Optimizing curve-based cryptography

Citation for published version (APA):
Chuengsatiansup, C. (2017). *Optimizing curve-based cryptography*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven.

Document status and date:
Published: 16/03/2017

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 15. May. 2025

Optimizing Curve-Based Cryptography

Chitchanok Chuengsatiansup

Printed by Printservice Technische Universiteit Eindhoven
Cover design by Sittiphol Phanvilai

The cover illustrates elliptic curves in Edwards form. The assembly code appearing inside the Edwards curve on the front cover is an excerpt of the Curve41417 implementation to compute scalar multiplication.

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN : 978-90-386-4233-8
NUR : 919

Optimizing Curve-Based Cryptography

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. F.P.T. Baaijens, voor een
commissie aangewezen door het College voor
Promoties, in het openbaar te verdedigen
op donderdag 16 maart 2017 om 16.00 uur

door

Chitchanok Chuengsatiansup

geboren te Bangkok, Thailand

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. B. Koren
1e promotor:	prof.dr. D.J. Bernstein
2e promotor:	prof.dr. T. Lange
leden:	dr. A. Blokhuis
	prof.dr. D.R. Kohel (Aix-Marseille Université)
	prof.dr.-ing. C. Paar (Ruhr-Universität Bochum)
	prof.dr. K.G. Paterson (Royal Holloway, University of London)
	prof.dr.ir. B. Preneel (University of Leuven)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Acknowledgments

I would like to take this opportunity to thank many people who have been helping, supporting and encouraging me throughout these four years of research in the Cryptographic Implementations group at Eindhoven University of Technology, The Netherlands. Without their help, works presented in this thesis would not have been possible.

First of all, I would like to express my very profound gratitude to my supervisors Daniel J. Bernstein and Tanja Lange for their excellent mentorship, continual support and wonderful encouragement. They always patiently provide me constructive feedback. Their guidance truly helps me to improve myself. My accomplishments would not have been possible without their support.

I am deeply grateful to Aart Blokhuis, David Kohel, Christof Paar, Kenny Paterson, and Bart Preneel for joining my Ph.D. committee. I greatly appreciate their time spent on reading my thesis and providing valuable comments. I also would like to thank Benne de Weger for his feedback on my thesis.

I especially thank Peter Schwabe for giving me a lot of advice, particularly on implementations. I would like to thank Benjamin Smith for helping me with math. I also thank Tung Chou for teaching me about GPUs and other topics.

I wish to express my sincere thanks to Vorapong Suppakitpaisarn, Phillip Rogaway, Damien Stehlé, and Tsuyoshi Takagi for giving me opportunities to visit their research groups and providing generous hospitality during the visits.

I would like to thank my co-authors Daniel J. Bernstein, Tung Chou, Andreas Hülsing, David Kohel, Eran Lambooij, Tanja Lange, Michael Naehrig, Ruben Niederhagen, Pance Ribarski, Peter Schwabe, and Christine van Vredendaal for the fruitful collaborations.

I also would like to thank my friends and colleagues in Eindhoven and all around the world. I especially thank Christine van Vredendaal who is my colleague, my office-mate and my co-author.

My sincere thanks also goes to Anita Klooster, our secretary, who is very nice and very supportive. I really appreciate her help from the very beginning such as moving into Eindhoven throughout the very end such as printing this thesis.

Last but not least, I would like to extend my appreciation to my parents and my brother who always be there to help and support me. Their love and understanding have provided me strength to overcome any obstacles.

