processors and this, moreover, incited research interest in computing outside the graphics-community. Until recently, programming GPUs was limited to graphics libraries such as OpenGL [180] and Direct3D [27], and for many applications, especially those based on integer-arithmetic, the performance improvements over CPUs was minimal, sometimes even degrading. The release of NVIDIA's G80 series and ATI's HD2000 series GPUs (which implemented the unified shader architecture), along with the companies' release of higher-level language support with Compute Unified Device Architecture (CUDA), Close to Metal (CTM) [158] and the more recent Open Computing Language (OpenCL) [93] facilitate the development of massively-parallel general purpose applications for GPUs [2, 155]. These general purpose GPUs have become a common target for numerically-intensive applications given their ease of programming (relative to previous generation GPUs), and ability to outperform CPUs in data-parallel applications, commonly by orders of magnitude.

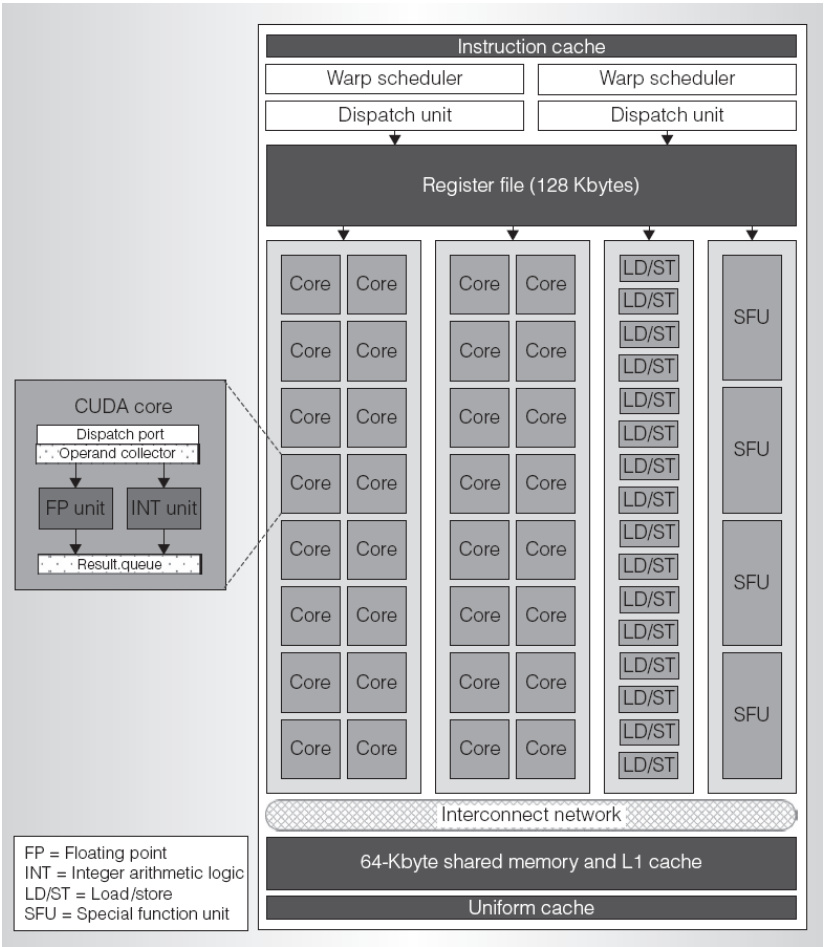

Figure 2.2: An overview of the Fermi streaming multiprocessor with its 32 CUDA processor cores. Figure taken from [152].

We focus on NVIDIA's GPU architecture with CUDA, more specifically the third generation GPU family known under the code name *Fermi* [154]. After the first generation G80 architecture, the first GPU to support the C-programming language, and the second generation GT200 architecture the Fermi architecture was released in 2010. One of the main features for our setting is the support of $32 \times 32 \to 32$-bit multiplication instructions, for both the least- and most-significant 32-bit of the multiplication result. The previous NVIDIA architecture families have native $24 \times 24 \to 32$-bit multiplication instructions.

We briefly recall some of the basic components of NVIDIA GPUs. More detailed information about the specification of CUDA as well as experiences using this parallel computer architecture can be found in [87,137,152,154,155]. Each GPU contains a number of streaming multiprocessors (SMs) and each SM consists of multiple scalar processor cores (SP); these number vary per graphics card. Typically, on the Fermi architecture, there are 32 SPs per

SM and around 16 SMs per GPU. C for CUDA is an extension to the C language that employs the massively parallel programming model called *single-instruction multiple-thread*. The programmer defines *kernel functions*, which are compiled for and executed on the SPs of each SM, in parallel: each light-weight thread executes the same code, operating on different data. A number of threads are grouped into a *thread block* which is scheduled on a single SM, the threads of which time-share the SPs. This hierarchy provides for threads within the same block to communicate using the on-chip shared memory and to synchronize their execution using barriers (a synchronization method which causes threads to wait until all threads reach a certain point).

On a lower level, threads inside each thread block are executed in groups of 32 called *warp*s. On the Fermi architecture each SM has two warp schedulers and two instruction dispatch units. This means that two instructions, from separate warps, can be scheduled and dispatched at the same time. By switching between the different warps, trying to fill the pipeline as much as possible, a high throughput rate can be sustained. When the code executed on the SP contains a conditional data-dependent branch all possibilities, taken by the threads inside this warp, are serially executed (threads which do not follow a certain branch are disabled). After executing these possibilities the threads within this warp continue with the same code execution. For optimal performance it is recommended to avoid multiple execution paths within a single warp.

The GPU has a large but relatively slow amount of global memory. Global memory is shared among all threads running on the GPU (on all SMs). Communication between threads inside a single thread block can be performed using the faster shared memory. Global memory accesses can be sped up significantly when ensuring the memory transactions are *coalesced*. If a warp requests data from global memory, the request is split into two separate memory requests, one for each half-warp (16 threads), each of which is issued independently. If the word size of the memory requested is 4, 8, or 16 bytes, the data requested by all threads lie in the same segment and are accessed in sequence (the k^{th} thread in the half-warp fetches the k^{th} word) then the global memory request is coalesced. In practice this means that a 64-byte memory transaction, a 128-byte memory transaction, or two 128-byte memory transactions are issued if the size of the words accessed by the threads is 4, 8, or 16, respectively. When this transfer is not coalesced, 16 separate 32-byte memory transactions are performed. More advanced rules might apply to decide if a global memory request is coalesced or not depending on the architecture used, see [155] for the specific details.

2.3 Multiplication

Integer multiplication, $n \times n \to 2n$ digits, is a well-studied research area. In this thesis we are mainly concerned with the multiplication of small- and medium-sized integers not larger than 1 500-bits. For the smaller (up to a few hundred bits) bit-sizes the fastest method in practice is the schoolbook, or textbook, multiplication which has run-time complexity $\mathrm{O}(n^2)$. See the left part of Algorithm 1 for a description of the radix-r schoolbook multiplication method.

Algorithm 1 The radix-r schoolbook (left) and interleaved Montgomery [145] (right) multiplication methods.

Input:
$$\begin{cases} A = \sum_{i=0}^{n-1} a_i r^i, B = \sum_{i=0}^{n-1} b_i r^i \\ \text{with } 0 \le a_i, b_i < r \end{cases}$$
Output:
$$\begin{cases} C = A \cdot B = \sum_{i=0}^{2n-1} c_i r^i \\ \text{with } 0 \le c_i < r \end{cases}$$
1. $C \leftarrow A \cdot b_0$
2. **for** $i = 1$ to $n-1$ **do**
3. $\quad C \leftarrow C + r^i (A \cdot b_i)$
4. **return** C

Input:
$$\begin{cases} A = \sum_{i=0}^{n-1} a_i r^i, B, M, \mu \text{ such that} \\ 0 \le a_i < r,\ 0 \le A, B < r^n,\ r^{n-1} \le M < r^n, \\ 2 \nmid M,\ \gcd(r, M) = 1,\ \mu = -M^{-1} \bmod r, \end{cases}$$
Output:
$$\begin{cases} C \equiv A \cdot B \cdot r^{-n} \bmod M \\ \text{such that } 0 \le C < r^n \end{cases}$$
1. $C \leftarrow 0$
2. **for** $i = 0$ to $n-1$ **do**
3. $\quad C \leftarrow C + a_i \cdot B$
4. $\quad q \leftarrow \mu \cdot C \bmod r$
5. $\quad C \leftarrow (C + q \cdot M)/r$
6. **if** $C \ge r^n$ **then**
7. $\quad C \leftarrow C - M$
8. **return** C

Another, asymptotically faster, multiplication method used in this thesis is Karatsuba multiplication [116] which has run-time complexity $\mathrm{O}(n^{\log_2(3)})$. This method is based on the divide-and-conquer paradigm and a recursive description is given in Algorithm 2. We typically use this method to multiply medium-sized (a few hundred bits and higher) integers.

2.3.1 Montgomery Modular Multiplication

The Montgomery modular multiplication method introduced in [145] consists of transforming each of the operands into a Montgomery representation and carry out the computation by replacing the conventional modular multiplications by Montgomery multiplications. This is suitable to speed up, for example, modular exponentiations which can be decomposed as a sequence of several modular multiplications. One of the advantages of this method is that the computational complexity is usually better compared to the classical method by a constant factor.

Given an n-word odd modulus M, such that $r^{n-1} \le M < r^n$, and an integer $X = \sum_{i=0}^{n-1} x_i \cdot 2^{w \cdot i}$. The Montgomery radix R is a constant such that $\gcd(R, M) = 1$ and $R > M$. For efficiency reasons, R is usually chosen as r^n where $r = 2^w$ is the radix of the system and w is the bit length of a word. The Montgomery residue of X is defined as $\widetilde{X} = X \cdot R \bmod M$. The Montgomery product of two integers is defined as $M(\widetilde{X}, \widetilde{Y}) = \widetilde{X} \cdot \widetilde{Y} \cdot R^{-1} \bmod M$. If $\widetilde{X} = X \cdot R \bmod M$ and $\widetilde{Y} = Y \cdot R \bmod M$ are Montgomery residues of X and Y, then $\widetilde{Z} = M(\widetilde{X}, \widetilde{Y}) = X \cdot Y \cdot R \bmod M$ is a Montgomery residue of $X \cdot Y \bmod M$. Algorithm 1 describes the radix-r interleaved Montgomery algorithm.

The conversion between the ordinary representation of an integer X to the Montgomery representation $\widetilde{X}$ can be performed using the Montgomery algorithm by computing $\widetilde{X} = M(X, R^2)$, provided that the constant $R^2 \bmod M$ is pre-computed. The conversion back

Algorithm 2 Karatsuba multiplication

Input: $\begin{cases} n \in \mathbf{Z},\ A = \sum_{i=0}^{n-1} a_i r^i,\ B = \sum_{i=0}^{n-1} b_i r^i,\ \text{with } 0 \le a_i, b_i < r \\ T:\ \text{some threshold for switching to schoolbook multiplication} \\ \text{Let } \widetilde{r} = r^{\lceil n/2 \rceil} \end{cases}$

Output: $C = A \cdot B = \sum_{i=0}^{2n-1} c_i r^i$ with $0 \le c_i < r$

1. **if** $n < T$ **then**
2. **return** $C \leftarrow \text{schoolbook}(A, B)$
3. $A \leftarrow A_0 + A_1\widetilde{r}, \quad 0 \le A_0, A_1 < \widetilde{r}$
4. $B \leftarrow B_0 + B_1\widetilde{r}, \quad 0 \le B_0, B_1 < \widetilde{r}$
5. $T_0 \leftarrow \text{Karatsuba}(A_0,\ B_0)$
6. $T_1 \leftarrow \text{Karatsuba}(A_1,\ B_1)$
7. $T_2 \leftarrow \text{Karatsuba}(A_0 + A_1,\ B_0 + B_1) - T_0 - T_1$
8. **return** $C \leftarrow (T_0 + T_2 \cdot \widetilde{r} + T_1 \cdot \widetilde{r}^2)$

from the Montgomery representation to the ordinary representation can be done by applying the Montgomery algorithm to the result and the number 1, i.e. $Z = M(\widetilde{Z}, 1)$.

In order to avoid the last conditional subtraction (lines 6 and 7 of the Montgomery algorithm shown in Algorithm 1), R may be chosen such that $4M < R$ and inputs and output are represented as elements of $\mathbf{Z}/2M\mathbf{Z}$ instead of $\mathbf{Z}/M\mathbf{Z}$, that is, operations are carried out in a redundant representation. It is easily shown that throughout the series of modular multiplications, outputs from multiplications can be reused as inputs and these values remain bounded [203]. This technique does not only speed-up modular multiplications but also lowers the success of timing attacks [126] as operations are data independent.

2.4 Elliptic Curves

Let $\mathbf{F}_p$ denote a finite field of prime cardinality $p > 3$. Any $a, b \in \mathbf{F}_p$ with $4a^3 + 27b^2 \neq 0$ define an elliptic curve $E_{a,b}$ over $\mathbf{F}_p$ (see for more details e.g. [185]). The *group of points* $E_{a,b}(\mathbf{F}_p)$ *of* $E_{a,b}$ *over* $\mathbf{F}_p$ is defined as the *zero point* $\mathfrak{o}$ along with the set of pairs $(x, y) \in \mathbf{F}_p \times \mathbf{F}_p$ that satisfy the *short Weierstrass equation*

$$y^2 = x^3 + ax + b \tag{2.1}$$

with the following additively written group law. For $\mathfrak{c} \in E_{a,b}(\mathbf{F}_p)$ define $\mathfrak{c} + \mathfrak{o} = \mathfrak{o} + \mathfrak{c} = \mathfrak{c}$. For non-zero $\mathfrak{c} = (x_1, y_1), \mathfrak{d} = (x_2, y_2) \in E_{a,b}(\mathbf{F}_p)$ define $\mathfrak{c} + \mathfrak{d} = \mathfrak{o}$ if $x_1 = x_2$ and $y_1 = -y_2$. Otherwise $\mathfrak{c} + \mathfrak{d} = (x, y)$ with $x = \lambda^2 - x_1 - x_2$ and $y = \lambda(x_1 - x) - y_1$, where

$$\lambda = \begin{cases} \dfrac{3x_1^2 + a}{2y_1} & \text{if } x_1 = x_2 \text{ (and thus } \mathfrak{c} = \mathfrak{d}) \\ \dfrac{y_1 - y_2}{x_1 - x_2} & \text{otherwise.} \end{cases}$$

Thus, using these *affine Weierstrass coordinates* to represent group elements, *doubling* (i.e., $\mathfrak{c} = \mathfrak{d}$) is different from *regular addition* (i.e., $\mathfrak{c} \neq \mathfrak{d}$).

In practice, different defining equations and coordinate systems can be used; cf. [22, 59] for an overview of the cost of point addition (the group operation) and scalar multiplication (repeated point addition). The Montgomery form $E_{a,b}$ [146], with $a^2 \neq 4$ and $b \neq 0$, is

$$by^2 = x^3 + ax^2 + x \quad \text{and} \quad by^2z = x^3 + ax^2z + xz^2 \tag{2.2}$$

in the affine and the homogeneous form.

Currently, the fastest known elliptic curves, in terms of the cost expressed in multiplications and squarings to compute the group operation, are the family of curves originating from a normal form for elliptic curves introduced by Edwards in 2007 [74]. These Edwards curves have been generalized by Bernstein and Lange [20, 21] and Bernstein et al. [13] showing their practical use in cryptology. A twisted Edwards curve is defined as (cf. [13])

$$ax^2 + y^2 = 1 + dx^2y^2 \quad \text{and} \quad (ax^2 + y^2)z^2 = z^4 + dx^2y^2 \tag{2.3}$$

in the affine and the homogeneous form respectively with $0 \neq a \neq d \neq 0$. An Edwards curve is a twisted Edwards curve with $a = 1$ and $d \in \mathbf{F}_p \setminus \{0, 1\}$. A triplet $(x : y : z)$ on the homogeneous twisted Edwards curve / Montgomery curve represents, when $z \neq 0$, the affine point $(x/z, y/z)$.

Currently the fastest known approach to perform elliptic curve point addition and duplication is due to Hisil et al. [105]. They propose to use an auxiliary coordinate to enhance the performance of the addition. A point on equation (2.3) is represented as $(x : y : t : z)$, where $t = xy/z$, and denoted as extended twisted Edwards coordinates [105].

Let (X_1, Y_1, T_1, Z_1) and (X_2, Y_2, T_2, Z_2) be distinct points, with $Z_1, Z_2 \neq 0$, represented in extended twisted Edwards coordinates. The addition $(X_1, Y_1, T_1, Z_1) + (X_2, Y_2, T_2, Z_2) = (X_3, Y_3, T_3, Z_3)$ can be computed as

$$\begin{aligned}
X_3 &= (X_1Y2 - Y_1X_2)(T_1Z_2 + Z_1T_2),\\
Y_3 &= (Y_1Y_2 + aX_1X_2)(T_1Z_2 - Z_1T_2),\\
T_3 &= (T_1Z_2 + Z_1T_2)(T_1Z_2 - Z_1T_2),\\
Z_3 &= (Y_1Y_2 + aX_1X_2)(X_1Y_2 - Y_1X_2).
\end{aligned}$$

When $a = -1$ the cost of an elliptic curve addition is eight multiplications (ignoring the cost of additions and subtractions). Note that an additional multiplication can be saved when either $Z_1 = 1$ or $Z_2 = 1$. In the setting of regular (non-extended) twisted Edwards coordinates the point addition costs ten multiplications, a single squaring and two multiplications by curve constants.

Computing the double $2(X_1, Y_1, T_1, Z_1) = (X_3, Y_3, T_3, Z_3)$, when $Z_1 \neq 0$, can be performed as (cf. [105])

$$\begin{aligned}
X_3 &= 2X_1Y_1(2Z_1^2 - Y_1^2 - aX_1^2),\\
Y_3 &= (Y_1^2 + aX_1^2)(Y_1^2 - aX_1^2),\\
T_3 &= 2X_1Y_1(Y_1^2 - aX_1^2),\\
Z_3 &= (Y_1^2 + aX_1^2)(2Z_1^2 - Y_1^2 - aX_1^2)
\end{aligned}$$

which is very similar to the doubling formula presented in [13] for twisted Edwards coordinates. When $a = -1$, the computation of a elliptic curve point doubling is four multiplications and four squarings. When using regular twisted Edwards coordinates this cost is reduced by a single multiplication. In [105] a mixing technique is described, which omits the calculation of the T-coordinate if possible when computing the elliptic curve scalar multiplication. Switching between extended and regular twisted Edwards coordinates obtains the best of both worlds: on average (see Section 7.2.4 for the details) it suffices to perform eight multiplications per elliptic curve addition and three multiplications and four squarings per elliptic curve doubling.

2.4.1 The Elliptic Curve Method

Introduced by Hendrik Lenstra Jr. in 1985 [136], the elliptic curve method (ECM) for integer factorization is analogous to the Pollard $p-1$ integer factorization method [164] and attempts to factor a composite integer $n = pq$ $(1 < p < q < n)$. The general idea behind ECM is as follows (we follow the description from [136]). First, pick a random point P and construct an elliptic curve E over $\mathbf{Z}/n\mathbf{Z}$ (cf. [132, Section 2.B]).

Next, compute the elliptic curve scalar multiplication $Q = kP \in E(\mathbf{Z}/n\mathbf{Z})$. The positive integer k is selected such that it is divisible by many small prime powers: e.g. $k = \mathrm{lcm}(1, 2, \ldots, B_1)$ for some bound $B_1 \in \mathbf{Z}$. If the order $\#E(\mathbf{F}_p)$ is B_1-powersmooth (an integer is defined to be B-powersmooth if none of its prime factors is greater than B) then $\#E(\mathbf{F}_p) \mid k$. In other words, $Q = kP$ and the neutral element of the curve become the same modulo p. In this event, a failure occurred in the group operation defined for $E(\mathbf{Z}/n\mathbf{Z})$ and the factor $p \mid \gcd(n, Q_z)$ where Q_z is the z-coordinate of the point Q when using projective coordinates. If $\gcd(n, Q_z)$ is not divisible by q then we have factored n.

Hasse proved (see e.g. [185, Theorem 1.1]) that the order $\#E(\mathbf{F}_p)$ is in the interval $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$. The advantage of ECM is that one can randomize by trying different curves, thus obtaining different orders. In Pollard's $p - 1$ one has only one choice – $\mathbf{Z}_p^*$ with order $p - 1$, and randomization of the order is not possible. It has been shown in [136] that the (heuristic) run-time of ECM depends mainly on p, the smallest non-trivial prime divisor of n. The expected run-time of ECM is based on a heuristic assumption, namely how the order of the elliptic curve and the integers in the range $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$ behave, and can be expressed using the L-function [132] (or L-notation) which is defined as

$$L_x[t, \gamma] = e^{\gamma (\ln x)^t (\ln \ln x)^{1-t}}, \tag{2.4}$$

where $t, \gamma \in \mathbf{R}$ and $0 \leq t \leq 1$. The run-time of ECM can be expressed as

$$O(L_p\left[\frac{1}{2}, (\sqrt{2} + o(1))\right] M(\log n)),$$

where $M(\log n)$ represents the complexity of multiplication modulo n and the $o(1)$ is for $p \to \infty$.

The approach described here is often referred to as "stage 1". There is a second stage continuation for ECM which takes as input a bound $B_2 \in \mathbf{Z}$ and succeeds (in factoring n) if

$Q = kP$ has prime order ℓ (for $B_1 < \ell < B_2$) in $E(\mathbf{F}_p)$. This means that $\#E(\mathbf{F}_p)$ is B_1-smooth except for one prime factor which is below B_2. There are several techniques [47,146,147] how to perform this second stage efficiently.

Note that the ECM is not the asymptotically fastest integer factorization method. The general number field sieve (GNFS) [133] is the fastest publicly known method to factor integers. The GNFS is a generalization of previous work performed by Coppersmith, Odlyzko and Schroeppel [61] and Pollard [162]. The exact details of the GNFS are not relevant for this thesis. The general idea is to find integer solutions x, y of the *congruence of squares* $x^2 \equiv y^2 \bmod n$ (where n is not a prime power). For a random such pair the probability is at least $\frac{1}{2}$ that n can be factored as $\gcd(x-y, n) \cdot \gcd(x+y, n)$. A whole family of factorization algorithms is based on this approach [72, 149, 168, 169]. The overall expected (heuristic) runtime of the GNFS method to factor a composite integer n is

$$L_n\left[\frac{1}{3}, \left(\left(\frac{64}{9}\right)^{\frac{1}{3}} + o(1)\right)\right],$$

the $o(1)$ is for $n \to \infty$. Note that Coppersmith proposed a faster version of NFS which uses a single linear polynomial and multiple non-linear polynomials (versus a single linear and a single non-linear polynomial in the original NFS) [60] to factor multiple numbers. This method requires some precomputation, including this time the constant $c = \left(\frac{64}{9}\right)^{\frac{1}{3}} \approx 1.923$ in GNFS is reduced to $c = 2\left(\frac{46+13\sqrt{13}}{108}\right)^{\frac{1}{3}} \approx 1.902$ or, when the precomputation is amortized over many factorizations to $c = 2\left(\frac{5+2\sqrt{6}}{18}\right)^{\frac{1}{3}} \approx 1.639$.

At one end of the integer factorization spectrum ECM is used to factor integers out of range for NFS consisting of thousands of bits. The current record ECM factor of 73-decimal digits (241-bit) has been found by an implementation especially targeted at numbers of a special form using optimized arithmetic (see Chapter 6). This factor has been found using stage 1 parameter $B_1 = 3{\cdot}10^9$ and stage 2 parameter $B_2 = 10^{14}$ and computing approximately 30 000 stage 1 curves and 8 800 stage 2 curves. The practical (cryptographic) impact of these ECM record factorizations is limited to two variants of the RSA cryptosystem, namely *RSA multiprime* [174] and *unbalanced RSA* [181]. The former gains a speedup by a factor of r^2 or $\frac{r^2}{4}$ for the private operation in vanilla RSA or CRT-RSA, respectively, by selecting RSA moduli (of appropriate size to be out of reach of NFS) consisting of the product of $r > 2$ primes of about the same size. In unbalanced RSA, the RSA modulus has two factors as usual, but one is chosen much smaller than the other. In these variants, r and the smallest factor must be chosen in such a way that ECM has a sufficiently low probability to find the resulting relatively small prime factor(s).

At the other end of the factorization spectrum ECM is used to rapidly factor many small (up to one or two hundred bits) integers inside NFS. The relation collection phase, one of the main phases of NFS, first generates a lot of composites which are divisible by small primes using sieving techniques and subsequently tries to factor these remaining composite integers. The process of trying to factor these composites is denoted as the cofactorization

Algorithm 3 The double-and-add algorithm.

Input: $\begin{cases} G \in E_{a,b}(\mathbf{F}_p) \\ s \in \mathbf{Z}_{>0},\ 2^{k-1} \leq s < 2^k, s = \sum_{i=0}^{k-1} s_i 2^i,\ \text{with } 0 \leq s_i < 2 \end{cases}$

Output: $P = sG \in E_{a,b}(\mathbf{F}_p)$

1. $P \leftarrow G$
2. **for** $i = k-2$ down to 0 **do**
3. $\quad P \leftarrow 2P$
4. $\quad$ **if** $s_i = 1$ **then**
5. $\quad\quad P \leftarrow P + G$

phase. To illustrate, the total time spent in the cofactorization procedure was roughly one third of the sieving time when factoring a 768-bit RSA modulus in [120] (currently the largest factorization of an integer without special form). Note that this one third includes the time of pseudo primality tests and different factorization methods: quadratic sieve [169], Pollard $p-1$ [164] and ECM. Before embarking upon an ECM factorization attempt a Pollard $p-1$ test is always performed first. The total time spent in ECM is hard to estimate precisely but is somewhere between 5 and 20 percent of the total sieving time. In this cofactorization phase only composites up to 140 bits were considered and ECM was used only for composites up to 109-bits. The parameters for stage 1 (stage 2) in ECM varied depending on the composite size and ranged from 150 (9 000) to 500 (36 000) where often only a single curve was tried with a maximum of around eight curves.

This area, using ECM for cofactorization, has seen a flurry of recent activity: see [68,84,95, 138, 160, 186, 208] for implementations of ECM targeted at small integers on reconfigurable hardware such as field-programmable gate arrays and [17, 18] for GPUs. In [17] the Cell architecture is covered as well. Kruppa compares a software implementation to hardware based solutions [128]. Methods to optimize the cofactorization phase are given by Kleinjung in [119].

2.4.2 Elliptic Curve Scalar Multiplication

The most common approach when computing the elliptic curve scalar multiplication, where we assume we want to compute sP with $P \in E_{a,b}(\mathbf{F}_p)$ and $1 < s \in \mathbf{Z}$, is using *addition chains* [177]. A finite sequence of positive integers $a_0 = 1, a_1, \ldots, a_r = s$ is called an addition chain of length r if every element a_i can be written as a sum $a_j + a_k$ of preceding elements. Let us briefly recall some of the popular techniques based on addition chains when computing the elliptic curve scalar multiplication.

One of the simplest methods to compute the elliptic curve scalar multiplication is the double-and-add algorithm (see Algorithm 3). This approach is also known as the square-and-multiply (referring to multiplicative notation). There is not much one can do to lower the required number of $k = \lceil \log_2(s) \rceil - 1$ duplications. The number of additions, on the other

hand, can be reduced using several techniques. Consider the example $s = 9\,997$, in binary this number is $9\,997_{10} = 10011100001101_2$. The following addition chain based on this binary representation

$$D^3 \to A \to D \to A \to D \to A \to D^5 \to A \to D \to A \to D^2 \to A$$
$$(((((2^3 + 2^0) \cdot 2^1 + 2^0) \cdot 2^1 + 2^0) \cdot 2^5 + 2^0) \cdot 2^1 + 2^0) \cdot 2^2 + 2^0 = 9\,997$$

can be used. This is exactly what Algorithm 3 does. This approach requires $k = \lceil \log_2(9\,997) \rceil - 1 = 13$ duplications and six additions. One can also use a w-bit window size [45], precomputing cP, with $1 \leq c < 2^w$. This requires a precomputation cost plus the cost for the addition chain. When using a $w = 2$-bit window size the precomputation is a single doubling to compute $2P$ and a single addition for $2P + P = 3P$. Next one can proceed as follows

$$(((((2 \cdot 2^2 + 1) \cdot 2^2 + 3) \cdot 2^2 + 0) \cdot 2^2 + 0) \cdot 2^2 + 3) \cdot 2^2 + 1 = 9\,997,$$

the totals cost becomes 13 duplications and seven additions. This cost is higher compared to the previous approach, which used a $w = 1$-window, but this can be remedied using sliding windows [198].

The idea behind sliding windows is to perform as many duplications as possible after an addition. This ensures that the additions are always performed using odd numbers which reduces the required number of precomputed points by a factor of two. Heuristically, one can expect, when using sliding windows, $\sum_{i=1}^{k} 2^{-i} = 1 - 2^{-k}$ zero bits (duplications) after an addition. In our example, when using a $w = 2$-bit window, the value $3P$ can precomputed with one addition and one duplication. Next, the value of $9\,997$ can be computed as

$$(((2^4 + 3) \cdot 2 + 1) \cdot 2^6 + 3) \cdot 2^2 + 1 = 9\,997.$$

The total cost becomes 13 duplications and five additions.

One could use signed windows [148], i.e. addition/subtraction chains, if point subtraction has roughly the same cost as point addition (as is the case in the setting of elliptic curves). Applied to the example we can write

$$((((2^2 + 1) \cdot 2^3 - 1) \cdot 2^5 + 1) \cdot 2 + 1) \cdot 2^2 + 1 = 9\,997$$

when using a $w = 1$-bit window size which costs 13 duplications and five additions/subtractions. When using a $w = 2$-bit window size this sequence becomes

$$(((2^2 + 1) \cdot 2^3 - 1) \cdot 2^4 + 1) \cdot 2^4 - 3 = 9\,997$$

at identical total cost. A more advanced method which requires slightly more precomputation but lowers the runtime is known as the fractional windowing method [144].

A survey related to addition chains is given in [91]. An overview of the costs, expressed in arithmetic operations in the finite field, of the elliptic curve scalar multiplication can be found in [22, 59]. Note that computing good (or optimal) addition chains, possibly within some constraints (give a "good" answer quickly or do not use too much memory), is a hard problem.

Algorithm 4 Montgomery ladder

Input: $\begin{cases} G \in E_{a,b}(\mathbf{F}_p) \\ n = \sum_{i=0}^{k-1} n_i 2^i, n \in \mathbf{Z}_{>0},\ 2^{k-1} \leq n < 2^k \end{cases}$

Output: $P = nG \in E_{a,b}(\mathbf{F}_p)$

1. $P \leftarrow G, Q \leftarrow G$
2. **for** $i = k-2$ down to 0 **do**
3. **if** $n_i = 1$ **then**
4. $(P, Q) \leftarrow (P + Q, 2Q)$
5. **else**
6. $(P, Q) \leftarrow (2P, P + Q)$

The Montgomery Ladder

A different approach to calculate the elliptic curve scalar multiplication is the Montgomery ladder. This technique was introduced by Montgomery in [146] in the setting of ECM. We give here the higher level description from [113]. Let $L_0 = s = \sum_{i=0}^{t-1} k_i 2^i$, define $L_j = \sum_{i=j}^{t-1} k_i 2^{i-j}$ and $H_j = L_j + 1$. Then,

$$L_j = 2L_{j+1} + k_j = L_{j+1} + H_{j+1} + k_j - 1 = 2H_{j+1} + k_j - 2.$$

One can update these two values using

$$(L_j, H_j) = \begin{cases} (2L_{j+1}, L_{j+1} + H_{j+1}) & \text{if } k_j = 0, \\ (L_{j+1} + H_{j+1}, 2H_{j+1}) & \text{if } k_j = 1. \end{cases}$$

A high-level overview of this approach is given in Algorithm 4. This approach is slower compared to, for instance, the double-and-add technique (Algorithm 3) since a duplication and addition are always performed per bit. This disadvantage is actually used as a feature in environments which are exposed to side-channel attacks and where the algorithms needs to process the exact same steps independent of the input parameters. It is not difficult to alter Algorithm 4 such that it becomes branch-free (using the bit n_i to select which point to double). In ECM the elliptic curve scalar multiplication is calculated using the Montgomery form (see equation (2.2)) which avoids computing on the y-coordinate. This is achieved as follows: given the x- and z-coordinate of the points P, Q and $P - Q$ one can compute the x- and z-coordinates of $P + Q$ (and similarly $2P$ or $2Q$). Avoiding computations on one of the coordinates results in a speedup in practice (see [146] for all the details).

2.5 The Pollard Rho Method

The Pollard rho algorithm was proposed in 1975 as an integer factorization method to find relative small factors of a given composite input integer [165]. Three years later, Pollard

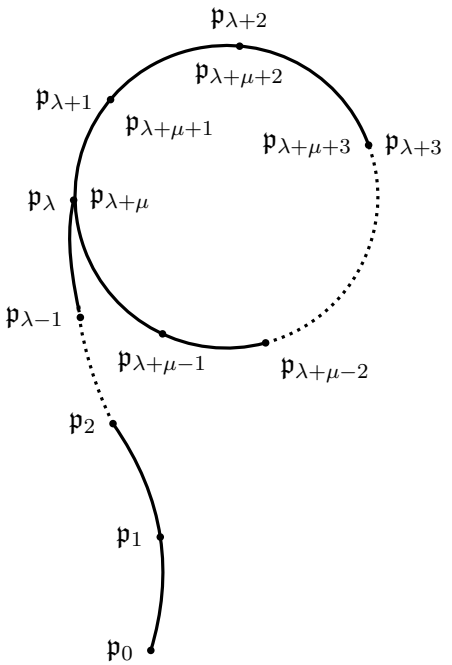

Figure 2.3: Representation of the ρ shape of the single-instance Pollard rho method. The points $\mathfrak{p}_i, \mathfrak{p}'_j$ represent points from two different walks.

adopted this method to solve the *discrete logarithm problem* (DLP) in generic groups [166]. Let $\mathbf{G}$ be a cyclic group of prime order n and let $\mathfrak{g} \in \mathbf{G}$ be a generator. The DLP is, given $\mathfrak{g}$ and $\mathfrak{h} \in \langle \mathfrak{g} \rangle$, to find $y = \log_{\mathfrak{g}} \mathfrak{h}$ (see Definition 1.2). We restrict ourselves in this description to the case where n is prime: a typical setting in cryptography. If n is composite one can reduce the computation of the discrete logarithm in an order n group to its prime order subgroups [161].

For arbitrary multipliers $u, v \in \mathbf{Z}$, $u\mathfrak{g} + v\mathfrak{h} \in \langle \mathfrak{g} \rangle$. A *collision* corresponds to random integer multipliers $u, v, \bar{u}, \bar{v}$ such that $u\mathfrak{g} + v\mathfrak{h} = \bar{u}\mathfrak{g} + \bar{v}\mathfrak{h}$. Unless $\bar{v} \equiv v \bmod n$, the value $m = \frac{u-\bar{u}}{\bar{v}-v} \bmod n$ solves the discrete logarithm problem after a collision has been found. Given an iteration function $f : \langle \mathfrak{g} \rangle \to \langle \mathfrak{g} \rangle$, the Pollard rho method calculates a sequence of points $\mathfrak{p}_{i+1} = f(\mathfrak{p}_i)$, $i \geq 0$ in order to find a collision. This sequence of points represents a walk through the set of points $\langle \mathfrak{g} \rangle$. Given $\mathfrak{p}_i = u_i\mathfrak{g} + v_i\mathfrak{h} \in \langle \mathfrak{g} \rangle$ and $u_i, v_i \in [0, n-1]$, f updates u_{i+1} and v_{i+1} and computes $\mathfrak{p}_{i+1}$ as $\mathfrak{p}_{i+1} = u_{i+1}\mathfrak{g} + v_{i+1}\mathfrak{h}$. The sequence is started from a random and known point $\mathfrak{p}_0 \in \langle \mathfrak{g} \rangle$ by selecting random values for u_0 and v_0. This sequence of points eventually collides (as operations are performed over a finite cyclic group). Let us denote λ and $\mu \geq 1$ as the smallest integers such that $\mathfrak{p}_\lambda = \mathfrak{p}_{\lambda+\mu}$ holds. The value λ is called the tail and μ the cycle length, graphically the walk through the set of points forms a ρ shape: see Figure 2.3. Assuming the iteration function is a random mapping of size n, i.e. f is equally probable among all functions $F : \langle \mathfrak{g} \rangle \to \langle \mathfrak{g} \rangle$, it can be shown [78, 100] that the asymptotic expected values of λ and μ are $\lambda = \mu = \sqrt{\frac{\pi n}{8}}$ when $n \to \infty$. Another way

of arriving at this average value is by regarding this walk as picking random objects (group elements) with replacement. Due to a result known as the birthday paradox this leads to the expected number of steps (or iterations) of $\sqrt{\frac{\pi n}{2}}$ [122, Exercise 3.1.12].

Finding a duplicate can be done by *Floyd's cycle finding method* [122, Exercise 3.1.6] requiring only a constant number of group elements: compute $(\mathfrak{p}_k, \mathfrak{p}_{2k})$ for $k = 1, 2, \ldots$ (where $\mathfrak{p}_k$ denotes the k^{th} point of the walk) until a collision occurs, i.e., $\mathfrak{p}_k = \mathfrak{p}_{2k}$. It can be seen (cf. [46]) that

$$k = \begin{cases} \mu & \text{if } \mu \equiv 0 \bmod \lambda \text{ and } \mu > 0 \\ \mu + \lambda - (\mu \bmod \lambda) & \text{otherwise.} \end{cases}$$

Since three calls to the iteration function f (one to compute $\mathfrak{p}_k$ and two for $\mathfrak{p}_{2k}$) are required to compute the next group elements the total number of calls to f is upper bounded by $3(\mu + \lambda)$.

An optimization of Floyd's cycle finding method is proposed by Brent [46]. A paper by Sedgewick, Szymanski and Yao [179] provide an algorithm which in the worst-case setting is asymptotically optimal. A stack based approach is introduced by Nivasch [153]. Details about optimizations for Pollard's rho algorithm, when implementing this method in practice and running multiple instances in parallel, are outlined in Chapter 4.

An alternative method to solve the DLP is *Shanks' baby step giant step* method [182] [123, Exercise 5.25] which builds a hash table containing $i\lceil\sqrt{n}\rceil\mathfrak{g}$ for $i = 0, 1, \ldots, \lceil\sqrt{n}\rceil$ and searches it for $\mathfrak{h} + j\mathfrak{g}$ for $j = 0, 1, 2, \ldots$, until a match is found. This works in time and memory on the order of $\sqrt{n}$. Pollard's rho method achieves expected runtime $O(\sqrt{n})$ and $O(\log n)$ memory or, if run in parallel, much less memory than Shanks' method: $O((\log n)^2)$ memory suffices [85, Exercise 16.23] when roughly $\sqrt{n}\log n$ out of n group elements are distinguished (see Chapter 4).

Chapter 3

High-Performance Arithmetic on Parallel Architectures

This chapter presents performance results for one of the key operations in ECC: modular multiplication. The performance results are obtained when running on two parallel architectures: the heterogeneous, multi-core, single instruction, multiple data (SIMD) Cell broadband engine (Cell) architecture and a number of different graphics processing unit (GPU) architecture families.

Our performance results set new speed records, in terms of throughput, for generic moduli, using interleaved Montgomery multiplication [145], and *special* modular multiplication for moduli ranging from 192 to 521 bits on the Cell. This range covers the current standardized parameters for ECC cryptosystems as specified by National Institute of Standards (NIST) [199]. Besides these special NIST primes, the prime of special form used in *curve25519* as proposed by Bernstein [12] is considered as well. These special primes are used to enhance the performance of ECC-based schemes in practice by exploiting the special form of the primes to construct a fast reduction step. Typically, the multiplication and special reduction are performed sequentially. For the separated multiplication step we consider schoolbook and Karatsuba multiplication [116] techniques. We use the straightforward methods to implement the fast reduction for the NIST recommended primes (see [188]). For the special prime in *curve25519* we use a different approach in order to compare with the proposed fast reduction from [12].

The performance results on the Cell are obtained by using the features of SIMD architectures. The implementations are specifically optimized for the Cell and take both the advantages (e.g., the rich instruction set and large register file) and disadvantages (e.g., the "small" $16 \times 16 \rightarrow 32$-bit multiplier) of this architecture into account. Furthermore, *multiple* streams of computations are interleaved to increase throughput. Multi-stream modular multiplication computations are useful in both a cryptanalytic and cryptographic setting. For instance, one could use multi-stream modular multiplication routines, either the generic or special variant, to speedup batch decryption for ECC-based schemes. Additionally, this work

shows the practical benefit of using the special over generic prime moduli on the Cell.

For the GPU we study a different setting. In order to asses the possibility to use the GPU as a cryptographic accelerator we present algorithms to compute the elliptic curve scalar multiplication (ECSM) (see Section 2.4.2), the core building block in ECC, for parallel computer architectures. An orthogonal perspective, compared to the Cell, is used and we aim to decrease the latency while trying to keep the throughput loss under control. The different design goals of the arithmetic between the Cell and the GPU architecture is motivated by the fact that the GPU has orders of magnitude more cores to its disposal compared to the Cell. In order to have a acceptable response time (e.g. the latency) one can compute the ESCM with multiple cores. Previous reports implementing ECC schemes using ECSM on GPUs [4, 17, 18, 193] use multiple cores to calculate the arithmetic in the finite field. Our approach differs: the modular arithmetic in the finite field is computed with a single thread (on a single core) to aim for high-throughput while the latency reduction is achieved by doing the elliptic curve arithmetic in parallel.

The presented algorithms are based on methods originating in cryptographic side-channel analysis [126] and are designed for a parallel computer architecture with a 32-bit instruction set. This makes the third generation of NVIDIA GPUs, the GTX 400/500 series known as *Fermi*, an ideal target platform. Despite the fact that our algorithms are not particularly optimized for the older generation GPUs, we show that this approach outperforms, in terms of low-latency, the results reported in literature while it at the same time sustains a high throughput. For the Fermi architecture the ECSM can be computed in 1.9 milliseconds (on the GTX 580), using an elliptic curve over a 224-bit prime field, with the additional advantage that the implementation can be made to run in constant time; i.e. resistant against timing attacks.

This chapter merges the two papers [30, 31] and parts of [35].

3.1 Fast Reduction using Special Primes

One way to speed up elliptic curve arithmetic is to enhance the performance of the finite field arithmetic by using a prime of a special form. The structure of such a prime is exploited by constructing a fast reduction method, applicable to this prime only. Typically, the multiplication and reduction are performed in two sequential phases. For the multiplication phase we consider the so-called schoolbook, or textbook, multiplication and the asymptotically faster Karatsuba multiplication techniques (see Chapter 2 for more details).

3.1.1 NIST Primes

In the FIPS 186-3 standard [199] NIST recommends the use of five prime fields when using the elliptic curve digital signature algorithm. These generalized Mersenne primes allow fast reduction based on the work by Solinas [188]. The five recommended primes are

Algorithm 5 Fast reduction modulo $p_{224} = 2^{224} - 2^{96} + 1$.

Input: Integer $c = (c_{13}, \ldots, c_1, c_0)$, each c_i is a 32-bit word, and $0 \leq c < p_{224}^2$.
Output: Integer $d \equiv c \bmod p_{224}$.
Define 224-bit integers:

$$\begin{array}{lrrrrrrr}
s_1 \leftarrow (& c_6, & c_5, & c_4, & c_3, & c_2, & c_1, & c_0), \\
s_2 \leftarrow (& c_{10}, & c_9, & c_8, & c_7, & 0, & 0, & 0), \\
s_3 \leftarrow (& 0, & c_{13}, & c_{12}, & c_{11}, & 0, & 0, & 0), \\
s_4 \leftarrow (& c_{13}, & c_{12}, & c_{11}, & c_{10}, & c_9, & c_8, & c_7), \\
s_5 \leftarrow (& 0, & 0, & 0, & 0, & c_{13}, & c_{12}, & c_{11})
\end{array}$$

return $(\mathrm{d} = s_1 + s_2 + s_3 - s_4 - s_5)$;

$$\begin{array}{ll}
p_{192} = 2^{192} - 2^{64} - 1, & p_{224} = 2^{224} - 2^{96} + 1, \\
p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1, & p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1, \\
p_{521} = 2^{521} - 1. &
\end{array}$$

Let us take p_{224} as an example since it is the prime considered in the GPU architecture setting in this chapter. The usage of the other primes in the setting of the Cell platform is similar. The prime p_{224}, together with the provided curve parameters from the FIPS 186-3, allows one to use 224-bit ECC which provides a 112-bit security level. This is the lowest strength for asymmetric cryptographic systems allowed by NIST's "SP 800-57 (1)" [156] standard from the year 2011 on (cf. [34] for a discussion about the migration to these new standards).

Reduction modulo p_{224} can be done efficiently: for $x \in \mathbf{Z}$ with $0 \leq x < (2^{224})^2$ and $x = x_L + 2^{224} x_H$ for $x_L, x_H \in \mathbf{Z}, 0 \leq x_L, x_H < 2^{224}$, define

$$\mathfrak{R}(x) = x_L + x_H(2^{96} - 1).$$

It follows that $\mathfrak{R}(x) \equiv x \bmod p_{224}$ and $\mathfrak{R}(x) \leq 2^{320} - 2^{96}$. Algorithm 5 shows the application of $\mathfrak{R}(\mathfrak{R}(x))$ for a machine word (limb) size of 32 bits, based on the work by Solinas [188]. Note that the resulting value $\mathfrak{R}(\mathfrak{R}(x)) \equiv x \bmod p_{224}$ with $-(2^{224} + 2^{96}) < \mathfrak{R}(\mathfrak{R}(x)) < 2^{225} + 2^{192}$.

The NIST curves over prime fields all have prime order. In order to translate this curve into a suitable Edwards curve over the same prime field (see Chapter 2), in order to use the faster elliptic curve arithmetic, the curve needs to have a group element of order four [20] (which is not the case with the prime order NIST curves). To comply with the NIST standard we choose not to use Edwards but Weierstrass curves.

An extensive study of a software implementation of the NIST-recommended elliptic curves over prime fields on the x86 architecture is given by Brown et al. [53].

3.1.2 Curve25519

The elliptic curve *curve25519* is proposed by Bernstein in [12]. Besides offering high-speed arithmetic, a list of other advantages can be found in the original article [12]. This curve is

over $\mathbf{F}_{p_{255}}$ with $p_{255} = 2^{255} - 19$. An element $x \in \mathbf{F}_{p_{255}}$ can be represented as

$$x = \sum_{i=0}^{9} x_i 2^{\lceil 25.5i \rceil}, \quad \text{with } -2^{25} \leq x_i \leq 2^{25}.$$

Bernstein proposes to implement the arithmetic using floating point instructions and therefore representation inside a CPU is achieved by using floating-point registers. The original article gives performance data obtained on a Pentium M architecture. Note that the faster Edwards curves can be used in combination with p_{255} since the curve described in [12] has a point of order four.

3.2 Applications

Modular multiplication is the main operation when computing the elliptic curve scalar multiplication. This, in its turn, is the core computation in almost all elliptic curve based cryptographic schemes. Enhancing the practical performance of modular multiplication results directly in faster elliptic curve based cryptographic protocols.

It might be less obvious to find applications that might benefit from processing multiple input streams, as we propose in this chapter for the Cell. To increase throughput, the 4-way SIMD instructions of the SPE are used to implement a modular multiplication routine which computes 4 streams, or a small multiple of 4 by interleaving these streams, in parallel (i.e. 4 modular multiplications are being processed concurrently). When a sequence of multiplications has to be computed, for instance in elliptic curve scalar multiplication, the algorithm performs the same operations in SIMD-mode on all inputs. When the scalar multipliers are different, a square-and-multiply algorithm needs to perform a different sequence of point additions and doublings, since this depends on the binary expansion of the scalar multiplier. Performing the same computations on multiple streams concurrently, when multiplying with different scalars, in a SIMD fashion might be suboptimal since all streams which are being processed in parallel need to perform the same computations. In this section we present some applications in cryptography and cryptanalysis where SIMD modular multiplication algorithms can be beneficial; i.e., where the same multiplier is used in multiple independent instances.

3.2.1 Cryptography

Cryptographic schemes often need to perform exponentiations with a randomly selected exponent, or scalar multiplications when using the additive group law as in the elliptic curve setting. If this exponent is used several times, in independent calculations, these operations can be performed in parallel in a SIMD fashion. For instance, in elliptic curve public-key schemes the ability to process multiple streams of modular multiplication computations can be used to speedup batch decryption. Examples of such schemes are the elliptic curve integrated encryption scheme (ECIES), proposed by Bellare and Rogaway [10] and standardized in [172], and the provably secure encryption curve scheme (PSEC), based on the work by

Fujisaki and Okamoto [83] and standardized in [109]. The decryption of a message consists of multiplying an elliptic curve point, as specified by the ciphertext, by the private key d in PSEC or by $h \cdot d$ in the case of ECIES, where $h \in \mathbf{Z}$ is a divisor of the cardinality of the elliptic curve and is constant for a given private key. When many messages need to be decrypted, using the same private key, SIMD algorithms as described in this article can be used to speedup computations.

In other settings, where the bitsize of the modulus is usually larger compared to the ECC setting, multi-stream modular multiplication computations can be useful as well. ElGamal encryption schemes [75] require two exponentiations with the same random exponent. Other related methods perform more exponentiations with the same exponent. The double base variant of ElGamal by Damgård, often referred to as Damgård ElGamal [67], performs three exponentiations. The "double" hybrid Damgård ElGamal, as proposed by Kiltz et al. [117], requires four exponentiations with the same exponent in every encryption.

3.2.2 Cryptanalysis

In cryptanalysis, multi-stream modular multiplication computations, for moduli sizes as considered in this article (in the 100-500 bit range), can be used to enhance the performance of the Pollard rho discrete logarithm algorithm [166], a method to solve the elliptic curve discrete logarithm problem (ECDLP) which is essential to assess the security of ECC (see Chapter 2 and 4). This approach is used, for instance, in Chapter 5, when solving a 112-bit ECDLP on the SPE architecture by working concurrently on 400 computations. Here, 70 percent of the total run-time is spent on the computation of modular multiplications.

Another cryptanalytic application is factoring integers. The integer factorization problem is essential to the security of cryptographic algorithms as RSA. The fastest known method to factor integers is the number field sieve [133, 162]. This method can use the elliptic curve factorization method (ECM) [136] (see Section 6.2) in a co-factorization phase. Performing elliptic curve arithmetic on multiple points in parallel allows the use of multi-stream modular multiplication methods. Related work by Bernstein et al. [18] gives performance details of a high-performance multi-stream implementation of modular arithmetic in ECM on graphics cards.

3.3 Representation of Long Integers

3.3.1 Representation of Long Integers on the SPU

To represent integers on the Cell one could directly use the 128-bit registers of the SPU to represent (part of) a single integer. But this simple-minded approach is not easily compatible with the SPU's instruction set.

For applications that allow high degrees of parallelization a 90-degree interpretative turn of the words is a better fit for the SPU's instruction set: instead of representing an m-bit integer using $\lceil \frac{m}{128} \rceil$ 128-bit registers, a **four-tuple** of long integers is laid out across the four-tuples of words of a sequence of 128-bit registers, thereby allowing the corresponding words of

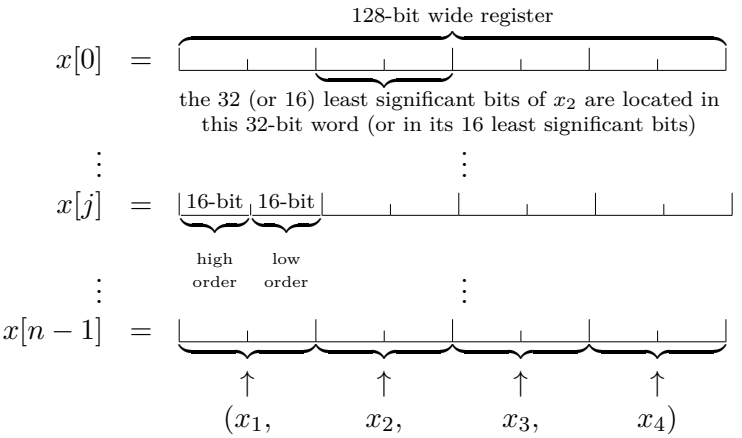

Figure 3.1: A four-tuple (x_1, x_2, x_3, x_4) of $32n$-bit or $16n$-bit integers represented by 128-bit registers $x[0], x[1], \ldots, x[n-1]$.

the four long integers, i.e., the words that belong to the same 128-bit register, to be processed simultaneously in SIMD fashion. Figure 3.1 illustrates two ways to map four-tuples of long integers to a sequence of 128-bit registers: one that uses all $4 \times 32 = 128$ bits of each register, and one where only $4 \times 16 = 64$ of the 128 bits per register are significant. This approach allows 4-way SIMD processing of four-tuples of identically sized long integers of any size.

Both methods represent four-tuples of long integers by *word slicing* a number of 128-bit registers. The choice of representation (4×32 or 4×16 bits used per 128-bit register) depends on the operation to be carried out. Each 128-bit register v is interpreted as a four-tuple (v_1, v_2, v_3, v_4) of 32-bit words. Here these words are interpreted as unsigned 32-bit integers.

In the first representation method, a sequence of ℓ 128-bit registers $x[0], x[1], \ldots, x[\ell-1]$ is used to represent a four-tuple (x_1, x_2, x_3, x_4) of 32ℓ-bit integers in their radix 2^{32} representation:

$$x_i = \sum_{j=0}^{\ell-1} x[j]_i 2^{32j}$$

for $i = 1, 2, 3, 4$. Thus, the ith word $x[j]_i$ of the 128-bit register $x[j]$ equals the coefficient of 2^{32j} in the radix 2^{32} representation of the ith 32ℓ-bit integer x_i, for $j = 0, 1, \ldots, \ell-1$ and $i = 1, 2, 3, 4$. This representation matches the SPU's 4-way SIMD integer additions and subtractions.

In the second representation method, a sequence of m 128-bit registers $y[0], y[1], \ldots,$ y[m-1] is used to represent a four-tuple (y_1, y_2, y_3, y_4) of $16m$-bit integers in their radix 2^{16} representation:

$$y_i = \sum_{j=0}^{m-1} (y[j]_i \bmod 2^{16}) 2^{16j}$$

for $i = 1, 2, 3, 4$ and where $0 \leq y[j]_i \bmod 2^{16} < 2^{16}$. Thus, the two least significant bytes of

the ith word $y[j]_i$ of the 128-bit register $y[j]$ contain the coefficient of 2^{16j} in the radix 2^{16} representation of the ith $16m$-bit integer y_i, for $j = 0, 1, \ldots, m-1$ and $i = 1, 2, 3, 4$. When used with the shift instruction `spu_sl`, this representation matches the SPU's 4-way SIMD unsigned multiply-and-add instruction `spu_mhhadd` (see Section 2.2.2 for the specification of these instructions).

Thus we use the 128-bit register width to hard-code 4-way SIMD processing of four-tuples of long integers. The values for ℓ (full-word radix 2^{32}) and m (bottom-half-word radix 2^{16}) depend on the modulus size.

3.3.2 Representation of Long Integers on the GPU

All recent GPU architectures support 32-bit instructions. Hence, long integers on the GPU are represented in the usual way by writing an m-bit number x in a radix-2^{32} representation:

$$x = \sum_{i=0}^{\lceil \frac{m}{32} \rceil} x_i 2^{32i} \text{ with } 0 \leq x_i < 2^{32}.$$

3.4 Finite Field Arithmetic

In order to speed up the modular calculations we represent the integers $x \in \mathbf{F}_p$ using a redundant representation. Instead of fully reducing x to the range $[0, p\rangle$, for an m-bit prime p, we use the slightly larger interval $[0, 2^m\rangle$. This redundant representation saves a multi-limb comparison to detect if we need to perform an additional subtraction after a number has been reduced to $[0, 2^{224}\rangle$. Using this representation, reduction can be done more efficiently, as outlined in this section, while it does not require more 32-bit limbs (or registers) to store the integers. Various operations need to be adopted in order to handle the boundary cases, this is outlined in this section.

The (modular) multiplication operations in this chapter are designed to operate on relatively small (≤ 521 bits) integers. On the widely available x86 and x86-64 architectures the threshold for switching from schoolbook multiplication to methods with a lower asymptotic run-time complexity (e.g. Karatsuba multiplication) is > 800 bits [92] (but this threshold depends on the word-size of the architecture). On these architectures the size of the operands on which the multiplication and addition instructions work is typically the same (either 32 or 64 bits).

On the Cell "only" a $16 \times 16 \rightarrow 32$ bits multiplication instruction is available (see Section 2.2.1), performing four multiplications in parallel, while the size of the 4-way SIMD operands to the addition instruction is 32 bits. Unlike the x86 architecture an integer multiply-and-add instruction is available. This allows the addition of two extra 16-bit values to a result of a 16-bit multiplication without generating a carry, since if $0 \leq a, b, c, d < 2^{16}$, then $a \cdot b + c + d < 2^{32}$. We consider both the schoolbook and Karatsuba multiplication for the special modular multiplication routines on the Cell architecture and only the schoolbook approach for the single case, 224-bit multiplication, considered on the GPUs.

For the GPU-architecture we aim to lower the latency. A common approach to achieve this is to compute the modular multiplications with multiple threads using a residue number system (RNS) [88, 142]. This might be one of the few available options to lower the latency for schemes which perform a sequence of data-dependent modular multiplications, such as in RSA where the main operation is modular exponentiation, but different approaches can be tried in the setting of elliptic curve arithmetic. We follow the ideas from [105,113] and choose, in contrast to for instance [4], to let a single thread compute a single modular multiplication. The parallelism is exploited at the elliptic curve arithmetic level where multiple instances of the finite field arithmetic are computed in parallel to implement the elliptic curve group operation.

3.4.1 Modular Addition and Subtraction

After an addition of a and b, with $0 \leq a, b < 2^{224}$, resulting in $a + b = c = c_H 2^{224} + c_L$ with $0 \leq c_L < 2^{224}$ and $0 \leq c_H \leq 1$ there are different strategies to compute the modular reduction of c. One can subtract p_{224} once ($c_H = 1$) or the result is already in the proper interval ($c_H = 0$) and subsequently continue using the 224 least significant bits of the outcome. In order to prevent divergent code on parallel computer architectures the value $c_H p_{224}$ (either 0 or p_{224}) could be pre-computed and subtracted after the addition of a and b. Note that an addition subtraction might be required in the unlikely event that $c_H = 1$ and $a + b - p_{224} \geq 2^{224}$.

A faster approach is to use the special form of the prime p_{224}. Since,

$$c = c_H 2^{224} + c_L \equiv c_L + c_H(2^{96} - 1) \bmod p_{224}, \tag{3.1}$$

this requires the computation of one addition of a and b and one addition with the pre-computed constant $c_H(2^{96} - 1)$, for $c_H \in \{0, 1\}$. Again, in the unlikely event that $c_H = 1$ and $c_L + 2^{96} - 1 \geq 2^{224}$ an additional subtraction is required. The probability to obtain a carry after adding the fourth 32-bit limb of $2^{96} - 1$ to c_L is so small that an early-abort strategy can be applied; i.e. all concurrent threads within the same warp assume that no carry is produced and continue executing the subsequent code, in the unlikely event that one or more of the threads produce a carry this results in divergent code and the other threads in this warp remain idle until the computation has been completed. This strategy decreases the number of instructions required to implement the modular addition and makes this latter approach preferable in practice.

For modular subtraction the same two approaches can be applied. In the first approach the modulus p_{224} is added to $c = a - b$ if there is a borrow out: i.e. $b > a$. An additional addition of p_{224} might be required since $0 > p_{224} - 2^{224} < a - b + p_{224}$. In the second approach $2p_{224} + a$ is computed before subtracting b, to ensure that the result is positive. Next, we proceed as in the addition scenario with the only difference that $c_H \in \{0, 1, 2\}$.

Algorithm 6 Radix-2^r schoolbook multiplication algorithm for architectures which have a multiply-and-add instruction. We use $r = 16$ and $r = 32$ for the Cell and GPU architecture respectively.

Input: Integers $a = \sum_{i=0}^{n-1} a_i 2^{ri}$, $b = \sum_{i=0}^{n-1} b_i 2^{ri}$, with $0 \leq a_i, b_i < 2^r$.

Output: Integer $c = a \cdot b = \sum_{i=0}^{2n-1} c_i 2^{ri}$, with $0 \leq c_i < 2^r$.

1. $d_i \leftarrow 0, \quad i \in [0, n-1]$
2. **for** $j = 0$ to $n-1$ **do**
3. $\quad (e, D_j) \leftarrow \texttt{split}(a_0 \cdot b_j + d_0)$
4. $\quad$ **for** $i = 1$ to $n-1$ **do**
5. $\quad\quad (e, d_{i-1}) \leftarrow \texttt{split}(a_i \cdot b_j + e + d_i)$
6. $\quad d_{n-1} \leftarrow e$
7. **return** $(c \leftarrow (d_{n-1}, d_{n-2}, \ldots, d_0, D_{n-1}, D_{n-2}, \ldots, D_0))$

3.4.2 Modular Multiplication

Algorithm 6 depicts schoolbook multiplication designed to run on SIMD architectures and is optimized for architectures with a native multiply-and-add instruction. After trivially unrolling the for-loops the algorithm is branch-free. Algorithm 6 splits the operands in r-bit words and takes advantage of the r-bit multiplier assumed to be available on the target platform. We use $r = 16$ and $r = 32$ for the Cell and GPU architecture respectively but this can be modified to work with any other word size on different architectures. After the multiply-and-add, and a possible extra addition of one r-bit word, the $2r$-bit result z is split into the r most and r least significant bits, x and y respectively. This is denoted by $(\lfloor \frac{z}{2^r} \rfloor, z \bmod 2^r) \leftarrow \texttt{split}(z)$ (see Section 2.2.2).

Multiplication on the SPU

On the SPE, Algorithm 6 operates on four-tuples of inputs simultaneously using the data representation from Figure 3.1.

On the SPE the splitting can be implemented in different ways, i.e. by using two odd `shuffle` instructions, or one even `and` and one odd `shuffle` instruction, or two even `and` instructions. The appropriate `split`ting implementation is chosen to balance the number of odd and even instructions, reducing the total number of required cycles. Note that when $i = 1$ the extra addition of d_{i+1} can be omitted. Hence, Algorithm 6 requires $n^2 \times$ `split`, $n^2 \times$ `muladd` and $n(n-2) \times$ `add` (when multiplying two $16n$-bit integers); this can be computed in $2n(n - \frac{3}{4})$ cycles, optimistically assuming all odd and even pairs can be dispatched simultaneously. Furthermore, this approximation ignores the function-call overhead and loading and storing the in- and output from the local store. Hence, an optimistic approximation for the computation of a single $16n \times 16n \rightarrow 32n$-bit schoolbook multiplication is $\frac{n}{2}\left(n - \frac{3}{4}\right)$ cycles

Algorithm 7 Radix-2^{32} Karatsuba multiplication algorithm for architectures which support vector instructions, n is even.

Input: $\begin{cases} \text{Integer } X = (x_{n-1}, \ldots, x_0), & \text{each } x_i \text{ is a 32-bit word.} \\ \text{Integer } Y = (y_{n-1}, \ldots, y_0), & \text{each } y_i \text{ is a 32-bit word.} \end{cases}$

Output: Integer $Z = (z_{2n-1}, \ldots, z_0) = X \cdot Y$, each z_i is a 32-bit word.

1. $(B_{n-1}, \ldots, B_0) \leftarrow \text{mul}((x_{n-1}, \ldots, x_{n/2}),\ (y_{n-1}, y_{n/2}))$
2. $(C_{n-1}, \ldots, C_0) \leftarrow \text{mul}((x_{n/2-1}, \ldots, x_0),\ (y_{n/2-1}, \ldots, y_0))$
3. $zero \leftarrow \text{carry}_1 \leftarrow \text{carry}_2 \leftarrow \{0\}$
4. **for** $i = 0$ to $n/2 - 1$ **do**
5. $\quad X_i \leftarrow \text{add\_extended}(x_{n/2+i},\ x_i,\ \text{carry}_1)$
6. $\quad Y_i \leftarrow \text{add\_extended}(y_{n/2+i},\ y_i,\ \text{carry}_2)$
7. $\quad \text{carry}_1 \leftarrow \text{gen\_carry\_extended}(x_{n/2+i},\ x_i,\ \text{carry}_1)$
8. $\quad \text{carry}_2 \leftarrow \text{gen\_carry\_extended}(y_{n/2+i},\ y_i,\ \text{carry}_2)$
9. $\text{mask}_1 \leftarrow \text{cmpgt}(\text{carry}_1, 0),\ \text{mask}_{\leftarrow}\text{cmpgt}(\text{carry}_2, 0)$
10. **for** $i = 0$ to $n/2 - 1$ **do**
11. $\quad s_i \leftarrow \text{select}(\text{zero}, Y_i, \text{mask}_1),\ t_i \leftarrow \text{select}(\text{zero}, X_i, \text{mask}_2)$
12. $c_1 \leftarrow \text{select}(\text{zero}, \text{carry}_1, \text{mask}_2)$
13. $(z_{n-1}, \ldots, z_{n/2}, A_{n/2-1}, \ldots, A_0) \leftarrow \text{mul}((X_{n/2-1}, \ldots, X_0),\ (Y_{n/2-1}, \ldots, Y_0))$
14. $\text{carry}_1 \leftarrow \text{carry}_2\{0\}$
15. **for** $i = n/2$ to $n - 1$ **do**
16. $\quad T \leftarrow \text{add\_extended}(z_i,\ s_{i-n/2},\ \text{carry}_1)$
17. $\quad A_i \leftarrow \text{add\_extended}(T,\ t_{i-n/2},\ \text{carry}_2)$
18. $\quad \text{carry}_1 \leftarrow \text{gen\_carry\_extended}(z_i,\ s_{i-n/2},\ \text{carry}_1)$
19. $\quad \text{carry}_2 \leftarrow \text{gen\_carry\_extended}(T,\ t_{i-n/2},\ \text{carry}_2)$
20. $A_n \leftarrow \text{add\_extended}(\text{carry}_1,\ \text{carry}_2,\ c_1)$
21. $\text{borrow}_1 \leftarrow \text{borrow}_2 \leftarrow \{1\}$
22. **for** $i = 0$ to $n - 1$ **do**
23. $\quad T \leftarrow \text{sub\_extended}(A_i,\ B_i,\ \text{borrow}_1)$
24. $\quad E_i \leftarrow \text{sub\_extended}(T,\ C_i,\ \text{borrow}_2)$
25. $\quad \text{borrow}_1 \leftarrow \text{gen\_borrow\_extended}(A_i,\ B_i,\ \text{borrow}_1)$
26. $\quad \text{borrow}_2 \leftarrow \text{gen\_borrow\_extended}(T,\ C_i,\ \text{borrow}_2)$
27. $E_n \leftarrow \text{sub}(A_n,\ zero,\ \text{borrow}_1),\ E_n \leftarrow \text{sub}(A_n,\ zero,\ \text{borrow}_2)$
28. $\text{carry}_1 \leftarrow 0$
29. **for** $i = n/2$ to $n - 1$ **do**
30. $\quad Z_i \leftarrow \text{add\_extended}(C_i,\ E_{i-n/2},\ \text{carry}_1)$
31. $\quad \text{carry}_1 \leftarrow \text{gen\_carry\_extended}(C_i,\ E_{i-n/2},\ \text{carry}_1)$
32. **for** $i = n$ to $n + n/2 - 1$ **do**
33. $\quad Z_i \leftarrow \text{add\_extended}(B_{i-n},\ E_{i-n/2},\ \text{carry}_1)$
34. $\quad \text{carry}_1 \leftarrow \text{gen\_carry\_extended}(B_{i-n},\ E_{i-n/2},\ \text{carry}_1)$
35. $Z_{n+n/2} \leftarrow \text{add\_extended}(B_{n/2},\ E_n,\ \text{carry}_1)$
36. $\text{carry}_1 \leftarrow \text{gen\_carry\_extended}(B_{n/2},\ E_n,\ \text{carry}_1)$
37. **for** $i = n + n/2 + 1$ to $2n - 1$ **do**
38. $\quad Z_i \leftarrow \text{add}(B_{i-n},\ \text{carry}_1)$
39. $\quad \text{carry}_1 \leftarrow \text{gen\_carry}(B_{i-n},\ \text{carry}_1)$
40. **return** $Z \leftarrow (Z_{2n-1}, \ldots, Z_{n/2}, C_{n/2-1}, \ldots, C_0)$

on average (when processing 4 streams in parallel).

A branch-free (when unrolled) Karatsuba multiplication algorithm optimized for vector architectures is given in Algorithm 7. This algorithm works on 32-bit words, which is the word size of the even 4-way SIMD addition and subtraction instructions on the SPE. Just as with the schoolbook multiplication this word size can trivially be modified. Algorithm 7 assumes that the bitsize of the input values is a multiple of 64 to split the operands evenly in two 32-bit multiples. These parts are multiplied using another multiplication routine `mul`, which is either a schoolbook or Karatsuba multiplication, which operates on inputs of half the size.

The $2m$-bit multiplication is split into two $m \times m$-bit and one $(m+1) \times (m+1)$-bit multiplications (see Chapter 2, Algorithm 2). In order to avoid the use of a probably more expensive multiplication by an extra limb (the $(m+1) \times (m+1)$-bit multiplication), three $m \times m$-bit multiplications are used. The correct result, for the $(m+1) \times (m+1)$-bit multiplication, is computed by creating select-masks from the most significant bit of each of the two operands. These are used to select the appropriate value (one of the inputs) or zero, which is added to the result of the $m \times m$-bit multiplication. Note that the initial borrow values, in line 21, are (counterintuitively) set to one. An extra subtraction of one is performed when the borrow is zero and no subtraction is performed when the borrow is one on the SPE.

Multiplication on the GPU

Recall that on the GPU we consider the 224-bit prime p_{224}. The $224 \times 224 \rightarrow 448$-bit multiplication is computed using the schoolbook multiplication method. For $r = 32$ a radix-2^{32} schoolbook multiplication algorithm is presented in Algorithm 6. This algorithm requires the computation of n times $\texttt{split}(a_0 \cdot b_j + d_0)$ and $n(n-1)$ times $\texttt{split}(a_i \cdot b_j + e + d_i)$, where $n = 7$ for the 224-bit multiplication. On the GTX 400 family of GPUs, where there are $32 \times 32 \rightarrow 32$-bit multiplication instructions to get the lower and higher 32-bits and 32-bit additions with carry in and out, the former can be implemented using four and the later using six instructions. A direct implementation of the schoolbook algorithm as presented in Chapter 2 (Algorithm 1) might result in a slightly lower instruction count, using the addition with carry in- and out, but has the disadvantage that it requires more storage (registers) compared to Algorithm 6. We benchmarked both approaches on different GPU families. The more memory efficient method as presented in Algorithm 6 is to be preferred in practice.

3.4.3 Fast Reduction

The special reduction algorithms used with the NIST primes do not fully reduce the input to the range $[0, p\rangle$ but to $[0, t \cdot p\rangle$, where p is the prime modulus used and t a small positive integer. In order to fully reduce multiple integers simultaneously using SIMD/SIMT instructions, several approaches can be applied. Obviously the reduction algorithm can be applied again. A most likely faster approach, when t is sufficiently small, is to subtract p repeatedly until the result is in the desired range $[0, p\rangle$. Since the arithmetic is executed on parallel architectures, the repeated subtraction is calculated by masking the value appropriately before subtracting,

Table 3.1: The values of the 32-bit unsigned limbs c_i of $t \cdot p_{224} = \sum_{i=0}^{7} c_i 2^{32i}$

t	$t \cdot p_{224} = \{c_7, \ldots, c_0\}$							
	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
0	0	0	0	0	0	0	0	0
1	0	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	0	0	1
2	1	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-2$	0	0	2
3	2	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-3$	0	0	3
4	3	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-4$	0	0	4

which needs to be performed up to $t-1$ times since multiple integer values are processed in parallel. This approach is used to avoid divergent code on the GPU. On SIMD-architectures, like the Cell, computing these values t can be different for the multiple streams which makes it hard to use branches.

An additional performance gain is possible at the expense of some storage. Select the desired multiple of the modulus p which needs to be subtracted from a look-up table, and perform a single subtraction. This can be achieved efficiently on the Cell, when operating on multiple integer values in parallel, using the `select` instruction. Using a redundant representation in $[0, 2^m\rangle$, for an m-bit modulus p, the most significant word, containing the possible carry, has to be inspected only to determine the multiple of p to subtract. Note that an extra single subtraction might be needed in the unlikely situation that the result after the subtraction is $> 2^m$. This rare case is implemented by a branch which is hinted to be false to reduce branch-overhead or avoid divergent code. The partially reduced numbers can be used as input to the same modular multiplication routines and if reduction to $[0, p\rangle$ is required this can be achieved at the cost of a single conditional multi-limb subtraction.

One can do more for the moduli of special form. For example consider the modulus $p_{224} = 2^{224} - 2^{96} + 1$. The output from the fast reduction routine as outlined in Algorithm 5, and denoted by *Red*, is not in the preferred range $[0, p_{224}\rangle$ nor in the range for the redundant representation $[0, 2^{224}\rangle$; instead, $-(2^{224} + 2^{96}) < Red(a \cdot b) < 2^{225} + 2^{192}$. In order to avoid working with negative (signed) numbers we modify the algorithm slightly such that it returns $d = s_1 + s_2 + s_3 - s_4 - s_5 + 2p_{224} \equiv c = a \cdot b \bmod p_{224}$ where $2^{224} - 2^{96} < d < 2^{226} + 2^{192}$ (instead of the value $s_1 + s_2 + s_3 - s_4 - s_5$ from Algorithm 5).

A refinement, in terms of storage, of the previous approaches to reduce the resulting value is by generating the desired values efficiently on-the-fly. We distinguish two cases (just as when doing the modular addition in Section 3.4.1); either subtract multiples of p_{224} or $2^{96}-1$. Selecting the correct multiple of p_{224} is illustrated in Table 3.1. The 32-bit unsigned limbs c_i of $t \cdot p_{224} = \sum_{i=0}^{7} c_i 2^{32i}$ for $0 \leq t < 5$ can be computed as $c_0 = t$, $c_1 = c_2 = 0$, $c_3 = 0 - t$. The values for c_4, c_5, c_6, c_7 can be efficiently constructed using masks depending on $t = 0$ or $t > 0$. When subtracting multiples of $2^{96} - 1 = \sum_{i=0}^{3} c_i 2^{32i}$ for $0 \leq t < 5$, the constants can

Algorithm 8 Radix-2^r Montgomery Multiplication Algorithm.

Input: $\begin{cases} \text{Integers } a = \sum_{i=0}^{n-1} a_i 2^{ri}, b = \sum_{i=0}^{n-1} b_i 2^{ri}, M = \sum_{i=0}^{n-1} M_i 2^{ri}, \widetilde{m} = -M^{-1} \bmod 2^r. \\ \text{such that } M \text{ is odd}, 0 \leq a, b < 2^{rn}, 2^{r(n-1)} \leq M < 2^{rn} \text{ and } 0 \leq a_i, b_i, M_i < 2^r \end{cases}$

Output: Integer $c = \sum_{i=0}^{n-1} c_i 2^{ri} \equiv a \cdot b \cdot 2^{-rn} \bmod M$.

1. $d_i \leftarrow 0, \quad i \in [0, n]$
2. **for** $i = 0$ to $n-1$ **do**
3. $\quad (e_0, d_0) \leftarrow \texttt{split}(a_0 \cdot b_i + d_0)$
4. $\quad$ **for** $j = 1$ to $n-1$ **do**
5. $\quad\quad (e_j, d_j) \leftarrow \texttt{split}(a_j \cdot b_i + d_j + e_{j-1})$
6. $\quad d_n \leftarrow d_n + e_{n-1}$
7. $\quad (*, q) \leftarrow \texttt{split}(d_0 \cdot \widetilde{m})$
8. $\quad (e_0, d_0) \leftarrow \texttt{split}(M_0 \cdot q + d_0)$
9. $\quad$ **for** $j = 1$ to $n-1$ **do**
10. $\quad\quad (e_j, d_{j-1}) \leftarrow \texttt{split}(M_j \cdot q + d_j + e_{j-1})$
11. $\quad (d_n, d_{n-1}) \leftarrow \texttt{split}(d_n + e_{n-1})$
12. **if** $d_n > 0$ **then**
13. $\quad (d_{n-1}, \ldots, d_1, d_0) \leftarrow (d_n, d_{n-1}, \ldots, d_1, d_0) - (M_{n-1}, \ldots, M_1, M_0)$
14. **return** $(c = (d_{n-1}, \ldots, d_1, d_0))$

be computed as

$$\begin{aligned} c_0 &= 0 - t \\ c_1 = c_2 &= \begin{cases} 0, & \text{if } t = 0, \\ 2^{32} - 1, & \text{if } t > 0. \end{cases} \\ c_3 &= \begin{cases} 0, & \text{if } t = 0, \\ t - 1, & \text{if } t > 0. \end{cases} \end{aligned}$$

The conditional statements can be converted to straight line (non-divergent) code to make the algorithms more suitable for parallel computer architectures.

3.4.4 Montgomery Multiplication on the SPU

The interleaved Montgomery multiplication, optimized for the use on vector architectures, is given in Algorithm 8. As presented, it uses 16-bit limbs and on the Cell four-tuples of inputs are processed concurrently (but Algorithm 8 can trivially be modified to operate on any radix size). A conditional subtraction step is needed at the end of the algorithm to ensure that the result is $< 2^{16n}$, for $16n$-bit inputs. This conditional subtraction is replaced by a comparison which creates a select mask, using this mask the value zero or the value of the modulus is selected and subtracted. This eliminates a branch which is to be avoided when processing multiple integer values in a SIMD fashion. For efficiency, the integer representation

is switched to a 2^{32} radix system when doing the final masking and subtraction in practice.

The same notation for the split function is used as in Section 3.4.2. Hence, Algorithm 8 requires $2n(n+1) \times$ `split`, $2n(n+1) \times$ `muladd` (when counting the multiplication in line 8 as a multiply-and-add) and $2n(n-1) \times$ `add` since the addition of d_j in line 5 when $j = 1$ can be omitted. For the conditional subtraction we first convert the integer representation to a 2^{32} radix system using $\lceil \frac{n}{2} \rceil$ `shuffle` instructions. Next we compare the carry (one `cmpgt` instruction) and mask the value which we are going to subtract using $\lceil \frac{n}{2} \rceil$ `and` instructions. The subtraction requires $\lceil \frac{n}{2} \rceil$ (extended) subtraction instructions and $\lceil \frac{n}{2} \rceil - 1$ (extended) generate borrow instructions.

Counting the number of instructions required in Algorithm 8 gives $4n^2 + 3\lceil \frac{n}{2} \rceil$ even and $\lceil \frac{n}{2} \rceil$ odd instructions plus $2n(n+1)$ times the split function. An optimistic estimate of the number of cycles using Algorithm 8 on a single SPE is $n^2 + \frac{9n}{8}$ cycles. This estimate ignoring overhead and assuming perfect scheduling, for a single computation of Montgomery multiplication on $16n$-bit inputs, when computing four computations in parallel.

3.5 Elliptic Curve Arithmetic on the GPU

In our setting we are interested, given a parallel computer architecture capable of launching a number of threads $\mathfrak{T}_i$, to lower the latency of the longest running thread $T_{max} = \max_i T_i$ as opposed to the total time of all resources combined $T_{sum} = \sum_i T_i$ where T_i is the time corresponding to thread $\mathfrak{T}_i$. Since high-throughput and low-latency are two orthogonal goals one cannot achieve both at the same time. Our approach is designed at the elliptic curve arithmetic level for low-latency while not sacrificing the throughput too much. To accomplish this we choose to aim for a high-throughput (and longer latency) design at the finite field arithmetic level: a single thread computes a single multiplication as described in Section 3.4.2. The elliptic curve point addition and duplication are processed simultaneously, significantly reducing the latency at the expense of potentially lowering the throughput.

Another desirable property of a parallel algorithm is that all threads follow the exact same procedure since this reduces the amount of divergent code. An active research area where such algorithms have been studied is in the context of cryptographic side-channel attacks. Side channel attacks [126] are attacks which use information gained from the physical implementation of a certain scheme to break its security; e.g. the elapsed time or power consumption. In order to avoid these types of attacks the algorithms must perform the same actions independent of the input to avoid leaking information. The approach we use is based on the Montgomery ladder (see Section 2.4.2) applied to projective Weierstrass coordinates [51, 77, 110] instead of Montgomery coordinates. Even though the y-coordinate can be recovered, this is not necessary in most of the elliptic curve based cryptographic schemes which only use the x-coordinate to compute the final result.

In particular, we adopt the formulas from [77]. Recall from Algorithm 4 in Section 2.4.2 that every iteration processes a single bit of the scalar at the cost of computing an elliptic

Table 3.2: Instruction overview to compute $(P+Q, 2Q) = (\tilde{P}, \tilde{Q}) = ((\tilde{P}_x, \tilde{P}_z), (\tilde{Q}_x, \tilde{Q}_z))$ using seven threads. The bold entries are pre-computed 224-bit integers, G_x is the x-coordinate of the input point to the Montgomery ladder.

Operation		Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7
(1)	**mul**	$t_0 = P_x Q_z$	$t_1 = Q_x P_z$	$t_2 = P_x Q_x$	$t_3 = P_z Q_z$	$t_4 = Q_x^2$	$t_5 = Q_z^2$	$t_6 = Q_x Q_z$
(2)	**triple**	$t_7 = 3t_3$	$t_8 = 3t_5$					
(3)	**add**	$t_9 = t_0 + t_1$	$t_{10} = t_4 + t_8$					
(4)	**sub**	$t_2 = t_2 - t_7$	$t_0 = t_0 - t_1$	$t_4 = t_4 - t_8$				
(5)	**mul**	$t_9 = t_9 t_2$	$t_3 = t_3^2$	$\tilde{P}_z = t_0^2$	$t_{10} = t_{10}^2$	$t_{11} = t_6 t_5$	$t_6 = t_6 t_4$	$t_5 = t_5^2$
(6)	**mul**	$t_9 = 2t_9$	$t_3 = \mathbf{4b} t_3$	$t_0 = G_x \tilde{P}_Z$	$t_{11} = \mathbf{8b} t_{11}$	$t_5 = \mathbf{4b} t_5$	$t_6 = 4t_6$	
(7)	**add**	$t_9 = t_9 + t_3$	$\tilde{Q}_z = t_5 + t_6$					
(8)	**sub**	$\tilde{P}_x = t_9 - t_0$	$\tilde{Q}_x = t_{10} - t_{11}$					

Table 3.3: Performance comparison of 224-bit elliptic curve scalar multiplication on different GPU platforms. A bold platform name indicates that this platform has compute capability 2.0 (Fermi) or higher (the latest (third) GPU-architecture family). Results when utilizing the entire GPU are expressed in operations (224-bit elliptic curve scalar multiplications) per second (op/s).

Ref	Platform	#GPUs	CUDA cores per GPU	Processor clock (MHz)	Modulus Type	Modulus Bit-size	Minimum latency [ms]	Maximum throughput [op/s]
[18] (scaled)	8800 GTS	1	96	1200	generic	280	-	3 018
	GTX 280	1	240	1296	generic	280	-	11 417
	GTX 295	2	240	1242	generic	280	-	21 103
[17]	GTX 295	2	240	1242	generic	210	-	259 534
[193]	8800 GTS	1	96	1200	special	224	305.0	1 413
[4]	8800 GTS	1	96	1200	special	224	30.3	3 138
	GTX 285	1	240	1476	special	224	24.3	9 990
New	GTX 295	2	240	1242	special	224	10.6	79,198
	GTX 465	1	352	1215	special	224	2.6	152 023
	GTX 480	1	480	1401	special	224	2.3	237 415
	GTX 580	1	512	1544	special	224	1.9	290,535

curve addition and doubling. Computation on the Y-coordinate is omitted as follows [77]

$$
\begin{aligned}
(P+Q, 2Q) = (\tilde{P}, \tilde{Q}) = ((\tilde{P}_x, \tilde{P}_z), (\tilde{Q}_x, \tilde{Q}_z)) = \\
\begin{cases}
\tilde{P}_x = & 2(P_xQ_z + Q_xP_z)(P_xQ_x + aP_zQ_z) \\
& +4bP_z^2Q_z^2 - G_x(P_xQ_z - Q_xP_z)^2 \\
\tilde{P}_z = & (P_xQ_z - Q_xP_z)^2 \\
\tilde{Q}_x = & (Q_x^2 - aQ_z^2)^2 - 8bQ_xQ_z^3 \\
\tilde{Q}_z = & 4(Q_xQ_z(Q_x^2 + aQ_z^2) + bQ_z^4).
\end{cases}
\end{aligned}
\tag{3.2}
$$

Note that G_x is the x-coordinate of the input point to the elliptic curve scalar multiplication algorithm. Using that the NIST standard defines $a = -3$, and slightly rewriting Eq. (3.2), results in the set of instructions presented in Table 3.2. Every row of the table is executed concurrently by the different threads, an empty slot means that the thread either remains idle or works on fake data. The bold entries in Table 3.2 are pre-computed at the initialization phase of the algorithm. The b value is one of the parameters which define the elliptic curve (together with a and p_{224}) and is provided in the standard. The b is invariant for the different concurrent elliptic curve scalar multiplications. Depending on the thread identifier, the pre-computed value is copied to the correct shared memory position which is used in operation number 6.

Using the instruction flow from Table 3.2 seven threads can compute a single elliptic curve scalar multiplication (ECSM) using the Montgomery ladder algorithm. The time T_{max} is three multiplications, two additions, two subtractions and a single triple operation in $\mathbf{F}_{p_{224}}$ to compute a single elliptic curve addition *and* duplication. This is in contrast with T_{sum} which consist of 18 multiplications, two triple, four additions, five subtractions and two multiplications by a power of two in $\mathbf{F}_{p_{224}}$. Although T_{sum} is significantly higher, compared to the cost to process one bit of the scalar using different coordinate representations and different ECSM algorithms, the latency T_{max} is roughly three multiplications to compute both an elliptic curve addition and doubling using seven threads. Running seven threads in parallel is the best, e.g. the highest number of concurrent running threads, we could achieve and it should be noted that this is suboptimal from different perspectives. First of all, a GPU platform using the CUDA paradigm typically dispatches threads in blocks whose size is a multiple of 32. Hence, in each subgroup of eight threads one thread remains inactive: decreasing the overall throughput. Secondly, 20 multiplications are used in Table 3.2 which results in one idle thread in the third multiplication (thread 7 in operation 6). On different parallel platforms, where i threads are processed concurrently, with $2 \leq i \leq 7$, the approach outlined in Table 3.2 can be computed using $\left\lceil \frac{20}{i} \right\rceil$ multiplications.

This approach is not limited to arithmetic modulo p_{224} but applies to any modulus. Given the (estimated) performance of a single modular multiplication, either using a special or a generic modulus, the approach from Table 3.2 can be applied such that the overall latency to multiply an elliptic curve point with a k-bit scalar is approximately the time to compute $k \cdot \left\lceil \frac{20}{i} \right\rceil$ single thread multiplications.

3.6 Performance Results and Discussion

3.6.1 Results on the Cell

We implemented the proposed generic and special modular multiplication algorithms using the C-programming language for the SPEs on the Cell architecture. Four, or a small multiple of four, computations are processed in parallel. The performance benchmarks are performed on a single SPE in the PlayStation 3 game console. We summarize these results, together with other (single and multi-stream computation) modular multiplication results, obtained from the literature, in Table 3.4. The metric of our performance results is the number of cycles for a single modular multiplication computation. Our performance results are obtained by averaging over long sequences, hundreds of millions, of different modular multiplications and include the timing benchmark overhead, the function call overhead, loading and storing the in- and output from the local store and possibly converting the in- and output from the different integer representations (from radix-2^{32} to radix-2^{16} and vice-versa).

Performance Comparison

Performance results obtained with the Multi-Precision Math (MPM) Library [108], provided by IBM in the example API for the Cell, are given in Table 3.4 for different bit-sizes. The MPM library implements a *single-stream* Montgomery multiplication computation. In order to obtain a faster implementation for specific bit-lengths (to make a fair comparison) we unrolled the various loops inside the MPM library. These unrolled versions are significantly faster compared to the standard MPM implementation; e.g., the unrolled 256-bit Montgomery multiplication is 1.4 times faster compared to the unmodified MPM implementation. Our multi-stream implementations have a higher latency compared to the unrolled MPM library but process multiple streams resulting in fewer cycles per single multiplication. For instance, in the setting of 256-bit moduli the unrolled MPM requires 877 cycles for a single multiplication while our implementation requires 1 188 cycles to compute four multiplications in parallel. From a throughput point of vies this is a speedup of almost a factor of three per single multiplication.