Contents

List of algorithms

List of figures

List of tables

List of symbols

E	elliptic curve
E'	twist of curve E
E/K	elliptic curve E defined over field K
$\mathbb{F}_p$	finite field with p elements where p is prime
$\mathbb{F}_q$	finite field with q elements where q is a power of a prime p
$\mathbb{F}_{q^k}$	extension field of $\mathbb{F}_q$ of degree k
$\mathbb{G}_1$	base-field subgroup in Type-3 pairing (group 1)
$\mathbb{G}_2$	trace-zero subgroup in Type-3 pairing (group 2)
$\mathbb{G}_T$	subgroup of rth roots of unity in $\mathbb{F}_{p^k}^*$ (target group)
K	arbitrary field
$\mathbb{P}^n$	projective n-space
e	(general) pairing
$e(P,Q)$	result of pairing P and Q
h	cofactor
nP	nth multiple of a group element P by $n \in \mathbb{Z}$
r	prime, usually an order of a group
$\mathbf{A}$	cost of field addition
$\mathbf{I}$	cost of field inversion
$\mathbf{M}$	cost of field multiplication
$\mathbf{m}$	cost of field multiplication by a constant
$\mathbf{m}_d$	cost of field multiplication by $d \in \mathbb{Z}$

S	cost of field squaring
CVP	closest vector problem
DAG	directed acyclic graph
DLP	discrete logarithm problem
ECC	elliptic-curve cryptography
ECDH	elliptic-curve Diffie–Hellman
ECDHE	ephemeral elliptic-curve Diffie–Hellman
ECDL	elliptic-curve discrete logarithm
ECM	elliptic-curve method for factorization
FFDL	finite-field discrete logarithm
GLS	Galbraith–Lin–Scott method
GLV	Gallant–Lambert–Vanstone method
IND-CCA	indistinguishability under chosen ciphertext attack
LWE	learning with errors
NFS	number-field sieve
NIST	National Institute of Standards and Technology
NTT	number-theoretic transform
RLWE	ring learning with errors
RSA	Rivest–Shamir–Adleman public-key cryptosystem
SVP	shortest vector problem

1

Introduction

Cryptography has a very long history and dates back to the time of the Roman Empire. Cryptography enables a sender (often called Alice) to convert a message (plaintext) into a secret code (ciphertext) before sending to a receiver (often called Bob). Bob is able to convert the ciphertext into the plaintext using secret information (called a key). An adversary (often called Eve) who eavesdrops upon the conversation cannot retrieve the plaintext from the ciphertext. This is because the secret key known to Bob is unknown to Eve.

Cryptography has been broadly used in applications involving sensitive information ranging from military or government usage for protecting confidential documents to daily personal conversations exchanged over the Internet. Cryptography is also crucial for online business or Internet banking, for example, when handling credit card information. Some people also use cryptography to encrypt personal data on their hard disks. Sometimes cards that people are carrying in their wallets have embedded cryptographic functions.

The security of a given cryptosystem usually relies on the hardness assumption of solving some problems in mathematics. One of the most widely deployed cryptographic schemes is the RSA cryptosystem. RSA is broken if one can factor the product of two large prime numbers. There are also many other mathematically hard problems that cryptosystems could base their security upon. Examples include the discrete logarithm problem which is the basis of curve-based cryptography.

Curve-based cryptography has recently gained a lot of attention and become popular because of its advantage of using a shorter key length compared to RSA cryptosystem to achieve the same level of security. Another advantage of curve-based cryptography is that it provides some features which enable many cryptographic schemes that are not achievable with RSA. For example, certain elliptic curves allow the computation of a bilinear map. Bilinearity is a very powerful property which enables various protocols based on those curves.

In real-world applications, users normally appreciate quick responses and become impatient having to wait for outputs. In some applications, speed is a big constraint. This also applies to cryptographic applications. Even though it is possible to lower the security level in order to gain some speedups, this may not be an option in many applications. For this reason, fast and secure cryptosystems are highly desired. This thesis addresses those demands by focusing on optimizing curve-based cryptography. This thesis presents various techniques and algorithms along different dimensions of optimizations.

There has been a rising concern about the coming of general-purpose quantum computers which threaten curve-based cryptography. Adversaries equipped with a quantum computer would be able to solve discrete logarithms in polynomial time and thus break curve-based cryptosystems. The RSA cryptosystem would also no longer be secure against quantum computers. Therefore, more and more researchers have been looking into alternative schemes that would resist the power of quantum computers. Lattice-based cryptosystems are some of the candidates for post-quantum cryptography and appear to be very promising in terms of performance. This thesis also presents an optimized lattice-based cryptosystem to be used in a post-quantum era.