In [62] Costigan and Schwabe implement elliptic curve arithmetic aimed at *curve25519* on the SPE architecture. The representation used differs slightly from, but is based on, the one proposed in [12]; an element $x \in \mathbf{F}_{p_{255}}$ is represented as $x = \sum_{i=0}^{19} x_i 2^{\lceil 12.75i \rceil}$. A multi-stream version working on four streams in parallel is implemented and hand-optimized in assembly and "perfectly" scheduled with the surrounding code in a larger function implementing elliptic curve arithmetic. This multi-stream implementation is estimated to compute a single modular multiplication in around 168 cycles [62], this does not include any overhead for saving and storing the in- and output registers to and from the local store, function call overhead and overhead due to benchmarking. In comparison, our implementation requires 175 cycles for a single modular multiplication using a different approach for the special reduction (see Section 3.4.2). This includes loading and storing the in- and output, function call and benchmarking overhead and additional latencies because not all code can be scheduled

Table 3.4: Performance results of (multi-stream) Montgomery (generic) multiplication or modular multiplication modulo the special prime p_i. In the special prime setting a separate multiplication (schoolbook (S) or Karatsuba (K)) and fast reduction phase are computed. The benchmarks are performed on a single SPE on a Cell in a PS3. The stated number of cycles c are the average to compute a single modular multiplication when processing s streams in parallel; the latency for this computation is $c \times s$ cycles. The optimistic estimates are from the formulas from Section 3.4.2 and do not include the special reduction cost.

From	Bitsize of the modulus	Method	#Streams	Performance (#cycles)	Estimate (#cycles)
Here	192	p_{192} (K)	8	105	
Here	192	p_{192} (S)	8	126	68
Here	192	Montgomery	8	176	151
Bernstein et al. [17]	195	Montgomery	6	189	
Here	224	p_{224} (K)	8	139	
Here	224	p_{224} (S)	8	143	93
Here	224	Montgomery	4	234	204
Costigan and Schwabe [62]	255	p_{255} (S)	4	168	
Here	255	p_{255} (K)	8	175	
Here	255	p_{255} (S)	8	182	122
Here	256	p_{256} (S)	8	192	122
Here	256	p_{256} (K)	4	193	
Here	256	Montgomery	4	297	265
MPM unrolled [108]	256	Montgomery	1	877	
MPM [108]	256	Montgomery	1	1 188	
Here	384	p_{384} (K)	4	389	
Here	384	p_{384} (S)	4	391	279
Here	384	Montgomery	4	665	590
MPM unrolled [108]	384	Montgomery	1	1 610	
MPM [108]	384	Montgomery	1	2 092	
Here	521	p_{521} (S)	4	622	500
Here	521	p_{521} (K)	4	723	
Here	512	Montgomery	4	1 393	1 042
MPM unrolled [108]	512	Montgomery	1	2 700	
MPM [108]	512	Montgomery	1	3 275	

perfectly (especially at the beginning and end of the function where stalls occur). Comparing the performance of the two different approaches for the reduction step is difficult since the reported performance results of two versions are in different settings; ours is a stand-alone multiplication function while the implementation from [62] is an inline version working on registers only. In [62] it is estimated that the time to load and store the in- and output requires 56 cycles in the setting of a single modular multiplication. When considering this cost our approach using the redundant representation looks preferable (since $175 < 168 + 56$), especially since we did not use any fine-tuned assembly code to achieve these results.

Improved multi-stream modular multiplication computations results, compared to [18], are given by Bernstein et al. [17]. Here, not only results for GPUs are reported but also for the Cell architecture as used in the PlayStation 3. In this setting Montgomery multiplication is implemented and optimized for one bit size: a 195-bit generic modulus. A radix-2^{13} system is used to represent 195-bit integers using 15 limbs, this has the advantage of accumulating multiple carries before an overflow occurs (on the SPE architecture) compared to a radix-2^{16} system but requires more limbs to represent the integers. When quadratically scaling our 192-bit performance result, in a similar fashion as done in [17], this leads to an estimate of $176 \cdot (\frac{195}{192})^2 = 182$ cycles; this is comparable to the 189 required cycles reported in [17].

Discussion

The performance data from Table 3.4 show that the modular multiplication using the special primes are in almost all cases, with the exception of p_{256} and p_{521}, roughly 1.7 times faster compared to the Montgomery multiplication implementations targeting the same bit-lengths. Our results show that p_{256} is 1.55 times faster than 256-bit Montgomery multiplication while p_{521} is 2.2 times faster compared to 512-bit Montgomery multiplication. This can be partially explained by the relatively complicated and easy structure of p_{256} and p_{521} respectively.

For p_{192} the version using Karatsuba multiplication is significantly (20 percent) faster compared to the version using schoolbook multiplication. For p_{224}, p_{255}, p_{256} and p_{384} the performance is similar while for p_{521} schoolbook multiplication is 16 percent faster. These differences can be explained due to extra load and store operations from and to the local store. For the smaller bitsizes almost all operations can be performed, after the initial loading from the inputs, on registers. For the larger values the available 128 registers are not sufficient and extra load and store instructions, leading to more instructions and possibly extra stalls, are required. This also explains why processing four streams instead of eight gives a higher performance for p_{384} and p_{521} (Table 3.4 shows only the fastest setting).

The number of cycles required for the Montgomery multiplication is 12 to 17 percent higher compared to the estimations for all special primes except p_{521}. This overhead is mainly caused by extra load and stores and due to the fact that the estimates are too optimistic (not every cycle a pair of instructions can be dispatched due to instruction dependencies). For the special prime p_{521} more than 33 percent of the estimated number of cycles is needed. After compiling our code to assembly, inspection shows that the significant overhead is due to the extra loads and stores. Note that loading the two input values, for the four streams in parallel, in registers (after conversion to radix-2^{16}) requires 66 registers which is more than

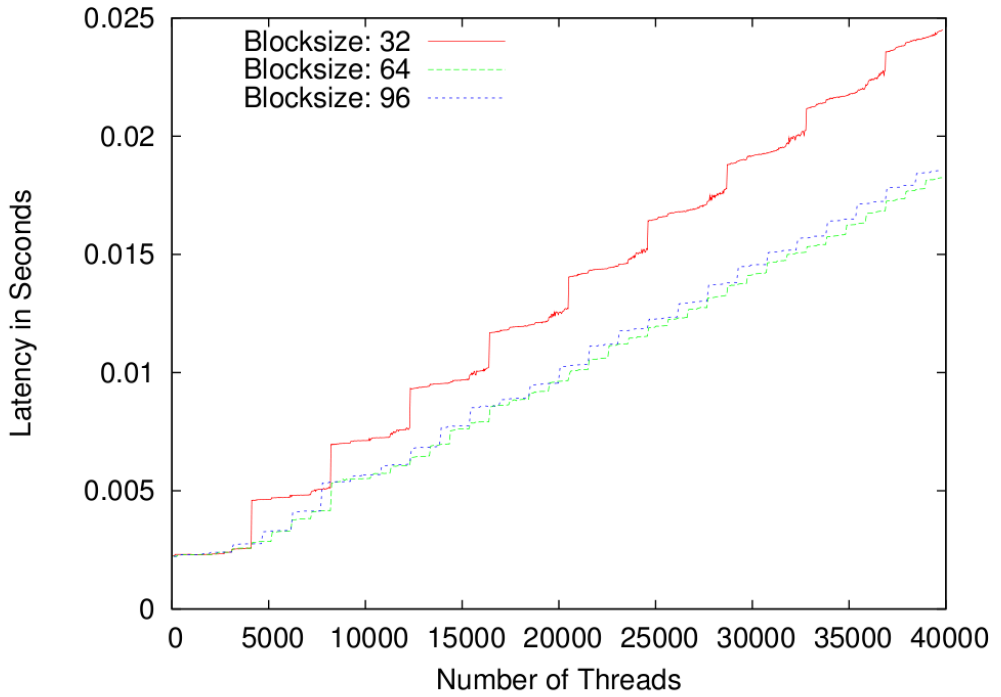

Figure 3.2: Latency results when varying the amount of dispatched threads for blocksize equal to 32 (red, top line), 64 (green, bottom line) and 96 (blue, middle line) on the GTX 580 GPU.

half of the available register space.

3.6.2 Results on Various GPUs

Table 3.3 states the performance results, using the approach from Table 3.2, when using our GPU implementation running on a variety of GPUs from both the older and newer generations of CUDA architectures. Our benchmark results include transferring the in- and output and allows to use different multipliers and elliptic curve points in the same batch. To be compatible with as many settings used in practice as possible, it is not assumed that the initial elliptic curve point is given in affine coordinates but instead in projective coordinates. This has a performance disadvantage: the amount of data which needs to be transferred from the host to the GPU is increased. While primarily designed for the GTX 400 (and newer) family, the bold entries GTX 465, 480 and 580 in Table 3.3, the performance on the older GTX 200 series in terms of latency and throughput are remarkably good.

Our fastest result is obtained on the GTX 580 GPU when computing a single 224-bit elliptic curve scalar multiplication and requires 1.94 milliseconds when dispatching eight threads. This is an order of magnitude faster, in term of response time, compared to the previous fastest low-latency implementation [4]. Figure 3.2 shows the latencies when varying

the amount of threads, eight threads are scheduled to work on a single ECSM, for different block-sizes on the GTX 580. There is a clear trade-off: increasing the block-size allows to hide the various latencies by performing a context switch and calculating on a different group of threads within the same block. On the other hand, every eight threads require their own memory and registers for the intermediate values as outlined in Table 3.2. Increasing the block size too much results in a performance degradation because the required memory for the entire block does not fit in the shared memory any more. As can be observed from Figure 3.2 a block-size of 64 results in the optimal practical performance when processing larger batches of ECSM computations.

To illustrate the computational power of the GPU even further let us consider the throughput when fixing the latency to 5 milliseconds. As can be seen from Figure 3.2, the GTX 580 can compute 916, 1024, or 960 224-bit elliptic curve scalar multiplications within this time limit when using a block-size of 32, 64, or 96 threads respectively. The best of these short runs already achieves a throughput of over 246 000 scalar multiplications per second, when using a blocksize of 64, which is already 0.85 of the maximum observed throughput obtained when processing much larger batches.

Performance Comparison

In [18] and the follow-up work [17] fast elliptic curve scalar multiplication is implemented using Edwards curves in a cryptanalytic setting. The GPU implementations optimize for high-throughput and implement generic modular arithmetic. The setting considered in [17,18] requires to perform an ECSM with a 11 797-bit scalar; in order to compare results we scale their figures by a factor $\frac{11\,797}{224}$. Comparing to these implementations is difficult because both the finite field and elliptic curve arithmetic differ from the approaches considered in this paper where the faster arithmetic on Edwards curves cannot be used. On the GTX 295 architecture, for which our algorithms are not designed, the throughput reported in [17] is 3.3 higher. The associated latency times are not reported.

The GPU implementations discussed in [4, 193] target the same special modulus as discussed in this paper. In [193] one thread per multiplication is used (optimizing for high-throughput) and multiplies the same elliptic curve point in all threads. This reduces the amount of data which needs to be transferred from the host machine to the GPU. The authors of [4] implement a low-latency algorithm by parallelizing the finite field arithmetic using RNS (see Section 3.4). Their performance data do not include the transfer time of the input (output) from the host (GPU) to the GPU (host). Both the GTX 285 and 295 belong to the same GPU family, the former is clocked 1.2 faster than the latter while the GTX 295 consists of two of these slower GPUs. Compared to [4] our minimum latency is more than twice lower while the maximum throughput on a *single* slower GPU of the GTX 295 is almost quadrupled.

3.7 Conclusions

In this chapter we presented techniques to efficiently implement modular multiplication algorithms to SIMD architectures (such as the Cell or GPUs). We considered Montgomery multiplication and various special reduction routines which are of interest for elliptic curve cryptography. The modular multiplication implementations, which use these faster reduction schemes, are at least 1.5 times faster compared to general purpose Montgomery multiplication for the same bitsize. The performance results of our multi-stream modular multiplication implementations for the synergistic processing elements of the Cell broadband engine architecture set new performance records for moduli of bit-length in the range $[192, 521]$ on this platform. These high-performing modular multiplication, generic or special, implementations can be used to speed up public-key cryptography; e.g. in batch elliptic curve decryption.

For the GPU platform we presented an algorithm which is particularly well-suited for parallel computer architectures to compute the scalar multiplication of an elliptic curve point; lowering the latency compared to the straight forward setting where each thread computes a separate scalar multiplication. When applied to a 224-bit standardized elliptic curve used in cryptography and computing with seven threads per elliptic curve scalar multiplication on a GTX 580 graphics processing unit the minimum time required is 1.9 milliseconds; improving on previous low-latency results by an order of magnitude. The latency could be reduced even further when computing both the elliptic curve and the finite field arithmetic concurrently.

Chapter 4

Pollard Rho Using the Negation Map

The difficulty of the *elliptic curve discrete logarithm problem* (ECDLP) underlies the security of cryptographic schemes based on elliptic curves over finite fields [124,143]. The best method known to solve ECDLP for curves without special properties is the parallelized [200] Pollard rho method [166]. A common optimization is to halve the search space by identifying a point with its inverse [73, 86, 204]. Because representatives for the equivalence classes can quickly be computed using the *negation map*, this equivalence relation may result in a speedup by a factor of up to $\sqrt{2}$ when solving the ECDLP. For the elliptic curves over binary extension fields $\mathbf{F}_{2^t}$ from [125], order t equivalence relations can be used as well, resulting in a speedup by a factor of up to $\sqrt{2t}$ [86, 204].

Usage of the negation map in the context of the Pollard rho method leads to *fruitless cycles*, useless cycles trapping the random walks. An analysis of their likelihood of occurrence appeared in [73]. Various methods have been proposed [86,204] to deal with them, all leading to costlier random walks and administrative overhead. The literature suggests that the resulting inefficiencies are negligible, and that a speedup by a factor of $\sqrt{2}$ is attainable [5, Section 19.5.5].

We analyze fruitless cycles and the previously published methods to avoid their ill effects and show that current approaches to escape from cycles suffer from *recurring cycles*. These may have contributed to the lack of practical usage of the negation map to solve prime field ECDLPs: it was not used for the solutions [55,99] of the 79-, 89-, 97- and 109-bit prime field Certicom challenges [54]. Neither was it used by the independent current 112-bit prime field record [36] (see Chapter 5).

We present and analyze alternative methods to deal with fruitless cycles. All our analyses are supported by experiments. We found that the negation map indeed leads to a speedup, but we have not been able to reach more than a factor of 1.29, somewhat short of the $\sqrt{2}$ that we had hoped for. We also found that the best attainable speedup depends on the platform one uses: for instance, if the Pollard rho method is parallelized in SIMD fashion, then it

is a challenge to achieve any speedup at all. This has consequences for the applicability of the negation map in large scale prime field ECDLP solution attempts. For such efforts, all participating processors must use the same random walk definition, so one may desire to gear the implementation towards processors with the best performance/price ratio, such as graphics cards.

The negation map (while dealing with cycles) slows down random walks in three ways. In the first place, on average more elliptic curve group operations are required per step of each walk. This is unavoidable and attempts should be made to minimize the number of additional operations. Secondly, dealing with cycles entails administrative overhead and branching, which cause a non-negligible slowdown when running multiple walks in SIMD-parallel fashion. Finally, the best way to counter the effect of the higher average number of group operations per step is making the walks "more random" by allowing a finer grained decision per step. However, the beneficial effects of this approach are, in most circumstances on current processors, wiped out by cache inefficiencies. It will be seen that it is best to strike a balance between the first and third of these slowdowns. The second slowdown somewhat affects regular PCs, but is a major obstacle to the negation map in SIMD environments.

This chapter is based on the article [38] and the journal version of this work [35].

4.1 r-Adding and $r+s$-Mixed Walks

Let p be a prime > 3, $a, b \in \mathbf{F}_p$ and $\mathfrak{g} \in E_{a,b}(\mathbf{F}_p)$ of prime order q be given such that the index $[E_{a,b}(\mathbf{F}_p) : \langle \mathfrak{g} \rangle]$ is small. For $\mathfrak{h} \in \langle \mathfrak{g} \rangle$ the ECDLP is to find an integer m such that $m\mathfrak{g} = \mathfrak{h}$. For curves without special properties, solving ECDLP is believed to require an effort on the order of $\sqrt{q}$.

Pollard's rho method uses an approximation of a truly random walk in $\langle \mathfrak{g} \rangle$. An index function $\ell : \langle \mathfrak{g} \rangle \mapsto [0, r-1]$ is chosen, for some small integer r, such that the ℓ-induced r-partition $\langle \mathfrak{g} \rangle = \cup_{i=0}^{r-1} \mathfrak{G}_i$, where $\mathfrak{G}_i = \{\mathfrak{x} : \mathfrak{x} \in \langle \mathfrak{g} \rangle, \ell(\mathfrak{x}) = i\}$, results in subsets $\mathfrak{G}_i$ of approximately the same cardinality. For random integer multipliers u_i, v_i, *addition constants* $\mathfrak{f}_i = u_i\mathfrak{g} + v_i\mathfrak{h} \in \langle \mathfrak{g} \rangle$ are pre-computed for $0 \leq i < r$, and the starting point of the walk is selected as a random but known multiple of $\mathfrak{g}$. Given a point $\mathfrak{p}$ of the walk calculate $\mathfrak{p} + \mathfrak{f}_{\ell(\mathfrak{p})} \in \langle \mathfrak{g} \rangle$ as the next point. This is called an *r-adding walk*. It is easy to keep track of the integer multipliers $u, v \in \{0, 1, \ldots, q-1\}$ such that $\mathfrak{p} = u\mathfrak{g} + v\mathfrak{h}$.

As shown by the following heuristic analysis from [7, Appendix B], which refines the arguments from [49], the average number of steps for an r-adding walk is somewhat larger than $\sqrt{\frac{\pi q}{2}}$. Let $p_i = \frac{\#\mathfrak{G}_i}{q}$. A point in the walk is said to be of class i if its predecessor upon its first occurrence belongs to $\mathfrak{G}_i$. If the nth point belongs to $\mathfrak{G}_j$ (with probability p_j) and the $(n+1)$st point produces the first collision, the collision point cannot be of class j (this happens with probability p_j), since then the collision would already have occurred in the previous step. Therefore, the conditional probability that the first collision occurs at step

$n+1$ is heuristically assumed to be

$$\frac{n}{q}\left(1-\sum_{j=0}^{r-1}p_j^2\right).$$

With $q'=\frac{q}{1-\sum_{j=0}^{r-1}p_j^2}$ this probability is $\frac{n}{q'}$, so that we get via the same arguments referred to above

$$\sqrt{\frac{\pi q'}{2}}=\sqrt{\frac{\pi q}{2(1-\sum_{j=0}^{r-1}p_j^2)}} \tag{4.1}$$

as a heuristic estimate for the average number of steps until the first collision.

Pollard, in [166], uses $r=3$ with addition constants $\mathfrak{f}_0=\mathfrak{h}$ and $\mathfrak{f}_2=\mathfrak{g}$, but replaces the $i=1$ case by the doubling $2\mathfrak{p}$ as follows

$$\mathfrak{p}_{i+1}=f(\mathfrak{p}_i)=\begin{cases}\mathfrak{f}_0+\mathfrak{p}_i, & \text{if } \mathfrak{p}_i\in\mathfrak{G}_0\\ 2\mathfrak{p}_i, & \text{if } \mathfrak{p}_i\in\mathfrak{G}_1\\ \mathfrak{f}_2+\mathfrak{p}_i, & \text{if } \mathfrak{p}_i\in\mathfrak{G}_2.\end{cases}$$

Although the successive points are not independent, further undermining the arguments in the above heuristics, it was shown in [118] that with high probability a collision occurs in $O(\sqrt{q})$ steps, if the partition is given by a random oracle. Together with the lowerbound result in the "generic algorithms" from [184] this implies that a collision occurs, with high probability, in $\Theta(\sqrt{q})$ steps. Teske, in [196,197] based on the work by Schnorr and Lenstra [176], suggests using larger r-values such as $r=20$. She shows that using random addition constants leads to fewer iterations and better performance on average, in accordance with the heuristics and even if none of the choices does an explicit doubling (as Pollard's $i=1$ case).

Inclusion of doublings leads to $r+s$-*mixed walks*: given a function $\ell:\langle\mathfrak{g}\rangle\mapsto[0,r+s-1]$ that induces an $r+s$-partition of $\langle\mathfrak{g}\rangle$, the next point equals $\mathfrak{p}+\mathfrak{f}_{\ell(\mathfrak{p})}$ if $0\leq\ell(\mathfrak{p})<r$, but $2\mathfrak{p}$ if $\ell(\mathfrak{p})\geq r$. The original walk by Pollard is a 2+1-mixed walk. The above heuristics apply to this case too, if we define the doublings as a single class hit with probability $p_D=\frac{\sum_{i=r}^{r+s-1}\#\mathfrak{G}_i}{q}$ (which should be $\approx\frac{s}{r+s}$). Experiments by Teske show that best performance is achieved when $\frac{1}{4}\leq\frac{s}{r}\leq\frac{1}{2}$ but that mixed walks are not significantly better than r-adding ones unless $r\leq 3$. Our experiments support the heuristics suggesting that the optimal ratio is close to zero (see also Table 4.1).

Per step the occurrence probability of the event $\mathfrak{p}=\mathfrak{f}_i$ (and thus potentially an immediate solution to the discrete logarithm problem) is negligible compared to the probability of a birthday collision. So, if r-adding as opposed to $r+s$-mixed walks are used, the possibility that doublings will occur can safely be ignored, making it efficient to SIMD-parallelize r-adding walks. This is further commented on below and exploited in Section 5.2.

Some types of elliptic curves allow faster variants of r-adding walks. For instance, for so-called Koblitz curves [125] over binary extension fields (which are not covered by our definition in Section 2.4), the Frobenius automorphism of the finite field can be used to define an efficient function ψ on the group of points of the elliptic curve. For instance, defining the successor of point $\mathfrak{p}$ as $\psi^i(\mathfrak{p})+\mathfrak{p}$ allows its quick computation [86].

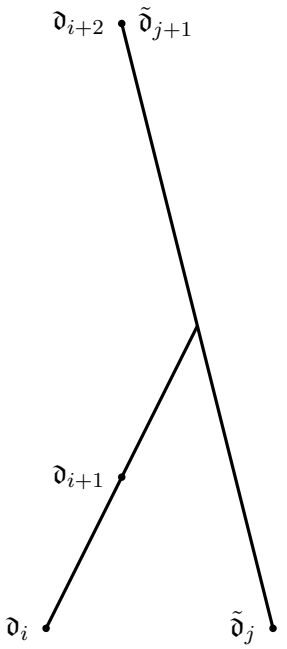

Figure 4.1: Representation of the λ shape of the multi-instance Pollard rho method illustrating when two (out of the many walks running in parallel) walks find the collision (the same distinguished point) $\mathfrak{d}_{i+2} = \tilde{\mathfrak{d}}_{j+1}$. The points $\mathfrak{d}_i, \tilde{\mathfrak{d}}_j$ represent distinguished points from the two different walks. Possibly there are many regular (non-distinguished) points between two subsequent distinguished points.

4.2 Parallelized Random Walks

Parallelization of Pollard's rho method does **not** consist of running random walks in parallel until one of them collides: on M processors the expected speedup would be only a factor of $\sqrt{M}$, so it would overall require $\sqrt{M}$ more processing power than a single processor. The proper way to parallelize Pollard's rho method [200], based on methods from [170, 171], achieves an M-fold speedup on M processors, thus requiring the same overall processing power as a single process in $\frac{1}{M}$th of the time. Different processes must be able to efficiently recognize whether, probably at different points in time, their walks have hit upon the same group element. To achieve this, each process generates a single random walk, each from its own random starting point, but all using the same index function ℓ and the same $\mathfrak{f}_i$'s. As soon as a walk hits upon a *distinguished point*, this point is reported to a central location, along with the corresponding integer multipliers u and v. If the latter would require too much central storage, information to regenerate the starting point should be provided such that, if needed, u and v can be recalculated. The walk may start afresh from a new random starting point, or it may continue. The idea is that as soon as two walks collide – without noticing it – they will keep taking the same steps (because they use the same ℓ and the same $\mathfrak{f}_i$'s) and will thus both ultimately reach the same distinguished point. This will be noticed when the colliding distinguished point is reported to the central location. The discrete logarithm

can then be computed from the two, hopefully distinct, pairs of integer multipliers that correspond to the same distinguished point. The parallel version of the Pollard rho method is often denoted as the Pollard lambda method since two colliding walks resemble the shape of the Greek letter λ (see Figure 4.1). Note that the parallel version of Pollard's rho method is not to be confused with Pollard's kangaroo algorithm [166, 167] (a different algorithm by Pollard to solve the discrete logarithm problem). Both have been called Pollard's lambda method.

A point is distinguished if it has an easily recognizable property that occurs with low enough probability to make it possible to store distinguished points on disk and to efficiently find collisions, but often enough for every walk to hit a distinguished point, eventually. When using distinguished points $O((\log q)^2)$ memory suffices [85, Exercise 16.23] when roughly $\sqrt{q}\log q$ out of q group elements are distinguished. Analysis of the distinguished point property is performed in [178] where the results from [200] are reaffirmed when $\sqrt{q} \ll \frac{q}{2^k} \ll q$; i.e. the distinguished point property should be chosen in such a way that at least one distinguished point is expected in each cycle (in this case one out of every 2^k points is expected to be a distinguished point).

4.3 Unique Point Representation

When using Pollard's rho method, group elements must be represented in a unique way to be able to decide to which partition they belong. When using the parallelized version, uniqueness is also useful to recognize if a point is distinguished. The fastest point representations that we are aware of that are applicable are the affine ones, such as the one in Section 2.4. It requires an inversion in $\mathbf{F}_p$ per group operation, i.e., per step of the walk. The resulting high inversion cost is amortized over many walks running in parallel, as described below.

4.4 Simultaneous Inversion

In the parallelized version of Pollard's rho method, Montgomery's *simultaneous inversion* method from [146] can be used to share the inversion with any number of synchronous but independent walks. Let n be some number of independent walks (typically all running on the same processor), and let $z_i \in \mathbf{F}_p^*$ denote the element that needs to be inverted for the computation of λ in the ith walk (with λ as in Section 2.4). With $w_0 = 1$, first combine the z_i's by calculating $w_i = z_i w_{i-1} \in \mathbf{F}_p^*$ for $i = 1, 2, \ldots, n$, then calculate $\bar{w} = w_n^{-1}$, and finally unravel the results: for $i = n, n-1, \ldots, 1$ in succession calculate $z_i^{-1} = \bar{w} w_{i-1}$ and replace $\bar{w}$ by $z_i \bar{w} = w_{i-1}^{-1}$. Avoiding useless multiplications, the cost $n\mathtt{I}$ of n inversions can thus be replaced by $3(n-1)\mathtt{M} + \mathtt{I}$. For relevant sizes of p it is safe to assume that $\mathtt{I}$ is much larger than $\mathtt{M}$, i.e., at least $\mathtt{I} > 5\mathtt{M}$ when using software (in hardware the difference can be made smaller [114]). For Pollard's rho method it leads to an amortized cost of about $6\mathtt{A} + \frac{1}{n}\mathtt{I} + 5\mathtt{M} + \mathtt{S}$ per step per walk. This makes affine Weierstrass coordinates the least costly point representation for this type of application, if n can be chosen sufficiently large.

Table 4.1: Number of steps required by Pollard's rho method in random elliptic curve groups of 32-bit prime order q over fields of random 32-bit prime cardinality p, divided by $\sqrt{\pi q/2}$ or by $\sqrt{\pi q/4}$ (without or with the negation map). Lowest and highest averages are over 10 measurements. Each measurement calculates the average number of steps taken until a collision occurs, over 100 000 collision searches where for each search a prime p and an elliptic curve over $\mathbf{F}_p$ are randomly selected until the order q of the group of points is prime. Overall average is the average of the 10 averages (thus, the average over one million searches). Expression (4.1) and (4.2) columns are the quotients as expected based on expressions (4.1) (with $p_i = \frac{1}{r}$ for $0 \leq i < r$) and (4.2) (with $p_i = \frac{1}{r+s}$ for $0 \leq i < r$ and $p_D = \frac{s}{r+s}$), respectively. Those expressions are for $q \to \infty$ and indeed for larger (smaller) q they give a better (worse) fit.

	Without negation map				With negation map			
	Averages			Expression	Averages			Expression
	lowest	overall	highest	(4.1)	lowest	overall	highest	(4.2)
8-adding	1.080	1.083	1.086	1.069	1.034	1.038	1.041	1.033
16-adding	1.034	1.036	1.039	1.033	1.013	1.016	1.019	1.016
32-adding	1.012	1.015	1.020	1.016	1.007	1.008	1.010	1.008
16 + 4-mixed	1.042	1.044	1.047	1.043	1.035	1.038	1.040	1.031
16 + 8-mixed	1.074	1.077	1.081	1.078	1.074	1.076	1.078	1.069

The disadvantage is, however, that the group operations are non-uniform: i.e. the addition and doubling are different operations. For SIMD implementation of two or more walks, this means that a regular addition step in one walk cannot be executed simultaneously with a doubling step in another walk. For regular r-adding walks this is not a problem because, as argued above, doubling steps will most likely not occur. Also, excluding $r + s$-mixed walks in a SIMD environment is not a big issue since such walks are not advantageous anyhow (in SIMD, threads could be regrouped to separate regular addition from doubling steps, but this may lead to considerable overhead). More importantly, it makes it harder to profit from the negation map, an optimization discussed in Section 4.5, in a SIMD environment, so elliptic curve parameterizations that allow identical addition and doubling operations remain relevant. Note that the one from [74] (see [20] and a series of follow-up papers) does not lead to a speedup if $\#E_{a,b}(\mathbf{F}_p)$ is prime, as in our case.

4.5 Using Automorphisms

Following [204], define an equivalence relation $\sim$ on $\langle \mathfrak{g} \rangle$ by $\mathfrak{p} \sim -\mathfrak{p}$ for $\mathfrak{p} \in \langle \mathfrak{g} \rangle$. Instead of searching $\langle \mathfrak{g} \rangle$ of size q, search $\langle \mathfrak{g} \rangle / \sim$ of size about $\frac{q}{2}$, where the equivalence class containing $\mathfrak{p}$ and $-\mathfrak{p}$ is represented by, for instance, the element with y-coordinate of least absolute value. Thus, using this *negation map* one would expect to save a factor of $\sqrt{2}$ in the number of iterations, at the cost of finding the representative after each step. The latter is fast since $-(x, y) = (x, -y)$ for $(x, y) \in \langle \mathfrak{g} \rangle$. Obviously, if $-\mathfrak{p}$ instead of $\mathfrak{p}$ is the representative, the integer multipliers u, v with $\mathfrak{p} = u\mathfrak{g} + v\mathfrak{h}$ must be replaced by $-u, -v$.

Adapting the earlier r-adding walk heuristics, it follows that for r-adding (or $r+s$-mixed)

walks the speedup by a factor of $\sqrt{2}$ that is generally reported in the literature is slightly too pessimistic. Let the definitions of p_i, p_D, and of class i be as in Section 4.2. Assume that the nth point belongs to $\mathfrak{G}_j$ and that the $(n+1)$st point produces the first collision while hitting the representative $\mathfrak{p}$, either directly or after negation. If this step is a doubling then the same heuristics as in Section 4.2 applies. This happens with probability p_D^2. Otherwise, we only exclude the case that as a result of just the addition the two predecessors hit the same point ($\mathfrak{p}$ or $-\mathfrak{p}$). This happens with probability $\frac{p_j^2}{2}$. Therefore, the conditional probability that the first collision occurs at step $n+1$ is heuristically assumed to be

$$\frac{2n}{q}\left(1 - p_D^2 - \sum_{j=0}^{r-1}\frac{p_j^2}{2}\right).$$

As above we get

$$\sqrt{\frac{\pi q}{4(1 - p_D^2 - \frac{1}{2}\sum_{j=0}^{r-1} p_j^2)}} \tag{4.2}$$

for the heuristically expected number of steps until the first collision. For the same parameter values this is more than a factor of $\sqrt{2}$ smaller than Expression (4.1).

Practical application of the negation map is complicated by *fruitless cycles*, as pointed out in [86, 204]. This is further discussed in Section 4.7. The group $\langle\mathfrak{g}\rangle$ may admit other trivially computable maps. For instance, for Koblitz curves the Frobenius automorphism of a degree-t binary extension field leads to a further $\sqrt{t}$-fold speedup [73, 86, 204]. This does not apply to the case considered in this article.

Small Scale Experimental Verification

For 32-bit primes q we checked the accuracy of the predictions based on expressions (4.1) and (4.2) and list the results in Table 4.1. With all averages larger than 1, both r-adding and $r+s$-mixed walks on average perform worse than truly random walks. For most walks with the negation map the averages are lower than their negation-less counterparts, indicating that the reduction factor in the expected number of steps is indeed larger than $\sqrt{2}$. This does not imply a speedup by the same factor, because to obtain the figures costly fruitless cycle detection methods had to be used. It can be seen that $r+s$-mixed walks are disadvantageous if $s > \frac{r}{4}$.

4.6 Tag-Tracing

Introduced in [57] to speed up r-adding walks, the idea of *tag-tracing* is that, given the low probability to hit a distinguished point, for most iterations a partial computation suffices. Given $\mathfrak{p}$ with $\ell(\mathfrak{p}) = i$ there is no need to fully calculate the next point $\mathfrak{q} = \mathfrak{p} + \mathfrak{f}_i$, unless it is a distinguished point, as long as there is enough information to compute $k = \ell(\mathfrak{q})$ in order to calculate $\mathfrak{q}$'s successor $\mathfrak{q} + \mathfrak{f}_k$. If a table containing the points $\mathfrak{f}_{ik} = \mathfrak{f}_i + \mathfrak{f}_k$ has been

precomputed, it would then suffice to fully compute $\mathfrak{q}$'s successor as $\mathfrak{p} + \mathfrak{f}_{ik}$. Or, better, by taking the largest τ that allows storage of the table containing the

$$\sum_{k=1}^{\tau} \binom{r+k-1}{k} = \binom{r+\tau}{\tau} - 1$$

sums over at most τ elements from $\{\mathfrak{f}_i : 0 \le i < r\}$, the same observation applies to $\mathfrak{p}$'s partially calculated first $\tau - 1$ successors, only fully calculating again its τth successor. The first partially calculated intermediate point that could be a distinguished point is fully calculated.

For discrete logarithms in multiplicative groups of finite fields, the group operation is modular multiplication. The partial calculation given in [57] suffices to recognize properly defined distinguished points and partition properties and leads to a tenfold speedup for 1024-bit prime fields. Generalization to ECDLP was left open.

ECDLP Tag-Tracing

The more complicated group operation in $\langle \mathfrak{g} \rangle$ makes it harder to apply the same idea to ECDLP. If only the x-coordinate is used for distinguishing and partition properties, calculation of the y-coordinate can be avoided, reducing the average cost per step by $\frac{\tau-1}{\tau}(2\mathtt{A} + \mathtt{M})$. Combined with simultaneous inversion, this leads to a speedup by a factor of approximately $\frac{6}{5}$ (we refer to this as ECDLP tag-tracing) at best (i.e., for large τ), but this comes at various disadvantages that, depending on the circumstances, may invalidate the speedup entirely.

Although initialization cost of the table can be ignored, the cost of retrieving its entries will grow with τ due to memory access latencies. In practice this implies that τ will be of moderate size, thereby lowering the computational speedup that would ideally be achievable. Slight improvements can be obtained by not storing rarely accessed entries (taking an infrequently occurring more costly step instead): for instance, the table entry corresponding to $\mathfrak{f}_0 + \mathfrak{f}_1 + \mathfrak{f}_2$ will be accessed six times as often as the one for $3\mathfrak{f}_0$.

ECDLP tag-tracing as proposed above is incompatible with the negation map, because the latter needs the y-coordinate that may not be computed while tag-tracing. One may conclude that usage of tag-tracing in most circumstances leads to a slow-down by a factor of $\frac{5}{6}\sqrt{2}$: only if r must be small (caches or very little memory) and occasional doubling is best avoided (SIMD) is it conceivable that the negation map is ineffective and that ECDLP tag-tracing (with small τ) gives a small speedup. We could have, but did not attempt to use ECDLP tag-tracing.

4.7 Fruitless Cycles

Straightforward application of the negation map to Pollard's rho method with r-adding or $r + s$-mixed walks does not work due to fruitless cycles. This section describes the current state-of-the-art of dealing with those cycles.

Length 2 Cycles

If a random walk step goes from $\mathfrak{p}$ to $-\mathfrak{p}-\mathfrak{f}_i$ (with probability $\frac{1}{2}$, for some i) and $-\mathfrak{p}-\mathfrak{f}_i \in \mathfrak{G}_i$ (with probability $\frac{1}{r}$), then the next point after $-\mathfrak{p}-\mathfrak{f}_i$ is $\mathfrak{p}$ again (with probability 1), thereby cancelling the effect of the previous step. It follows that a fruitless 2-cycle starts from a random point with probability $\frac{1}{2r}$, cf. [73, Proposition 31]. This 2-cycle is denoted as

$$\mathfrak{p} \xrightarrow{(i,-)} -(\mathfrak{p}+\mathfrak{f}_i) \xrightarrow{(i,-)} \mathfrak{p}.$$

Here "(i,s)" with $s \in \{-,+\}$ indicates that addition constant $\mathfrak{f}_i$ is added to a point $\mathfrak{p}$ after which the result is left as is ($s=+$) or negated ($s=-$) to find the correct representative ($\mathfrak{p}+\mathfrak{f}_i$ if $s=+$, or $-\mathfrak{p}-\mathfrak{f}_i$ if $s=-$). Any walk with two consecutive steps "$(i,-)$" is trapped in an infinite loop. Because this happens with probability $\frac{1}{2r}$, all walks can be expected to end up in fruitless cycles after a moderate number of steps when the negation map is used with r-adding walks.

Looking Ahead to Reduce 2-cycles

To reduce the occurrence of 2-cycles, Wiener and Zuccherato propose to use a more costly iteration function that results in a lower probability that two successive points belong to the same partition [204]. This can be achieved by using the first i of $\ell(\mathfrak{p})$, $\ell(\mathfrak{p})+1$, …, $\ell(\mathfrak{p})+r-1$ such that $i \bmod r \neq \ell(\sim(\mathfrak{p}+\mathfrak{f}_i))$, if such an index exists (here and in the sequel indices i in $\mathfrak{f}_i$ are understood to be taken modulo r). Thus, define the next point as $f(\mathfrak{p})$ with $f : \langle\mathfrak{g}\rangle \to \langle\mathfrak{g}\rangle$ defined by

$$f(\mathfrak{p}) = \begin{cases} E(\mathfrak{p}) & \text{if } j = \ell(\sim(\mathfrak{p}+\mathfrak{f}_j)) \text{ for } 0 \leq j < r \\ \sim(\mathfrak{p}+\mathfrak{f}_i) & \text{with } i \geq \ell(\mathfrak{p}) \text{ minimal s.t. } \ell(\sim(\mathfrak{p}+\mathfrak{f}_i)) \neq i \bmod r. \end{cases}$$

The function $E : \langle\mathfrak{g}\rangle \to \langle\mathfrak{g}\rangle$ may restart the walk at a new random initial point. The latter is expected to happen once every r^r steps and will therefore not affect the efficiency. The expected cost per step of the walk is increased by a factor of $\sum_{i=0}^{r} \frac{1}{r^i}$, which lies between $1+\frac{1}{r}$ and $1+\frac{1}{r-1}$.

Dealing with Fruitless Cycles in General

Although the look-ahead technique reduces the frequency of 2-cycles, they may still occur [204]. This is elaborated upon in Section 4.8. Even so, it is well known that just addressing 2-cycles does not solve the problem of fruitless cycles, because longer cycles will occur as well. Reducing their occurrence requires additional overhead on top of what is already incurred to reduce 2-cycles. Given that fruitless cycles are unavoidable, they must be effectively dealt with when they occur.

In [86] a general approach is proposed to detect cycles and to escape from them: after α steps record a length β sequence of successive points and compare the next point to these β points. If a cycle is detected a cycle representative $\mathfrak{p}$ is chosen deterministically from which

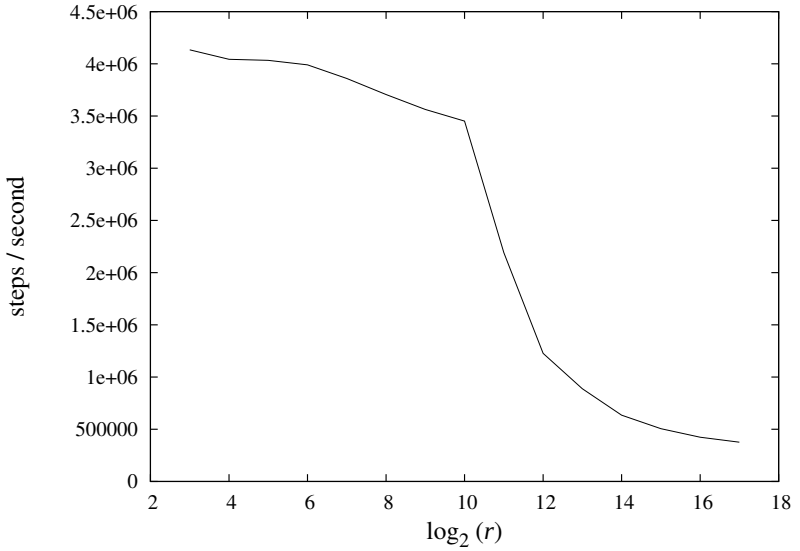

Figure 4.2: Total number of steps per second as a function of r, taken by 200 parallel r-adding walks sharing the modular inversion and not using the negation map, for Pollard's rho method applied to a 131-bit prime ECDLP.

the cycle is escaped. One may add $\mathfrak{f}_{\ell(\mathfrak{p})+c}$ for a fixed $c \in [2, r-1]$ (the choice $c = 1$ is bad as it could lead to an immediate cycle recurrence). Instead one may add a distinct precomputed value $\mathfrak{f}'$ that does not depend on the escape-point, or one may add $\mathfrak{f}''_{\ell(\mathfrak{p})}$ from a distinct list of r precomputed values $\mathfrak{f}''_0, \mathfrak{f}''_1, \ldots, \mathfrak{f}''_{r-1}$.

In the next section we discuss fruitless cycles in greater detail and propose alternative methods that avoid problems that the method from [86] may run into.

4.8 Improved Fruitless Cycle Handling

The probability to enter a fruitless cycle decreases with increasing r [73]. This does not imply that it suffices to take r large enough to make the probability sufficiently low. Figure 4.2 depicts the effect of increasing r-values on the performance of an r-adding walk, measured as number of steps per second. The performance deterioration can be attributed to the increasing rate of cache misses during retrieval of the addition constants $\mathfrak{f}_i$. The effect varies between processors, implementations, and elliptic curves. It is worsened for more contrived walks, such as those using the negation map where cycle reduction, detection and escape methods are unavoidable. Unless the expected overall number of steps (of order $\sqrt{q}$) is too small to be of interest, r cannot be chosen large enough to both avoid fruitless cycles and achieve adequate performance. Therefore, in this section we concentrate on other ways to deal with fruitless cycles. We first discuss short-cycle reduction techniques, next discuss cycle detection methods and analyze their behavior, and finally propose alternative methods.

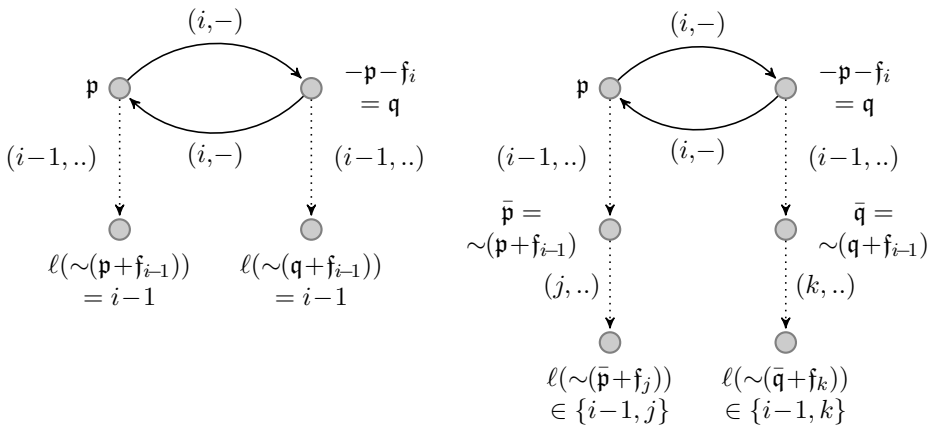

Figure 4.3: 2-cycles caused by 2-cycle reduction (left) and 4-cycle reduction. The dotted steps are prevented.