Overview

Chapter 2 provides fundamental background related to the contents of this thesis. This chapter starts by explaining algorithms to multiply polynomials and presenting different number representations and their advantages. Then it provides definitions of elliptic curves, explains operations and algorithms related to curve-based cryptography, and describes cryptographic pairings based on elliptic-curve groups. This chapter finishes by describing a class of attacks called side-channel attacks and countermeasures against them.

Chapter 3 presents new speed records for arithmetic on a large family of elliptic curves with cofactor 3: specifically, $8.73\mathbf{M}$ per bit for 256-bit variable-base single-scalar multiplication when curve parameters are chosen properly. This is faster than the best results known for cofactor 1, showing for the first time that points of order 3 are useful for performance and narrowing the gap to the speeds of curves with cofactor 4.

Chapter 4 introduces new formulas and algorithms reducing the number of field multiplications required for scalar multiplication on curves with cofactor 4. The new speeds rely on advances in tripling speeds and on advances in constructing double-base chains. The new tripling formulas for twisted Edwards curves takes just $11.4\mathbf{M}$, and the new algorithm for constructing an optimal double-base chain for n takes just $(\log n)^{2.5+o(1)}$ bit operations. This chapter also extends the new algorithm to double-base double-scalar multiplication.

Chapter 5 analyzes the cost of breaking ECC under the following assumptions:

ECC is using a standardized elliptic curve that was actually chosen by an attacker, and the attacker is aware of a vulnerability in some curves that are not publicly known to be vulnerable. This chapter investigates how hard it is for attackers to get their curves to be used.

Chapter 6 introduces constant-time ARM Cortex-A8 ECDH software that is faster than the fastest ECDH option in the latest version of OpenSSL but achieves a security level above 2^{200} using a prime above 2^{400}. The new speeds are achieved in a quite different way from typical prime-field ECC software: they rely on a synergy between Karatsuba's method and choices of radix smaller than the CPU word size.

Chapter 7 presents software setting new speed records for high-security constant-time variable-base-point Diffie–Hellman. The new speeds rely on a synergy between state-of-the-art formulas for genus-2 hyperelliptic curves and a modern trend towards vectorization in CPUs. This chapter also introduces several new techniques for efficient vectorization of Kummer-surface computations.

Chapter 8 introduces PandA, a software framework for *Pairings and Arithmetic*. This software framework provides protocol designers and implementors an easy access to a toolbox of all functions needed for implementing pairing-based cryptographic protocols. An example of PandA usage is shown by implementing Boneh-Lynn-Shacham (BLS) signatures.

Chapter 9 changes focus from curve-based to lattice-based cryptography. This chapter proposes NTRU Prime, which tweaks NTRU to use rings without special structures; proposes Streamlined NTRU Prime, a public-key cryptosystem optimized from an implementation perspective, subject to the standard design goal of IND-CCA2 security; finds high-security post-quantum parameters for Streamlined NTRU Prime; and optimizes a constant-time implementation of those parameters. The performance results are surprisingly competitive with the best previous speeds for lattice-based cryptography.

2

Preliminaries

This chapter serves as an introduction to essential mathematical background that would help readers to understand subsequent contents in the following chapters of this thesis. Note that only the background on topics that are directly relevant to the contents of this thesis are presented. Another purpose of this chapter is to take an opportunity to specify notations that appear in the succeeding chapters.

The contents of this chapter are categorized into 5 topics which are organized as follows:

Polynomial multiplications to compute the underlying arithmetic are explained in Section 2.1 which includes two algorithms that achieve asymptotically faster run time than the schoolbook method, namely, Karatsuba multiplication (Section 2.1.1) and Toom–Cook multiplication(Section 2.1.2).