4.8.1 Short Fruitless Cycle Reduction

2-cycles

Unfortunately, the look-ahead technique to reduce 2-cycles presented above introduces new 2-cycles. The dotted lines in the left example in Figure 4.3 are the steps taken by the regular iteration function, the new cycle is depicted by the solid lines which are the steps taken as a result of $f(\mathfrak{p})$ and $f(\mathfrak{q})$. This new cycle occurs with probability $\frac{1}{2r^3}$. It is the most likely 2-cycle introduced by the look-ahead technique.

Lemma 4.1. *The probability to enter a fruitless 2-cycle when looking ahead to reduce 2-cycles while using an r-adding walk is*

$$\frac{1}{2r}\left(\sum_{i=1}^{r-1}\frac{1}{r^i}\right)^2 = \frac{(r^{r-1}-1)^2}{2r^{2r-1}(r-1)^2} = \frac{1}{2r^3} + O\left(\frac{1}{r^4}\right).$$

Proof. With i as in the definition of f, the probability is r^{-c} that $i \geq \ell(\mathfrak{p}) + c$ for $0 \leq c < r$ (considering the case $E(\mathfrak{p})$ as $i = \infty$), hence $i = \ell(\mathfrak{p}) + c$ with probability $\frac{r-1}{r}\frac{1}{r^c}$.

We compute the probability of entering a cycle consisting of points $\mathfrak{p}$ and $\mathfrak{q}$ starting at $\mathfrak{p}$. Let $j = \ell(\mathfrak{p})$ and $k = \ell(\mathfrak{q})$, and let the steps from $\mathfrak{p}$ to $\mathfrak{q}$ and back be adding $\mathfrak{f}_{j+c}$ and $\mathfrak{f}_{k+d}$, respectively. This implies that $j + c \equiv k + d \bmod r$ and that the step from $\mathfrak{p}$ to $\mathfrak{q}$ involves a negation. From the definition of f it follows that $\ell(\mathfrak{q}) \not\equiv j + c \bmod r$, thus $d \neq 0$ and by symmetry $c \neq 0$. Since j is given and k is determined by j, c and d, the probabilities must be summed over all possible c and d. The probability for a c, d pair is the product of the following probabilities:

- $\frac{r-1}{r}\frac{1}{r^c}$ for the first step being c;
- $\frac{1}{2}$ for the sign;

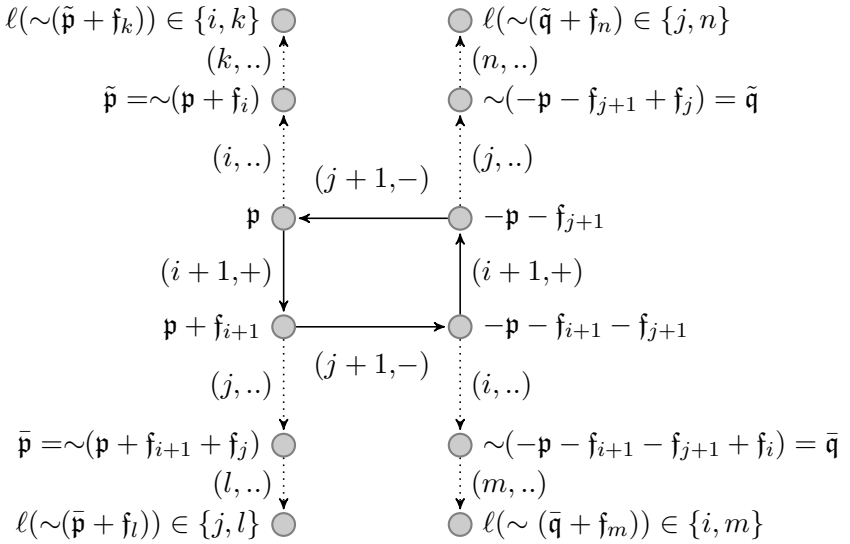

Figure 4.4: A 4-cycle when the 4-cycle reduction method is used.

- $\frac{1}{r-1}$ for $\ell(\sim(\mathfrak{p}+\mathfrak{f}_{j+c}))=k$
  (we know already that $\ell(\sim(\mathfrak{p}+\mathfrak{f}_{j+c}))\not\equiv j+c\not\equiv k \bmod r$);
- $\frac{1}{r^d}$ for the second step being d (since $\ell(\sim(\mathfrak{q}+\mathfrak{f}_{k+d}))\not\equiv k+d \bmod r$).

This results in the probability $\dfrac{1}{2r}\displaystyle\sum_{c=1}^{r-1}\sum_{d=1}^{r-1}\frac{1}{r^c}\frac{1}{r^d}$. □

We conclude that, even when the look-ahead technique is used, 2-cycles are still too likely to occur for relevant values of q and r. Some of the new 2-cycles are prevented by other short-cycle reduction methods, but the remaining ones must be dealt with using detection and escape methods. This is discussed below.

4-cycles

Unless the addition constants $\mathfrak{f}_i$ have been chosen poorly (e.g. $\mathfrak{f}_i=\mathfrak{f}_j+\mathfrak{f}_k$), 3-cycles do not occur as a direct result of the negation map, so that 4-cycles are the next type of short cycles to be considered. Excluding again that the $\mathfrak{f}_i$ have unlikely properties, a fruitless 4-cycle without proper sub-cycle is of the form

$$\mathfrak{p}\ \xrightarrow{(i,+)}\ \mathfrak{p}+\mathfrak{f}_i\ \xrightarrow{(j,-)}\ -\mathfrak{p}-\mathfrak{f}_i-\mathfrak{f}_j\ \xrightarrow{(i,+)}\ -\mathfrak{p}-\mathfrak{f}_j\ \xrightarrow{(j,-)}\ \mathfrak{p}.$$

The cycle may be entered at any of its four points. Hence, a fruitless 4-cycle starts from a random point with probability $\frac{r-1}{4r^3}$. This is a lower bound for the probability of occurrence of 4-cycles when looking ahead to reduce 2-cycles.

An extension of the 2-cycle reduction method looks ahead to the first two successors of a point, thereby reducing the frequency of 2-cycles and 4-cycles, while still being deterministic:

$$g(\mathfrak{p}) = \begin{cases} E(\mathfrak{p}) & \text{if } j \in \{\ell(\mathfrak{q}), \ell(\sim(\mathfrak{q} + \mathfrak{f}_{\ell(\mathfrak{q})}))\} \text{ or } \ell(\mathfrak{q}) = \ell(\sim(\mathfrak{q} + \mathfrak{f}_{\ell(\mathfrak{q})})) \\ & \text{where } \mathfrak{q} = \sim(\mathfrak{p} + \mathfrak{f}_j), \text{ for } 0 \le j < r, \\ \mathfrak{q} = \sim(\mathfrak{p} + \mathfrak{f}_i) & \text{with } i \ge \ell(\mathfrak{p}) \text{ minimal s.t.} \\ & i \bmod r \ne \ell(\mathfrak{q}) \ne \ell(\sim(\mathfrak{q} + \mathfrak{f}_{\ell(\mathfrak{q})})) \ne i \bmod r. \end{cases}$$

Compared to $f(\mathfrak{p})$, the probability that E is called increases from $(\frac{1}{r})^r$ to at least $(\frac{2}{r})^r$ because $\ell(\sim(\mathfrak{q} + \mathfrak{f}_{\ell(\mathfrak{q})})) \in \{j \bmod r, \ell(\mathfrak{q})\}$ with probability $\frac{2}{r}$ for each j. This iteration function is at least $\frac{r+4}{r}$ times slower than the standard one, because with probability $\frac{2}{r}$ at least two additional group operations need to be carried out, an effect that is slightly alleviated by a factor of $(\frac{r-1}{r})^{\frac{1}{2}}$ since the image of g is a subset of $\langle\mathfrak{g}\rangle$ of cardinality approximately $\frac{r-1}{r}q$. The value $\sim(\mathfrak{q} + \mathfrak{f}_{\ell(\mathfrak{q})})$ can be stored for use in the next iteration. Usage of g reduces the occurrence of 4-cycles, and also prevents some of the 2-cycles newly introduced by the 2-cycle reduction method (such as the one depicted on the left in Figure 4.3). But g introduces new types of 2-cycles and 4-cycles as well, both of which do indeed occur in practice. A newly introduced 2-cycle is shown in the right example in Figure 4.3. There the points $\bar{\mathfrak{p}}$ and $\bar{\mathfrak{q}}$ are $\notin \mathfrak{G}_{i-1} \cup \mathfrak{G}_i$. This 2-cycle occurs with probability $\frac{2(r-2)^2}{(r-1)r^4}$, which is therefore a lower bound for the probability of 2-cycles when using the 4-cycle reduction method. Figure 4.4 depicts an example of a newly introduced 4-cycle: the points reached via dotted lines belong to a partition different from their predecessors. The probability that such a 4-cycle starts from a random point is at least $\frac{4(r-2)^4(r-1)}{r^{11}}$.

We have not been able to design or to find in the literature short-cycle reduction methods that do not introduce other (lower probability) short cycles. We therefore turn our attention to cycle detection and escape methods.

4.8.2 Cycle Detection and Escape

Recurring Cycles

The cycle detection and escape method from [86] described in Section 4.7, does not prevent recurrence to the same cycle. When using $\mathfrak{f}_{\ell(\mathfrak{p})+c}$ to escape (we fixed $c = 4$ as it worked as well as any other choice $\ne 1$), Figure 4.5 depicts how the (wavy) escape from the (solid) 4-cycle recurs to the 4-cycle via one of the dotted possibilities. The probability of recurrence depends on the escape method and on which point in the cycle the walk recurs to. With $\mathfrak{f}_{\ell(\mathfrak{p})+c}$ as escape, immediate recurrence to the escape point happens with probability $\frac{1}{2r}$ when no cycle reduction is used, recurrence happens with probability at least $\frac{1}{2r^2}$ with 2-cycle reduction, and with probability at least $\frac{(r-2)^2}{r^4}$ with 4-cycle and thus 2-cycle reduction. Similar recurrences occur, with lower probabilities, when $\mathfrak{f}'$ or $\mathfrak{f}''_{\ell(\mathfrak{p})}$ are used to escape.

Lemma 4.2. *Lower bounds for the probabilities to enter 2-cycles or 4-cycles or to recur to cycles for three different cycle escape methods are listed in Table 4.2 if no cycle reduction,*

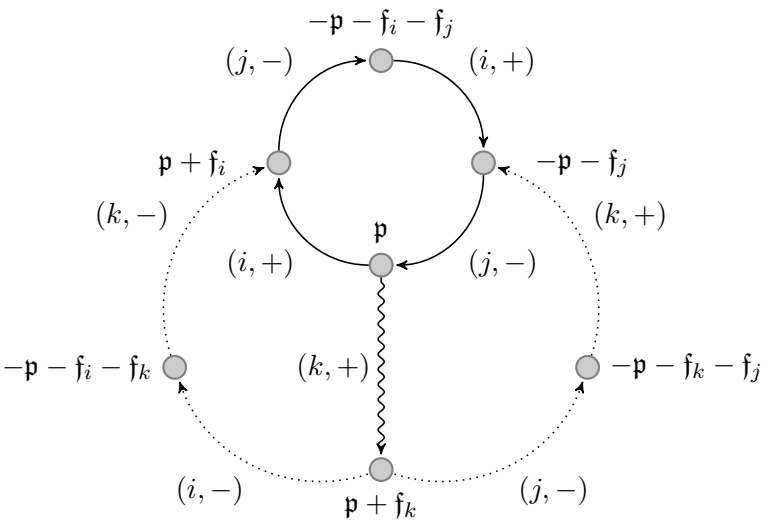

Figure 4.5: Escaping from a fruitless 4-cycle, and recurring to it ($i \neq j \neq k \neq i$).

or 2-cycle reduction (f), or 4-cycle reduction (g) is used, along with a lower bound for the slowdown factor caused by f or g.

Proof. The proofs for many entries of Table 4.2 were given earlier. We prove the entries in rows five and six.

Let $\mathfrak{p}$ be the escape point and let $\mathfrak{q}$ be the point it escapes to. Using $\mathfrak{f}'$ or $\mathfrak{f}''_{\ell(\mathfrak{p})}$ one can recur to the escape point $\mathfrak{p}$ by entering another cycle at $\mathfrak{q}$ and escaping from it at $\mathfrak{q}$ again. This new cycle could be a 2-cycle. For this to happen the first escape step to $\mathfrak{q}$ has to involve a negation (probability $\frac{1}{2}$), a 2-cycle has to be entered at $\mathfrak{q}$ (probabilities in first row, but see below), the escape point of this 2-cycle has to be $\mathfrak{q}$ (probability $\frac{1}{2}$), and, in the case of $\mathfrak{f}''_i$, the partition that $\mathfrak{q}$ belongs to has to be the same as the one $\mathfrak{p}$ belongs to (probability $\frac{1}{r}$). In the case of 4-cycle reduction the probability to enter a 2-cycle at $\mathfrak{q}$ is slightly lower since we do not have the information that $\ell(\sim(\mathfrak{q}+\mathfrak{f}_{\ell(\mathfrak{q})})) \neq \ell(\mathfrak{q})$; a calculation analogous to the one done at the end of Section 4.8.1 produces the values listed in the table. □

6-cycles

With proper $\mathfrak{f}_i$ and no sub-cycle, a common 6-cycle is of the form

$$\mathfrak{p} \xrightarrow{(i,+)} \mathfrak{p}+\mathfrak{f}_i \xrightarrow{(j,-)} -\mathfrak{p}-\mathfrak{f}_i-\mathfrak{f}_j \xrightarrow{(k,+)} -\mathfrak{p}-\mathfrak{f}_i-\mathfrak{f}_j+\mathfrak{f}_k \xrightarrow{(i,+)} -\mathfrak{p}-\mathfrak{f}_j+\mathfrak{f}_k \xrightarrow{(j,-)} \mathfrak{p}-\mathfrak{f}_k \xrightarrow{(k,+)} \mathfrak{p}$$

($i \neq j \neq k \neq i$) where with appropriate sign changes steps four and five may be swapped. It may be entered at any of its six points and occurs, when using 4-cycle reduction, with probability $\frac{1}{4r^3}+O(\frac{1}{r^4})$. A lower bound to recur to it follows by multiplying this probability with the recurring probabilities from Table 4.2.

Table 4.2: Summary of effect of cycle reduction, detection, and escape methods. With the exception of the two bold entries, all figures are lower bounds.

Successor of $\mathfrak{p}$:		$\mathfrak{p}+\mathfrak{f}_{\ell(\mathfrak{p})}$	$f(\mathfrak{p})$	$g(\mathfrak{p})$
Corresponding cycle reduction method:		none	2-cycle	4-cycle
Probability to enter	2-cycle	$\mathbf{\frac{1}{2r}}$	$\frac{1}{2r^3}$	$\frac{2(r-2)^2}{(r-1)r^4}$
	4-cycle	$\mathbf{\frac{r-1}{4r^3}}$	$\frac{r-1}{4r^3}$	$\frac{4(r-2)^4(r-1)}{r^{11}}$
	2ω-cycle for $\omega \in \mathbf{Z}_{>2}$, see [73]	$\Omega(r^{-\omega})$	$\Omega(r^{-\omega})$	$\Omega(r^{-\omega})$
Probability to recur to a cycle after escaping it from $\mathfrak{p}$ to	$\sim(\mathfrak{p}+\mathfrak{f}_{\ell(\mathfrak{p})+c})$	$\frac{1}{2r}$	$\frac{1}{2r^2}$	$\frac{(r-2)^2}{r^4}$
	$\sim(\mathfrak{p}+\mathfrak{f}')$	$\frac{1}{8r}$	$\frac{1}{8r^3}$	$\frac{(r-2)^2}{2r^5}$
	$\sim(\mathfrak{p}+\mathfrak{f}''_{\ell(\mathfrak{p})})$	$\frac{1}{8r^2}$	$\frac{1}{8r^4}$	$\frac{(r-2)^2}{2r^6}$
Slowdown factor of iteration function		n/a	$\frac{r+1}{r}$	$\frac{r+4}{r}$

4.8.3 Alternative Approaches

The purpose of using the negation map is to obtain a speedup, hopefully by a factor of $\sqrt{2}$. From Figure 4.2 it follows that large r-values cannot be used. From Table 4.2 it follows that for small r-values and relevant q-values fruitless cycles are likely to occur and recur. Medium r-values look the most promising, but are not compatible with all environments.

Since fruitless cycle occurrence and recurrence cannot be rooted out, alternative methods are needed if we want to make the negation map useful. In this section several possibilities are offered.

Heuristic 4.1. *A cycle with at least one doubling is most likely not fruitless.*

Proof. Let $\mathfrak{p} = u\mathfrak{g}+v\mathfrak{h}$ be a point on the cycle. The subsequent points are obtained by adding one of the $\mathfrak{f}_i$ or by doubling, and negating if needed, thus are up to sign linear combinations of the $\mathfrak{f}_i$ and a power-of-two multiple of $\mathfrak{p}$. If $c \geq 1$ is the number of doublings in the cycle, we get a relation of the form

$$\mathfrak{p} = \pm 2^c\mathfrak{p} + \sum_{i=0}^{r-1} c_i\mathfrak{f}_i = \pm 2^c\mathfrak{p} + \sum_{i=0}^{r-1} c_i u_i \mathfrak{g} + \sum_{i=0}^{r-1} c_i v_i \mathfrak{h} \qquad \text{and thus}$$

$$\left((1 \mp 2^c)u - \sum_{i=0}^{r-1} c_i u_i\right)\mathfrak{g} + \left((1 \mp 2^c)v - \sum_{i=0}^{r-1} c_i v_i\right)\mathfrak{h} = 0,$$

where $c_i \in \mathbf{Z}$. Since $1 \mp 2^c \neq 0$, the expression $\left((1 \mp 2^c)u - \sum_{i=0}^{r-1} c_i u_i\right)$ is most likely not divisible by the group order. This also holds if $\{\mathfrak{f}_i : 0 \leq i < r\}$ is enlarged with $\mathfrak{f}'$ or with $\{\mathfrak{f}''_i : 0 \leq i < r\}$. This concludes our heuristic argument. □

Cycle Reduction by Doubling

The regular structure required for cycles is caused by repeated addition and subtraction using the same set of constants. This structure would be broken effectively by using an occasional doubling, i.e., a mixed walk. If such walks are used, the heuristics suggest that cycles occur only between two doublings. If the doubling frequency is sufficiently high, only short cycles would have to be dealt with.

As borne out by expressions (4.1) and (4.2) when using the idealized values $p_i = \frac{1}{r+s}$ for $0 \leq i < r$ and $p_D = \frac{s}{r+s}$ for $r > 0$, and as supported by the experiments reported in Table 4.1, an $r + s$-mixed walk with $s > 1$ always displays noticeably less random behavior than a well-partitioned r'-adding walk for any $r' > r$. Nevertheless, using properly tuned $r + s$-mixed walks may be a way to address the cycle problem while avoiding impractically large r-values.

However, $r + s$-mixed walks have disadvantages caused by the underlying arithmetic. Given the relative speeds of addition and doubling, an $r + s$-mixed walk is $\frac{r+7s/6}{r+s}$ times slower than an r-adding walk. In a SIMD environment where many walks are processed simultaneously, per step a fraction of about $\frac{r}{r+s}$ of the walks will do an addition, whereas the others do a doubling. If the addition and doubling code differ, as is the case for the affine Weierstrass representation, the two types of steps cannot be executed simultaneously. Thus, in such environments, to avoid a slowdown by a factor of more than 2 one needs to swap walks to make all parallel step-operations identical (at non-negligible overhead), or one has to settle for a suboptimal affine point representation that allows identical code. SIMD-application of the negation map and the possibility of another point representation are subjects for further study (see Section 4.11).

Doubling Based Cycle Reduction and Escape

Taking into account that doubling should not be used too frequently, usage could be limited to cycle reduction or escape. This would not solve the SIMD-issue, but the relative inefficiency and non-randomness would be addressed. If doublings are used to escape from fruitless cycles, they would not recur, as that would contradict the heuristics. Cycle reduction using doubling replaces $f(\mathfrak{p})$ and $g(\mathfrak{p})$ by $\bar{f}(\mathfrak{p})$ and $\bar{g}(\mathfrak{p})$, respectively, where

$$\bar{f}(\mathfrak{p}) = \begin{cases} \sim(\mathfrak{p} + \mathfrak{f}_{\ell(\mathfrak{p})}) & \text{if } \ell(\mathfrak{p}) \neq \ell(\sim(\mathfrak{p} + \mathfrak{f}_{\ell(\mathfrak{p})})), \\ \sim(2\mathfrak{p}) & \text{otherwise,} \end{cases}$$

$$\bar{g}(\mathfrak{p}) = \begin{cases} \mathfrak{q} = \sim(\mathfrak{p} + \mathfrak{f}_{\ell(\mathfrak{p})}) & \text{if } \ell(\mathfrak{q}) \neq \ell(\mathfrak{p}) \neq \ell(\sim(\mathfrak{q} + \mathfrak{f}_{\ell(\mathfrak{q})})) \neq \ell(\mathfrak{q}), \\ \sim(2\mathfrak{p}) & \text{otherwise.} \end{cases}$$

It follows from the heuristics that these functions avoid recurring fruitless cycles.

Alternative Cycle Detection

Because shorter cycles are more frequent, a potentially interesting modification of the cycle detection method from [86] (described at the end of Section 4.7) would be to occasionally

compare a point to its kth successor, where k is the least common multiple of all even short cycle lengths that one wants to catch. Detecting, for instance, cycles up to length 12 requires only $\frac{1}{120}$th comparison per step. This can be done in several steps, recording every 12th point to catch 4- and 6-cycles, recording every 10th of these recorded points to catch 8- and 10-cycles, etc. It can be combined with the regular method with large α and β to catch longer cycles infrequently.

However, if a cycle has been detected the k points need to be recorded as before, so an escape point can be chosen deterministically. This argues against using large k. It also suggests that an improvement can be expected only if cycles occur with low probability, and therefore that the improvement will be marginal at best (cf. α and β choices in Section 4.9). For this reason we did not conduct extensive experiments with this method.

4.9 Comparison

We implemented and compared on a traditional non-SIMD platform all previously published and newly proposed methods to deal with fruitless cycles when using the negation map. Here we report on our findings. It quickly turned out that the cycle detection methods from [86] when combined with doubling based cycle reduction and escape, are considerably more efficient than $r + s$-mixed walks with their on average slower steps and less random behavior. Mixed walks are therefore not further discussed. Experiments with the alternative cycle detection method were quickly abandoned as well.

For each combination of iteration function, escape method, and r-value a search was conducted to determine the α and β to be used for the cycle detection method from [86]. Using a heuristic argument that for $\beta = 2k$ with k much smaller than r, cycles of length $\geq \beta$ occur with probability on the order of $\frac{(k-1)!}{(2r)^k}$, values for k that make this probability low enough resulted in good initial values for the search for close to optimal α and β. To give some examples (this notation is explained in more detail later in the section), for "f, e," (2-cycle reduction and escape by adding $\mathfrak{f}_{\ell(\mathfrak{p})+4}$) we used $\alpha = 31$ and $\beta = 20$ for $r = 16$, $\alpha = 3264$ and $\beta = 12$ for $r = 128$, and $\alpha = 52\,418$ and $\beta = 10$ for $r = 256$. For "$\bar{f}$, $\bar{\mathrm{e}}$" (2-cycle reduction using doubling and escape by doubling) and the same r-values we used the same β-values but replaced the α-values by 1 618, 838 848, and 53 687 081, respectively.

Each of the benchmarks presented in Table 4.3 was run on a single core of an AMD Phenom 2.2GHz 4-core processor, with each of the four cores processing a different combination. A 10-bit distinguishing property was used to get a significant amount of data in a reasonable amount of time. This somewhat affects the performance, but not the cycle behavior as walks continue after hitting a distinguished point. The figures in millions as given in the table are thus an underestimate for the actual per-core yield in units when a more realistic 30-bit distinguishing property would be used (since $2^{30}/2^{10} = 2^{20} \approx 10^6$).

In order to be able to compare the long term yield figures, the expected number of steps must be taken into account using expressions 4.1 and 4.2. As a result, the yields are corrected by a factor of $(\frac{r-1}{r})^{\frac{1}{2}}$ for the iteration functions that do not use the negation map, and by a factor of $(\frac{2r-1}{r})^{\frac{1}{2}}$ for the others, with an extra factor of $(\frac{r}{r-1})^{\frac{1}{2}}$ for g and $\bar{g}$. After this

Table 4.3: Long term yield when using different cycle reduction and escaping techniques (and an r-adding walk). After the colon (:) the speed-up when using the negation map is presented. The bold entries show the settings with the highest speedup. More detailed information is given in the text.

	$r = 16$	$r = 32$	$r = 64$	$r = 128$	$r = 256$	$r = 512$
Without negation map						
	7.29: 0.98	7.28: 0.99	**7.27**: 1.00	7.19: 0.99	6.97: 0.96	6.78: 0.94
With negation map						
†	0.00: 0.00	0.00: 0.00	0.00: 0.00	0.00: 0.00	0.00: 0.00	0.00: 0.00
just g	0.00: 0.00	0.00: 0.00	0.00: 0.00	0.00: 0.00	0.04: 0.01	3.59: 0.70
just $\bar{g}$	0.00: 0.00	0.00: 0.00	0.00: 0.00	0.75: 0.15	4.90: 0.96	5.90: 1.16
just e''	0.00: 0.00	0.00: 0.00	0.00: 0.00	0.61: 0.12	4.94: 0.97	5.73: 1.12
just $\bar{\mathrm{e}}$	3.34: 0.64	4.89: 0.95	5.85: 1.14	6.10: 1.19	6.28: 1.23	6.18: 1.21
f, e	0.00: 0.00 $\frac{9.4e8}{0.0e0}$ *0.08*	0.00: 0.00 $\frac{6.6e8}{0.0e0}$ *0.48*	1.52: 0.30 $\frac{1.0e8}{0.0e0}$ *1.28*	5.93: 1.16 $\frac{3.6e7}{0.0e0}$ *1.37*	6.47: 1.27 $\frac{2.9e7}{0.0e0}$ *1.38*	6.36: 1.25 $\frac{2.5e7}{0.0e0}$ *1.39*
f, e'	0.00: 0.00 $\frac{3.9e8}{0.0e0}$ *0.86*	3.24: 0.63 $\frac{8.0e7}{0.0e0}$ *1.30*	6.04: 1.18 $\frac{4.6e7}{0.0e0}$ *1.35*	6.41: 1.25 $\frac{3.3e7}{0.0e0}$ *1.38*	6.29: 1.23 $\frac{2.9e7}{0.0e0}$ *1.38*	6.21: 1.22 $\frac{2.6e7}{0.0e0}$ *1.39*
f, e''	0.00: 0.00 $\frac{1.3e8}{0.0e0}$ *1.22*	5.34: 1.04 $\frac{6.0e7}{0.0e0}$ *1.33*	6.21: 1.21 $\frac{4.2e7}{0.0e0}$ *1.36*	6.30: 1.23 $\frac{3.3e7}{0.0e0}$ *1.38*	6.20: 1.21 $\frac{2.9e7}{0.0e0}$ *1.38*	5.99: 1.17 $\frac{2.7e7}{0.0e0}$ *1.39*
f, $\bar{\mathrm{e}}$	3.71: 0.72 $\frac{9.2e7}{9.9e5}$ *1.27*	6.36: 1.24 $\frac{6.8e7}{2.8e5}$ *1.32*	6.50: 1.27 $\frac{4.2e7}{6.5e4}$ *1.36*	6.57: **1.29** $\frac{3.3e7}{1.5e4}$ *1.38*	6.47: 1.27 $\frac{2.9e7}{3.8e3}$ *1.38*	6.30: 1.25 $\frac{2.7e7}{9.7e2}$ *1.39*
g, e	0.00: 0.00 $\frac{8.7e8}{0.0e0}$ *0.19*	0.01: 0.00 $\frac{3.7e8}{0.0e0}$ *0.91*	4.89: 0.96 $\frac{6.6e7}{0.0e0}$ *1.34*	6.22: 1.22 $\frac{4.2e7}{0.0e0}$ *1.37*	6.23: 1.22 $\frac{3.3e7}{0.0e0}$ *1.38*	6.05: 1.19 $\frac{1.3e7}{0.0e0}$ *1.41*
g, e'	0.00: 0.00 $\frac{7.8e8}{0.0e0}$ *0.32*	0.01: 0.00 $\frac{3.0e8}{0.0e0}$ *1.00*	5.32: 1.05 $\frac{6.0e7}{0.0e0}$ *1.35*	6.26: 1.23 $\frac{4.1e7}{0.0e0}$ *1.37*	6.25: 1.23 $\frac{3.0e7}{0.0e0}$ *1.38*	6.11: 1.20 $\frac{5.5e7}{0.0e0}$ *1.35*
g, e''	0.00: 0.00 $\frac{7.6e8}{0.0e0}$ *0.34*	1.09: 0.21 $\frac{1.2e8}{0.0e0}$ *1.27*	5.37: 1.13 $\frac{6.0e7}{0.0e0}$ *1.35*	6.08: 1.20 $\frac{4.2e7}{0.0e0}$ *1.37*	6.06: 1.19 $\frac{3.5e7}{0.0e0}$ *1.38*	5.86: 1.15 $\frac{4.3e7}{0.0e0}$ *1.37*
g, $\bar{\mathrm{e}}$	0.76: 0.15 $\frac{3.3e8}{1.6e5}$ *0.97*	5.91: 1.17 $\frac{1.7e8}{6.0e4}$ *1.19*	6.02: 1.18 $\frac{8.1e7}{8.1e3}$ *1.32*	6.25: 1.23 $\frac{5.4e7}{1.0e3}$ *1.35*	6.13: 1.20 $\frac{4.0e7}{1.2e2}$ *1.37*	6.00: 1.18 $\frac{2.7e7}{9.0e0}$ *1.39*
$\bar{f}$, e	0.00: 0.00 $\frac{8.7e8}{2.4e6}$ *0.18*	0.00: 0.00 $\frac{4.3e8}{1.7e7}$ *0.80*	2.70: 0.53 $\frac{5.4e7}{1.5e7}$ *1.34*	5.96: 1.16 $\frac{1.1e7}{7.7e6}$ *1.41*	6.34: 1.24 $\frac{1.0e7}{3.9e6}$ *1.41*	6.20: 1.21 $\frac{1.4e7}{1.9e6}$ *1.40*
$\bar{f}$, e'	0.01: 0.0 $\frac{2.6e8}{4.3e7}$ *1.03*	4.24: 0.82 $\frac{6.8e7}{2.9e7}$ *1.31*	6.32: 1.23 $\frac{3.9e7}{1.5e7}$ *1.36*	6.43: 1.26 $\frac{3.2e7}{7.6e6}$ *1.38*	6.33: 1.24 $\frac{2.8e7}{3.8e6}$ *1.38*	6.20: 1.22 $\frac{2.7e7}{1.9e6}$ *1.39*
$\bar{f}$, e''	1.34: 0.26 $\frac{8.9e7}{5.2e7}$ *1.27*	5.80: 1.13 $\frac{5.3e7}{2.9e7}$ *1.33*	6.23: 1.22 $\frac{3.9e7}{1.5e7}$ *1.36*	6.21: 1.22 $\frac{3.6e7}{7.5e6}$ *1.37*	6.15: 1.20 $\frac{2.8e7}{3.8e6}$ *1.38*	6.00: 1.18 $\frac{2.6e7}{1.9e6}$ *1.39*
$\bar{f}$, $\bar{\mathrm{e}}$	5.58: 1.06 $\frac{6.1e7}{4.2e7}$ *1.31*	6.14: 1.18 $\frac{3.7e7}{3.0e7}$ *1.36*	6.34: 1.23 $\frac{1.8e7}{1.5e7}$ *1.39*	6.42: 1.25 $\frac{1.1e7}{7.7e6}$ *1.41*	6.27: 1.23 $\frac{1.0e7}{3.9e6}$ *1.41*	6.07: 1.19 $\frac{1.4e7}{1.9e6}$ *1.40*
$\bar{g}$, e	2.56: 0.51 $\frac{1.4e8}{9.9e7}$ *1.23*	5.80: 1.15 $\frac{7.9e7}{5.6e7}$ *1.31*	6.02: 1.18 $\frac{5.1e7}{2.9e7}$ *1.35*	6.09: 1.20 $\frac{4.1e7}{1.5e7}$ *1.37*	6.19: 1.21 $\frac{2.6e7}{7.6e6}$ *1.39*	5.74: 1.13 $\frac{7.7e6}{3.9e6}$ *1.41*
$\bar{g}$, e'	4.74: 0.94 $\frac{1.2e8}{1.0e8}$ *1.25*	5.88: 1.16 $\frac{7.8e7}{5.6e7}$ *1.31*	6.14: 1.21 $\frac{5.3e7}{2.9e7}$ *1.35*	6.28: 1.23 $\frac{3.9e7}{1.5e7}$ *1.37*	6.05: 1.19 $\frac{2.6e7}{7.6e6}$ *1.39*	5.80: 1.14 $\frac{7.7e6}{3.9e6}$ *1.41*
$\bar{g}$, e''	4.72: 0.94 $\frac{1.2e8}{1.0e8}$ *1.25*	5.80: 1.15 $\frac{7.7e7}{5.6e7}$ *1.31*	6.08: 1.20 $\frac{5.3e7}{2.9e7}$ *1.35*	6.05: 1.19 $\frac{3.8e7}{1.5e7}$ *1.37*	5.91: 1.16 $\frac{1.8e7}{7.6e6}$ *1.40*	5.67: 1.11 $\frac{7.7e6}{3.9e6}$ *1.41*
$\bar{g}$, $\bar{\mathrm{e}}$	4.83: 0.96 $\frac{1.2e8}{1.0e8}$ *1.25*	5.87: 1.16 $\frac{7.9e7}{5.6e7}$ *1.31*	6.09: 1.20 $\frac{5.2e7}{2.9e7}$ *1.35*	6.16: 1.21 $\frac{4.0e7}{1.5e7}$ *1.37*	6.09: 1.20 $\frac{2.6e7}{7.6e6}$ *1.39*	5.70: 1.12 $\frac{7.7e6}{3.9e6}$ *1.41*

correction, the best iteration function without the negation map is the one with $r = 64$. Comparing that one with each iteration function that uses the negation map, thus boosting the latter's yield ratio by a factor of $C = ((\frac{2r-1}{r})/(\frac{63}{64}))^{\frac{1}{2}}$ or $C = ((\frac{2r-1}{r-1})/(\frac{63}{64}))^{\frac{1}{2}}$ for g and $\bar{g}$, leads to the long term speedup figure. Note that the correction factor C depends on the iteration function, and is close to and for some r larger than $\sqrt{2}$.

The numbers in Table 4.3 have the following meaning. For the (iteration function, escape method, r-value) combinations specified, the non-italics entries list the long term yield (millions of distinguished points, found during the *second* half hour when running a given setting for an hour) and the long term speedup over the best r-value ($r = 64$) without the negation map, taking into account the correction factor C. Cycle detection and subsequent escape by adding $\mathfrak{f}_{\ell(\mathfrak{p})+4}$, $\mathfrak{f}'$, $\mathfrak{f}''_{\ell(\mathfrak{p})}$ and by doubling is indicated by "e," "e′," "e″" and by "ē," respectively. The iteration functions f (2-cycle reduction), g (4-cycle and 2-cycle reduction), $\bar{f}$ (2-cycle reduction using doubling), and $\bar{g}$ (4-cycle and 2-cycle reduction using doubling) are as in Sections 4.7, 4.8.1 and 4.8.3. The yields are for 256 parallel walks (sharing the inversion) for a 131-bit ECDLP with a 131-bit prime order group. The yields during the first half hour are almost consistently higher, considerably so for poorly performing combinations. They are not meaningful and are thus not listed.

We measured to what extent our failure to achieve a speedup by a factor of $\sqrt{2}$ can be blamed on cycle detection and escape and other overheads, and which part is due to the higher average cost of the iteration function. For most combinations in Table 4.3 we counted the number S of *useful* steps performed when doing 10^9 group operations, while keeping track of the number D of doublings among them. Here a step is useful if it is not taken as part of a fruitless cycle, so all D doublings are useful. Without the negation map, S would be 10^9 and $D = 0$; this is the basis for the comparison. With the negation map, $A = 10^9 - S$ is counted as the number of *additional* additions due to cycle reductions or fruitless cycles. The inherent slowdown of that iteration function is then $1 + \frac{A+D/6}{S}$, so that it can achieve a speedup by a factor of at most $\frac{CS}{S+A+D/6} = \frac{C(10^9-A)}{10^9+D/6}$, with C being the correction factor as defined above. The italics entries in Table 4.3 are A above D, followed by the maximal achievable speedup factor of $\frac{C(10^9-A)}{10^9+D/6}$. The rows starting with † apply to the cases: "no reduction, no escape," "just f," "just $\bar{f}$," "just e," and "just e′."

Non-doubling 2-cycle reduction (f) with doubling-based cycle escape ($\bar{\text{e}}$) and $r = 128$ performed best, with an overall speedup by a factor of 1.29: although fewer distinguished points are found than for the best case without the negation map ($r = 64$), there is a considerable overall gain because fewer distinguished points (by a factor of C, for the relevant C) should suffice. For $r = 16$ most iteration functions with the negation map perform poorly.

Based on Table 4.3 and Figure 4.2, we conclude that our failure to better approach the optimal speedup by a factor of $\sqrt{2}$ is due to an onset of cache effects combined with various overheads. The italics figures from Table 4.3 make us believe that improvements may be obtained when using better implementations.

Previous Results

The only publication that we know that presents practical data about Pollard's rho method used with the negation map is [76]. Only relatively small ECDLPs were solved (42- and 43-bit prime fields) and small r-values were avoided. The adverse cycle behavior that we witnessed can therefore not be expected and we doubt if the results reported are significant for the sizes that we consider. Only mixed walks were used, and an overall speedup by a factor of about 1.35 was reported. Cycle escaping was done by jumping to the sum of all points in a cycle, which cannot be expected to work in general because the sum may depend just on the addition constants.

4.10 Conclusion

It was shown that the tag-tracing method from [57] can in principle be applied in elliptic curve context as well, but that scenarios are limited where the proposed method could lead to a speedup.

With judicious application of doubling, usage of the negation map to solve ECDLPs over prime fields using Pollard's rho method can indeed be recommended. In the best of circumstances that we have been able to create, however, the speedup falls short of the hoped for $\sqrt{2}$, but is with 1.29 still considerable.

4.11 Follow-Up Work

After the publication of our work in [38] a follow-up work appeared by Bernstein, Lange, and Schwabe [24]. Here some techniques are presented, in the setting of the 112-bit ECDLP (see Chapter 5), to eliminate the branches required to compute the negation map in SIMD-environments. This removes one of the main obstacles to use the negation map in such parallel environments. Although no direct comparison between a regular (non-negation map) and a negation map implementation are being made, the authors claim to achieve a speedup "very close to $\sqrt{2}$" on the Cell broadband engine. In [24] it is estimated that "under 4% of the cycles per iteration are spent on operations that can be blamed on negation". Note that a speedup of $\sqrt{2} - 0.04 \approx 1.37$ is in correspondence with the theoretical speedup figures as presented in Table 4.3 taking into account that [24] used an 2048-adding walk on the cache-less SPE of the Cell.

Chapter 5

Solving ECDLPs on the Cell

In this chapter an implementation of Pollard's rho method to solve prime field ECDLPs on the Cell processor, the processor that is the heart of the Sony PlayStation 3 (PS3) game console is described. The underlying modular arithmetic is targeted at single instruction multiple data (SIMD) platforms and is mostly branch-free. It can take advantage of prime moduli of a special form using efficient "sloppy reduction." We used the implementation to set a new prime field ECDLP record for a 112-bit prime of the proper special form. The calculation was performed on EPFL's cluster of about 215 PS3s.

The previous prime field ECDLP record, reported in 2002, involved a 109-bit prime [55]. The following may explain the apparent lack of interest to set new ECDLP records. The expected cost to solve a particular ECDLP on any combination of platforms can be extrapolated from a relatively short calculation, given implementations of Pollard's rho method. Cryptographically relevant ECDLPs turn out to be firmly out of reach, despite occasional improvements of Pollard's rho method. Given easy estimation of overall cost and infeasibility of cryptographically relevant problems, not much is gained by solving an ECDLP, in particular of a cryptographically irrelevant size. This is unlike integer factorization where the only convincing way to show the feasibility and estimate the cost of a record-breaking calculation is completing it (cf. the orders of magnitude difference between the actual cost reported in [120] and the estimate in [175]).

We present a parallelized implementation of Pollard's rho method on a cluster of PS3 game consoles, devices that are relatively inexpensive given their processing power. The parallelization exists on five distinct levels: each PS3 runs independently of all others, on each PS3's Cell processor six cores work independently of each other, and each of these cores simultaneously runs 50 times two interleaved 4-fold SIMD processes. The top two levels are merely 'embarrassingly parallel', the first at the physical PS3 level, the other provided by the Cell processor's multi-core design. The 50 simultaneous copies serve to amortize a high cost modular inversion, interleaving is done to improve throughput, and 4-way SIMD exploits the core's arithmetic instruction set.

The first projects on EPFL's PS3 cluster concerned cryptographic hash collisions [192]. To

ascertain the cluster's reliability and stability for projects requiring long integer arithmetic, a rough version of Pollard's rho method for prime field ECDLP was run for a few weeks. Because it turned out to work satisfactorily and because no other project was ready to be deployed, it was left running. As this soon led to the completion of a non-negligible fraction of the total expected work for the ECDLP at hand, it was decided to further optimize the code and, some misgivings notwithstanding, to attempt to solve it. Although our choice of improvements that could be carried through was limited by the early design decisions (as we did not want to start afresh), the overall expected runtime was reduced by more than 60% in the course of the calculation. It may be further reduced by adopting a variety of changes in the initial design [24].

Apart from the prime field ECDLP record we present an efficient 4-way SIMD binary modular inversion and a fast branch-free sloppy reduction and normalization modulo primes of the form $\frac{2^{32\ell}\pm m}{c}$, for relatively small $\ell, m, c \in \mathbf{Z}_{>0}$. These methods are designed for cryptanalytic applications in a SIMD environment. The sloppy reduction may not be suitable for cryptographic applications, because it can produce an incorrect result. When solving ECDLPs, however, it suffices if calculations are most of the time correct: as expected based on our heuristics, sloppy reduction never produced an incorrect result. Many of these methods can be used on SIMD platforms other than the Cell processor, such as graphics cards.

In the second part of this chapter we describe an approach to solve the Certicom challenge ECC2K-130 challenge. This challenge states an ECDLP using a Koblitz curve [125], an elliptic curve defined over a particular type of binary extension field. This work is part of a larger project which aims to solve this challenge using a variety of platforms [7]. Many optimization techniques used to achieve the fast arithmetic do not require independent parallel computations (batching) and are therefore not only relevant in the context of cryptanalytical applications but can also be used to accelerate cryptographic schemes in practice.

Section 5.1 contains material from [35, 36] while Section 5.4 is based on parts of [40].

5.1 A 112-bit Prime Field ECDLP

The first part of this chapter concentrates on curve "secp112r1" from [173] (also defined in the Wireless Transport Layer Security Specification [80] as *curve number 6*). Let $R = 2^{128}$ and $\tilde{p} = R - 3$, then $p = \frac{\tilde{p}}{11 \cdot 6949}$ is prime. The elliptic curve $E_{a,b}$ over $\mathbf{F}_p$ defined by $a = p - 3$ and

$$b = 2061118396808653202902996166388514$$

has a group $E_{a,b}(\mathbf{F}_p)$ of prime order $q = p + 1 + 4407293269000505$ and is generated by

$$\mathfrak{g} = (1882814650579725348922237787137 52, 3419875491033170827167861896082688).$$

This curve and generator were created "verifiably at random" [173], implying that solving ECDLP in $\langle \mathfrak{g} \rangle = E_{a,b}(\mathbf{F}_p)$ should not be unexpectedly easy due to a built-in trapdoor. Because no corresponding challenge ECDLP was included in [173], we defined one ourselves in a "verifiably not pre-cooked" manner by taking $\mathfrak{h} = (x, y) \in \langle \mathfrak{g} \rangle$ for an unforgeable value

of x. With $x = \lfloor(\pi - 3)10^{34}\rfloor$, this leads to the 112-bit prime ECDLP where

$$\mathfrak{h} = (\ 141592653589793238462643383279502 8,\\ 384675960649470672428613962388554 4) \in E_{a,b}(\mathbf{F}_p),$$

is given and an m with $m\mathfrak{g} = \mathfrak{h}$ must be found.