Elliptic-curve cryptography is introduced in Section 2.2 along with various curve shapes (Section 2.2.1) and coordinate systems (Section 2.2.2) to represent curves.

Scalar multiplication is discussed in Section 2.3 together with algorithms to compute this operation, namely, double-and-add (Section 2.3.1) and the Montgomery ladder (Section 2.3.2). This section also explains number representations for expressing a scalar which focuses on representations other than the (unsigned) binary format, namely, non-adjacent form (Section 2.3.3) and double-base chains (Section 2.3.4).

Cryptographic pairings are described in Section 2.4 which consists of types of pairings (Section 2.4.1), twists of curves (Section 2.4.2), and pairing-friendly curves (Section 2.4.3).

Side-channel attacks are discussed in Section 2.5 especially countermeasures such as constant-time implementations (Section 2.5.1), conditional branch selection via arithmetic (Section 2.5.2), and constant-time table lookup (Section 2.5.3).

2.1 Polynomial multiplications

One of the fundamental arithmetic operations is multiplication. Even though multiplication can be easily explained as performing repeated additions, the simple descriptive word “repeated” hides all the complexity of its real computation. In other words, the computational complexity of multiplication is much worse than that of addition. While addition requires only $O(n)$, simple (schoolbook) multiplication requires $\Theta(n^2)$. Because of this huge gap between addition and multiplication, there have been many researchers focusing on optimizing multiplication.

Integer multiplications can be generalized to polynomial multiplications. This can be seen by representing integers in polynomial form. For example, $1234 = 1000 + 200 + 30 + 4 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10 + 4$, thus 1234 can be written in polynomial form as $1x^3 + 2x^2 + 3x + 4$ where $x = 10$. Therefore, multiplying integers can be thought of as multiplying polynomials. For example, 1234×5678 can be written in polynomial form as $(1x^3 + 2x^2 + 3x + 4) \times (5x^3 + 6x^2 + 7x + 8)$ where $x = 10$. The advantage of representing numbers in polynomial form is that it helps illustrating the number of small multiplications (“limb” multiplications) required.

For example, if the multiplication operation only allows computing the product of one decimal digit at a time, then it makes sense to use the form above with $x = 10$. If the multiplication operation allows multiplying two decimal digits at a time, then representing the product as $(12x + 34) \times (56x + 78)$ where $x = 100$ makes more sense. Counting the number of multiplications, the former case (where $x = 10$) requires $4 \times 4 = 16$ multiplications, whereas the latter case (where $x = 100$) requires only $2 \times 2 = 4$ multiplications.

This can be further extended to computer’s multiplication instructions, for example, 32-bit and 64-bit multiplication. The former case corresponds to multiplying 32-bit numbers by 32-bit numbers at a time, while the latter case corresponds to multiplying 64-bit numbers by 64-bit numbers at a time. To help analyzing and visualizing, it is usually helpful to represent numbers in radix 2^{32} or 2^{64} depending on whether the underlying architecture supports 32-bit or 64-bit instructions. Therefore, using polynomial representations, numbers can be rewritten in different formats, for example, $x = 2^{32}$ for radix 2^{32} or $x = 2^{64}$ for radix 2^{64}.

Recall in schoolbook multiplication, the most basic algorithm to compute multiplication, the computational complexity is $\Theta(n^2)$. There exist other algorithms which achieve better time complexity than this, namely, Karatsuba and Toom–Cook. The idea behind these techniques is to split polynomials into smaller pieces and perform multiplication on those smaller pieces then combine the results to obtain the entire product, i.e., similar to the above example of splitting 1234 into four or two pieces.

2.1.1 Karatsuba

The Karatsuba algorithm [KO63] is a multiplication algorithm that achieves a faster asymptotic complexity than the quadratic schoolbook algorithm. It reduces the number of single-limb multiplications needed to multiply two n-limb numbers from n^2

(as in schoolbook multiplication) down to at most about $n^{\log_2 3} \approx n^{1.585}$ single-limb multiplications.

The algorithm works by splitting $2n$-limb numbers F and G into n-limb numbers F_0, F_1, G_0 and G_1 where $F = F_0 + F_1 t^n$ and $G = G_0 + G_1 t^n$ where t is the radix, then using the Karatsuba identity:

$$(F_0 + F_1 t^n)(G_0 + G_1 t^n) = F_0 G_0 + t^n((F_0 + F_1)(G_0 + G_1) - F_0 G_0 - F_1 G_1) + t^{2n} F_1 G_1.$$

By using this equality, the four multiplications of F_0G_0, F_0G_1, F_1G_0 and F_1G_1 can be computed using only the three multiplications of F_0G_0, F_1G_1 and $(F_0 + F_1)(G_0 + G_1)$ plus some extra additions. This means that the number of multiplications is reduced by 25%. As previously mentioned, the computational complexity of addition is much lower than that of multiplication, thus additions are normally ignored in theoretical complexity analysis. However, the number of additions is still significant in practice.

A later improvement led to an identity which Bernstein in [Ber09a] calls the "refined Karatsuba identity":

$$(F_0 + t^n F_1)(G_0 + t^n G_1) = (1 - t^n)(F_0 G_0 - t^n F_1 G_1) + t^n (F_0 + F_1)(G_0 + G_1).$$

Notice that the number of multiplications stays the same but the number of additions decreases.

Example 2.1: *Compute the product of 1234 and 5678 using refined Karatsuba with $t = 10$ and $n = 2$.*

- Before applying Karatsuba, let

$$\begin{aligned} F &= 1234 && = F_0 + F_1 t^n && = 34 + 12 \cdot 10^2, \\ G &= 5678 && = G_0 + G_1 t^n && = 78 + 56 \cdot 10^2. \end{aligned}$$

- Applying the refined Karatsuba identity,

$$(F_0 + t^n F_1)(G_0 + t^n G_1) = (1 - t^n)(F_0 G_0 - t^n F_1 G_1) + t^n (F_0 + F_1)(G_0 + G_1).$$

- Then, the three multiplications that need to be computed are:

$$\begin{aligned} c_0 &= F_0 G_0 && = 34 \times 78 && = 2652, \\ c_1 &= F_1 G_1 && = 12 \times 56 && = 672, \\ c_2 &= (F_0 + F_1)(G_0 + G_1) && = (34 + 12) \times (78 + 56) && = 6164. \end{aligned}$$

- The result of the recombination is

$$\begin{aligned} &(1 - 10^2)(c_0 - c_1 10^2) + c_2 10^2 \\ &\qquad = (1 - 10^2)(2652 - 672 \cdot 10^2) + 6164 \cdot 10^2 \\ &\qquad = 7006652, \end{aligned}$$

which equals $1234 \times 5678 = 7006652$.

The Karatsuba algorithm can also be applied recursively if the number of digits is at least four. The first level of Karatsuba reduces the number of multiplications from $4((n/2) \times (n/2))$ to $3((n/2) \times (n/2))$. By applying another Karatsuba to each of the $(n/2) \times (n/2)$ multiplication, this further reduces the number of multiplications in each from $4((n/4) \times (n/4))$ to $3((n/4) \times (n/4))$. That is, from $4((n/2) \times (n/2))$ to $3(3((n/4) \times (n/4)))$. In general, if $T(n)$ denotes the total number of operations that are required to perform the multiplication of two n-digit numbers, then Karatsuba's algorithm requires

$$T(n) = 3T(\lceil n/2 \rfloor) + cn + d$$

for some constants c and d.