5.2 Pollard's Rho Method on the PS3

To solve the ECDLP from Section 5.1 with Pollard's rho method, each core of the Cell processes four walks simultaneously in 4-way SIMD fashion, two of those SIMD-processes are interleaved, and as many as possible of these interleaved processes are batched, to amortize the inversion cost in the best possible way (Section 4.2). Although the description below focuses on the 4-way SIMD parallelization of the Cell processor's cores, many ideas apply to wider SIMD environments as well, such as graphics cards.

The 4-way SIMD long integer representation, tailored to the Cell's instructions is used (described in Section 2.2.2). The interleaved 4-way SIMD arithmetic modulo the specific p (Section 5.1) is described in Section 5.2.1. To gain speed, results may be incorrect; it is argued that it may be expected that bad cases do not occur (though an example is given). Section 5.2.2 describes a 4-way SIMD implementation of binary modular inversion. Timings and the solution to the ECDLP are given in Section 5.3.

5.2.1 4-way SIMD Long Integer SPU-Arithmetic

With $R = 2^{128}$, reduction modulo the multiple $\tilde{p} = R-3$ of the prime $p = \frac{\tilde{p}}{11\cdot 6949}$ (Section 5.1) can be done using *sloppy reduction modulo* $\tilde{p}$, which is faster than reduction modulo p but which may produce an incorrect result, with a probability that is argued to be negligible. When working in $\mathbf{F}_p$ we use a redundant representation modulo $\tilde{p}$. Only when required for distinguishing and partition properties (Section 4.2) we switch to a unique value modulo p using a quick Montgomery-like step [145]. All methods in this section allow any number of SIMD threads. See [9, 11, 12, 65, 188, 189], for instance, for previous work involving primes of a special form. We are not aware of earlier publication of modular arithmetic similar to sloppy reduction or an analysis thereof.

Sloppy Reduction Modulo $\tilde{p}$

For $z \in \mathbf{Z}$ with $0 \le z < R^2$ and $z = z_0 + Rz_1$ for $z_0, z_1 \in \mathbf{Z}, 0 \le z_0, z_1 < R$, define

$$\mathfrak{R}(z) = z_0 + 3z_1.$$

From $\tilde{p} = R - 3$ it follows that $\mathfrak{R}(z) \equiv z \bmod \tilde{p}$ and $\mathfrak{R}(z) \equiv z \bmod p$. With $\mathfrak{R}(z) = y = y_0 + y_1R$ for $y_0, y_1 \in \mathbf{Z}, 0 \le y_0, y_1 < R$, it follows from $\mathfrak{R}(z) = z_0 + 3z_1 \le 4R - 4$ that $y_1 \le 3$. If $y_1 = 3$, then $y_0 + y_1R = y_0 + 3R \le 4R - 4$ and thus $y_0 \le R - 4$. Using $y_0 \le R - 1$ when $y_1 \le 2$, it follows that $\mathfrak{R}(y) = y_0 + 3y_1 \le R + 5$.

Algorithm 9 Sloppy reduction modulo $\tilde{p}$ of a four-tuple of 256-bit integers.

Input: a four-tuple (c_1, c_2, c_3, c_4) of 256-bit integers in radix 2^{16} represented by sixteen 128-bit registers $c[0], c[1], \ldots, c[15]$.

Output: a four-tuple (t_1, t_2, t_3, t_4) of 128-bit integers $t_i = \mathfrak{S}(c_i) \bmod R$, for $i = 1, 2, 3, 4$, in radix 2^{16} represented by eight 128-bit registers $t[0], t[1], \ldots, t[7]$.

1: Let r be a register with $r_1 = r_2 = r_3 = r_4 = 3 \cdot 2^{16}$
2: /* the 16 most significant bits of the words of r all represent 3 */
3: /* Compute the first application of $\mathfrak{R}$ */
4: **for** $k = 0$ to 7 **do**
5: $\quad t[k] \leftarrow$ `spu_mhhadd`(`spu_sl`$(c[k+8], 16), r, c[k])$
6: **for** $k = 0$ to 6 **do**
7: $\quad (s, t[k]) \leftarrow$ `spu_split`$(t[k])$
8: $\quad t[k+1] \leftarrow$ `spu_add`$(t[k+1], s)$
9: $(s, t[7]) \leftarrow$ `spu_split`$(t[7])$
10: /* Compute the second application of $\mathfrak{R}$ */
11: $t[0] \leftarrow$ `spu_mhhadd`(`spu_sl`$(s, 16), r, t[0])$
12: $(s, t[0]) \leftarrow$ `spu_split`$(t[0])$
13: **if** `spu_orx`$(s) \neq 0$ **then**
14: $\quad t[1] \leftarrow$ `spu_add`$(t[1], s)$
15: $\quad$ **for** $k = 1$ to 6 **do**
16: $\quad\quad (s, t[k]) \leftarrow$ `spu_split`$(t[k])$
17: $\quad\quad t[k+1] \leftarrow$ `spu_add`$(t[k+1], s)$
18: /* truncate modulo R by ignoring there may be an $i \in \{1, 2, 3, 4\}$ with $t[7]_i \geq 2^{16}$ */
19: **return** $t[0], t[1], \ldots, t[7]$

Define $\mathfrak{S}(z) = \mathfrak{R}(\mathfrak{R}(z))$. Then $\mathfrak{S}(z) < R + 6$ and $\mathfrak{S}(z) \equiv z \bmod \tilde{p}$ (and thus $\mathfrak{S}(z) \equiv z \bmod p$). If all values in the range of $\mathfrak{S}$ occur with approximately the same probability, then $\mathfrak{S}(z) \geq R$ with probability close to $\frac{6}{R+6}$, which is small. Thus, the truncated value $\mathfrak{S}(z) \bmod R \in \{0, 1, \ldots, R-1\}$ is most likely equivalent to z modulo $\tilde{p}$. For relevant z-values, i.e, products of two 128-bit integers, it is argued below that $\mathfrak{S}(z) \geq R$ with probability only about $\frac{1}{R}$, so low that $\mathfrak{S}$ may indeed simply be truncated, rather than applying $\mathfrak{R}$ a third time (which would always be correct modulo $\tilde{p}$ and p). Sloppy reduction modulo $\tilde{p}$ of z is therefore defined as $\mathfrak{S}(z) \bmod R \in \{0, 1, \ldots, R-1\}$.

The SPU calculation of 4-way SIMD sloppy reduction modulo $\tilde{p}$ of a four-tuple of 256-bit integers in radix 2^{16} representation is done by the algorithm depicted in Algorith 9. Without the `if`-statement in line 13 (while keeping lines 14-17) it is branch-free (and slower).

Incorrectness Probability of Sloppy Reduction Modulo $\tilde{p}$ of Products

Let $0 \leq x, y < R$ and let $xy = a + b\tilde{p}$ for integers a, b with $0 \leq a < \tilde{p}$ and $0 \leq b \leq R+1$. Define c as the smallest integer such that $0 \leq cR + a - 3b < R$. It then follows from $xy = cR + a - 3b + (b-c)R$ that $\mathfrak{R}(xy) = cR + a - 3c$. If $a - 3c \geq 0$, then $\mathfrak{S}(xy) = a < \tilde{p}$ so that sloppy reduction modulo $\tilde{p}$ produces the correct result. If $a - 3c < 0$, then $\mathfrak{R}(xy) = (c-1)R + R + a - 3c$. With $cR < R - a + 3b \leq R + 3(R+1)$ so that $c \leq 4$ and thus $0 \leq R + a - 3c < R$, it follows that $\mathfrak{S}(xy) = R + a - 3c + 3c - 3 = R + a - 3$. Because also $\mathfrak{S}(xy) < R + 6$, the cases where $\mathfrak{S}(xy) \geq R$ (and sloppy reduction modulo $\tilde{p}$ is incorrect) are $3 \leq a \leq 8$.

Because $\mathfrak{S}(xy) \in \{R, R+1, \ldots, R+5\}$ for pairs (x, y) for which sloppy reduction modulo $\tilde{p}$ of xy is incorrect, it follows that $\mathfrak{S}(xy)$ is coprime to $\tilde{p}$, implying that x and y are co-prime to $\tilde{p}$. But if for such a pair it is the case that $\gcd(x, \tilde{p}) = 1$, then $\gcd(y, \tilde{p}) = 1$ as well.

Writing $a = i + 3k$, where $i \in \{0, 1, 2\}$ and $k \in \{1, 2\}$, it follows from $a - 3c < 0$ that $c \geq k + 1$. Since c is minimal such that $0 \leq cR + a - 3b < R$, it follows that $kR + a - 3b < 0$ and thus $b > \frac{a+kR}{3}$. With $xy = a + b\tilde{p}$ and $a \geq 3k$ this implies $xy > 3k + \frac{(3k+kR)\tilde{p}}{3} = 3k + \frac{k(R+3)(R-3)}{3} = \frac{kR^2}{3}$. Thus $x, y > \frac{kR}{3}$, since $0 \leq x, y < R$. The number of pairs (x, y) with $x, y > \frac{kR}{3}$ and $xy > \frac{kR^2}{3}$ is approximated as

$$\frac{(3-k)R^2}{3} - \int_{\frac{kR}{3}}^{R} \frac{kR^2}{3x} dx = \frac{(3-k)R^2}{3} - \frac{kR^2}{3} \log\left(\frac{3}{k}\right).$$

For $3k \leq a < 3(k+1)$ the probability that $xy \equiv a \bmod \tilde{p}$ for a pair (x, y) may be approximated as $\frac{3}{R} \cdot \frac{\phi(\tilde{p})}{\tilde{p}}$ (where ϕ denotes Euler's totient function). This leads to

$$\left(\frac{\phi(\tilde{p})}{\tilde{p}}\right) \cdot R \cdot \sum_{k=1,2} \left(3 - k - k \log\left(\frac{3}{k}\right)\right)$$

as a heuristic approximation for the total number of pairs (x, y) where sloppy reduction modulo $\tilde{p}$ of the product xy produces an incorrect result. Because $\frac{\phi(\tilde{p})}{\tilde{p}} \approx 0.90896$, the sum equals $3 - \log\left(\frac{27}{4}\right) \approx 1.09046$, and $0.90896 \cdot 1.09046 \approx 0.99118$, we find a heuristic upper bound of $\frac{1}{R}$ for the probability that sloppy reduction modulo $\tilde{p}$ of xy is incorrect, assuming that x and y are drawn at random.

Incorrectness probability for other moduli

Sloppy reduction may be advantageous for other primes of the form $\frac{2^{32\ell} \pm m}{c}$ for relatively small $\ell, m, c \in \mathbf{Z}_{>0}$. For $\ell = 6, 8$, $m = 38$, $c = 2$ [12,30], and the functions $\mathfrak{R}'(z_0 + z_1 2^{32\ell}) = z_0 + m z_1$ and $\mathfrak{S}'(z) = \mathfrak{R}'(\mathfrak{R}'(z))$, sloppy reduction modulo either of the two primes $2^{32\ell-1} - \frac{m}{2}$ is defined as $\mathfrak{S}' \bmod 2^{32\ell}$, i.e., truncation of $\mathfrak{S}'$ to 32ℓ bits (this works for $\ell = \frac{1}{2}$ and $\ell = 1$ too). A heuristic upper bound of $\frac{343}{2^{32\ell}}$ for the probability that sloppy reduction modulo $2^{32\ell} - m$ of xy is incorrect, for random non-negative $x, y < 2^{32\ell}$, follows as above. It uses

$$\sum_{k=1}^{m-1} \left(m - k - k \log\left(\frac{m}{k}\right)\right) \approx 342.552$$

Algorithm 10 Radix 2^{16} schoolbook multiplication of two four-tuples of $16m$-bit integers.

Input: two four-tuples $(a_1, a_2, a_3, a_4), (b_1, b_2, b_3, b_4)$ of $16m$-bit integers in radix 2^{16} represented by $2m$ 128-bit registers $a[0], a[1], \ldots, a[m-1], b[0], b[1], \ldots, b[m-1]$.

Output: a four-tuple (c_1, c_2, c_3, c_4) of $32m$-bit integers $c_i = a_i \cdot b_i$, for $i = 1, 2, 3, 4$, in radix 2^{16} represented by $2m$ 128-bit registers $c[0], c[1], \ldots, c[2m-1]$.

1: **for** $k = 0$ to $m-1$ **do**
2: $c[m+k] \leftarrow 0$
3: $a[k] \leftarrow$ `spu_sl`$(a[k], 16)$
4: $b[k] \leftarrow$ `spu_sl`$(b[k], 16)$
5: **for** $j = 0$ to $m-1$ **do**
6: $(e[0], c[j]) \leftarrow$ `spu_split`(`spu_mhhadd`$(a[0], b[j], c[m]))$
7: **for** $k = 1$ to $m-1$ **do**
8: $(e[k], c[m+k-1]) \leftarrow$ `spu_split`(`spu_add`(`spu_mhhadd`$(a[k], b[j], c[m+k]), e[k-1]))$
9: /* $a[k]_i \cdot b[j]_i + c[m+k]_i + e[k-1]_i \leq (2^{16}-1)^2 + 2^{16} - 1 + 2^{16} - 1 = 2^{32} - 1$ for $i = 1, 2, 3, 4$ */
10: $c[2m-1] \leftarrow e[m-1]$
11: **return** $c[0], c[1], \ldots, c[2m-1]$

and an argument involving $c = 2$ that is somewhat more contrived than the $\phi(\tilde{p})$-argument above: for odd a both x and y must be odd and integration is over the odd x values only, for even a each odd x leads to a single even y whereas each even x leads to two y values. Thus, the summation hides the observation that $\frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \left(\frac{1}{2} + \frac{1}{2} \cdot 2 \right) = 1$.

Sloppy Multiplication Modulo $\tilde{p}$

Algorithm 10 depicts the algorithm for the SPU calculation of 4-way SIMD schoolbook multiplication of two four-tuples of $16m$-bit integers in radix 2^{16} representation [30]. Note that Algorithm 10 is essentially the same as Algorithm 6 with $r = 16$ but using the notation from this chapter. The only subtlety in Algorithm 10 is that none of the two 4-way SIMD additions in line 8 (`spu_add` and as part of `spu_mhhadd`) generates a carry. Algorithm 9 and Algorithm 10 are compatible: using Algorithm 10 with $m = 8$, its four-tuple output can be simultaneously reduced modulo $\tilde{p}$ using Algorithm 9, and the latter's four-tuple output can again be used as one of the four-tuple inputs for Algorithm 10. Sloppy multiplication modulo $\tilde{p}$ consists of a call to Algorithm 10 with $m = 8$ followed by a call to Algorithm 9.

All four outputs of Algorithm 9 have a small probability not to be unique modulo $\tilde{p}$ (with only the residue classes 0, 1, and 2 modulo $\tilde{p}$ allowing two representations), but the outputs are not unique modulo p. Unique representations modulo p are obtained as indicated below. As analyzed above, each output has a small probability to be incorrect: for instance, when $2 \bmod \tilde{p}$ is represented as $2 + \tilde{p} = R - 1$ and squared, the value $\mathfrak{S}((R-1)^2) = R + 1$ is truncated to the incorrect result 1.

Algorithm 11 Division by 2^{16} modulo p of a four-tuple of 128-bit integers.

Input: a four-tuple (x_1, x_2, x_3, x_4) of 128-bit integers in radix 2^{16} represented by eight 128-bit registers $x[0], x[1], \ldots, x[7]$.

Output: a four-tuple (y_1, y_2, y_3, y_4) of 128-bit integers $y_i \equiv x_i 2^{-16} \bmod p$, for $i = 1, 2, 3, 4$, in radix 2^{16} represented by eight 128-bit registers $y[0], y[1], \ldots, y[7]$.

1: Let $p[0], p[1], \ldots, p[6]$ be 128-bit registers representing $p_1 = p_2 = p_3 = p_4 = p$ in radix 2^{16}
2: /* Put p's bits in the 16 most significant locations */
3: **for** $k = 0$ to 6 **do**
4: $\quad p[k] \leftarrow$ `spu_sl`$(p[k], 16)$
5: $\nu \leftarrow$ `spu_sl`(`spu_mulo`$(x[0], r), 16)$ where r is a register with $r_1 = r_2 = r_3 = r_4 = 47325$
6: $(y[0], d) \leftarrow$ `spu_split`(`spu_mhhadd`$(p[0], \nu, x[0]))$ /* d is zero */
7: **for** $k = 1$ to 6 **do**
8: $\quad (y[k], y[k-1]) \leftarrow$ `spu_split`(`spu_add`(`spu_mhhadd`$(p[k], \nu, y[k-1]), x[k]))$
9: $(y[7], y[6]) \leftarrow$ `spu_split`(`spu_add`$(x[7], y[6]))$
10: **return** $y[0], y[1], \ldots, y[7]$

Unique Representation Modulo p

Given a four-tuple (x_1, x_2, x_3, x_4) of integers modulo $\tilde{p}$ in $\{0, 1, \ldots, R-1\}$, a unique representation modulo p is required for each x_i at the end of each step of Pollard's rho method. Least non-negative remainders modulo p require computation of $x_i \bmod p \in \{0, 1, \ldots, p-1\}$ for $i = 1, 2, 3, 4$. A faster way to obtain unique representations modulo p is to simultaneously calculate all $x_i 2^{-16} \bmod p \in \{0, 1, \ldots, p-1\}$. This is not the same as $x_i \bmod p \in \{0, 1, \ldots, p-1\}$, but that is not a problem as long as the distinguishing and partition properties are properly defined.

The computation of $x_i 2^{-16} \bmod p$ is done using a single Montgomery reduction [145] iteration in radix 2^{16}. Because $\frac{-1}{p} \equiv 47325 \bmod 2^{16}$, the value $\nu_i = \frac{-x_i}{p} \bmod 2^{16} = 47325 x_i \bmod 2^{16}$ satisfies $x_i + \nu_i p \equiv 0 \bmod 2^{16}$, so that $y_i = (x_i + \nu_i p)/2^{16} \equiv x_i 2^{-16} \bmod p$. A unique representation in $\{0, 1, \ldots, p-1\}$ of y_i modulo p is obtained by observing that

$$y_i \leq \frac{R - 1 + (2^{16} - 1)p}{2^{16}} < 3p,$$

so one of y_i, $y_i - p$, or $y_i - 2p$ is in $\{0, 1, \ldots, p-1\}$.

A 4-way SIMD algorithm to perform the calculation of (y_1, y_2, y_3, y_4) given a four-tuple (x_1, x_2, x_3, x_4) as above is depicted in Algorithm 11 (which in practice should be replaced by a version that uses radix 2^{32} as opposed to radix 2^{16} for the additions to the x_i values). The unique representation is then obtained by two applications of the 4-way SIMD modular subtraction algorithm depicted in Algorithm 12 with $\ell = 4$. Algorithm 12 uses masks to avoid branching, and can simply be changed to have radix 2^{32} inputs or output. If it is used with $b_i = m_i = p$, then the resulting c_i equals the input a_i if $a_i < p$ but c_i equals $a_i - p$ if $a_i \geq p$, for $i = 1, 2, 3, 4$ simultaneously, as required.

Pipelining

To reduce bottlenecks in the even and the odd pipelines, the implementations of all algorithms presented here attempt to balance the two pipelines by shifting instructions between the two. Bottlenecks are also reduced by interleaving two 4-way SIMD processes, thereby considerably increasing overall throughput and reducing overall latency, sacrificing the (mostly irrelevant) latency per walk of Pollard's rho method.

Simultaneous Inversion

With $r = 16$ as chosen in Section 4.1 it is possible to store the data for 50 sequential interleaved 4-way SIMD walks in the SPU's Local Store, synchronizing the walks at the point where the modular inverses are calculated. Per SPU we use the simultaneous inversion from Section 4.4 in a nested manner, not sharing inversions among multiple SPUs as the computational advantages would be outweighed by synchronization and communication overhead.

Let $z_{ijk} \in \mathbf{F}_p^*$ for $1 \leq k \leq 50$, $1 \leq j \leq 2$ and $1 \leq i \leq 4$ denote the 400 elements for which the inversions will be shared per SPU. Using 99 (partially interleaved) 4-way SIMD sloppy multiplications modulo $\tilde{p}$ the four-tuple $(\nu_1, \nu_2, \nu_3, \nu_4)$ of products $\nu_i = \prod_{k=1}^{50} \prod_{j=1,2} z_{ijk} \bmod \tilde{p}$ is calculated, for $i = 1, 2, 3, 4$ simultaneously, while keeping the partial products. The four inverses $\nu_i^{-1} \bmod p$ are then calculated using simultaneous inversion at the cost of $3 \times (4 - 1) = 9$ modular multiplications and one modular inversion (described in Section 5.2.2), using a SIMD tree-based approach for the combination and unraveling. Finally, the individual inverses $z_{ijk}^{-1} \bmod p$ are calculated (in a representation modulo $\tilde{p}$) at the cost of twice 99 4-way SIMD sloppy multiplications modulo $\tilde{p}$, by unraveling in 4-way SIMD fashion.

5.2.2 SIMD Modular Inversion on the SPU

The calculation of the modular inverse of a positive integer b in a residue class of the odd modulus $a = p$ is outlined by the algorithm depicted in Algorithm 13. It uses the binary version of the Euclidean algorithm from [115] to compute an almost Montgomery inverse $b^{-1}2^k \bmod p$ for some integer k, because that allows fast implementation on the SPU. The factor $2^k \bmod p$ is removed by table look-up of the value $2^{-k} \bmod p$ (which equals $\frac{2^{1-k} \bmod p}{2}$ if $(2^{1-k} \bmod p) \in \{0, 1, 2, \ldots, p-1\}$ is even and $\frac{(2^{1-k} \bmod p)+p}{2}$ otherwise) followed by sloppy multiplication modulo $\tilde{p}$ from Section 5.2.1.

Let $d = \gcd(a, b)$. Let y be a solution of $by \equiv d \bmod a$. The algorithm has invariants

$$\begin{gathered} k_u, k_v \geq 0, \\ u, v > 0, \\ u(2^{k_u+k_v} y) \equiv rd \bmod a, \\ v(2^{k_u+k_v} y) \equiv sd \bmod a, \\ \gcd(u, v) = d,\ us - vr = a, \\ 2^{k_u} u \leq a, \\ 2^{k_v} v \leq b, \\ r \leq 0 < s. \end{gathered} \tag{5.1}$$

Algorithm 12 Modular subtraction of two four-tuples of 32ℓ-bit integers in radix 2^{16} representation.

Input: a four-tuple (m_1, m_2, m_3, m_4) of 32ℓ-bit integer moduli in radix 2^{32} represented by ℓ 128-bit registers $m[0], m[1], \ldots, m[\ell-1]$ (typically, but not necessarily, the four moduli are the same); two four-tuples $(a_1, a_2, a_3, a_4), (b_1, b_2, b_3, b_4)$ of 32ℓ-bit integers in radix 2^{16} with $0 \le a_i$ and $0 \le b_i \le m_i$ for $i = 1, 2, 3, 4$, represented by 4ℓ 128-bit registers $a[0], a[1], \ldots, a[2\ell-1], b[0], b[1], \ldots, b[2\ell-1]$.

Output: a four-tuple (c_1, c_2, c_3, c_4) of 32ℓ-bit integers in radix 2^{16} with $0 \le c_i \equiv (a_i - b_i) \bmod m_i$ for $i = 1, 2, 3, 4$, represented by 2ℓ 128-bit registers $c[0], c[1], \ldots, c[2\ell-1]$

Let β be a register with $\beta_1 = \beta_2 = \beta_3 = \beta_4 = 1$, for four borrows that are initially empty
Let γ be a register with $\gamma_1 = \gamma_2 = \gamma_3 = \gamma_4 = 0$, for four carries that are initially empty
/* Convert a and b input registers to radix 2^{32} */
for $k = 0$ to $\ell - 1$ **do**
    $u[k] \leftarrow$ `spu_merge`$(a[2k+1], a[2k])$
    $v[k] \leftarrow$ `spu_merge`$(b[2k+1], b[2k])$
/* Do the subtraction */
for $k = 0$ to $\ell - 1$ **do**
    $c[k] \leftarrow$ `spu_subx`$(u[k], v[k], \beta)$
    $\beta \leftarrow$ `spu_genbx`$(u[k], v[k], \beta)$
/* Set the masks for the negative c_i's, i.e., the zero β_i's */
$\mu \leftarrow$ `spu_cmpeq`$(\beta, 0)$ /* where 0 consists of 128 zero bits */
/* if $\beta_i = 0$ (implying that ith mask μ_i is all ones), then add m_i to c_i */
for $k = 0$ to $\ell - 1$ **do**
    $\nu \leftarrow$ `spu_and`(m_i, μ)
    $t[k] \leftarrow$ `spu_addx`$(c[k], \nu, \gamma)$
    $\gamma \leftarrow$ `spu_gencx`$(c[k], \nu, \gamma)$
/* Convert from radix 2^{32} to radix 2^{16} */
for $k = 0$ to $\ell - 1$ **do**
    $(c[2k+1], c[k]) \leftarrow$ `spu_split`$(t[k])$
return $c[0], c[1], \ldots, c[2\ell-1]$

The values of u and v are bounded by a and b, respectively. The invariant $a = us - vr \ge s - r$ bounds r and s. For $\ell = 4$ both r and s fit in 128 bits. When the loop exits the subscript $k_u + k_v$ is bounded as follows:

$$2^{k_u + k_v} \le (2^{k_u} u)(2^{k_v} v) \le ab.$$

At that point $u = v = \gcd(u, v) = d$. If $v > 1$ then b is not coprime to a and the modular

Algorithm 13 Outline of a single modular inverse computation using 4-way SIMD arithmetic.

Input: $\begin{cases} a, b, \ell \text{ where } a \text{ is odd, } a, b > 0, \text{ and } \ell \text{ is the radix } 2^{32} \text{ length of } a; \\ \text{assume availability of a large enough table of } 2^{-k} \bmod a \text{ for } k = 0, 1, 2, 3, \ldots . \end{cases}$

Output: "Not relatively prime," or a residue class $b^{-1} \bmod a$.

1: Let (u, r, v, s) be a four-tuple of 32ℓ-bit integers, represented in radix 2^{32} using ℓ 128-bit registers, with initial value $(a, 0, b, 1)$.
2: Let (k_u, k_v) be a pair of 32-bit integers, represented using a 128-bit register, with initial value $(0, 0)$
3: **while** true **do**
4: Find t_u such that 2^{t_u} divides u and t_v such that 2^{t_v} divides v (see text)
5: $(k_u, k_v) \leftarrow (k_u + t_u, k_v + t_v)$
6: $(u, r, v, s) \leftarrow (u/2^{t_u}, r \cdot 2^{t_v}, v/2^{t_v}, s \cdot 2^{t_u})$
7: **if** $u > v$ **then**
8: $(u, r, v, s) \leftarrow (u - v, r - s, v, s)$
9: **else if** $v > u$ **then**
10: $(u, r, v, s) \leftarrow (u, r, v - u, s - r)$
11: **else if** v equals 1 **then**
12: **return** $s \cdot 2^{-(k_u + k_v)} \bmod a$
13: **else**
14: **return** Not relatively prime

inverse computation fails. Otherwise $d = 1$ and the output $z = s \cdot (2^{-k_u - k_v})$ satisfies

$$\begin{aligned} z &= zd \equiv s \cdot (2^{-k_u - k_v})d \\ &\equiv (v 2^{k_u + k_v} y) 2^{-k_u - k_v} \equiv vy = y \bmod a. \end{aligned}$$

At the start of every iteration at least one of u and v is odd, by (5.1). If t_u and t_v are picked as large as possible, then the new u and v will both be odd, so that after the subtraction and next iteration's shift $u + v$ will be reduced by at least a factor of 2.

The trailing zero bit count of a positive integer k is the population count of $\overline{k} \wedge (k - 1)$. Examining u and v simultaneously can therefore be done using the SPU's population count instruction; however, it acts only on 8-bit data, so the resulting t_u and t_v may not be maximal. This increases the number of iterations performed by Algorithm 13 by about 1%: with maximal t_u and t_v the number of iterations would be close to 0.706 times the bitlength of a, as analyzed in [122]. Algorithm 13 needs on average almost 80 iterations for inversion modulo p.

The four differences $u - v$, $r - s$, $v - u$, and $s - r$ are evaluated simultaneously. The loop is exited if neither $u - v$ nor $v - u$ needs a borrow. Otherwise, depending on the sign of $u - v$ a mask is created to build a fast branch-free selector of the parts of (u, r, v, s) that must be updated. This, and the fact that we know that the inputs are co-prime, avoids the four branches from Algorithm 13. The implementation does not take advantage of the decreasing

sizes of u and v or of the initial small sizes of r and s, but treats them all as 32ℓ-bit integers. Nevertheless, it is quite efficient because only 4-way SIMD operations are carried out on the four-tuple (u, r, v, s). For $\ell = 4$ it is about 8.5 times faster than the implementation from [108].

5.3 Timings and Solution of the Prime Field ECDLP

With parameters as selected above, the clock cycle counts for the various operations are listed in Table 5.1. It lists both the number of clock cycles used by a single operation for eight walks in parallel (organized as two interleaved 4-way SIMD processes), but also the *artificial* number of clock cycles used per operation and iteration in the third and fifth column, respectively: artificial because a single sloppy multiplication modulo $\tilde{p}$ for one walk is not completed in 54 clock cycles, but $8 \times 54 \approx 430$ clock cycles suffice to do eight multiplications, one for each of eight walks.

Table 5.1 refers only to the cost of regular point addition, as iterations do not perform doublings: this saves code (and thus space) and makes the main inner-loop of the parallel walks branch-free at a negligible risk to drop off the curve (as argued in Section 4.2). The "Miscellaneous" category accounts for the retrieval of the $\mathfrak{f}_i$'s, data-shuffling, distinguished point checking, and all other overheads including occasional branching.

At 3.2GHz, an SPU performs about seven million iterations per second. With a 24-bit distinguishing property (of the unique representation of $x2^{-16} \bmod p \in \{0, 1, 2, \ldots, p-1\}$), a single PS3 (six SPUs) produced on average five distinguished points every two seconds, i.e., at most 160-bytes per second in uncompressed format. The ethernet connecting a server with the 215 PS3s could easily handle the required bandwidth.

Approximately 8.5×10^{16} elliptic curve additions were carried out to find that $m\mathfrak{g} = \mathfrak{h}$ for

$$m = 312521636014772477161767351856699.$$

This number of elliptic curve additions is close to the number $\sqrt{\frac{\pi q}{2}} \approx 8.36 \times 10^{16}$ of iterations expected based on the birthday paradox. It is also close to the number of iterations expected based on Eq. (4.1), namely $\sqrt{\frac{\pi q}{2(1-\frac{1}{16})}} \approx 8.64 \times 10^{16}$, which takes into account that we used a 16-adding walks. This effort translates into more than 10^{18} additions and multiplications modulo the 112-bit prime number p (or, most of the time, its 128-bit multiple $\tilde{p} = 2^{128} - 3$), and thus to well over 2^{60} operations on 32-bit or 64-bit integers. With our latest software the calculation would have taken less than four months. Because earlier versions were less efficient, the actual calculation took from January 13 to July 8, 2009.

Slightly more than five billion distinguished points were collected. All distinguished points received were correct, indicating that none of the 5×10^{17} sloppy reductions modulo $\tilde{p}$ was incorrect (each had probability argued to be less than $2^{-128} \approx 10^{-38.53}$ to be incorrect, see Section 5.2.1), and that none of the walks dropped off the curve due to an overlooked doubling (which too would have happened with negligible probability, see Section 4.2) – or that if such mishaps occurred they magically cancelled each others' effect (a possibility that can safely be ruled out).

Table 5.1: Average (Avg) clock cycle count for the operations (op) carried out during an iteration (it) of Pollard's rho method on a single SPU that performs 50 sequential processes, each consisting of two interleaved 4-way SIMD iterations (computing on 8 walks), for a total of 400 simultaneous walks per SPU.

Operation (sloppy modulus $\tilde{p} = 2^{128} - 3$, modulus $p = \frac{\tilde{p}}{11 \cdot 6949}$)	Avg #cycles per 2×4-SIMD ops (8 walks)	Avg #cycles per op (1 walk)	Op per it	Avg #cycles per it (1 walk)
Sloppy multiplication modulo $\tilde{p}$ (multiplication+reduction)	430 $(318 + 112)$	54 $(40 + 14)$	6	322
Modular subtraction	40 (40 even, 24 odd)	5	6	30
Modular inversion	n/a	4941	$\frac{1}{400}$	12
Unique representation mod p	192	24	1	24
Miscellaneous	544	68	1	68
Throughput (average #cycles per iteration for a single walk)				456
Latency (average #cycles per iteration for 400 simultaneous walks per SPU)				$182 \cdot 10^3$

5.4 An Approach to Solve ECC2K-130

In this second part of the chapter an approach is presented to solve an ECDLP where the elliptic curve is a so-called Koblitz curve [125] over the finite field $\mathbf{F}_{2^{131}}$. This setting is different from the rest of this thesis where only elliptic curves over prime fields ($E(\mathbf{F}_p)$ with $p > 3$ prime) are considered. More specifically, the target curve is defined in the Certicom challenge [54], a list of curves and parameters provided by Certicom as a challenge to solve, and is denoted as ECC2K-130.

The Cell implementation discussed here is one of the two approaches to perform the finite field arithmetic which are described in [40]. Note that [40] zooms in on Section 6 of [7] and describes the implementation of the parallel Pollard rho algorithm for the Synergistic Processor Elements of the Cell architecture in more detail. In [40] a bit-sliced [26] approach and a non-bitsliced (standard) approach are studied in the setting of implementing the parallel Pollard rho method when solving the ECDLP for ECC2K-130. As expected, since a bitsliced approach fits a computer more naturally, we found that the bitsliced approach outperforms the "standard" approach. But the speedup for the bitsliced approach was less than we had anticipated. The details of this standard (non-bitsliced) approach are given in this section.

Many optimization techniques for the non-bitsliced version do not require independent parallel computations (batching) and are therefore not only relevant in the context of crypt-analytical applications but can also be used to accelerate cryptographic schemes in practice. To the best of our knowledge this is the first work to describe an implementation of high-speed binary-field arithmetic for the Cell.

5.4.1 ECC2K-130 and Choice of Iteration Function

The specific ECDLP addressed in this paper is given in the Certicom challenge list [54] as challenge ECC2K-130. The elliptic curve is a Koblitz curve $E : y^2 + xy = x^3 + 1$ over the finite field $\mathbf{F}_{2^{131}}$; the two given points P and Q have order l, where l is a 129-bit prime. The challenge is to find an integer k such that $Q = [k]P$. Here we will only give the definition of distinguished points and the iteration function used in our implementation. For a detailed description please refer to [7], for a discussion and comparison to other possible choices also see [6].

Let us denote by $\mathrm{HW}(x)$ the Hamming weight of an integer x. We define a point $R_i \in E(\mathbf{F}_{2^{131}})$ as *distinguished* if the Hamming weight of the x-coordinate in normal basis representation $\mathrm{HW}(x(R_i))$ is smaller than or equal to 34. Our iteration function is defined as

$$R_{i+1} = f(R_i) = \sigma^j(R_i) + R_i,$$

where σ is the Frobenius endomorphism and

$$j = ((\mathrm{HW}(x_{R_i})/2) \pmod 8) + 3.$$

Using a restricted set of Frobenius powers is not new and was used in the computation of the smaller ECDLPs over Koblitz by Harley [99]. Restricting j to eight values has some advantages for hardware implementations and this choice of j makes sure to avoid entering small fruitless cycles (see for more details [7]).

The restriction of σ to $\langle P \rangle$ corresponds to scalar multiplication with some integer r. For an input $R_i = a_i P + b_i Q$ the output of f will be $R_{i+1} = (r^j a_i + a_i)P + (r^j b_i + b_i)Q$. When a collision has been detected, it is possible to recompute the two corresponding iterations and update the coefficients a_i and b_i following this rule. This gives the coefficients to compute the discrete logarithm.

5.4.2 Computing the Iteration Function

Computing the iteration function requires one application of σ^j and one elliptic-curve addition. Furthermore we need to convert the x-coordinate of the resulting point to normal basis, if a polynomial-basis representation is used, and check whether it is a distinguished point.

Many applications use so-called inversion-free coordinate systems to represent points on elliptic curves (see, e.g., [98, Section 3.2]) to speed up the computation of point multiplications. These coordinate systems use a redundant representation for points. Identifying distinguished points requires a unique representation, which is why we use the affine Weierstrass representation to represent points on the elliptic curve. Elliptic-curve addition in affine Weierstrass coordinates on the given elliptic curve requires two multiplications, one squaring, six additions, and a single inversion in $\mathbb{F}_{2^{131}}$ (see, e.g. [19]). Application of σ^j means computing the 2^j-th powers of the x- and the y-coordinate. In total, one iteration takes two multiplications, a single squaring, two computations of the form r^{2^m} (where r is an integer, see the previous subsection), with $3 \leq m \leq 10$, a single inversion, a single conversion to

normal-basis, and a single Hamming-weight computation. In the following we will refer to computations of the form r^{2^m} as *m-squaring*.

5.4.3 Polynomial or Normal Basis?

Another choice to make for both bitsliced and non-bitsliced implementations is the representation of elements of $\mathbb{F}_{2^{131}}$: Polynomial bases are of the form $(1, z, z^2, z^3, \ldots, z^{130})$, so the basis elements are increasing powers of some element $z \in \mathbb{F}_{2^{131}}$. Normal bases are of the form $(\alpha, \alpha^2, \alpha^4, \ldots, \alpha^{2^{130}})$, so each basis element is the square of the previous one.

Performing arithmetic in normal-basis representation has the advantage that squaring elements is just a rotation of coefficients. Furthermore we do not need any basis transformation before computing the Hamming weight in normal basis. On the other hand, implementations of multiplications in normal basis are widely believed to be much less efficient than those of multiplications in polynomial basis.

In [202], von zur Gathen, Shokrollahi and Shokrollahi proposed an efficient method to multiply elements in type-II normal basis representation. This approach is used in [7] and optimized in [23]. The bitsliced implementation uses this multiplier while the standard approach uses polynomial arithmetic as outlined in the next section.

5.5 The Non-Bitsliced Implementation

For the non-bitsliced implementation, we decided not to implement arithmetic in a normal-basis representation. The main reason is that the required permutations, splitting and reversing of the bits, as required for the conversions in the Shokrollahi multiplication algorithm (see for more details [7, 23, 202]) are too expensive to outweigh the gain of having no basis change and faster m-squarings.

The non-bitsliced implementation uses a polynomial-basis representation of elements in $\mathbf{F}_{2^{131}} \cong \mathbf{F}_2[z]/(z^{131}+z^{13}+z^2+z+1)$. Field elements in this basis can be represented using 131 bits. On the SPE architecture this is achieved by using two 128-bit registers, one containing the three most significant bits. As described in Section 5.4.2 the functionality of addition, multiplication, squaring and inversion are required to implement the iteration function. Since the distinguished-point property is defined on points in normal basis, a basis change from polynomial to normal basis is required as well. In this section the various implementation decisions for the different (field-arithmetic) operations are explained.

The implementation of addition is trivial and requires two `XOR` instructions. These are instructions going to the even pipeline; each of them can be dispatched together with one instruction going to the odd pipeline. The computation of the Hamming weight is implemented using the `CNTB` instruction, which counts the number of ones per byte for all 16 bytes of a 128-bit vector concurrently, and the `SUMB` instruction, which sums the four bytes of each of the four 32-bit parts of the 128-bit input. The computation of the Hamming weight requires four cycles.

In order to eliminate (or reduce) stalls due to data dependencies we interleave different iterations. Our experiments show that interleaving a maximum of eight iterations maximizes

Algorithm 14 The reduction algorithm for the ECC2K-130 challenge used in the non-bitsliced version. The algorithm is optimized for architectures with 128-bit registers.

Input: $C = A \cdot B = a + b \cdot z^{128} + c \cdot z^{256}$, such that $A, B \in \mathbf{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$ and a, b, c are 128-bit strings representing polynomial values.

Output: $D = C \bmod (z^{131} + z^{13} + z^2 + z + 1)$.

1: $c \leftarrow (c \ll 109) + (b \gg 19)$
2: $b \leftarrow b$ `AND` $(2^{19} - 1)$
3: $c \leftarrow c + (c \ll 1) + (c \ll 2) + (c \ll 13)$
4: $a \leftarrow a + (c \ll 16)$
5: $b \leftarrow b + (c \gg 112)$
6: $x \leftarrow (b \gg 3)$
7: $b \leftarrow b$ `AND` 7
8: $a \leftarrow a + x + (x \ll 1) + (x \ll 2) + (x \ll 13)$
9: **return** $(D = a + b \cdot z^{128})$

performance. We process 32 of such batches in parallel, computing on 256 iterations in order to reduce the cost of the inversion (see Section 4.4). Every iteration all 256 points need to be inspected if they satisfy the distinguished point property. Hence, all 256 points are converted to normal basis. We keep track of the lowest Hamming weight of the x-coordinate among these points. This can be done in a branch-free way eliminating the need for 256 expensive branches (to test if the Hamming weight is ≤ 34). Then, before performing the simultaneous inversion, only one branch is used to check if one of the points is distinguished (by looking at the lowest Hamming weight of the 256 concurrent points). If one or more distinguished points are found, we have to process all 256 points again to determine and output the distinguished points. Note that this happens only very infrequently (since the probability that a point is distinguished is $2^{-25.27}$ [7]).

5.5.1 Multiplication

If two polynomials $A, B \in \mathbf{F}_2[z]/(z^{131}+z^{13}+z^2+z+1)$ are multiplied in a straightforward way using 4-bit lookup tables (containing the multiples from 0 up to 2^4-1), the table entries would be 134-bit wide. Storing and accumulating these entries would require operations (`SHIFT` and `XOR`) on two 128-bit limbs. To avoid computing on two limbs all the time we describe a method which splits the 131-bit polynomials A and B in such a way that most intermediate values fit in a single 128-bit limb. With $0 \leq A, B < 2^{131}$ we denote that the polynomials A and B can be represented using 131 bits. Let us write A and B as $A = A_l + A_h \cdot z^{128}$ and $B = B_l + B_h \cdot z^{128}$ respectively with $0 \leq A_l, B_l < 2^{128}$ and $0 \leq A_h, B_h < 2^3$.

Split A as

$$A = A_l + A_h \cdot z^{128} = \widetilde{A}_l + \widetilde{A}_h \cdot z^{121}$$

with $0 \leq \widetilde{A}_l < 2^{121}$ and $0 \leq \widetilde{A}_h < 2^{10}$. This allows us to build a 4-bit lookup table from $\widetilde{A}_l$ whose entries fit in 124 bits (a single 128-bit limb). Furthermore, the product of $\widetilde{A}_l$ and an 8-bit part of B fits in a single 128-bit limb. While accumulating such intermediate results

we only need byte-shift instructions (which can be computed efficiently using the shuffle instruction on the Cell). In this way we calculate the product $\widetilde{A}_l \cdot B = \widetilde{A}_l \cdot (B_l + B_h \cdot z^{128})$.

When calculating $\widetilde{A}_h \cdot B$ we split B as

$$B = B_l + B_h \cdot z^{128} = \widetilde{B}_l + \widetilde{B}_h \cdot z^{15}$$

with $0 \leq \widetilde{B}_l < 2^{15}$ and $0 \leq \widetilde{B}_h < 2^{116}$. Then we calculate $\widetilde{A}_h \cdot \widetilde{B}_l$ and $\widetilde{A}_h \cdot \widetilde{B}_h$ using two 2-bit lookup tables from $\widetilde{B}_l$ and $\widetilde{B}_h$. We choose to split 15 bits from B in order to facilitate the accumulation of partial products in

$$\begin{aligned} C &= A \cdot B \\ &= (\widetilde{A}_l + \widetilde{A}_h \cdot z^{121}) \cdot B \\ &= \widetilde{A}_l \cdot (B_l + B_h \cdot z^{128}) + \widetilde{A}_h \cdot (\widetilde{B}_l + \widetilde{B}_h \cdot z^{15}) \cdot z^{121} \\ &= \widetilde{A}_l \cdot B_l + \widetilde{A}_l \cdot B_h \cdot z^{128} + \widetilde{A}_h \cdot \widetilde{B}_l \cdot z^{121} + \widetilde{A}_h \cdot \widetilde{B}_h \cdot z^{136} \end{aligned}$$

since $121 + 15 = 136$ which is divisible by 8 allowing fast byte-oriented arithmetic.