Example 2.2: *Compute the product of 1234 and 5678 using two-level refined Karatsuba with $t = 10$, $n_1 = 2$ and $n_2 = 1$.*

- For the first level of Karatsuba, let

$$\begin{aligned} F &= 1234 && = F_0 + F_1 t^{n_1} && = 34 + 12 \cdot 10^2, \\ G &= 5678 && = G_0 + G_1 t^{n_1} && = 78 + 56 \cdot 10^2. \end{aligned}$$

- Applying the Karatsuba identity,

$$FG = (1 - t^{n_1})(F_0G_0 - t^{n_1}F_1G_1) + t^{n_1}(F_0 + F_1)(G_0 + G_1).$$

- Then, the three multiplications that need to be computed are:

$$\begin{aligned} c_0 &= F_0G_0 && = 34 \times 78, \\ c_1 &= F_1G_1 && = 12 \times 56, \\ c_2 &= (F_0 + F_1)(G_0 + G_1) && = (34 + 12) \times (78 + 56). \end{aligned}$$

- For the second level of Karatsuba, let

$$\begin{aligned} F_0 &= 34 && = F_{00} + F_{01} t^{n_2} && = 4 + 3 \cdot 10, \\ F_1 &= 12 && = F_{10} + F_{11} t^{n_2} && = 2 + 1 \cdot 10, \\ F_2 &= (F_0 + F_1) = 34 + 12 = 46 && = F_{20} + F_{21} t^{n_2} && = 6 + 4 \cdot 10, \\ G_0 &= 78 && = G_{00} + G_{01} t^{n_2} && = 8 + 7 \cdot 10, \\ G_1 &= 56 && = G_{10} + G_{11} t^{n_2} && = 6 + 5 \cdot 10, \\ G_2 &= (G_0 + G_1) = 78 + 56 = 134 && = G_{20} + G_{21} t^{n_2} && = 4 + 13 \cdot 10. \end{aligned}$$

- Apply the Karatsuba identity three times,

$$\begin{aligned}
F_0G_0 &= (1-t^{n_2})(F_{00}G_{00} - t^{n_2}F_{01}G_{01}) + t^{n_2}(F_{00}+F_{01})(G_{00}+G_{01}),\\
F_1G_1 &= (1-t^{n_2})(F_{10}G_{10} - t^{n_2}F_{11}G_{11}) + t^{n_2}(F_{10}+F_{11})(G_{10}+G_{11}),\\
F_2G_2 &= (1-t^{n_2})(F_{20}G_{20} - t^{n_2}F_{21}G_{21}) + t^{n_2}(F_{20}+F_{21})(G_{20}+G_{21}).
\end{aligned}$$

- Then, the nine multiplications that need to be computed are:

$$\begin{aligned}
c_{00} &= F_{00}G_{00} && = 4\times 8 && = 32,\\
c_{01} &= F_{01}G_{01} && = 3\times 7 && = 21,\\
c_{02} &= (F_{00}+F_{01})(G_{00}+G_{01}) && = (4+3)\times(8+7) && = 105,\\
c_{10} &= F_{10}G_{10} && = 2\times 6 && = 12,\\
c_{11} &= F_{11}G_{11} && = 1\times 5 && = 5,\\
c_{12} &= (F_{10}+F_{11})(G_{10}+G_{11}) && = (2+1)\times(6+5) && = 33,\\
c_{20} &= F_{20}G_{20} && = 6\times 4 && = 24,\\
c_{21} &= F_{21}G_{21} && = 4\times 13 && = 52,\\
c_{22} &= (F_{20}+F_{21})(G_{20}+G_{21}) && = (6+4)\times(4+13) && = 170.
\end{aligned}$$

- Recombination of second level Karatsuba resulted in:

$$\begin{aligned}
F_0G_0 &= (1-10)(32-21\cdot 10)+105\cdot 10 && = 2652,\\
F_1G_1 &= (1-10)(12-5\cdot 10)+33\cdot 10 && = 672,\\
F_2G_2 &= (1-10)(24-52\cdot 10)+170\cdot 10 && = 6164.
\end{aligned}$$

- Recombination of first level Karatsuba resulted in:

$$FG = (1-10^2)(2652-672\cdot 10^2)+6164\cdot 10^2 = 7006652.$$

This is the same result as in Example 2.1 above.