The reduction can be done efficiently by taking the form of the irreducible polynomial into account. Given the result C from a multiplication or squaring, $C = A \cdot B = C_h \cdot z^{131} + C_l$, the reduction is calculated using the trivial observation that

$$C_h \cdot z^{131} + C_l \equiv C_l + (z^{13} + z^2 + z^1 + 1)C_h \bmod (z^{131} + z^{13} + z^2 + z + 1).$$

Algorithm 14 shows the reduction algorithm optimized for architectures which can operate on 128-bit operands. This reduction requires ten `XOR`, 11 `SHIFT` and two `AND` instructions. On the SPU architecture the actual number of required `SHIFT` instructions is 15 since the bit-shifting instructions only support shifting up to seven bits (in 4-way 32-bit SIMD fashion). Larger bit-shifts are implemented combining both a byte- and a bit-shift instruction. When interleaving two independent modular multiplication computations, parts of the reduction and the multiplication of both calculations are interleaved to reduce latencies, save some instructions and take full advantage of the available two pipelines.

When doing more than one multiplication containing the same operand, we can save some operations. By doing the simultaneous inversion in a binary-tree style we often have to compute the products $A \cdot B$ and $A' \cdot B$. In this case, we can use the 2-bit lookup tables from $\tilde{B}_l$ and $\tilde{B}_h$. Using these optimizations in the simultaneous inversion a single multiplication plus reduction takes 149 cycles averaged over the five multiplications required per iteration (when interleaving two multiplications to increase throughput).

5.5.2 Squaring

The modular squaring is implemented by inserting a zero bit between each two consecutive bits of the binary representation of the input (to compute the squaring) and next reduce the result as described in Algorithm 14. The squaring can be efficiently implemented using the `SHUFFLE` and `SHIFT` instructions. Just as with the multiplication two squaring computations are interleaved to reduce latencies. A single squaring takes 34 cycles.

5.5.3 Basis Conversion and m-Squaring

The repeated Frobenius map σ^j requires at least six and at most 20 squarings to compute r^{2^m} for $3 \leq m \leq 10$ for both the x- and y-coordinate (see Section 5.4.2), when computed as a series of single squarings. This can be computed in at most $20 \times 34 = 680$ cycles ignoring loop overhead using our single squaring implementation.

To reduce this number a time-memory tradeoff technique is used. We precompute the values

$$T[k][j][i_0 + 2i_1 + 4i_2 + 8i_3] = (i_0 \cdot z^{4j} + i_1 \cdot z^{4j+1} + i_2 \cdot z^{4j+2} + i_3 \cdot z^{4j+3})^{2^{3+k}},$$

for $0 \leq k \leq 7$, $0 \leq j \leq 32$, $0 \leq i_0, i_1, i_2, i_3 \leq 1$. We have $T[k][j][i] \in \mathbf{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$. These precomputed values are stored in two tables, for both limbs needed to represent the number, of $8 \times 33 \times 16$ elements of 128-bit each. This table requires 132 KB which is more than half of the available space of the local store.

Given a coordinate a of an elliptic-curve point and an integer $0 \leq m \leq 7$ the computation of the m-squaring $a^{2^{3+m}}$ can be computed as

$$\sum_{j=0}^{32} T[m][j][\lfloor (a/2^{4j}) \rfloor \bmod 2^4].$$

This requires 2×33 `LOAD` and 2×32 `XOR` instructions, due to the use of two tables, plus the calculation of the appropriate address to load from. Our assembly implementation of the m-squaring function requires 96 cycles, this is 1.06 and 3.54 times faster compared to performing three (3×34 cycles) and ten (10×34 cycles) sequential squarings respectively.

For the basis conversion we used a similar time-memory tradeoff technique. We enlarged the two tables by adding $1 \times 33 \times 16$ elements required to compute the basis conversion. This allows to use the m-squaring implementation, calling the function with an index to these extra elements, saving code size. For the computation of the basis conversion we proceed exactly the same as for the m-squarings, only the initialization of the corresponding table elements is different.

5.5.4 Modular Inversion

From Fermat's little theorem it follows that the modular inverse of $a \in \mathbf{F}_{2^{131}}$ can be obtained by computing $a^{2^{131}-2}$. This can be implemented using 8 multiplications, 6 m-squarings (using $m \in \{2, 4, 8, 16, 32, 65\}$) and 3 squarings. When processing many iterations in parallel the inversion cost per iteration is small compared to the other main operations such as multiplication. Considering this, and due to code-size considerations, we calculate the inversion using the fast routines we already have at our disposal: multiplication, squaring and m-squaring, for $3 \leq m \leq 10$. In total the inversion is implemented using 8 multiplications, 14 m-squarings and 7 squarings. All these operations depend on each other; hence, the interleaved (faster) implementations cannot be used. Our implementation of the inversion requires 3784 cycles.

We also implemented the binary extended greatest common divisor [190] to compute the inverse. This latter approach turned out to be roughly 2.1 times slower.

Table 5.2: Cycle counts per input for all operations on one SPE of a 3192 MHz Cell Broadband Engine. The value B in the last row denotes the batch size for Montgomery inversions.

	Non-bitsliced, polynomial basis	Bitsliced, polynomial basis	Bitsliced, normal basis
Squaring	34	3.164	2.563
m-squaring	96	$m \times 3.164$	2.563
Conditional m-squaring	—	$m \times 3.164 + 4.047$	3.539
Multiplication	149	117.914	130.102
Addition	2	3.844	
Inversion	3784	1354.102	1063.531
Conversion to normal basis	96	29.281	—
Hamming-weight computation	4	6.594	
Pollard's rho iteration	1148 ($B = 256$)	889.406 ($B = 12$)	788.625 ($B = 14$) 745.531 ($B = 512$)

5.5.5 Results

To the best of our knowledge there were no previous attempts to implement fast binary-field arithmetic on the Cell. The cycle counts for all field operations are summarized in Table 5.2 for both approaches. Our experiments showed that on the Cell processor the bitsliced implementation of highly parallel binary-field arithmetic is more efficient than the standard (non-bitsliced) implementation. For applications that do not process large batches of different independent computations the non-bitsliced approach remains of interest.

Using the bitsliced normal-basis implementation—which uses DMA transfers to main memory to support a batch size of 512 for Montgomery inversions—on all six SPUs of a Sony Playstation 3 in parallel, we can compute 25.57 million iterations per second. The expected total number of iterations required to solve the ECDLP given in the ECC2K-130 challenge is $2^{60.9}$ (see [7]). This number of iterations can be computed in 2,654 Playstation 3 years.

5.6 Conclusion

In the first part of this chapter we developed SIMD multiplication modulo primes of the form $\frac{2^{32\ell} \pm m}{c}$ for small $\ell, m, c \in \mathbf{Z}_{>0}$ that achieves a speedup of approximately 30% over more traditional methods. It uses a redundant representation modulo $2^{32\ell} \pm m$ and a truncation-based reduction method, whose probability to produce an incorrect result has been argued to be very small. The method is suitable for error-tolerant applications, such as cryptanalytic ones.

As an application, we have shown the cryptanalytic potential of a commonly available toy by using a cluster of PlayStation 3 game consoles to solve an elliptic curve discrete logarithm problem over a 112-bit prime field. The runtimes and their extrapolations provide upper bounds for the effort required to solve larger instances of the same problem using a larger

network of game consoles. Such a network is in principle accessible using programs such as BOINC [3]. Although surreptitious application of such programs would not be difficult to arrange for any miscreant who desires to do so, the effort required to solve a "practically relevant" problem remains staggering.

In the second part of this chapter we have outlined a novel approach to implement fast (non-bitsliced) binary-field arithmetic. Although it turned out that a bitsliced approach to implement the arithmetic is faster in practice for this setting. The standard approach (unlike the bitsliced approach) can be used to speed up arithmetic in single-stream settings such as cryptography.

Chapter 6

Efficient SIMD arithmetic modulo a Mersenne number

Numbers of a special form often allow faster modular arithmetic operations than generic moduli. This is exploited in a variety of applications and has led to a substantial body of literature on the subject of fast special arithmetic. Speeding up calculations using special moduli was already proposed in the mid-1960s by Merrill [142] in the setting of residue number systems (RNS) [88]. Other applications range from speeding up fast Fourier transform based multiplication [64], enhancing the performance of digital signal processing [69, 187, 195], to faster elliptic curve cryptography (ECC; [124, 143]), such as in [12].

Another application area of special moduli is in factorization attempts of so-called *Cunningham numbers*, numbers of the form $b^n \pm 1$ for $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to high powers. This long term factorization project, originally reported in the Cunningham tables [66] and still continuing in [52], has a long and distinguished record of inspiring algorithmic developments and large-scale computational projects [48, 49, 130, 134, 149, 163]. Factorizations from [52] with $b = 2$ are used in formal correctness proofs of floating point division methods [101]. Several of these developments [133] turned out to be applicable beyond special form moduli, and are relevant for security assessment of various common public-key cryptosystems.

This chapter concerns efficient arithmetic modulo a Mersenne number, an integer of the form $2^M - 1$. These numbers, and a larger family of numbers called generalized Mersenne numbers [8, 58, 188], have found many arithmetic applications ranging from number theoretic transforms [44] to cryptography. In the latter they are used to run calculations concurrently using RNS [9] or to improve the speed of finite field arithmetic in ECC based schemes [188, 199]. The great internet Mersenne prime search project [89] is based on an implementation of the Lucas-Lehmer primality test [129, 139] for Mersenne numbers in the many-million-bit range. Hence, efficient arithmetic modulo a Mersenne number is a widely studied subject, not just of interest in its own right but with many applications.

Our interest in arithmetic modulo a Mersenne number was triggered by a potential (special) number field sieve (NFS) project [133], for which we need a list of composites dividing

$2^M - 1$ for exponents M in the range from 1000 to 1200. The Cunningham tables contain over 20 composite Mersenne numbers (or composite factors thereof) in the desired range that have not been fully factored yet. It may be expected that some of these composites are not suitable candidates for our list because they can be factored faster using the elliptic curve method (ECM) for integer factorization [136] than by means of special NFS (SNFS). The only way to find out whether ECM is indeed preferable, is by subjecting each candidate to an extensive ECM effort (which, though it may be substantial, is small compared to the effort that would be required by SNFS): only candidates that ECM failed to factor should be included in the list.

The efficiency of ECM factoring attempts relies on the efficiency of integer arithmetic modulo the number being factored. Given the need to do extensive ECM pre-testing for over 20 composite Mersenne numbers, we developed arithmetic operations modulo a Mersenne number suitable for implementation of ECM on the platform that we intended to use for the calculations: the Cell processor as found in the Sony PlayStation 3 (PS3) game console. Because each ECM effort consists of a large number of independent attempts that can be executed in single instruction multiple data (SIMD) mode and because each core of the Cell processor can be interpreted as a 4-way SIMD environment, our arithmetic modulo a Mersenne number is geared towards SIMD implementation.

This chapter is published as [39].

6.1 Arithmetic Modulo $2^M - 1$ on the SPE

In this section we describe the SPE-arithmetic that we developed for arithmetic modulo $N = 2^M - 1$, for M in the range from 1000 to 1200 (allowing larger values as well). Notice that the following description can easily be carried over to numbers of the form $2^M + 1$. Assume that $M < 13 \cdot 96 - 2 = 1246$ (larger M-values can be accommodated by putting $M < u \cdot v - 2$ with $v \cdot (2^{u-1})^2 < 2^{31}$). Our approach aims to optimize overall throughput as opposed to minimize per process latency. Two variants are presented: a first approach where addition and subtraction are fast at the cost of a radix conversion before and after the multiplication, and an alternative approach where radix conversions are avoided at the cost of slower addition and subtraction. This second variant turns out to be faster for our ECM application. In applications with a different balance between the various operations the first approach could be preferable, so it is described as well. All our methods are particularly suited to SPE-implementation, but the approach may have broader applicability. See Section 2.1 for the notation of the integer representation.

6.1.1 Related work

In [62] an SPE implementation is presented using arithmetic modulo the special prime $2^{255} - 19$ introduced in [12]. The SPE-performance of generic versus generalized Mersenne moduli is compared in [30] (see Chapter 3). SPE-arithmetic for moduli in the 200-bit range is presented in [17,56]; on PS3s the former is more than twice faster than the latter. Different

approaches to implement arithmetic over a binary extension field on SPEs are stated in [40] (see Section 5.4).

Our usage of a small radix to avoid carries (cf. below) is not new [64], [122, Section 4.6], [17]. In [17] signed radix-2^{13} representation is used along with the SPE's $16 \times 16 \to 32$-bit multiplication instruction to develop fast multiplication modulo 195-bit moduli. Each addition done during a single schoolbook multiplication is carry-less, as for polynomial multiplication, requiring normalization to radix-2^{13} representation only at the end of the big multiplication.

6.1.2 Representation of 4-tuples of Integers Modulo N

Integers are represented similarly as presented in Section 3.3.1. Each 128-bit SPE register is interpreted as being partitioned into four 32-bit *words*. With s 128-bit registers thought to be stacked on top of each other, where $32s \geq M$, four different integers modulo N can be represented using four disjoint parallel columns, each consisting of s words: denoting the ith word of the jth register by w_{ij} for $i \in \{1, 2, 3, 4\}$ and $j = 0, 1, \ldots, s-1$, the sequence $(w_{ij})_{j=0}^{s-1}$ is interpreted as the radix-2^{32} representation of the $32s$-bit integer $\sum_{j=0}^{s-1} w_{ij} 2^{32i}$. More generally, for any $t \leq 32$ of one's choice, the sequence $(w_{ij})_{j=0}^{s-1}$ may represent the integer $\sum_{j=0}^{s-1} w_{ij} 2^{ti}$ whose value depends on the interpretation of the words w_{ij}: as an unnormalized radix-2^t representation if the w_{ij} are interpreted as non-negative integers (normalized and unique if $w_{ij} < 2^t$ as well), and as a signed k-bit radix-2^t representation, for some $k \leq 32$, if the w_{ij} are interpreted as signed k-bit integers.

It should be understood that the integer operations described below are carried out in 4-way SIMD fashion on the SPE.

6.1.3 Addition and Subtraction Modulo N

Addition and subtraction in 4-way SIMD fashion on a pair of 4-tuples of integers modulo N in radix-2^t representation, with each 4-tuple represented by a stack of s registers of 128-bits (where $ts \geq M$), is done by applying s additions or subtractions to the matching pairs of registers (one from each stack), combined with a moderate number of carry propagations. Since N is Mersenne, the reduction modulo N (when needed) usually affects only two of the radix-2^t digits. More digits are affected with probability $2^{-1-t-(M \bmod t)}$, in which case it causes a slight stall for the other three calculations in the 4-tuple.

For $t = 32$ the SPE's built-in carry generation instructions are used. For smaller t-values more work needs to be done. We describe the calculation of $c = a + b \bmod N$ and $d = a - b \bmod N$ (so-called *addition-subtraction* of a and b) given the signed radix-2^{13} representations $a = \sum_{j=0}^{95} a_j 2^{13j}$ and $b = \sum_{j=0}^{95} b_j 2^{13j}$ (cf. Step 5 in Section 6.1.7). Note that $-2^{12} \leq a_j, b_j < 2^{12}$. The following 5 steps are carried out:

1. Let $a'_j = a_j + 2^{12}$ for $0 \leq j < 96$. Now all $0 \leq a'_j < 2^{13}$.
2. Set $c_j = a'_j + b_j$ and $d_j = a'_j - b_j$ for $0 \leq j < 96$. We have $-2^{12} \leq c_j < 2^{13} + 2^{12} - 1$ and $-2^{12} + 1 \leq d_j < 2^{13} + 2^{12}$.

3. Now we can propagate the carries.

   - Initialize the carry τ as 0.
   - For $j = 0$ to 95 in succession do the following
   - first replace τ by $\tau + c_j$,
   - next replace c_j by $\tau \bmod 2^{13}$ (so that $0 \le c_j < 2^{13}$),
   - and finally replace τ by $\lfloor \tau/2^{13} \rfloor$ (which can be negative).

   The resulting τ is a carry corresponding to $\tau \cdot 2^{13 \cdot 96}$; modulo N this carry is taken care of by adding $\tau \cdot 2^{\alpha}$ to c_β (for $\gamma = 13 \cdot 96 - M$, $\beta = \lfloor \gamma/13 \rfloor$ and $\alpha = \gamma - 13\beta \in [0, 12]$) followed by a few more carry propagations. If there is still a carry, which occurs rarely, use a more expensive function.

4. Repeat the previous step with c replaced by d.

5. Set $c_j = c_j - 2^{12}$ and $d_j = d_j - 2^{12}$ for $0 \le j < 96$ (subtracting the value in step 1).

Steps 1, 2, and 5 allow arbitrary parallelization. Table 6.1 lists SPE clock cycle counts for the addition operations modulo $2^{1193} - 1$: it can be seen that for signed radix-2^{13} representation they are about twice as slow as for radix-2^{32} representation.

6.1.4 Multiplication Modulo N using Radix Conversions

Given a pair of 4-tuples of M-bit integers, the four pairwise products result in a 4-tuple of $2M$-bit integers. The four reductions modulo N can in principle be done by means of a few of the above 4-tuple additions and subtractions modulo N. Here we present our first approach that uses two different radix representations, thereby making it possible to take advantage of the fast radix-2^{32} addition and subtraction modulo N. In Section 6.1.6 another approach is described that is based on signed radix-2^{13} representation.

The multiplication modulo N of two M-bit integers a and b given by their radix-2^{32} representations, each using 39 words of 32 bits, proceeds in three steps. The steps are:

1. conversion of inputs a and b to signed radix-2^{13} representation;

2. carry-less calculation of the $2M$-bit product $a \cdot b$ in signed 32-bit radix-2^{13} representation;

3. reduction modulo N and conversion to radix-2^{32} representation of the $2M$-bit product $a \cdot b$, resulting in $c = a \cdot b \bmod N \in \{0, 1, \ldots, N-1\}$.

The following sections describe the steps in more detail.

Conversion of Inputs to Signed Radix-2^{13} Representation

Given the radix-2^{32} representation of the precomputed constant $C_0 = 2^{12} \cdot \sum_{j=0}^{95} 2^{13j}$, first calculate the radix-2^{32} representation of $a + C_0$, in the usual way requiring carries. Next, using masks and shifts, extract the radix-2^{13} representation $(\tilde{a})_{j=0}^{95}$ of $a + C_0$, and finally subtract C_0 again by calculating $a_j = \tilde{a}_j - 2^{12}$, for $j = 0, 1, \ldots, 95$ (because $a_{96} = 0$ for our choice of M, it is dropped).

This approach (first adding $2^{12} \cdot \sum_{j=0}^{95} 2^{13j}$ and finally subtracting this value from the individual digits a_j) is used because it allows the last two steps to run in parallel. Furthermore it can be run twice as fast (while requiring fewer registers) if two 13-bit chunks are packed into a single 32-bit word. Applying the same method to b, we find signed radix-2^{13} representations of the inputs, below regarded as polynomials

$$P_a(X) = \sum_{j=0}^{95} a_j X^j, \quad P_b(X) = \sum_{j=0}^{95} b_j X^j \in \mathbf{Z}[X]$$

with $P_a(2^{13}) = a$ and $P_b(2^{13}) = b$.

Carry-less Calculation of the $2M$-bit Product in Signed 32-bit Radix-2^{13} Representation

The product polynomial $P(X) = P_a(X)P_b(X) = \sum_{j=0}^{190} p_j X^j$ corresponds to the carry-less product calculation of a and b as represented by $(a_j)_{j=0}^{95}$ and $(b_j)_{j=0}^{95}$, respectively. Its coefficients satisfy $|p_j| \leq 96 \cdot (2^{12})^2 < 2^{31}$, which allows computation modulo 2^{32}, resulting in a signed 32-bit radix-2^{13} representation $(p_j)_{j=0}^{190}$ of the product $a \cdot b = P(2^{13})$. If $M < 13 \cdot w$ with $w < 96$, the degree of $P(X)$ will be at most $2w - 2 < 190$, which leads to savings here and in the description below.

The polynomial $P(X)$ is calculated using three levels of Karatsuba multiplication [116] (but see Section 6.1.6 for the possibility to use more levels), resulting in 27 pairs of polynomials $(P_a^{(k)}(X), P_b^{(k)}(X))$ of degree ≤ 11, for $k = 1, 2, \ldots, 27$ (in the more general case where $M < u \cdot v - 2$ we would use $16 - u$ levels). This leads to 27 independent polynomial multiplications $Q^{(k)}(X) = P_a^{(k)}(X)P_b^{(k)}(X)$, done using carry-less schoolbook multiplications. The polynomial $P(X)$ is then obtained by carry-less additions and subtractions of the appropriate $Q^{(k)}(X)$'s.

Reduction Modulo N and Conversion to Radix-2^{32} Representation of the $2M$-bit Product

Given a signed 32-bit radix-2^{13} representation $(p_j)_{j=0}^{190}$ of the $2M$-bit product $a \cdot b$, regarded as the polynomial $P(X) = \sum_{j=0}^{190} p_j X^j$ with $P(2^{13}) = a \cdot b$, the radix-2^{32} representation $(c_i)_{i=0}^{38}$ of the M-bit number $c \equiv P(2^{13}) \bmod N$ is calculated. We use the following precomputed constants:

- $C_1 \equiv -2^{31} \cdot \sum_{j=0}^{190} 2^{13j} \bmod N,\ 0 \leq C_1 < N$. This constant allows, in a similar fashion as when doing modular addition and subtraction, to work with positive coefficients in step 1 below resulting in parallelization possibilities.

- Integers k_j, l_j and m_j such that

$$13j = m_j M + 32l_j + k_j$$
$$\text{with } 0 \leq 32l_j + k_j < M \text{ and } 0 \leq k_j < 32,$$

  for $0 \leq j < 191$. Note that $m_j \in \{0, 1, 2\}$ because $M > 827$ (and $M < 1246$). These constants are used to split the positive coefficients accordingly (see step 2 below).

Given these values, the following four steps are carried out, the correctness of which easily follows by inspection:

1. For $0 \leq j < 191$, compute $\tilde{p}_j = p_j + 2^{31}$ (this allows arbitrary parallelization), so that $0 \leq \tilde{p}_j < 2^{32}$. As a result

$$\left(\sum_{j=0}^{190} \tilde{p}_j \cdot 2^{13j}\right) + C_1 \equiv P(2^{13}) \bmod N.$$

2. For $0 \leq j < 191$, left shift $\tilde{p}_j$ over k_j bits and right shift $\tilde{p}_j$ over $32 - k_j$ bits, to obtain d_j, e_j such that

$$\tilde{p}_j \cdot 2^{13j} \equiv d_j \cdot 2^{32l_j} + e_j \cdot 2^{32(l_j+1)} \bmod N$$

   (this again allows arbitrary parallelization).

3. Let $v_0 = 0$. For $0 \leq i < 39$, let

$$u_i = \sum_{j \text{ s.t. } l_j = i} d_j + \sum_{j \text{ s.t. } l_j + 1 = i} e_j, \tag{6.1}$$

   (where the indices j can be precomputed) and compute

$$\tilde{c}_i = (v_i + u_i) \bmod 2^{32} \in \{0, 1, \ldots, 2^{32} - 1\},$$
$$v_{i+1} = \lfloor (v_i + u_i)/2^{32} \rfloor$$

   (this allows partial parallelization). Finally, compute $\tilde{c}_{39} = v_{39} + \sum_{j \text{ s.t. } l_j = 38} e_j$.

   Using Eq. (6.1), reduction modulo N is effected by disregarding m_j (since $2^{m_j M} \equiv 1 \pmod{2^M - 1}$) and grouping together identical d_j-values and identical e_j-values. As a result, $(\tilde{c}_i)_{i=0}^{39}$ is the radix-2^{32} representation of a number $\tilde{c}$ with $\tilde{c} + C_1 \equiv c \bmod N$.

4. Calculate $c \equiv \tilde{c} + C_1 \bmod N$. Although the numbers are slightly bigger ($\tilde{c}$ is one 32-bit limb too large), this calculation is in principle the same as regular addition modulo N.

6.1.5 Optimizations

Swapping Even for Odd Instructions

Modular arithmetic mostly relies on the SPE's arithmetic instructions, which are even pipeline instructions. Following the approach from [43, 157] one may replace an even instruction by one or more odd ones with the same effect. Although this may increase the latency for the functionality of each replaced even instruction and the number of instructions, balancing the counts of even and odd instructions often increases the throughput. This method was used throughout our implementation. Examples are sketched below.

Modular Squaring

When squaring polynomials of degree at most 11, half of the mixed products, i.e., $\frac{12^2-12}{2} = 66$ multiplications, can be saved by doubling their resulting 21 sums. Of these sums, the eleven for coefficients of odd degree can be doubled for free during the conversion to radix-2^{32}, by using for odd j precomputed integers $\tilde{k}_j$, $\tilde{l}_j$, and $\tilde{m}_j$ such that

$$\begin{gathered} 13j + 1 = \tilde{m}_j M + 32\tilde{l}_j + \tilde{k}_j \\ \text{with } 0 \leq 32\tilde{l}_j + \tilde{k}_j < M \text{ and } 0 \leq \tilde{k}_j < 32, \end{gathered}$$

instead of k_j, l_j, and m_j, as defined earlier. The ten remaining sums need to be doubled before they are added to the corresponding squared input coefficient. Let $V = \{v_0, v_1, v_2, v_3\}$ be a 128-bit vector and v_i 32-bit words. Doubling the values in V can using a single 4-way 32-bit shift instruction: $W = V \ll 1 = \{v_0 \ll 1, v_1 \ll 1, v_2 \ll 1, v_3 \ll 1\}$. However, a doubling can also be performed by four odd pipeline instructions.

- Shifting the full 128-bit quadword one bit to the left and store the result in V'. Now the most significant bit of the V'_i, for $i \in 1, 2, 3$, has been shifted into the least significant byte of V'_{i-1}.
- To correct this use the shuffle instruction to extract the least significant byte from each quadword V_i and store these in W (setting the remainder of the bytes to zero).
- Shift the full 128-bit quadword W one position to the left (now the least significant bytes are correct).
- Use the shuffle instruction to get the correct bytes from W and V' to construct the desired output.

Note that computing two doublings only six instructions are required since the second and third step can shuffle and shift the eight least significant bytes from both quadwords. The ten remaining doublings could thus be squeezed in the odd pipeline, including all load and storage overheads (all 21 doublings would not have fit in the odd pipeline). As a result, all doublings required for squaring can be computed at no extra cost by calculating this using odd instructions.

Conversion to Radix-2^{32}

The computation of d_j and e_j requires shifts by k_j and $32-k_j$, respectively, for $0 \le j < 191$, for a total of 382 even pipeline shift instructions. If $k_j \equiv 0 \bmod 8$, each shift can be replaced by a single odd pipeline byte reordering instruction (or by no instruction if $k_j = 0$). Shift counts bigger than eight can be replaced by three odd pipeline instructions.

M-Dependent Optimization

For $0 \le j < 191$ and most M we have $\sum_{j \text{ s.t. } l_j+1=i} e_j < 2^{32}$, since e_j is obtained by a right shift over $32-k_j > 0$ bits and the shift amounts usually differ. Thus, for such M the second summation in Eq. (6.1) does not generate carries.

We have written a program that generates SPE code for each value of M, with the applicable C_0, C_1, k_j, l_j, m_j, $\tilde{k}_j$, $\tilde{l}_j$, and $\tilde{m}_j$ hard-coded and including all optimizations mentioned so far. The resulting code thus depends on the value of M used, with slightly varying performance between different M-values. Representative instruction and cycle counts for 4-way SIMD multiplication and squaring modulo $2^{1193}-1$ on a single SPE are given in Table 6.1. Because $\frac{78}{144}\cdot 3\,905 \approx 2\,115$, the 2 130 cycles required for the calculation of the $Q^{(k)}$'s while squaring is very close to what one would expect based on the 3 905 cycles required for multiplication.

6.1.6 Further Speedups

Initial estimates indicated that the speed advantage of the radix-2^{32} additions would outweigh the disadvantage of the conversion (in Section 6.1.4) to signed radix-2^{13} representations required for the carry-less product calculation. Only after the code based on the methods described above had been used for about nine months (obtaining the results as reported in Section 6.2) and two further improvements had been developed, this issue was revisited. The two improvements, described in this section, apply to the first approach as well. The alternative version of the method from Section 6.1.4 that normalizes (and reduces) the signed 32-bit radix-2^{13} product to its signed radix-2^{13} representation (as opposed to converting and reducing the product to radix-2^{32} representation, as in Section 6.1.4) is presented in Section 6.1.7.

Using $C_1 \equiv 0 \bmod N$ in Section 6.1.4

Let $\gamma = 13\cdot 191+18-M$, $\beta = \lfloor \gamma/13 \rfloor$ and $\alpha = \gamma - 13\beta$. To get non-negative $\tilde{p}_j$'s in the first step of Section 6.1.4, it suffices to put $\tilde{p}_0 = p_0 + 2^{31}$, $\tilde{p}_j = p_j + 2^{31} - 2^{18}$ for $1 \le j < 191$, and next to replace $\tilde{p}_\beta$ by $\tilde{p}_\beta - 2^\alpha$ to make sure that the sum of all values added to $\sum_{j=0}^{190} p_j 2^{13j}$ telescopes to zero modulo N. Here we use that $p_j \ge -96(2^{12})(2^{12}-1) > -2^{31}+2^{19} > -2^{31}+2^{18}$ and that $-2^{31}+2^{19} > -2^{31}+2^{18}+2^\alpha$ (or $-2^{31}+2^{19} > -2^{31}+2^\alpha$ if $\beta = 0$). In this way C_1 in Section 6.1.4 is replaced by a value that is zero modulo N. This saves an addition (by C_1) in the final calculation of c in the fourth step of Section 6.1.4.

Table 6.1: SPE cycle counts for 4-way SIMD operations modulo $2^{1193}-1$. The first two rows of data refer to addition and subtraction related figures, separated on the left hand side and combined on the right hand side. The remaining rows of data refer to multiplication (on the left hand side) and squaring (on the right hand side) related figures. The measured number of cycles is indicated by **m**.

instructions		cycles			instructions		cycles	
even	odd		**m**		even	odd		**m**
$a+b$ or $a-b$					$a+b$ and $a-b$			
120	117	144	**180**	radix-2^{32}	222	180	235	**268**
301	296	332	**363**	signed radix-2^{13}	553	394	571	**645**
$a \cdot b$				**original, radix 2^{32} inputs and output (Section 6.1.4)**	a^2			
708	722	752		$P_a(X), P_b(X)$, and $P_a^{(k)}(X)$, $P_b^{(k)}(X)$ for $1 \le k \le 27$ / $P_a(X)$ and $P_a^{(k)}(X)$ for $1 \le k \le 27$	354	361	376	
3889	1137	3905		$Q^{(k)}(X)$ for $1 \le k \le 27$	2107	2055	2130	
1138	1078	1163		$P(X)$ and (d_j, e_j) for $0 \le j < 191$	1139	1086	1171	
906	907	936		$\tilde{c}_i$ for $0 \le i < 39$ and c	900	905	931	
6641	3844	6756	**6971**	total	4500	4407	4608	**4814**
$a \cdot b$				**signed radix-2^{13} inputs and output (Sections 6.1.6, 6.1.7)**	a^2			
3622	1510	3637		$P_a^{(k)}(X)$, $P_b^{(k)}(X)$, and $Q^{(k)}(X)$ for $1 \le k \le 27$	2220	1921	2243	
1292	1172	1308		$P(X)$, steps 1, 2 and part of steps 3, 4 of Section 6.1.7	1299	1264	1340	
544	508	568		Steps 5, 6 and remainder of steps 3, 4 of Section 6.1.7	544	508	568	
5458	3190	5513	**5666**	total	4063	3693	4151	**4306**

Karatsuba Multiplication with Multiply-and-Add

A more substantial improvement is obtained by noting that for 26 out of the 27 k-values in Section 6.1.4 the coefficients of the polynomials $P_a^{(k)}(X)$ and $P_b^{(k)}(X)$ are signed 15-bit integers. Therefore, for these k another level of Karatsuba multiplication can be used for the calculation of $Q^{(k)}(X)$, while taking advantage of the SPE's multiply-and-add instructions. Some details are described below.

Let e, e', f, f' be four polynomials of degree at most $n-1$. To multiply the two polynomials $e + e'X^n$ and $f + f'X^n$ of degree at most $2n-1$, calculate $g = e - e'$ and $h = f' - f$ using n subtractions each (note the asymmetry). Defining $ef = U + U'X^n$, $e'f' = V + V'X^n$ and $gh = W + W'X^n$, we have to calculate

$$(e + e'X^n)(f + f'X^n) = U + (U' + W + U + V)X^n + (V + W' + U' + V')X^{2n} + V'X^{3n}.$$

This is done by calculating (using multiply-and-add when relevant) U and U' in n^2 operations, next $U' + V$ and V' using another n^2 operations, $U' + V + U$ (n additions) and $U' + V + V'$ ($n-1$ additions), and finally $U' + V + U + W$ and $U' + V + V' + W'$ using n^2 operations.

In this way this final level of Karatsuba multiplication requires $3n^2 + 4n - 1$ operations. In our case this can be reduced to $3n^2 + 3n - 1$ since the computation of g and h are twice as fast using 8-way SIMD 16-bit subtractions. With $n = 6$ this becomes 125 operations for the calculation of each of the 26 $Q^{(k)}(X)$'s to which this applies; the 27th one can be done in 144 operations, for a total of 3 394 even instructions to calculate all $Q^{(k)}(X)$'s. For $n = 3$ we get $3n^2 + 3n - 1 = 35 < 6^2$, but the remaining parts of the 12-to-6-Karatsuba step take more than 20 operations, so more than $3 \times 35 + 20 = 125$ operations per $Q^{(k)}(X)$.

Improving the method from section 6.1.4 using Sections 6.1.6 and 6.1.6 would lead to a speedup of slightly less than 10% for modular multiplication and a much smaller speedup for modular squaring. We have not used this improvement as it led to only a small speedup of the ECM application. Instead we combined these improvements with the method presented in Section 6.1.7 below as it was expected (and turned out) to lead to a more substantial speedup for the ECM application.

6.1.7 Multiplication Modulo N using Signed Radix-2^{13}

Multiplication modulo N with inputs and output in signed radix-2^{13} representation (and thus relatively slow addition operations) is obtained from the description in Section 6.1.4 by omitting the conversion, keeping the polynomial multiplication in place (possibly improved with the Karatsuba multiplication), and by replacing the reduction by the reduction and normalization step described below.

Reduction Modulo N and Normalization to Signed Radix-2^{13} Representation of the $2M$-bit Product

Given a signed 32-bit radix-2^{13} representation $(p_j)_{j=0}^{190}$ of the $2M$-bit product $a \cdot b$, regarded as the polynomial $P(X) = \sum_{j=0}^{190} p_j X^j$ with $P(2^{13}) = a \cdot b$, the signed radix-2^{13} representation $(c_j)_{j=0}^{95}$ of the M-bit number $c \equiv P(2^{13}) \bmod N$ is calculated.

1. Compute $(\tilde{p}_j)_{j=0}^{190}$ as described in Section 6.1.6.

2. For $0 \le j < 96$ replace $\tilde{p}_j$ by $\tilde{p}_j + 2^{12}$. (All additions in steps 1 and 2 are combined at a total cost of 191 even addition instructions for steps 1 and 2.)

3. For $96 \le j < 191$ let p'_j and p''_j be words such that $\tilde{p}_j = p'_j + p''_j 2^{16}$ and $0 \le p'_j, p''_j < 2^{16}$, and replace p'_j by $p'_j 2^{k'_j}$ and p''_j by $p''_j 2^{k''_j}$ using odd instructions, where

$$13j = m'_j M + 13l'_j + k'_j \text{ and}$$

$$13j + 16 = m''_j M + 13l''_j + k''_j$$

with $0 \le 13l'_j + k'_j, 13l''_j + k''_j < M$ and $0 \le k'_j, k''_j < 13$.

4. For $96 \le j < 191$ replace $\tilde{p}_{l'_j}$ by $\tilde{p}_{l'_j} + p'_j$ and $\tilde{p}_{l''_j}$ by $\tilde{p}_{l''_j} + p''_j$ using a total of 190 even instructions. (No overflow occurs because $p'_j, p''_j \le 2^{28}$ and $p_j < (j+1)2^{24}$ for $0 \le j < 96$.)

5. Perform Step 3 of the addition-subtraction method in Section 6.1.3 with c (consisting of halfwords) replaced by $\tilde{p}$ (consisting of words). The carry τ can become as big as $2^{19} - 1$.

6. For $0 \le j < 96$ calculate the halfword $c_j = \tilde{p}_j - 2^{12}$.

Steps 1, 2, 3, 4, and 6 allow arbitrary parallelization. Step 3 and 4 perform the modular reduction and normalization. The resulting SPE clock cycle counts are listed in Table 6.1.

6.1.8 Comparison with other SPE Implementations

Because an SPE runs at 3.192GHz and six are available per PS3, it follows from Table 6.1 that a single PS3 can perform 13.5 (17.8) million multiplications (squarings) modulo $2^{1193} - 1$ per second. This compares to 182 million and 138 million multiplications modulo 192-bit and 224-bit special moduli, respectively, as reported for a single PS3 in [30] (see Chapter 3), i.e., less than an 11-fold slowdown for 5-fold bigger special moduli.

For generic moduli the same carry-less Karatsuba-based multiplication applies. The basic approach to the more cumbersome reduction would reduce our performance by a factor of at most three, but we expect we can do much better. Compared to the roughly 102 million modular multiplications for generic moduli in the 200-bit range, as reported for a single PS3 in [17], we would get at worst a 20-fold slowdown for 6-fold bigger generic moduli.

6.2 Application to ECM

Recall from Section 2.4.1 that each ECM trial consists of two stages, stage one with bound B_1, which is compute intensive but requires little memory, followed by a memory-hungry stage two with bound B_2. Depending on the number of trials and the two bounds, the probability can

be estimated that a factor up to a specific size, if present, will be found. To have probability at least $\frac{e-1}{e} \approx 0.632$ to find a factor of up to 65 decimal digits (when present), 24 000 ECM trials with $B_1 = 3 \cdot 10^9$ and $B_2 \approx 10^{14}$ (the default B_2 of GMP-ECM) suffice [209]. For the same bounds and success probability, 110 000 trials suffice to find a 70-digit factor (when present). Before our work the largest prime factor ever found using ECM had 68 decimal digits [206].

Using the GMP-ECM package [207, 209], with B_1 and B_2 as above, on a single core of a 2.2GHz Athlon 2427, stage one for an ECM trial for $2^M - 1$ with M around 1 200 takes on the order of six hours, stage two takes about one hour requiring many GBytes of RAM (for generic composites of comparable size each stage takes about twice as long; more precise timings are presented in Table 6.4 in Section 6.2.2 below). For each composite of the form $2^M - 1$ with $1\,000 \leq M \leq 1\,200$ this implies up to 20 core years for an ECM attempt to find a 65-digit factor, and up to 90 core years for a 70-digit one. This should be compared to an SNFS effort ranging from on the order or 70 ($M \approx 1\,000$) to several thousand ($M \approx 1\,200$) core years. Thus, the larger M, the harder we should first try with ECM, commensurate with the expected SNFS effort and the probability that a candidate has a small factor.

Stage one can easily be run in parallel in SIMD fashion for *any* number of trials. During a large scale ECM effort, overall throughput of trials is, within reason, a more important performance measure than latency per trial: for instance, being able to process four trials simultaneously in one day is better than processing (on the same platform) one trial every eight hours.

Rationale to use Cell processors for ECM on $2^M - 1$.

Factoring numbers of the form $2^M - 1$ is a "popular" activity [52] and hunting for relatively small factors is not hard given several freely available ECM packages. Nevertheless, given the efforts involved, we considered it likely that several of the unfactored composites $2^M - 1$ with $1000 \leq M \leq 1200$ have a factor that can be found more economically by ECM than by SNFS. Given our research interest in the ones that cannot (relatively) easily be factored by ECM, we decided on an ECM effort down our list of at least 20 candidates, aiming to find all factors of up to, roughly, 65 digits. Since it was meant to be a simple production run, we chose to use the off-the-shelf GMP-ECM package, because it is free, easy to use, has an excellent track-record, and can take advantage of the special form of the number $2^M - 1$. Other packages may be faster, but we were not familiar with them [16]. Notice, that if some small factors of $2^M - 1$ are known it is still faster to use the arithmetic modulo this Mersenne number than modulo the smaller composite.

The overall computation for these 20 candidates requires at least $20 \times 20 = 400$ core years and can in principle be done on regular server-clusters. But that would be a waste of resources, because about $\frac{6}{7}$th of the time is spent in stage one, which requires little memory thereby underutilizing the available RAM.

We also have access to a cluster of 215 PS3s, and thus to 215 Cell processors comprising a total of 1290 SPEs with little memory per SPE. It could therefore be more economical for us to use those SPEs to do all stage one calculations, and to do the relatively small stage

Table 6.2: SPE effort for 4-way SIMD stage one ECM trials for $N = 2^{1193} - 1$, $B_1 = 3 \cdot 10^9$ (where "cpc" = "cycles per call").

operation mod N	number of calls	radix-2^{32}		signed radix-2^{13}	
		cpc	hours	cpc	hours
$a \cdot b$	26 193 284 192	6971	15.89	5666	12.92
a^2	13 358 576 558	4814	5.60	4306	5.00
$a + b$, $a - b$	18 990 126 989	268	0.44	645	1.12
$a + b$	523 868 924	180	0.01		
$a - b$	523 868 924	180	0.01		
		total	21.95		19.05

two effort whenever servers with adequate RAM would otherwise be idle. To test this we ported stage one of GMP-ECM to the SPE, trying a variety of home-grown SPE-specific arithmetic packages (which were already known to outperform [108]). In the course of these early experiments we stumbled upon a 63-digit prime factor (of $2^{1187} - 1$). This showed that conducting a thorough ECM search indeed makes sense, and stimulated development of the much faster SPE-arithmetic modulo $2^M - 1$ described in Section 6.1.

It was not our goal to improve the ECM package that we put on top of our enhanced arithmetic. It is likely that improvements reported over GMP-ECM that are based on different elliptic curve arithmetic or representations, such as, for instance, described and implemented in [15,16], apply to our overall performance figures as well. See for a more detailed discussion Chapter 7.

ECM on the Cell Processor to Support (S)NFS

Although ECM factorizations have little cryptographic significance, this does not imply that ECM *performance* is cryptographically irrelevant as well. In [18], for instance, it is observed that high performance ECM implementations on relatively inexpensive devices (given their computational power, such as on graphics cards (GPUs)), may be helpful for future (S)NFS projects. A particularly memory-hungry step of (S)NFS, *sieving*, generates large quantities of fairly small (100- to 200-bit) composites that must be factored. That task requires little memory and is therefore best outsourced to cheap devices, so sieving is not interrupted and all resources are used in a cost-conscious fashion.

6.2.1 ECM on the Cell Applied to $2^M - 1$

Table 6.2 lists the numbers of modular arithmetic operations carried out by stage one of a single ECM trial with bound $B_1 = 3 \cdot 10^9$ when using GMP-ECM. When run on an SPE, four stage one trials are run simultaneously. With the operations from Section 6.1, their cycle counts (cf. Table 6.1), and the SPE's 3.192GHz clock speed, this leads to an estimated time of less than 22 hours on a single SPE to complete four stage one ECM trials with bound

$B_1 = 3 \cdot 10^9$ using our first approach from Section 6.1.4, and a more than 10% speedup when using the approach from Section 6.1.7 along with the improvements from Section 6.1.6. The measured wall-clock times are slightly larger than the estimates. For applications where additions play a more important role, the method from Section 6.1.4 may outperform the method from Section 6.1.7 (where both methods are enhanced using Section 6.1.6).

With six SPEs per Cell processor and 215 Cell processors in the PS3-cluster, $4 \times 6 \times 215 = 5160$ stage one ECM trials can be processed in less than 20 hours. With 24 000 trials, stage one for a 65-digit search takes less than four days; stage one for the 110 000 trials for a 70-digit search takes two and a half weeks. Using our multi-core adaptation of stage two of GMP-ECM, the corresponding stage two calculations (with $B_2 = 103\,971\,375\,307\,818$) take the same time when using 4 cores per node on a 56-node cluster (with two hexcore processors per node): each trial takes 15 minutes on 4 cores, using at most 16 GBytes of RAM. Thus, the efforts of the two clusters involved in our calculations are well matched.

After nine months of sustained calculations for several M-values (using the slower approach from Section 6.1.4), seven new factors were found, in the following order: a 63-digit factor for $M = 1187$, the 73-digit factor

$$1\,808\,422\,353\,177\,349\,564\,546\,512\,035\,512\,530\,001$$
$$279\,481\,259\,854\,248\,860\,454\,348\,989\,451\,026\,887$$

for $M = 1181$, another 73-digit factor,

$$1\,042\,816\,042\,941\,845\,750\,042\,952\,206\,680\,089\,794$$
$$415\,014\,668\,329\,850\,393\,031\,910\,483\,526\,456\,487,$$

for $M = 1163$, a 66-digit factor for $M = 1073$, a 63-digit factor for $M = 1051$, a 68-digit factor for $M = 1139$, and a 70-digit factor for $M = 1237$. The 241-bit, 73-digit prime factor of $2^{1181} - 1$ is the current ECM record, beating the previous record by 5 digits. The factor was found after somewhat more than 25 000 stage one trials at approximately the 8800th corresponding stage two trial, implying that we were quite lucky finding it (GMP-ECM [209] reports that finding a 73-decimal digit factor (if present) requires the computation of 259 058 curves given our B_1 and B_2 parameters). It was found for $\sigma = 4\,000\,027\,779$ (cf. [209]) with elliptic curve group order factoring into primes at most B_1 with the exception of one prime between B_1 and B_2:

$$2^4 \cdot 3^2 \cdot 13 \cdot 23 \cdot 61 \cdot 379 \cdot 13\,477 \cdot 272\,603 \cdot$$
$$12\,331\,747 \cdot 19\,481\,797 \cdot 125\,550\,349 \cdot 789\,142\,847 \cdot$$
$$1\,923\,401\,731 \cdot 10\,801\,302\,048\,203.$$

Less, but still considerable, luck was involved in finding the second 73-digit factor (a bit smaller at 240 bits): it was found after about 50 000 ECM trials for $\sigma = 3\,000\,085\,158$ and group order

$$2^2 \cdot 3^2 \cdot 5 \cdot 23 \cdot 1\,429 \cdot 28\,229 \cdot 139\,133 \cdot 249\,677 \cdot$$
$$389\,749 \cdot 15\,487\,861 \cdot 47\,501\,591 \cdot 111\,707\,179 \cdot$$
$$431\,421\,191 \cdot 13\,007\,798\,103\,359.$$

Table 6.3: Factors found of $2^M - 1$ using ECM on the Cell with the arithmetic described in Section 6.1.4 of this chapter, and with $B_1 = 3 \cdot 10^9$ and $B_2 \approx 10^{14}$.

M	targeted composite	completed number of trials		result
		stage one	stage two	
1051	c310	23 136	9 186	p63 · c248
1073	c281	24 504	1 460	p66 · p215
1139	c313	49 080	35 490	p68 · p246
1163	c318	50 152	47 768	p73 · p246
1181	c291	25 393	8 808	p73 · p218
1187	c266	15 089	9 860	p63 · p204
1237	c373	71 556	70 809	p70 · c303

So far our example number $2^{1193} - 1$, with known factor 121687, stubbornly resisted all ECM efforts to be factored after running 142 162 ECM trials on it. For the numbers $2^M - 1$ that we fail to factor using ECM, such as (so far) $M = 1193$, our efforts will result in a reasonable degree of confidence that they will not have a prime factor of 65 digits or less. Only for $M = 1051$ and $M = 1237$ did we find composite cofactors: for $M = 1051$ the attempt was continued and the 63-factor was indeed re-found where it could be predicted (once it had been found), but the c248 cofactor remained unfactored.

Table 6.3 lists all results obtained using the slower approach from Section 6.1.4, with ck and pk denoting a k-digit composite and prime, respectively. For exponents $M \in [1000, 1140]$ ($M \in [1141, 1200]$) not stated in Table 6.3 roughly 50 000 (100 000) ECM trials have been completed with bounds as above without finding a factor. Although we hope, during our continuing efforts using the faster approach from Sections 6.1.6 and 6.1.7, not to miss factors up to the 65-digit range, with ECM one can never be sure. Should we wish to find out, using SNFS is probably the best option.

Using the improved arithmetic we have so far found one factorization: for $M = 961$ we found that c254 = p61 · p193 after 1190 curves with $B_1 = 10^9$ and $B_2 = 25\,427\,965\,563\,016$. The improved arithmetic is also being used for numbers of the form $2^M + 1$ and several factors have already been found.

6.2.2 Comparison Between Cell and Regular Processors

A single PS3 processes 24 stage one ECM trials for $2^{1193} - 1$ in 19.2 hours. To put this number into perspective, we did the same computation using GMP-ECM 6.3 powered by GMP 5.0.1 [82] (both the latest versions at the time of writing) using all cores on a variety of processors, with optimal multiplication parameters obtained using the tune-up script, and taking advantage of the special Mersenne-arithmetic available in GMP-ECM. Table 6.4 lists the results. On a per-core basis, and accounting for the ratio in clock-speeds, our special 4-way SPE Mersenne arithmetic turns out to about $\frac{4}{3}$ times more effective than the regular Mersenne arithmetic from GMP-ECM 6.3 when run on Intel processors, despite the fact that

Table 6.4: Time to complete 24 stage one ECM trials for $2^{1193} - 1$ with $B_1 = 3 \cdot 10^9$.

processor	GHz	cores	hours	
			Mersenne	generic
Intel Core i7 920	2.67	4	46.28	83.52
Intel Core2 Quad Q9550	2.83	4	47.26	85.93
AMD Opteron 1381	2.50	4	33.78	58.46
AMD Opteron 6168	1.90	12	15.32	25.44
PlayStation 3	3.19	6	19.20	

the SPE does not have 64-bit or 32-bit integer multiplications. The lack of such multipliers is clearly to the SPE's disadvantage when comparing it to the AMD processor with its much faster (than Intel) integer multiplication. The more recent generations of processors, like the 12-core AMD Opteron, are catching up with the performance of the PS3.

6.3 Conclusion

For integers M in the range from 1000 to 1200 we presented our Cell processor implementation of multiplication of M-bit integers, processing 24 such multiplications in parallel on a single PlayStation 3 game console, and used it to obtain efficient multiplication modulo $2^M - 1$. The ideas underlying our implementation apply to many arithmetic contexts of cryptologic relevance. We focused on application to elliptic curve factoring, which led to the three largest ECM factors found so far[1].

[1] In January 2012 S. Wagstaff found a 72-decimal digit factor of $3^{713} - 1$ using ECM, moving our 70-decimal digit factor of $2^{1237} - 1$ to the fourth place.

Chapter 7

ECM at Work

Today, more than 25 years after its invention by Hendrik Lenstra Jr., the elliptic curve method [136] (ECM) remains the asymptotically fastest integer factorization method for finding relatively small prime factors of large integers. Although it is not the fastest general purpose integer factorization method, when factoring a composite integer $n = pq$ with $p \approx q \approx \sqrt{n}$ the number field sieve [133, 163] (NFS) is asymptotically faster, it has recently received a renewed research interest due to the discovery of an interesting normal form for elliptic curves introduced by Edwards [74].

In this chapter we optimize ECM by exploiting the fact that the same scalar is often used when computing the elliptic curve scalar multiplication (ECSM) in practice. This allows one to prepare particularly good addition chains for these fixed scalars. Our approach is inspired by the ideas used in the ECM implementation by Dixon and Lenstra [71] from 1992. In [71] the total cost to compute the ECSM, in terms of point duplications and point additions, is lowered by testing if the ECSM of small product of primes is cheaper (requires less point additions) than processing the primes one at a time (or all at once using a single large batch).

Inspired by this technique we generalize this idea; many billions of integers, which are constructed such that they can be computed using addition chains with a high duplication/addition ratio, are tested for smoothness and factored. Combining some of these integers using a greedy approach results in *more efficient* ECSM algorithms when the scalar is fixed (in terms of memory consumption and run-time performance).

Arithmetic using Edwards curves is faster than using Montgomery curves [146] (see Section 2.4), the approach used in most ECM implementations. In order to obtain this efficient arithmetic, when using Edwards curves, addition chains using large windowing methods are used (cf. [22] for a summary of these techniques). The memory (storage) requirement grows roughly linearly with the input parameters of ECM while it is an independent low constant value (14 residues modulo n) when using Montgomery curves.

We study two variants of our approach. A version which can compute the ECSM *without* requiring any additional memory, besides the in- and output point, and a more efficient version which requires a small amount of memory. These two versions are applied in two settings

Table 7.1: A summary of the cost of elliptic curve addition and duplication when using Montgomery or Edwards curves with different coordinate systems. The cost is expressed in modular multiplications (**M**), squarings (**S**) and multiplication by a curve constant (**d**). The notation $z_1 = 1$ indicates that the z-coordinate of one of the input points is equal to one (an affine point).

Projective coordinate system	Addition	Duplication
Montgomery	$4\mathbf{M} + 2\mathbf{S}$	$2\mathbf{M} + 2\mathbf{S} + 1\mathbf{d}$
Twisted Edwards	$10\mathbf{M} + 1\mathbf{S} + 2\mathbf{d}$	$3\mathbf{M} + 4\mathbf{S} + 1\mathbf{d}$
$a = -1$	$10\mathbf{M} + 1\mathbf{S} + 1\mathbf{d}$	$3\mathbf{M} + 4\mathbf{S}$
$a = -1$, $z_1 = 1$	$9\mathbf{M} + 1\mathbf{S} + 1\mathbf{d}$	$3\mathbf{M} + 3\mathbf{S}$
Extended Twisted Edwards	$9\mathbf{M} + 1\mathbf{d}$	$4\mathbf{M} + 4\mathbf{S} + 4\mathbf{d}$
$a = -1$	$8\mathbf{M}$	$4\mathbf{M} + 4\mathbf{S}$
$a = -1$, $z_1 = 1$	$7\mathbf{M}$	$4\mathbf{M} + 3\mathbf{S}$

of ECM: for large input parameters (when using ECM to find factors of large integers) and for small input parameters (which is of cryptanalytic interest). This makes our approach particularly interesting for environments where the memory (per thread) is constrained; e.g. graphics processing units.

7.1 ECM in Practice

Traditionally, ECM is implemented using Montgomery coordinates (see Section 2.4.1) and uses the various techniques described in [207]. The most-widely used ECM implementation is GMP-ECM [209] and this implementation, or modifications to it, is responsible for setting all recent ECM record factorizations. After the invention of Edwards curves (see Section 2.4) Bernstein, Birkner, Lange, and Peters explored the possibility to use these curves in the ECM setting [15]. A follow-up paper [14] discusses the usage of the "$a = -1$" twisted Edwards curves. The main reason to use Edwards curves is performance. The cost to implement elliptic curve addition and duplication when using projective Montgomery or (extended) twisted Edwards is summarized in Table 7.1. There are two implementations of ECM using Edwards curves available called GMP-EECM and EECM-MPFQ (see the web-page [16]). Both are designed to run on relatively small integers used in a cofactorization phase of the number field sieve (see Section 2.4.1).

Since different approaches are used to compute the elliptic curve scalar multiplication when using either Montgomery or Edwards curves the numbers in Table 7.1 do not show the *total* cost to compute the ECSM. Table 7.2 compares the required total number of modular multiplications and squarings required in GMP-ECM and EECM-MPFQ for different typical B_1 values used in ECM. These numbers show that using Edwards curves result in fewer multiplications and squarings. However, the required storage for GMP-ECM (Montgomery curves) is independent of B_1 while it grows almost linearly with the size of B_1 and is significantly higher, due to the use of width-w windowing methods, for EECM-MPFQ (Edwards curves, see [15, Table 4.1]).

Table 7.2: Performance comparison between GMP-ECM and EECM-MPFQ in terms of multiplications (**M**) and squarings (**S**) in the finite field. The number of residues modulo n (R) which needs to be kept in memory is shown for GMP-ECM and EECM-MPFQ in the $a = -1$ setting.

B1	GMP-ECM [209]			
	#**S**	#**M**	#**S**+#**M**	#R
256	1 066	2 025	3 091	14
512	2 200	4 210	6 410	14
1024	4 422	8 494	12 916	14
12 288	53 356	103 662	157 018	14
49 152	214 130	417 372	631 502	14
262 144	1 147 928	2 242 384	3 390 312	14
1 048 576	4 607 170	9 010 980	13 618 150	14

B1	EECM-MPFQ [15]					
		($a = 1$)		($a = -1$)		
	#**S**	#**M**	#**S**+#**M**	#**M**	#**S**+#**M**	#R
256	1 436	1 707	3 143	1 638	3 074	38
512	2 952	3 303	6 255	3 183	6 135	62
1 024	5 892	6 363	12 255	6 144	12 036	134
12 288	70 780	69 870	140 650	68 006	138 786	1 046
49 152	283 272	269 991	553 263	263 599	546 871	2 122
262 144	1 512 100	1 395 435	2 907 535	1 366 396	2 878 496	9 286
1 048 576	6 050 208	5 462 496	11 512 704	5 359 737	11 409 945	32 786

7.2 Elliptic Curve Constant Scalar Multiplication

Most of the addition/subtraction chain based approaches to compute the ECSM used in practice use the w-bit windowing technique, for some (optimal) width w to reduce the number of required elliptic curve additions. As discussed in Section 2.4.2, the total number of elliptic curve additions may be reduced significantly by using this approach but one also needs to store more points: 2^{w-1} when using sliding windows. In environments where the available memory per thread is low, these methods cannot be used or one is forced to settle for a suboptimal window size. A prime example of such a platform are graphics processing units (GPUs); e.g. one of the latest GPU architectures [154] (Fermi) shares 64KB fast shared memory per 32 processors and each processor typically time-shares multiple threads.

We investigate different approaches to lower the number of additions *and* the storage required to compute the scalar product. Our approach is inspired by the results reported by Dixon and Lenstra [71] in 1992. Suppose we have a scalar $k = \prod_{i=0}^{\ell-1} p_i$, where $\{p_0, p_1, \ldots, p_{\ell-1}\}$ is a list of primes less than B_1. Typically, the ECSM is implemented processing one such p_i at a time [207]. In [71] it is suggested to process the p_i in *batches*; i.e. multiply a batch of p_i's at a time such that the weight of the product $w(\prod_i p_i)$, the number of ones in the binary representation of $\prod_i p_i$, is (much) lower than the sum of the individual weights $\sum_i w(p_i)$. If this is the case then the number of required additions is reduced when using the straight forward

double-and-add approach. Moreover, the storage requirement is small since the usage of large windows is avoided. The search for such low-weight products is performed by partitioning, using a greedy search, the set of prime powers in subsets of cardinality of at most three (the cardinality three was chosen only from a practical point of view). This lowered the weight by approximately a factor three [71]. As an example the following triple is given

$$1028107 \cdot 1030639 \cdot 1097101 = 1162496086223388673$$
$$w(1028107) = 10, \quad w(1030639) = 16, \quad w(1097101) = 11,$$
$$w(1162496086223388673) = 8,$$

where the multiplication of primes of weights 10, 16, and 11 results in a integer of weight eight. The resulting composite integer can be computed using an addition chain requiring only seven additions and 60 duplications using the naive double-and-add algorithm.

In this section we explore different methods to find numbers which can be constructed using even better (higher) duplication/addition ratios. These methods do not aim to construct sequences by combining the different p_i (as in [71]) but use an opposite approach by factoring many integers which can be constructed using a relatively low number of additions and subsequently combining these integers such that all p_i's are used.

7.2.1 Addition/Subtraction Chains With Restrictions

In order to generate integers which can be computed using an addition/subtraction chain with a high duplication/addition ratio we need to construct and denote addition chains of a certain length m. In this section we define and explain the notation used to denote the addition/subtraction chains.

Let us first define the set of symbols $\mathscr{O}$, used to denote our chains, consisting of the symbols D, A, S used for duplication, addition and subtraction respectively:

$$\mathscr{O} = \{D_i \mid i \in \mathbf{Z}\} \cup \{A_{i,j} \mid i, j \in \mathbf{Z}, i > j\} \cup \{S_{i,j} \mid i, j \in \mathbf{Z}, i > j\},$$

where the subscripts indicate on which element we compute (this is made more precise later). The set of all m-tuples, ordered lists of m elements, of symbols in $\mathscr{O}$ with the restriction that no elements can be used which have not yet been generated is

$$\mathcal{O}_m = \{(o_{m-1}, \ldots, o_0) \in \mathscr{O}^m \mid o_k \in \{D_i \mid i \leq k\} \cup \{A_{i,j} \mid i \leq k\} \cup \{S_{i,j} \mid i \leq k\}, 0 \leq k < m\}.$$

In order to construct an addition/subtraction chain from such an m-tuple of symbols we define a function $\sigma_m : \mathscr{O} \times \mathbf{Z}^{m+1} \to \mathbf{Z}^{m+2}$ such that $(o, (t_m, \ldots, t_0 = 1)) \mapsto (t_{m+1}, t_m, \ldots, t_0 = 1)$ where

$$t_{m+1} = \begin{cases} 2t_i & \text{if } o = D_i, \\ t_i + t_j & \text{if } o = A_{i,j}, \\ t_i - t_j & \text{if } o = S_{i,j}. \end{cases}$$

Given an m-tuple of symbols $(o_{m-1}, \ldots, o_0) \in \mathcal{O}_m$ the $(m+1)$-tuple of integers associated to this addition/subtraction chain is

$$\sigma_{m-1}(o_{m-1}, \sigma_{m-2}(o_{m-2}, \ldots, \sigma_0(o_0, 1)\ldots)),$$

the *resulting integer* produced by this chain is t_m. As an example consider the 7-tuple of symbols $(S_{6,0}, D_5, D_4, A_{3,0}, D_2, D_1, D_0) \in \mathcal{O}_7$ which corresponds to the 8-tuple of integers in the addition/subtraction chain $(35, 36, 18, 9, 8, 4, 2, 1)$ computed as

$$\sigma_7(S_{6,0}, \sigma_6(D_5, \sigma_5(D_4, \sigma_4(A_{3,0}, \sigma_3(D_2, \sigma_2(D_1, \sigma_1(D_0, 1))))))).$$

The function σ is the correspondence between a tuple of symbols and the actual addition/subtraction chain. The example shows how to compute the resulting integer 35 using one subtraction, one addition and five duplications.

A duplication can always be assumed to apply to the previously generated element in σ_i (instead of duplicating any previous element), since one can reorder the symbols in the tuple such that duplication always occurs on the last element without changing the resulting integer t_{m+1}. In some cases this results in a shorter sequence when one duplicates the same element multiple times: e.g. the sequence $(A_{3,0}, D_0, D_0, D_0) \in \mathcal{O}_4$ which corresponds to the 5-tuple $(3, 2, 2, 2, 1)$ can also be computed using $(A_{1,0}, D_0) \in \mathcal{O}_2$ corresponding to the 3-tuple $(3, 2, 1)$. Hence, we change the definition of $\mathscr{O}$ to

$$\mathscr{O} = \{D\} \cup \{A_{i,j} \mid i, j \in \mathbf{Z}, i > j\} \cup \{S_{i,j} \mid i, j \in \mathbf{Z}, i > j\},$$

and the value of t_{m+1} in σ_m to

$$t_{m+1} = \begin{cases} 2t_m & \text{if } o = D, \\ t_i + t_j & \text{if } o = A_{i,j}, \\ t_i - t_j & \text{if } o = S_{i,j}, \end{cases}$$

to incorporate this change. Although the set of tuples $\mathcal{O}_m$ consists of the most generic type of addition/subtraction chains, a significant amount of tuples corresponds to chains which perform useless (unnecessary) computations. An example is computing the addition (and subtraction) of two previous values without using this result. To address this we define a more restricted set of tuples $\mathcal{P}_m \subset \mathcal{O}_m$ as

$$\mathcal{P}_m = \{(o_{m-1}, \ldots, o_0) \in \mathcal{O}_m \mid o_k \in \{D\} \cup \{A_{i,j} \mid i = k\} \cup \{S_{i,j} \mid i = k\}, 0 \leq k < m\}.$$

These additional restrictions ensure that, just as for the duplication, we only add or subtract to the last integer in the sequence to obtain the next one. Such chains are known as *Brauer chains* or *star addition chains* [96, Section C6].

In this setting we write A_j and S_j for $A_{i,j}$ and $S_{i,j}$, respectively, and $k > 0$ subsequent instances of D are denoted as D^k. The previous example can now be written as $S_0 D^2 A_0 D^3 \in \mathcal{P}_7$ by abusing the notation: omitting the brackets and comma's. In practice we would generate sequences of symbols such that a number of elliptic curve additions $\mathbf{A}$ and duplications $\mathbf{D}$ are fixed and look at sequences of symbols of length $m = \mathbf{A} + \mathbf{D}$ which use $\mathbf{A}$ times A_j or S_j and $\mathbf{D}$ times D. Different tuples might compute the same integer result. Using our example, the number 35 can be obtained with $\mathbf{D} = 5$ and $\mathbf{A} = 2$ in different ways

$$\begin{aligned} 35 &= (2^3 + 1) \cdot 2^2 - 1 && S_0 D^2 A_0 D^3 \in \mathcal{P}_7 \\ &= (2^4 + 1) \cdot 2 + 1 && A_0 D A_0 D^4 \in \mathcal{P}_7. \end{aligned}$$

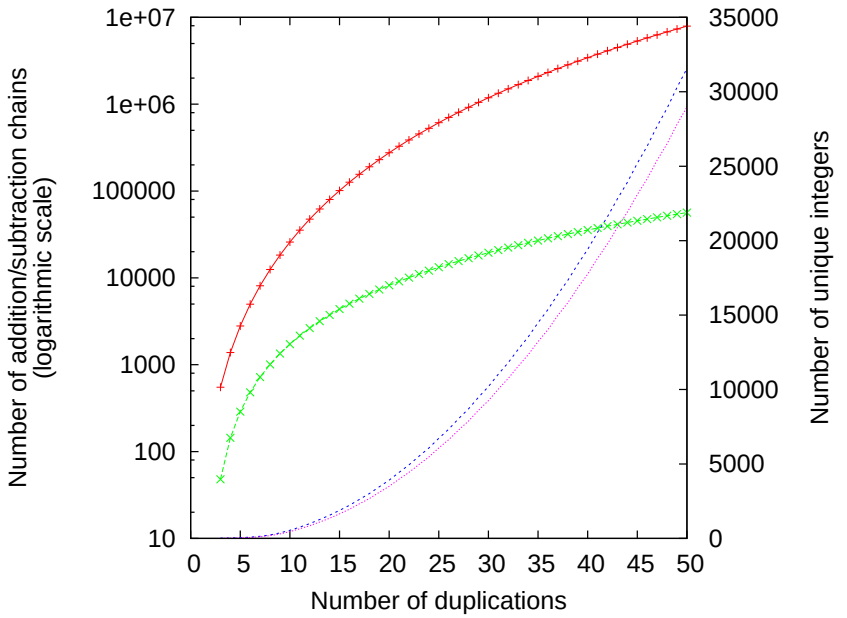

Figure 7.1: The two top lines on the left denote the number of generated addition/subtraction chains computing odd resulting integers with $\mathcal{P}_m$ (upper (red) line) and $\mathcal{Q}_m$ (lower (green) line) when fixing $\mathbf{A}$=3 and varying the number of duplications from one to fifty. The lower two lines show the number of *unique* integers corresponding to these chains where the upper line corresponds to $\mathcal{P}_m$.

7.2.2 Generating Addition/Subtraction Chains

We have defined some notation (the set of symbols $\mathscr{O}$), sets of m-tuples with different restrictions and how to connect these m-tuples to addition/subtraction chains with the help of σ_m. In this subsection we discuss how to efficiently generate the resulting integers t_{m+1} in two settings: a low-storage and no-storage approach.

The Low-Storage Setting

Let $\mathbf{A}$ be the number of elliptic curve additions and $\mathbf{D}$ the number of elliptic curve duplications (with $\mathbf{D} \geq \mathbf{A}$). The generation of *all* the tuples in $\mathcal{P}_m$, with $m = \mathbf{A} + \mathbf{D}$, results in many resulting integers t_{m+1} which are identical. Removing these duplicate values can be achieved by first generating and storing all the resulting integers and subsequently sorting and uniqueing this large dataset. To avoid storing all the resulting integers for a given pair $(\mathbf{A}, \mathbf{D})$, which requires a significant amount of storage as we will see later in this chapter,

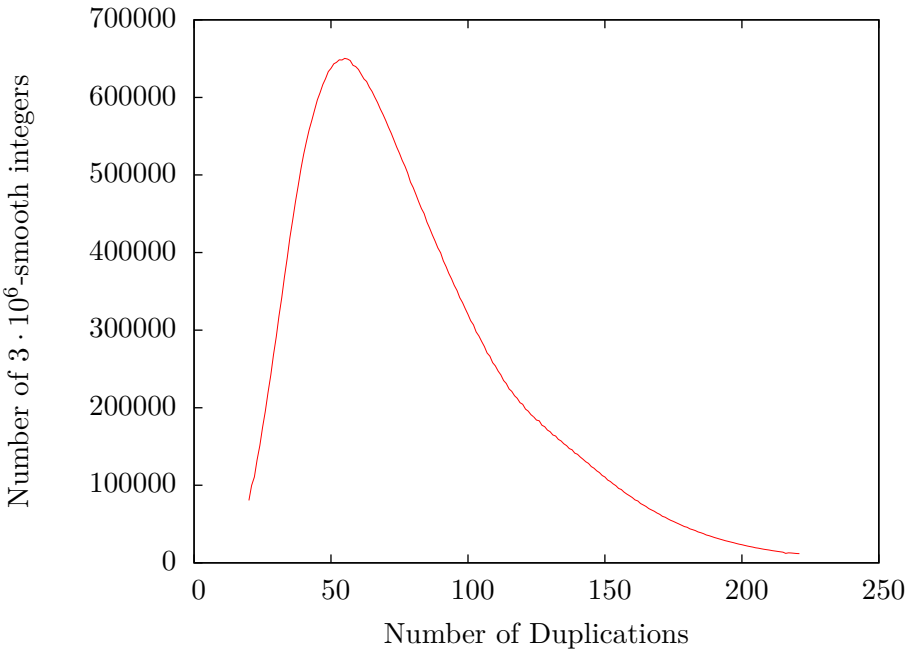

Figure 7.2: The number of unique $3 \cdot 10^6$-smooth integers produced by the low-storage addition/subtraction chains ($\mathcal{Q}$) when $\mathbf{A} = 4$ and $20 \leq \mathbf{D} \leq 221$.

and to avoid sorting this huge data set we define a more restricted set of rules $\mathcal{Q}_m$ as follows

$$\mathcal{Q}_m = \{(o_{m-1}, \ldots, o_0) \in \mathcal{P}_m \mid \quad o_k \in \{D\} \cup \{A_i, S_i \mid o_{k-1} = D \wedge (i = 0 \ \vee \ o_{i-1} \in \{A_\ell, S_\ell\})\}, \\ o_0 = D, o_{m-1} \in \{A_i, S_i\}, 0 < k < m-1\}.$$

We have $\mathcal{Q}_m \subset \mathcal{P}_m \subset \mathcal{O}_m$. The restrictions used in the definition of $\mathcal{Q}_m$ ensure the resulting integer is odd and only addition (or subtraction) of an odd number to the current (even) number is allowed. This approach significantly reduces the amount of chains which produce the same resulting integer at the cost of slightly reducing the number of unique integers produced.

To illustrate, Figure 7.1 shows the number of tuples generated by $\mathcal{P}_m$ and $\mathcal{Q}_m$ when using $\mathbf{A} = 3$ additions and $3 \leq \mathbf{D} \leq 50$ duplications resulting in odd integers. For $\mathbf{D} = 50$ the total number of tuples generated by $\mathcal{P}_{53}$ is more than 140 times higher compared to $\mathcal{Q}_{53}$ while the number of unique odd resulting integers is only 1.09 times higher.

All chains resulting from $\mathcal{Q}_m$ start with a duplication and end in either an addition or subtraction. Unless it is the last operation, an addition or subtraction is always followed by a duplication. Hence, there are $\binom{\mathbf{D}-1}{\mathbf{A}-1}$ ways to place the remaining $\mathbf{A}-1$ additions/subtractions and $\mathbf{D}-1$ duplications in the $m-2$ positions. Since every addition can be substituted by a subtraction the number of possibilities is multiplied by a factor $2^{\mathbf{A}}$. By definition of $\mathcal{Q}_m$, only an odd number (a result of an addition or subtraction) can be added or subtracted to an even number (a result of a duplication): this increases the number of possible tuples by a

factor of $\mathbf{A}!$. Hence, the total number of resulting integers produced by $\mathcal{Q}_m$, using $\mathbf{A}$ elliptic curve additions and $\mathbf{D}$ elliptic curve duplications, is

$$\binom{\mathbf{D}-1}{\mathbf{A}-1} \cdot \mathbf{A}! \cdot 2^{\mathbf{A}} = 2^{\mathbf{A}} \cdot \mathbf{A} \cdot \prod_{i=1}^{\mathbf{A}-1} (\mathbf{D} - \mathbf{A} + i).$$

The list of $(m+1)$ integers u_i corresponding to the m-tuple of symbols from $\mathcal{Q}_m$ can be efficiently generated recursively using

$$u_{i+1} = \begin{cases} 2u_i \\ u_i \pm u_j \quad \text{for } j < i \text{ and } u_i \equiv 0 \bmod 2 \not\equiv u_j \end{cases}$$

with $u_0 = 1$ and ensuring that the final operation is not a duplication (to make the resulting integer odd). Hence, the next integer in the sequence can always be obtained by duplication or adding a previous odd number u_j to the current even integer u_i. The number of times a different u_j is used for addition/subtraction determines the required amount of storage needed. In practice we generate all sequences using a fixed number of duplications and additions making sure that the resulting storage requirement is never too large. Figure 7.2 illustrates the number of unique $3 \cdot 10^6$-smooth integers produced when fixing $\mathbf{A} = 4$ and varying $20 \leq \mathbf{D} \leq 221$.

The No-Storage Setting

The second setting we consider is constructing chains which do not require any additional stored points, besides the in- and output (and possibly some auxiliary variables required to calculate the elliptic curve group operation). This means we are looking for resulting integers which can be computed using addition/subtraction chains which only use duplications and add or subtract the input point. Using our notation we can define the set of tuples $\mathcal{R}_m \subset \mathcal{Q}_m$ as

$$\mathcal{R}_m = \{(o_{m-1}, \ldots, o_0) \in \mathcal{Q}_m \mid o_k \in \{A_0, S_0, D\}, 0 \leq k < m\}.$$

All no-storage chains which can be constructed using $\mathbf{A}$ elliptic curve additions and $\mathbf{D}$ elliptic curve duplications are of the form

$$2^{\mathbf{D}} + \sum_{n_i} \pm 2^{n_i}, \quad \text{with } 0 = n_1 < n_2 < \ldots < n_i < \ldots < n_{\mathbf{A}} < \mathbf{D}. \tag{7.1}$$

We have $2^{\mathbf{D}}$ since the chain starts with a duplication and $n_1 = 0$ since we end with an addition or subtraction. In the other cases the first element $2^0 = 1$ is added or subtracted and subsequently duplicated the appropriate number of times. Using the same argument as in the low-storage setting the number of resulting integers generated by $\mathcal{R}_m$ using $\mathbf{A}$ additions and $\mathbf{D}$ duplications is $\binom{\mathbf{D}-1}{\mathbf{A}-1} \cdot 2^{\mathbf{A}}$. Hence, the no-storage setting produces a factor of $\mathbf{A}!$ fewer resulting integers compared to the low-storage setting.

7.2.3 Combining Addition/Subtraction Chains

Recall that, given a bound B_1, we want to multiply an elliptic curve point with the integer $k = \prod_{i=0}^{\ell-1} p_i = \text{lcm}(1, \ldots, B_1)$ where the product ranges over ℓ (not necessarily distinct) primes. Given the techniques from the previous section we can generate a list of integers $S = \{s_0, \ldots, s_i, \ldots, s_{m-1}\}$ which can be constructed using a known number of additions and duplications. Let add(s) denote the number of required elliptic curve additions (or subtractions) and dup(s) the number of elliptic curve duplications in the addition/subtraction chain to construct s. To find the set $S' \subset S$ of these integers such that $k = \prod_{s_i \in S'} s_i$ we do the following

1. Let $\hat{S} = \{s_i \mid s_i \in S \text{ and } s_i \text{ is } B_1\text{-smooth}\}$. For all $s_j \in \hat{S}$ store $(s_j, (\hat{s}_{j,0}, \ldots, \hat{s}_{j,t_j-1}))$ such that $s_j = \prod_{v=0}^{t_j-1} \hat{s}_{j,v}$ and $\hat{s}_{j,v}$ prime.

2. Among the smooth integers search for m' integers $s_j \in \hat{S}$ such that the prime divisors $\hat{s}_{j,v}$ of these m' integers exactly match all prime divisors of k (or match a significant amount of prime divisors of k). Let $S' = \{s_0, \ldots, s_u, \ldots, s_{m'-1}\}$ such that

$$\prod_{u=0}^{m'-1} s_u = \prod_{u=0}^{m'-1} \prod_{v=0}^{t_u-1} \hat{s}_{u,v} = k = \prod_{i=0}^{\ell-1} p_i = \text{lcm}(1, \ldots, B_1).$$

   One of the main search criteria is that $\sum_{u=0}^{m'-1} \text{add}(s_u)$ is low.

The meaning of "low" is still undefined. Ideally we aim to lower the cost, in terms of elliptic curve additions, of the different addition chains to construct the s_u compared to the cost of the addition chain to construct k using more advanced (e.g. signed sliding window) techniques (denoted by a not well-defined add'). Hence, we hope to find s_u's such that

$$\sum_{u=0}^{m'-1} \text{add}(s_u) = \sum_{u=0}^{m'-1} \text{add}\left(\prod_{v=0}^{t_u-1} \hat{s}_{u,v}\right) < \text{add'}\left(\prod_{u=0}^{m'-1} \prod_{v=0}^{t_u-1} \hat{s}_{u,v}\right) = \text{add'}(k).$$

Testing a large list of numbers for B_1-smoothness and, if this is the case, outputting the prime factorization, can be done using the optimized test for divisibility by small primes as introduced in [81, Section 4]. The main idea is to first build the product $k = \prod_{i=0}^{\ell-1} p_i = \text{lcm}(1, \ldots, B_1)$ using a binary tree. For a fixed B_1 this has to be done only once. Next, the B_1-smooth s_i are detected by removing all prime factors using a remainder tree (see for the exact algorithm [81]).

Finding the optimal set S', which results in the minimum number of elliptic curve additions in the addition chains, is in general a difficult problem. We choose to use a greedy approach which results in satisfactory results. Select an integer $s_j = \prod_{v=0}^{t_j-1} \hat{s}_{j,v}$ such that all the prime divisors $\hat{s}_{j,v}$ are still needed (i.e. $s_j \mid k$) and the addition/subtraction chain for s_j is good: the ratio dup(s_j)/add(s_j) is high. Once such an s_j has been found the list of primes we are searching for is updated (replace k with k/s_j) and this greedy approach is repeated.

A refinement to this approach is to also take the size of the prime factors $\hat{s}_{j,v}$ into account. A strategy could be to first collect B_1-smooth integers with only large prime divisors; since the majority of the prime powers dividing k are large. The idea is to attach a score to a B_1-smooth integer given its prime factorization with respect to the currently unmatched prime factors in k. Given the current ℓ unmatched primes in $k = \prod_{i=0}^{\ell-1} p_i$ the ratio of j-bit primes is defined as

$$a_j(p_0, \ldots, p_{\ell-1}) := \frac{\#\{i \mid \lceil \log_2(p_i) \rceil = j, 0 \leq i < \ell - 1\}}{\ell},$$

where $1 \leq j \leq \lceil \log_2(B_1) \rceil$. Next the *score* of s_i given k is defined as

$$\text{score}\left(s_i = \prod_{j=0}^{u-1} \hat{s}_{i,j}, k = \prod_{i=0}^{\ell-1} p_i\right) = \sum_{h=1}^{\lceil \log_2(B_1) \rceil} \frac{a_h(\hat{s}_{i,0}, \ldots, \hat{s}_{i,u-1})}{a_h(p_0, \ldots, p_{\ell-1})}$$

for the non-zero $a_h(p_0, \ldots, p_{\ell-1})$. The higher the score the more small prime divisors are likely to be present. In general, for a given ratio, we select the integers which have a low score.

To illustrate, consider $B_1 = 1024$. Initially, the different a_i are

$$\begin{array}{lll} a_2 = 0.032 & a_3 = 0.037 & a_4 = 0.021 \\ a_5 = 0.053 & a_6 = 0.037 & a_7 = 0.069 \\ a_8 = 0.122 & a_9 = 0.229 & a_{10} = 0.399 \end{array}$$

(with $\sum_{i=2}^{10} a_i = 1$). Almost 40 percent of all the primes fall in the largest (10-bit) category. An example of a low score-integer is

$$11529215054666795009 = 743 \cdot 719 \cdot 677 \cdot 461 \cdot 457 \cdot 449 \cdot 337$$

where the size of the smallest prime is 9-bit, the score is 3.57 and this integer can be computed using 63 duplications and five additions as

$$A_0 D^{11} A_0 D^{12} A_0 D^{10} A_0 D^{28} A_0 D^2 \in \mathcal{R}_{68}.$$

On the other hand, an example of a high-score integer, consisting of mainly small primes, is

$$1048575 = 41 \cdot 31 \cdot 11 \cdot 5^2 \cdot 3,$$

its score is significant higher (29.62) and it can be computed with 20 duplications and a single subtraction as $S_0 D^{20} \in \mathcal{R}_{21}$.

This approach is outlined in Algorithm 15. Note that the values of a_i need to be recalculated after prime factors have been removed from the list corresponding to k. In Algorithm 15 this is done after the while-loop in lines 11-14 when the new scores are computed. In practice one could modify the running condition of this while-loop from ($s_i \mid k$ and $i < j$) to ($s_i \mid k$ and $i < j/c$) for some $0 < c \in \mathbf{Z}$ to ensure more frequent updating of the a_i.

Algorithm 15 Given a bound B_1 and a set of B_1-smooth integers $\{s_0, \ldots, s_{\ell-1}\}$, which can be computed with an addition/subtraction chain using $\text{add}(s_i)$ and $\text{dup}(s_i)$ elliptic curve additions and duplications respectively, together with the prime factorization of these integers ($s_i = \prod_j \hat{s}_{i,j}$) the algorithm attempts to output triples $(s_j, \text{add}(s_j), \text{dup}(s_j))$ such that $\text{lcm}(1, \ldots, B_1)/\prod_j s_j$ is small. This algorithm considers scores $\leq s_{\text{thres}}$ only and combines integers s_i for which $\dfrac{\text{dup}(s_i)}{\text{add}(s_i)} \geq r$ where r starts at r_h and is decreased until r_l.

Input:
- Bound $B_1 \in \mathbf{Z}$,
- Set of integers $\{s_0, \ldots, s_{\ell-1}\}$ with $s_i = \prod_j \hat{s}_{i,j}$ for $\hat{s}_{i,j}$ prime and $0 \leq i < \ell$,
- Upper- and lower bound on the duplication/addition ratio: r_h and r_l
- A threshold value for the score: s_{thres}

Output: Output triples $(p_i, \text{add}(p_i), \text{dup}(p_i))$ such that $\prod_i p_i = \text{lcm}(1, \ldots, B_1)$

1. $k \leftarrow \text{lcm}(1, \ldots, B_1)$
2. **for** $r = r_h$ to r_l **do**
3. found $\leftarrow$ true
4. **while** found=true **do**
5. found $\leftarrow$ false, $j \leftarrow 0$
6. **for** $0 \leq i < \ell$ **do**
7. **if** $s_i \mid k$ and $\dfrac{\text{dup}(s_i)}{\text{add}(s_i)} \geq r$ and $\text{score}(s_i, k) \leq s_{\text{thres}}$ **then**
8. $\text{score}_j \leftarrow (\text{score}(s_i, k), s_i)$, j++
9. sort score_i for $0 \leq i < j$ with respect to $\text{score}(s_i)$ and the s_i's accordingly
10. $i = 0$
11. **while** $s_i \mid k$ and $i < j$ **do**
12. output $(s_i, \text{add}(s_i), \text{dup}(s_i))$
13. /* Remove the prime divisor of $s_i = \prod_j \hat{s}_{i,j}$ from k */
14. $k \leftarrow \frac{k}{s_i}$, found $\leftarrow$ true, i++
15. output $(k, \text{add}(k), \text{dup}(k))$

A Randomized Variant

In the current state, Algorithm 15 returns a single solution given a set of input parameters. To increase the amount of different results, and hereby hopefully improving these results, we randomize the selection process of the integer with the best score in line 12 of Algorithm 15. With probability $x \in \mathbf{R}$ $(0 < x < 1)$ select the current s_i or, with probability $1 - x$, skip this s_i and repeat this procedure for the next integer s_{i+1}. If $i + 1 \geq j$, i.e. we have reached the end of the list, one could either end the while-loop or select the best score which was skipped.

Table 7.3: The top table shows the number of integers generated which addition/subtraction chain using **A** and **D** elliptic curve additions and duplications respectively. All these integers were tested for $2.9 \cdot 10^9$-smoothness and, if smooth, the prime divisors are stored. The **bold** ranges indicate that 2^{31} random integers per single **A**, **D** combination were tested for smoothness instead of the full range. The bottom table shows the number of unique B_1-smooth integers in the no-storage and low-storage setting for different values of B_1.

No-storage setting			Low-storage setting		
A	**D**	#smoothness tests	**A**	**D**	#smoothness tests
1	$5 - 200$	$3.920 \cdot 10^2$	1	$5 - 250$	$4.920 \cdot 10^2$
2	$10 - 200$	$7.946 \cdot 10^4$	2	$10 - 250$	$2.487 \cdot 10^5$
3	$15 - 200$	$1.050 \cdot 10^7$	3	$15 - 250$	$1.235 \cdot 10^8$
4	$20 - 200$	$1.035 \cdot 10^9$	4	$20 - 250$	$6.101 \cdot 10^{10}$
5	$25 - 200$	$8.114 \cdot 10^{10}$	5	$25 - 153$	$2.511 \cdot 10^{12}$
			5	**$154 - 220$**	**$1.439 \cdot 10^{11}$**
6	$30 - 124$	$2.858 \cdot 10^{11}$	**6**	**$60 - 176$**	**$2.513 \cdot 10^{11}$**
7	$35 - 55$	$2.529 \cdot 10^{10}$			
Total		$3.932 \cdot 10^{11}$	Total		$2.967 \cdot 10^{12}$

B_1	No-Storage	Low-Storage
256	$2.412 \cdot 10^5$	$9.012 \cdot 10^6$
512	$1.442 \cdot 10^6$	$3.013 \cdot 10^7$
1 024	$5.466 \cdot 10^6$	$7.271 \cdot 10^7$
12 288	$1.149 \cdot 10^8$	$5.711 \cdot 10^8$
49 152	$3.152 \cdot 10^8$	$1.250 \cdot 10^9$
262 144	$7.757 \cdot 10^8$	$2.889 \cdot 10^9$
1 048 576	$1.380 \cdot 10^9$	$5.121 \cdot 10^9$
3 000 000	$1.991 \cdot 10^9$	$7.271 \cdot 10^9$
2 900 000 000	$1.054 \cdot 10^{10}$	$3.930 \cdot 10^{10}$

Combining the Remaining Primes

After Algorithm 15 finishes it returns (in line 15) k: the product of remaining unmatched prime factors. The associated cost for this addition chain is calculated assuming a double-and-add algorithm (see Chapter 2) is used. To lower the number of additions required, if the number of primes in this list is not too high, we use similar techniques as described in [71]. We use a brute-force program which calculates the cost of the addition chains when multiplying n of these prime divisors (of k) for $1 \leq n \leq 5$. These costs are sorted and using a greedy approach the best ones (lowest addition cost) are selected.

7.2.4 Additional Multiplications

The fastest arithmetic for Edwards curves is due to Hisil et al. [105] (see Section 2.4). They propose to use extended twisted Edwards coordinates, which are twisted Edwards coordinates

plus an auxiliary coordinate. This allows faster addition but slower duplication. Using a mixing technique, by switching between extended twisted Edwards and regular twisted Edwards, the overall cost for scalar multiplication is reduced [105]. This is realized by performing the duplications using the cheaper regular twisted Edwards coordinates when a duplication is followed by a duplication. When an addition is required after a duplication one can use the duplication formula in the extended twisted Edwards coordinates (which does not need the auxiliary coordinate as input) at the cost of an extra multiplication to compute the auxiliary coordinate of the result. Next, the fast addition is performed in extended twisted Edwards coordinates; one multiplication (to compute the auxiliary coordinate of the output) can be saved, cancelling the extra multiplication used when doubling, since a duplication is always performed after an addition in ECSM-algorithms. This approach assumes that both inputs of the elliptic curve addition are in extended twisted Edwards coordinates. This is the case for simple double-and-add algorithms and (signed) windowing algorithms where the computation of the auxiliary coordinates of the lookup table are a minor overhead.

In both our settings, the low- and no-storage, this does not hold. Converting a point from twisted Edwards coordinates to extended twisted Edwards coordinates requires a single multiplication. The computation of the large elliptic curve scalar product is done by processing batches of prime products (the s_i) at a time. All the additions or subtractions in the addition/subtraction chain to compute s_i require that the points are in extended twisted Edwards coordinates. When needed, the odd intermediate results are stored in extended twisted Edwards coordinates at a cost of a single additional multiplication. The cost of computing a low-storage addition/subtraction chain $(o_{m-1}, \ldots, o_0) \in \mathcal{Q}_m$ is increased by x multiplications, where $x = \#\{i \mid o_i \in \{A_j, S_j\}, 0 \leq i < m\}$; i.e. the unique number of indices used in the additions and subtractions. This increases the cost of no-storage chains by $\#\{\text{addition chains used}\} - 2$ multiplications (since $x = 1$ for almost all s_i): we can save one multiplication due to the powers of 2 (which are EC-addition free) and the other multiplication is saved if we assume that the input point is already in extended twisted Edwards coordinates. In the low-storage setting this number of additional multiplications might be higher.

7.3 Results

When fixing the number of additions and duplication one can generate all the possible resulting integers which can be constructed using an addition/subtraction chain as described in the previous section. Table 7.3 summarizes the ranges we have covered showing that we have tested more than 10^{12} integers for $2.9 \cdot 10^9$-smoothness. The bold ranges in the low-storage setting indicate that 2^{31} random integers resulting from an addition/subtraction chain per single **A**, **D** combination have been tested for $2.9 \cdot 10^9$-smoothness (instead of the full range). We separated our data-set in two: one part for the no-storage setting and both parts to be used in the low-storage setting. Table 7.3 also summarizes the number of integers which passed the B_1-smoothness test for varying B_1-parameters. Let us provide some information to give an idea about the effort required to test these numbers for smoothness. The smooth-

Table 7.4: Example of the best addition chain found for $B1 = 256$ in the no-storage setting.

#D	#A	product	addition chain
11	1	$89 \cdot 23$	S_0D^{11}
14	2	$197 \cdot 83$	$S_0D^5S_0D^9$
15	2	$193 \cdot 191$	$S_0D^{12}A_0D^3$
15	2	$199 \cdot 19 \cdot 13$	$A_0D^{14}A_0D^1$
18	1	$109 \cdot 37 \cdot 13 \cdot 5$	A_0D^{18}
19	2	$157 \cdot 53 \cdot 7 \cdot 3 \cdot 3$	$S_0D^6S_0D^{13}$
21	3	$223 \cdot 137 \cdot 103$	$A_0D^{10}A_0D^{10}A_0D^1$
23	3	$179 \cdot 149 \cdot 61 \cdot 5$	$S_0D^{13}A_0D^5S_0D^5$
28	1	$127 \cdot 113 \cdot 43 \cdot 29 \cdot 5 \cdot 3$	S_0D^{28}
30	3	$181 \cdot 173 \cdot 167 \cdot 11 \cdot 7 \cdot 3$	$A_0D^{11}A_0D^{16}A_0D^3$
33	5	$211 \cdot 73 \cdot 67 \cdot 59 \cdot 47 \cdot 3$	$S_0D^6A_0D^2A_0D^{11}S_0D^3S_0D^{11}$
36	4	$241 \cdot 131 \cdot 101 \cdot 79 \cdot 31 \cdot 11$	$A_0D^2A_0D^{16}A_0D^{16}A_0D^2$
41	4	$233 \cdot 229 \cdot 163 \cdot 139 \cdot 107 \cdot 17$	$S_0D^9S_0D^4S_0D^{11}S_0D^{17}$
49	5	$251 \cdot 239 \cdot 227 \cdot 151 \cdot 97 \cdot 71 \cdot 41$	$S_0D^3S_0D^{29}A_0D^4A_0D^8A_0D^5$
8	0	2^8	D^8
361	38	**Total**	

ness testing implementation requires (when using $B_1 = 2.9 \cdot 10^9$) at most 4.6GB of memory which is shared among the 8 cores of a Intel Xeon E5430 (2.66GHz) which compute on the product tree in parallel. The smoothness computations ran on 5 such nodes (40 cores) in parallel for more than half a year and one of these nodes was occasionally used for the combining experiments (using the approach as outlined in Algorithm 15). The run-time of the greedy approach to combine the chains varies from seconds (for the low B_1 values) to almost a day for the large B_1 values for multiple runs. For these large B_1 values most of the time is consumed by reading the factorization data from disk, once this has been put in memory multiple runs (using the probabilistic version) can be performed quickly.

Table 7.4 shows an example for $B_1 = 256$ in the no-storage setting. All the prime powers $p^e \leq 256$ with p prime, $e \in \mathbf{Z}$ such that $p^{e+1} > 256$ are used. The total cost, in terms of modular multiplications and squarings, for these 15 addition chains is $361 \times (3\mathbf{M} + 4\mathbf{S}) + 38 \times 8\mathbf{M} + 13\mathbf{M} = 1\,444\mathbf{S} + 1\,400\mathbf{M}$ where the 13 additional multiplications are due to adding or subtracting the input point in all except the first and last chain in Table 7.4. Only additions or subtractions with the input point are performed: no storage besides the in- and output is required.

Table 7.5 shows the results obtained using Algorithm 15 on our dataset (see Table 7.3). The memory required is expressed in the number of residues (R), integers modulo n, which need to be kept in memory. In the setting of EECM-MPFQ [15] we assume that only the input point needs to be kept in memory while we assume that two points (the input point and the current active point) are required in the no- and low-storage setting. The implementation of the elliptic curve group operation is assumed to require at most two auxiliary variable

Table 7.5: The number of modular multiplications (**M**) and squarings (**S**) required to calculate the elliptic curve additions (**A**) and duplications (**D**) for various B_1 when factoring an integer n with ECM. The memory required is expressed as the number of residues (R), integers modulo n, which are kept in memory.

Cost \ B_1	256	512	1024	12 288	49 152	262 144
	EECM-MPFQ [15]					
#**M**	1 608	3 138	6 116	67 693	260 372	1 351 268
#**S**	1 436	2 952	5 892	70 780	283 272	1 512 100
#**M** + #**S**	3 044	6 090	12 008	138 473	543 644	2 863 368
A	69	120	215	1 864	6 392	29 039
D	359	738	1 473	17 695	70 818	378 025
$\#R$	30	48	102	786	1 593	6 966
	No Storage Setting					
#**M**	1 400	2 842	5 596	65 873	262 343	1 389 078
#**S**	1 444	2 964	5 912	70 768	283 168	1 511 428
#**M** + #**S**	2 844	5 806	11 508	136 641	545 511	2 900 506
A	38	75	141	1 564	6 113	31 280
D	361	741	1 478	17 692	70 792	377 857
$\#R$	10	10	10	10	10	10
	Low Storage Setting					
#**M**	1 383	2 776	5 481	64 634	255 852	1 354 052
#**S**	1 448	2 964	5 908	70 740	283 056	1 510 796
#**M** + #**S**	2 831	5 740	11 389	135 374	538 908	2 864 848
A	35	65	124	1 366	5 127	25 956
D	362	741	1 477	17 685	70 764	377 699
$\#R$	22	22	22	26	26	26

(residues). Hence, the no-storage setting requires memory for $2 \times 4 + 2 = 10$ residues modulo n.

Note that the performance results for EECM-MPFQ presented in Table 7.5 differ from the ones in Table 7.2. The numbers in Table 7.2 are the real performance numbers obtained when running the EECM-MPFQ software. The improved numbers in Table 7.5 are a lowerbound when a different approach, involving inversions, is used (see also [22, Section 4]). The idea is to normalize the precomputed points to their affine representation. This has two advantages: it reduces the memory cost since three out of the four coordinates have to be stored (when using extended twisted Edwards coordinates) and faster elliptic curve arithmetic can be used (see Table 7.1). This normalization costs inversions, which are expensive, but this cost is not incorporated in the results from Table 7.5. In more detail, one can proceed as follows. For the precomputation cost we assume that the input is doubled and this result is normalized (at the cost of an inversion). Next, the other precomputations (the odd multiples) can be computed using the faster elliptic curve addition formula (since one of the inputs has its

z-coordinate equal to one). These points are normalized as well using Montgomery's simultaneous inversion [146] (see Section 4.4); the inversions are traded for three multiplications and normalizing the x-, y- and t-coordinate cost another three multiplications (and the cost for the single inversion is again not considered). Hence, the total cost to compute the ECSM, given v precomputed points, **A** elliptic curve additions and **D** elliptic curve duplications, is roughly $((7+6)v + 7\mathbf{A} + 3\mathbf{D})$ multiplications and $4\mathbf{D}$ squarings. This approach will most likely be faster (when considering the cost for the inversions) for the large B_1 values. For the small B_1 ($< 1\,204$) the cost of the inversion might outweight the advantages. Nevertheless, we use these optimistic figures (in terms of storage and performance) to compare against.

The low-storage setting requires at most additional storage for four points (see Table 7.5). Which is more than the no-storage setting but significantly less compared to the approach described in [15]. For small B_1-values the number of multiplications and squarings is significant less compared to the windowing methods. For instance, when $B_1 = 256$ the number of multiplications and squarings using addition chains is 0.93 (0.93) times the effort required when using windowing based methods while reducing the memory by a factor 1.4 (3.0) when using the low-storage (no-storage) approach. The smaller B_1 values (256, 512 and 1024) are typical parameters used in the cofactorization step of the NFS. The larger B_1 values are used for finding factors of large composite integers (where $B_1 = 12\,288$ corresponds to searching for 20 decimal digit factors and $B_1 = 3\,000\,000$ to 40 decimal digit factors).

The performance difference deteriorates when the B_1-value increases. When $B_1 = 49\,152$ ($B_1 = 262\,144$) the performance of the no-storage setting is worse by a factor 1.003 (1.013) compared to the windowing based methods used in [15]. But since the no-storage setting uses only 0.006 (0.001) times the amount of storage this approach is to be preferred in settings where there is not much memory or when the access to this memory is slow. When comparing the no-storage setting to GMP-ECM, which uses Montgomery curves, less memory is required while 0.864 (0.856) times the amount of modular multiplications and squarings used in GMP-ECM need to be computed when using $B_1 = 49\,152$ ($B_1 = 262\,144$).

7.4 Conclusion

Using the relatively new Edwards curves combined with the fast arithmetic when using the extended twisted Edwards coordinates is faster than using Montgomery curves in the setting of ECM. This speed-up comes at a price, as the memory requirement grows roughly linearly with the size of B_1 when using Edwards curves. We have presented techniques, inspired by the approach from Dixon and Lenstra, which use the fact that the same B_1-parameter is often used in practice, allowing one to perform some precomputations. We tested over 10^{12} integers, which resulted from additions/subtractions chains with a low addition/duplication ratio, for smoothness. Using a greedy approach these integers were combined for different popular choices of B_1. Our results show that for small B_1 values, we are both faster and require less memory compared to the current state-of-the-art. For large B_1 values the performance results are similar while we only require a fraction of the memory used by the algorithms in the current Edwards ECM implementations. This makes our approach extremely suitable for memory-constrained parallel architectures like GPUs.

CURRICULUM VITAE

PERSONAL INFORMATION

Name: Joppe Willem Bos
E-mail: joppe.bos@epfl.ch

Date of Birth: 4 November 1982
Nationality: Dutch

EDUCATION

- École Polytechnique Fédérale de Lausanne (Swiss Federal Institute of Technology), Lausanne, Switzerland
  2007 – February 2012
  PhD Student at the Laboratory for Cryptologic Algorithms (LACAL) under supervision of Prof. A. K. Lenstra.
  Thesis title: *On the Cryptanalysis of Public-Key Cryptography*
- Microsoft Research, Redmond, USA
  August 2011 – October 2011
  12-week internship under supervision of Dr. P. L. Montgomery working on factoring large integers on graphics processing units
- University of Amsterdam, Amsterdam, Netherlands
  Field of Study: Master Grid Computing (2004 – 2006)
  Master research project title: *The Number Field Sieve – The Sieving Stage: A Different Approach*
  Bachelor Computer Science (2002 – 2004)

RELEVANT EMPLOYMENT HISTORY

- Company: ClusterVision BV
  Location: Amsterdam, Netherlands
  Function: Software Engineer; Implementation (in C++) of a high performance cluster management daemon
  Time: August 2006 – January 2007

SKILLS

- Languages Skills
  Dutch: Native language English: Excellent
- Software Skills
  Programming languages skills include: C, C++, Java, Perl and assembly (on misc. platforms including x86, x86-64, Cell and GPU).
  Familiar with a wide variety of software libraries including: CUDA, GMP, OpenCL, OpenMPI and OpenSSL.
  Experience using different OSes, including *nix and Windows.
- Projects
  2010: Involved in finding the record factor of 73 decimal digits using the elliptic curve method for integer factorization.
  2010: Involved in the factorization of RSA-768: the current integer factorization record.
  2009: Involved in solving a 112-bit prime elliptic curve discrete logarithm problem: the current record.

INTERESTS

My research interests include cryptanalysis, fast (parallel) arithmetic and efficient implementations of cryptologic algorithms on parallel architectures with a focus on elliptic curve cryptography and integer factorization algorithms.

Bibliography

[1] D. Aggarwal and U. M. Maurer. Breaking RSA generically is equivalent to factoring. In A. Joux, editor, *Eurocrypt 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 36–53. Springer, Heidelberg, 2009.

[2] AMD. ATI CTM Reference Guide. Technical Reference Manual, 2006.

[3] D. P. Anderson. BOINC: a system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[4] S. Antao, J.-C. Bajard, and L. Sousa. Elliptic curve point multiplication on GPUs. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 192–199, 2010.

[5] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2006.

[6] D. V. Bailey, B. Baldwin, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, G. van Damme, G. de Meulenaer, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, C. Paar, F. Regazzoni, P. Schwabe, and L. Uhsadel. The Certicom challenges ECC2-X. Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2009, 2009. `http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf`.

[7] D. V. Bailey, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, H.-C. Chen, C.-M. Cheng, G. van Damme, G. de Meulenaer, L. J. D. Perez, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, R. Niederhagen, C. Paar, F. Regazzoni, P. Schwabe, L. Uhsadel, A. V. Herrewege, and B.-Y. Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. `http://eprint.iacr.org/2009/541`.

[8] J.-C. Bajard, L. Imbert, and T. Plantard. Modular number systems: Beyond the Mersenne family. In H. Handschuh and M. A. Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 159–169. Springer, Heidelberg, 2004.

[9] J.-C. Bajard, N. Meloni, and T. Plantard. Efficient RNS bases for cryptography. In *IMACS'05 : World Congress: Scientific Computation Applied Mathematics and Simulation*, 2005. `http://hal-lirmm.ccsd.cnrs.fr/lirmm-00106470/PDF/D547.PDF`.

[10] M. Bellare and P. Rogaway. Minimizing the use of random oracles in authenticated encryption schemes. In Y. Han, T. Okamoto, and S. Qing, editors, *Information and Communication Security – ICICS 1997*, volume 1334 of *Lecture Notes in Computer Science*, pages 1–16. Springer, Heidelberg, 1997.

[11] A. Bender and G. Castagnoli. On the implementation of elliptic curve cryptosystems. In G. Brassard, editor, *Crypto 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 186–192. Springer, Heidelberg, 1990.

[12] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, Heidelberg, 2006.

[13] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards curves. In S. Vaudenay, editor, *Africacrypt*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer, Heidelberg, 2008.

[14] D. J. Bernstein, P. Birkner, and T. Lange. Starfish on strike. In M. Abdalla and P. S. L. M. Barreto, editors, *Latincrypt*, volume 6212 of *Lecture Notes in Computer Science*, pages 61–80. Springer, Heidelberg, 2010.

[15] D. J. Bernstein, P. Birkner, T. Lange, and C. Peters. ECM using Edwards curves. Cryptology ePrint Archive, Report 2008/016, 2008. `http://eprint.iacr.org/`.

[16] D. J. Bernstein, P. Birkner, T. Lange, and C. Peters. EECM: ECM using Edwards curves. `http://eecm.cr.yp.to/`, 2010.

[17] D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang. The billion-mulmod-per-second PC. In *Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2009*, pages 131–144, 2009.

[18] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on graphics cards. In A. Joux, editor, *Eurocrypt 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 483–501. Springer, Heidelberg, 2009.

[19] D. J. Bernstein and T. Lange. Explicit-formulas database. `http://www.hyperelliptic.org/EFD/` (accessed 2010-01-05).

[20] D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In K. Kurosawa, editor, *Asiacrypt*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, Heidelberg, 2007.

[21] D. J. Bernstein and T. Lange. Inverted Edwards coordinates. In S. Boztas and H. feng Lu, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 4851 of *Lecture Notes in Computer Science*, pages 20–27. Springer, Heidelberg, 2007.

[22] D. J. Bernstein and T. Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. In G. L. Mullen, D. Panario, and I. E. Shparlinski, editors, *Finite Fields and Applications*, volume 461 of *Contemporary Mathematics Series*, pages 1–19. American Mathematical Society, 2008.

[23] D. J. Bernstein and T. Lange. Type-II optimal polynomial bases. In M. A. Hasan and T. Helleseth, editors, *Arithmetic of Finite Fields – WAIFI 2010*, volume 6087 of *Lecture Notes in Computer Science*, pages 41–61. Springer, Heidelberg, 2010.

[24] D. J. Bernstein, T. Lange, and P. Schwabe. On the correct use of the negation map in the

Pollard rho method. In D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, editors, *Public Key Cryptography – PKC 2011*, volume 6571 of *Lecture Notes in Computer Science*, pages 128–146. Springer, Heidelberg, 2011.

[25] M. Bevand. MD5 Chosen-Prefix Collisions on GPUs. Black Hat, 2009. Whitepaper.

[26] E. Biham. A fast new DES implementation in software. In E. Biham, editor, *Fast Software Encryption – FSE 1997*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, Heidelberg, 1997.

[27] D. Blythe. The Direct3D 10 system. *ACM Transactions on Graphics*, 25(3):724–734, 2006.

[28] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society*, 46(2):203–213, 1999.

[29] D. Boneh and R. Venkatesan. Breaking RSA may not be equivalent to factoring. In K. Nyberg, editor, *Eurocrypt 1998*, volume 1403 of *Lecture Notes in Computer Science*, pages 59–71. Springer, Heidelberg, 1998.

[30] J. W. Bos. High-performance modular multiplication on the Cell processor. In M. A. Hasan and T. Helleseth, editors, *Arithmetic of Finite Fields – WAIFI 2010*, volume 6087 of *Lecture Notes in Computer Science*, pages 7–24. Springer, Heidelberg, 2010.

[31] J. W. Bos. Low-latency elliptic curve scalar multiplication, 2012. Submitted for publication.

[32] J. W. Bos, N. Casati, and D. A. Osvik. Multi-stream hashing on the PlayStation 3. In *Applied Parallel Computing – PARA 2008*, volume 6126 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 2008. To appear.

[33] J. W. Bos and M. E. Kaihara. Montgomery multiplication on the Cell. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics – PPAM 2009*, volume 6067 of *Lecture Notes in Computer Science*, pages 477–485. Springer, Heidelberg, 2010.

[34] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. On the security of 1024-bit RSA and 160-bit elliptic curve cryptography. Cryptology ePrint Archive, Report 2009/389, 2009. `http://eprint.iacr.org/`.

[35] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *International Journal of Applied Cryptography*, 2(3):212–228, 2012.

[36] J. W. Bos, M. E. Kaihara, and P. L. Montgomery. Pollard rho on the PlayStation 3. In *Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2009*, pages 35–50, 2009. `http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf`.

[37] J. W. Bos and T. Kleinjung. ECM at work, 2012. Work in progress.

[38] J. W. Bos, T. Kleinjung, and A. K. Lenstra. On the use of the negation map in the Pollard rho method. In G. Hanrot, F. Morain, and E. Thomé, editors, *Algorithmic Number Theory – ANTS-IX*, volume 6197 of *Lecture Notes in Computer Science*, pages 67–83. Springer, Heidelberg, 2010.

[39] J. W. Bos, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Efficient SIMD arithmetic modulo a Mersenne number. In *IEEE Symposium on Computer Arithmetic –*

ARITH-20, pages 213–221. IEEE Computer Society, 2011.

[40] J. W. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe. ECC2K-130 on Cell CPUs. In D. J. Bernstein and T. Lange, editors, *Africacrypt 2010*, volume 6055 of *Lecture Notes in Computer Science*, pages 225–242. Springer, Heidelberg, 2010.

[41] J. W. Bos, O. Özen, and J.-P. Hubaux. Analysis and optimization of cryptographically generated addresses. In P. Samarati, M. Yung, F. Martinelli, and C. A. Ardagna, editors, *Information Security Conference – ISC 2009*, volume 5735 of *Lecture Notes in Computer Science*, pages 17–32. Springer, Heidelberg, 2009.

[42] J. W. Bos, O. Özen, and M. Stam. Efficient hashing using the AES instruction set. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 507–522. Springer, Heidelberg, 2011.

[43] J. W. Bos and D. Stefan. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 279–293. Springer, Heidelberg, 2010.

[44] S. Boussakta and A. Holt. New transform using the Mersenne numbers. *Vision, Image and Signal Processing, IEE Proceedings* -, 142(6):381–388, December 1995.

[45] A. Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45:736–739, 1939.

[46] R. P. Brent. An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics*, 20:176–184, 1980.

[47] R. P. Brent. Some integer factorization algorithms using elliptic curves. *Australian Computer Science Communications*, 8:149–163, 1986.

[48] R. P. Brent. Factorization of the tenth Fermat number. *Mathematics of Computation*, 68(225):429–451, 1999.

[49] R. P. Brent and J. M. Pollard. Factorization of the eighth Fermat number. *Mathematics of Computation*, 36(154):627–630, 1981.

[50] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[51] E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In D. Naccache and P. Paillier, editors, *Public Key Cryptography – PKC 2002*, volume 2274 of *Lecture Notes in Computer Science*, pages 335–345. Springer, Heidelberg, 2002.

[52] J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, and S. S. Wagstaff, Jr. *Factorizations of* $b^n \pm 1$*,* $b = 2, 3, 5, 6, 7, 10, 11, 12$ *Up to High Powers*, volume 22 of *Contemporary Mathematics*. American Mathematical Society, First edition, 1983, Second edition, 1988, Third edition, 2002. Electronic book available at: `http://homes.cerias.purdue.edu/~ssw/cun/index.html`, 1983.

[53] M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the NIST elliptic curves over prime fields. In D. Naccache, editor, *CT-RSA*, volume 2020 of *Lecture Notes in Computer Science*, pages 250–265. Springer, Heidelberg, 2001.

[54] Certicom. Certicom ECC Challenge. `http://www.certicom.com/images/pdfs/cert_`

ecc_challenge.pdf, 1997.

[55] Certicom. Press release: Certicom announces elliptic curve cryptosystem (ECC) challenge winner. http://www.certicom.com/index.php/2002-press-releases/38-2002-press-releases/340-notre-dame-mathematician-solves-eccp-109-encryption-key-problem-issued-in-1997, 2002.

[56] H.-C. Chen, C.-M. Cheng, S.-H. Hung, and Z.-C. Lin. Integer number crunching on the Cell processor. *International Conference on Parallel Processing*, pages 508–515, 2010.

[57] J. H. Cheon, J. Hong, and M. Kim. Speeding up the Pollard rho method on prime fields. In J. Pieprzyk, editor, *Asiacrypt 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 471–488. Springer, Heidelberg, 2008.

[58] J. Chung and M. A. Hasan. More generalized Mersenne numbers. In M. Matsui and R. J. Zuccherato, editors, *Selected Areas in Cryptography*, volume 3006 of *Lecture Notes in Computer Science*, pages 335–347. Springer, Heidelberg, 2003.

[59] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In K. Ohta and D. Pei, editors, *Asiacrypt 1998*, volume 1514 of *Lecture Notes in Computer Science*, pages 51–65. Springer, Heidelberg, 1998.

[60] D. Coppersmith. Modifications to the number field sieve. *Journal of Cryptology*, 6(3):169–180, 1993.

[61] D. Coppersmith, A. M. Odlyzko, and R. Schroeppel. Discrete logarithms in GF(p). *Algorithmica*, 1(1):1–15, 1986.

[62] N. Costigan and P. Schwabe. Fast elliptic-curve cryptography on the Cell Broadband Engine. In B. Preneel, editor, *Africacrypt 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 368–385. Springer, Heidelberg, 2009.

[63] N. Costigan and M. Scott. Accelerating SSL using the vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3. Cryptology ePrint Archive, Report 2007/061, 2007. http://eprint.iacr.org/2007/061.

[64] R. Crandall and B. Fagin. Discrete weighted transforms and large-integer arithmetic. *Mathematics of Computation*, 62(205):305–324, 1994.

[65] R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system, October 1992. U.S. patent number 5,159,632.

[66] A. J. C. Cunningham and H. J. Woodall. Factorizations of $y^n \pm 1$, $y = 2, 3, 5, 6, 7, 10, 11, 12$ up to high powers. Frances Hodgson, London, 1925.

[67] I. Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In J. Feigenbaum, editor, *Crypto 1991*, volume 576 of *Lecture Notes in Computer Science*, pages 445–456. Springer, Heidelberg, 1991.

[68] G. de Meulenaer, F. Gosset, G. M. de Dormale, and J.-J. Quisquater. Integer factorization based on elliptic curve method: Towards better exploitation of reconfigurable hardware. In *Field-Programmable Custom Computing Machines – FCCM 2007*, pages 197–206. IEEE Computer Society, 2007.

[69] V. Dimitrov, T. Cooklev, and B. Donevsky. Generalized Fermat-Mersenne number theoretic transform. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 41(2):133–139, February 1994.

[70] B. Dixon and A. K. Lenstra. Fast massively parallel modular arithmetic. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 99–110, 1993.

[71] B. Dixon and A. K. Lenstra. Massively parallel elliptic curve factoring. In R. A. Rueppel, editor, *Eurocrypt 1992*, volume 658 of *Lecture Notes in Computer Science*, pages 183–193. Springer, Heidelberg, 1993.

[72] J. D. Dixon. Asymptotically fast factorization of integers. *Mathematics of Computation*, 36(153):255–260, 1981.

[73] I. M. Duursma, P. Gaudry, and F. Morain. Speeding up the discrete log computation on curves with automorphisms. In K.-Y. Lam, E. Okamoto, and C. Xing, editors, *Asiacrypt 1999*, volume 1716 of *Lecture Notes in Computer Science*, pages 103–121. Springer, Heidelberg, 1999.

[74] H. M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44:393–422, July 2007.

[75] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. Blakley and D. Chaum, editors, *Crypto 1984*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, Heidelberg, 1985.

[76] A. E. Escott, J. C. Sager, A. P. L. Selkirk, and D. Tsapakidis. Attacking elliptic curve cryptosystems using the parallel Pollard rho method. *CryptoBytes Technical Newsletter*, 4(2):15–19, 1999. `ftp.rsasecurity.com/pub/cryptobytes/crypto4n2.pdf`.

[77] W. Fischer, C. Giraud, E. W. Knudsen, and J.-P. Seifert. Parallel scalar multiplication on general elliptic curves over $\mathbb{F}_p$ hedged against non-differential side-channel attacks. Cryptology ePrint Archive, Report 2002/007, 2002. `http://eprint.iacr.org/`.

[78] P. Flajolet and A. M. Odlyzko. Random mapping statistics. In J.-J. Quisquater and J. Vandewalle, editors, *Eurocrypt 1989*, volume 434 of *Lecture Notes in Computer Science*, pages 329–354. Springer, Heidelberg, 1990.

[79] T. H. Flowers. The design of colossus. *IEEE Annals of the History of Computing*, 5:239–252, 1983.

[80] W. A. P. Forum. Wireless transport layer security specification. See `http://www.openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf`, 2001.

[81] J. Franke, T. Kleinjung, F. Morain, and T. Wirth. Proving the primality of very large numbers with fastECPP. In D. A. Buell, editor, *Algorithmic Number Theory – ANTS-VI*, volume 3076 of *Lecture Notes in Computer Science*, pages 194–207. Springer, Heidelberg, 2004.

[82] Free Software Foundation, Inc. *GMP: The GNU Multiple Precision Arithmetic Library*, 2011. Available at `http://www.gmplib.org/`.

[83] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In M. J. Wiener, editor, *Crypto 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, Heidelberg, 1999.

[84] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi. Implementing the elliptic curve method of factoring in reconfigurable hardware. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems*

– *CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 119–133. Springer, Heidelberg, 2006.

[85] S. Galbraith. Mathematics of public key cryptography (version 0.6). `http://www.isg.rhul.ac.uk/~sdg/crypto-book/crypto-book.html`, 2010.

[86] R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Improving the parallelized Pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000.

[87] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008.

[88] H. L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, 8:140–147, 1959.

[89] GIMPS Home Page. The great internet Mersenne prime search. `http://www.mersenne.org`, 2010.

[90] O. Goldreich, S. Goldwasser, and S. Halevi. Public-key cryptosystems from lattice reduction problems. In B. S. Kaliski Jr., editor, *Crypto 1997*, volume 1294 of *Lecture Notes in Computer Science*, pages 112–131. Springer, Heidelberg, 1997.

[91] D. M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27:129–146, April 1998.

[92] T. Granlund. GMP small operands optimization. In *Software Performance Enhancement for Encryption and Decryption – SPEED 2007*, 2007.

[93] K. Group. OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[94] M. Gschwind. The Cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35:233–262, 2007.

[95] T. Güneysu, T. Kasper, M. Novotny, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57:1498–1513, 2008.

[96] R. Guy. *Unsolved problems in number theory*, volume 1. Springer Verlag, 3rd edition, 2004.

[97] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, pages 195–206. ACM, 2010.

[98] D. Hankerson, A. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, Heidelberg, New York, 2004.

[99] R. Harley. Elliptic curve discrete logarithms project. `http://pauillac.inria.fr/~harley/`.

[100] B. Harris. Probability distributions related to random mappings. *The Annals of Mathematical Statistics*, 31:1045–1062, 1960.

[101] J. Harrison. Isolating critical cases for reciprocals using integer factorization. In *IEEE Symposium on Computer Arithmetic – (Arith-16)*, pages 148–157. IEEE Computer Society, 2003.

[102] O. Harrison and J. Waldron. AES encryption implementation and analysis on commod-

ity graphics processing units. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 209–226. Springer, Heidelberg, 2007.

[103] O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th conference on Security symposium*, pages 195–209. USENIX Association, 2008.

[104] O. Harrison and J. Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In B. Preneel, editor, *Africacrypt 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 350–367. Springer, Heidelberg, 2009.

[105] H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In J. Pieprzyk, editor, *Asiacrypt 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer, Heidelberg, 2008.

[106] J. Hoffstein, J. Pipher, and J. H. Silverman. Ntru: A ring-based public key cryptosystem. In J. Buhler, editor, *Algorithmic Number Theory – ANTS-III*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, Heidelberg, 1998.

[107] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *High-Performance Computer Architecture – HPCA 2005*, pages 258–262. IEEE, 2005.

[108] IBM. Multi-precision math library. Example Library API Reference. Available at `http://public.dhe.ibm.com/software/dw/cell/SDK_Example_Library_API_v3.1.pdf`.

[109] ISO/IEC 18033-2. Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers, 2006.

[110] T. Izu and T. Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In D. Naccache and P. Paillier, editors, *Public Key Cryptography – PKC 2002*, volume 2274 of *Lecture Notes in Computer Science*, pages 371–374. Springer, Heidelberg, 2002.

[111] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *USENIX conference on Networked systems design and implementation – NSDI'11*, pages 1–14. USENIX Association, 2011.

[112] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) 1: RSA Cryptography Specifications Version 2.1. RFC 3447, RSA Laboratories, 2003.

[113] M. Joye and S.-M. Yen. The Montgomery powering ladder. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 1–11. Springer, Heidelberg, 2003.

[114] M. E. Kaihara and N. Takagi. A hardware algorithm for modular multiplication/division. *IEEE Transactions on Computers*, 54(1):12–21, 2005.

[115] B. S. Kaliski Jr. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.

[116] A. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. Number 145 in Proceedings of the USSR Academy of Science, pages 293–294, 1962.

[117] E. Kiltz, K. Pietrzak, M. Stam, and M. Yung. A new randomness extraction paradigm

for hybrid encryption. In A. Joux, editor, *Eurocrypt 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 590–609. Springer, Heidelberg, 2009.

[118] J. H. Kim, R. Montenegro, Y. Peres, and P. Tetali. A birthday paradox for Markov chains, with an optimal bound for collision in the Pollard rho algorithm for discrete logarithm. *The Annals of Applied Probability*, 20(2):495–521, 2010.

[119] T. Kleinjung. Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024-bit integers. In *Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2006*, 2006.

[120] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In T. Rabin, editor, *Crypto 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, Heidelberg, 2010.

[121] T. Kleinjung, J. W. Bos, A. K. Lenstra, D. A. Osvik, K. Aoki, S. Contini, J. Franke, E. Thomé, P. Jermini, M. Thiémard, P. Leyland, P. L. Montgomery, A. Timofeev, and H. Stockinger. A heterogeneous computing environment to solve the 768-bit RSA challenge. *Cluster Computing*, pages 1–16, 2010.

[122] D. E. Knuth. *Seminumerical Algorithms*. The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, USA, 3rd edition, 1997.

[123] D. E. Knuth. *Sorting and Searching*. The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, USA, 2nd edition, 1998.

[124] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[125] N. Koblitz. CM-curves with good cryptographic properties. In J. Feigenbaum, editor, *Crypto 1991*, volume 576 of *Lecture Notes in Computer Science*, pages 279–287. Springer, Heidelberg, 1992.

[126] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Crypto 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, Heidelberg, 1996.

[127] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Crypto 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, Heidelberg, 1999.

[128] A. Kruppa. A software implementation of ECM for NFS. Research Report RR-7041, INRIA, 2009. `http://hal.inria.fr/inria-00419094/PDF/RR-7041.pdf`.

[129] D. H. Lehmer. An extended theory of Lucas' functions. *Annals of Mathematics*, 31(3):419–448, 1930.

[130] D. N. Lehmer. Hunting big game in the theory of numbers. Scripta Mathematica, March 1933.

[131] A. K. Lenstra. Unbelievable security: Matching AES security using public key systems. In C. Boyd, editor, *Asiacrypt 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 67–86. Springer, Heidelberg, 2001.

[132] A. K. Lenstra and H. W. Lenstra, Jr. Algorithms in number theory. In J. van Leeuwen,

editor, *Handbook of Theoretical Computer Science (Volume A: Algorithms and Complexity)*, pages 673–715. Elsevier and MIT Press, 1990.

[133] A. K. Lenstra and H. W. Lenstra, Jr. *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verslag, 1993.

[134] A. K. Lenstra, H. W. Lenstra Jr., M. S. Manasse, and J. M. Pollard. The factorization of the ninth Fermat number. *Mathematics of Computation*, 61(203):319–349, 1993.

[135] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[136] H. W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[137] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[138] D. Loebenberger and J. Putzka. Optimization strategies for hardware-based cofactorization. In M. J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 170–181. Springer, Heidelberg, 2009.

[139] E. Lucas. Théorie des fonctions numériques simplement périodiques. *American Journal of Mathematics*, 1(2):184–196, 1878.

[140] S. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68, 2007.

[141] R. McEliece. A public-key cryptosystem based on algebraic coding theory. *Deep Space Network Progress Report*, 44:114–116, 1978.

[142] R. D. Merrill. Improving digital computer performance using residue number theory. *Electronic Computers, IEEE Transactions on*, EC-13(2):93–101, April 1964.

[143] V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Crypto 1985*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, Heidelberg, 1986.

[144] B. Möller. Improved techniques for fast exponentiation. In P. J. Lee and C. H. Lim, editors, *Information Security and Cryptology*, volume 2587 of *Lecture Notes in Computer Science*, pages 298–312. Springer, Heidelberg, 2002.

[145] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[146] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[147] P. L. Montgomery. *An FFT extension of the elliptic curve method of factorization*. PhD thesis, University of California, 1992.

[148] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *Informatique Thèorique et Applications/Theoretical Informatics and Applications*, 24:531–544, 1990.

[149] M. A. Morrison and J. Brillhart. A method of factoring and the factorization of F_7. *Mathematics of Computation*, 29(129):183–205, 1975.

[150] A. Moss, D. Page, and N. P. Smart. Toward acceleration of RSA using 3D graphics hardware. In S. D. Galbraith, editor, *Proceedings of the 11th IMA international conference on Cryptography and coding*, Cryptography and Coding 2007, pages 364–383. Springer-Verlag, 2007.

[151] National Security Agency. Fact sheet NSA Suite B Cryptography. `http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml`, 2009.

[152] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, 2010.

[153] G. Nivasch. Cycle detection using a stack. *Information Processing Letters*, 90(3):135–140, 2004.

[154] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi, 2009.

[155] NVIDIA. NVIDIA CUDA Programming Guide 3.2, 2010.

[156] N. I. of Standards and Technology. Special publication 800-57: Recommendation for key management part 1: General (revised). `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf`.

[157] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In S. Hong and T. Iwata, editors, *Fast Software Encryption – FSE 2010*, volume 6147 of *Lecture Notes in Computer Science*, pages 75–93. Springer, Heidelberg, 2010.

[158] J. Owens. GPU architecture overview. In *Special Interest Group on Computer Graphics and Interactive Techniques – SIGGRAPH 2007*, page 2. ACM, 2007.

[159] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, California, fourth edition, 2009.

[160] J. Pelzl, M. Šimka, T. Kleinjung, M. Drutarovský, V. Fischer, and C. Paar. Area-time efficient hardware architecture for factoring integers with the elliptic curve method. *Information Security, IEE Proceedings on*, 152(1):67–78, 2005.

[161] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, 24:106–110, 1978.

[162] J. M. Pollard. Factoring with cubic integers. pages 4–10 in [133].

[163] J. M. Pollard. The lattice sieve. pages 43–49 in [133].

[164] J. M. Pollard. Theorems on factorization and primality testing. *Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974.

[165] J. M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[166] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.

[167] J. M. Pollard. Kangaroos, monopoly and discrete logarithms. *Journal of Cryptology*, 13:437–447, 2000.

[168] C. Pomerance. Analysis and comparison of some integer factoring algorithms. In H. W. Lenstra, Jr. and R. Tijdeman, editors, *Computational Methods in Number Theory*, pages 89–139, Amsterdam, 1982. Mathematisch Centrum.

[169] C. Pomerance. The quadratic sieve factoring algorithm. In T. Beth, N. Cot, and

I. Ingemarsson, editors, *Eurocrypt 1984*, volume 209 of *Lecture Notes in Computer Science*, pages 169–182. Springer, Heidelberg, 1985.

[170] J.-J. Quisquater and J.-P. Delescaille. How easy is collision search? application to DES (extended summary). In J.-J. Quisquater and J. Vandewalle, editors, *Eurocrypt 1989*, volume 434 of *Lecture Notes in Computer Science*, pages 429–434. Springer, Heidelberg, 1990.

[171] J.-J. Quisquater and J.-P. Delescaille. How easy is collision search. new results and applications to DES. In G. Brassard, editor, *Crypto 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 408–413. Springer, Heidelberg, 1990.

[172] C. Research. Standards for efficient cryptography 1: Elliptic curve cryptography. Standard SEC1, Certicom, 2000.

[173] C. Research. Standards for efficient cryptography 2: Recommended elliptic curve domain parameters. Standard SEC2, Certicom, 2000.

[174] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[175] RSA the security division of EMC. The RSA challenge numbers. Formerly on `http://www.rsa.com/rsalabs/node.asp?id=2093`, now on `http://en.wikipedia.org/wiki/RSA_numbers`.

[176] C. P. Schnorr and H. W. Lenstra, Jr. A Monte Carlo factoring algorithm with linear storage. *Mathematics of Computation*, 43(167):289–311, 1984.

[177] A. Scholz. Aufgabe 253. *Jahresbericht der deutschen Mathematiker-Vereingung*, 47:41–42, 1937.

[178] E. Schulte-Geers. Collision search in a random mapping: some asymptotic results. Talk at ECC 2000, The Fourth Workshop on Elliptic Curve Cryptography, Essen, Germany, 2000, Slides available from `http://www.cacr.math.uwaterloo.ca/conferences/2000/ecc2000/slides.html`, 2000.

[179] R. Sedgewick, T. G. Szymanski, and A. C. Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11(2):376–390, 1982.

[180] M. Segal and K. Akeley. The OpenGL graphics system: A specification (version 2.0). *Silicon Graphics, Mountain View, CA*, 2004.

[181] A. Shamir. RSA for paranoids. CryptoBytes Technical Newsletter. `ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto1n3.pdf`.

[182] D. Shanks. Class number, a theory of factorization, and genera. In D. J. Lewis, editor, *Symposia in Pure Mathematics*, volume 20, pages 415–440. American Mathematical Society, 1971.

[183] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[184] V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *Eurocrypt 1997*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266. Springer, Heidelberg, 1997.

[185] J. H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Gradute Texts in Mathematics*. Springer-Verlag, 1986.

[186] M. Šimka, J. Pelzl, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, and V. Fischer. Hardware factorization based on elliptic curve method. In *Field-Programmable Custom Computing Machines – FCCM 2005*, pages 107–116. IEEE Computer Society, 2005.

[187] A. Skavantzos and P. Rao. New multipliers modulo $2^n - 1$. *IEEE Transactions on Computers*, 41:957–961, 1992.

[188] J. A. Solinas. Generalized Mersenne numbers. Technical Report CORR 99–39, Centre for Applied Cryptographic Research, University of Waterloo, 1999.

[189] J. A. Solinas. Cryptographic identification and digital signature method using efficient elliptic curve, May 2005. U.S. patent number 6,898,284.

[190] J. Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397–405, 1967.

[191] M. Stevens, A. K. Lenstra, and B. de Weger. Predicting the winner of the 2008 US presidential elections using a Sony PlayStation 3. `http://www.win.tue.nl/hashclash/Nostradamus/`.

[192] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In S. Halevi, editor, *Crypto 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 55–69. Springer, Heidelberg, 2009.

[193] R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 79–99. Springer, Heidelberg, 2008.

[194] O. Takahashi, R. Cook, S. Cottier, S. H. Dhong, B. Flachs, K. Hirairi, A. Kawasumi, H. Murakami, H. Noro, H. Oh, S. Onish, J. Pille, and J. Silberman. The circuit design of the synergistic processor element of a Cell processor. In *International conference on Computer-aided design – ICCAD 2005*, pages 111–117. IEEE Computer Society, 2005.

[195] F. Taylor. Large moduli multipliers for signal processing. *Circuits and Systems, IEEE Transactions on*, 28(7):731–736, July 1981.

[196] E. Teske. Speeding up Pollard's rho method for computing discrete logarithms. In J. Buhler, editor, *Algorithmic Number Theory – ANTS-III*, volume 1423 of *Lecture Notes in Computer Science*, pages 541–554. Springer, Heidelberg, 1998.

[197] E. Teske. On random walks for Pollard's rho method. *Mathematics of Computation*, 70(234):809–825, 2001.

[198] E. G. Thurber. On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$. *Duke Mathematical Journal*, 40:907–913, 1973.

[199] U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-3, 2009. `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`.

[200] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[201] H. C. van Tilborg. *Encyclopedia of Cryptography and Security*. Springer-Verlag, 2005.

[202] J. von zur Gathen, A. Shokrollahi, and J. Shokrollahi. Efficient multiplication using type 2 optimal normal bases. In C. Carlet and B. Sunar, editors, *Arithmetic of Finite Fields – WAIFI 2007*, volume 4547 of *Lecture Notes in Computer Science*, pages 55–68. Springer, Heidelberg, 2007.

[203] C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35(21):1831–1832, 1999.

[204] M. J. Wiener and R. J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In S. Tavares and H. Meijer, editors, *Selected Areas in Cryptography – (SAC) 1998*, volume 1556 of *Lecture Notes in Computer Science*, pages 190–200. Springer New York, 1999.

[205] J. Yang and J. Goodman. Symmetric key cryptography on modern graphics hardware. In K. Kurosawa, editor, *Asiacrypt*, volume 4833 of *Lecture Notes in Computer Science*, pages 249–264. Springer, Heidelberg, 2007.

[206] yoyo@home and M. Thompson. Found GMP-ECM top50 factor. `http://www.loria.fr/~zimmerma/records/p68`, 2009.

[207] P. Zimmermann and B. Dodson. 20 years of ECM. In F. Hess, S. Pauli, and M. E. Pohst, editors, *Algorithmic Number Theory – ANTS-VII*, volume 4076 of *Lecture Notes in Computer Science*, pages 525–542. Springer, Heidelberg, 2006.

[208] R. Zimmermann, T. Güneysu, and C. Paar. High-performance integer factoring with reconfigurable devices. In *Field Programmable Logic and Applications – FPL 2010*, pages 83–88. IEEE, 2010.

[209] P. Zimmermann et al. GMP-ECM (elliptic curve method for integer factorization). `https://gforge.inria.fr/projects/ecm/`, 2010.