2.1.2 Toom–Cook

The Toom–Cook algorithm [Coo66] is another multiplication algorithm that achieves a faster asymptotic complexity than the quadratic schoolbook algorithm. The Toom–Cook algorithm can be thought of as a generalization of the Karatsuba algorithm. In the Toom–Cook algorithm, a number can be split into any number k of pieces. Therefore, Karatsuba is a specific case where numbers are split into 2 pieces ($k=2$). Sometimes people refer to this general method of multiplying two numbers as "Toom-3". However, this term should be used only for a specific case where $k=3$. The general algorithm of multiplying two numbers itself should be referred to as "Toom–Cook".

Given two polynomials $f = \sum_i f_i x^i$ and $g = \sum_i g^i x^i$, the Toom–Cook algorithm computes $f \cdot g = h = \sum_i h_i x^i$ in 5 steps, namely, decomposition, evaluation, multiplication, interpolation, and recombination. The idea is to decompose the original large integer inputs f and g into smaller size integers and rewrite f and g in polynomial form. Then those polynomials are evaluated at various points. After that the algorithm multiplies the smaller integers and interpolates the coefficients h_i. Finally, the polynomials are recombined into the integer result. The complexity of the Toom–Cook algorithm is about $n^{\log_k(2k-1)}$. Note that substituting $k = 2$ matches the complexity of the Karatsuba algorithm described in the previous subsection.

Decomposition. This step splits numbers into k pieces, i.e., rewrites numbers f and g in radix $x = t^\ell$ where

$$\ell = \max\left(\left\lfloor \frac{\lfloor \log_t f \rfloor}{k} \right\rfloor, \left\lfloor \frac{\lfloor \log_t g \rfloor}{k} \right\rfloor\right) + 1.$$

For example, to compute $f \cdot g$, first f is decomposed as $f = f_0 + f_1 x + f_2 x^2 + \cdots + f_{k-1} x^{k-1}$. Similarly, g is decomposed as $g = g_0 + g_1 x + g_2 x^2 + \cdots + g_{k-1} x^{k-1}$. Note that it is possible to split f and g into different numbers of pieces. Details of splitting into unequal numbers of pieces are omitted since this thesis only focuses on splitting into equal numbers of pieces (except for the last piece, which might be smaller than the others). Let h be the polynomial product of $f \cdot g$, then h can be written as $h = h_0 + h_1 x + h_2 x^2 + \cdots + h_{2k-2} x^{2k-2}$.

Evaluation. To compute the polynomial product of $f(x) \cdot g(x)$, the polynomials $f(x)$ and $g(x)$ are evaluated at various points. Then the obtained values are multiplied (next step) in order to get evaluations of the polynomial product of $f(x) \cdot g(x)$.

Note that a degree-d polynomial is uniquely determined by $d + 1$ points. For example, a line is a polynomial of degree 1. Thus, given 2 points, there is exactly one line that passes both given points, i.e., a line is uniquely determined by 2 points. The algorithm works fine regardless of which points are used for evaluation. It is easier to evaluate at small integer values such as $0, 1, -1$. It is also convenient to evaluate at ∞, which means taking the leading coefficient. It is also better to evaluate using both a point and its negative in order to cancel out some values.

Multiplication. The multiplications in this step can be done using schoolbook multiplication. If the inputs are still large, it is also possible to recursively apply another Toom–Cook (or Karatsuba, or other multiplication algorithms) to further reduce the size of the inputs.

Interpolation. Using values of h obtained from the evaluation and multiplication steps, each coefficient h_i can be solved by the method of variable elimination and substitution.

Recombination. This step is simply recombine each piece of h_i in order to obtain the final result of the large number multiplication.

> **Example 2.3:** *Compute the product of 123456 and 234567 using Toom-3, i.e.,* $k = 3$.
>
> - Step1: decomposition.

- Let $f = 123456$ and $g = 234567$.
- Let $t = 10$ (since numbers are represented in base 10).
- Compute

$$\ell = \max\left(\left\lfloor \frac{\lfloor \log_{10} f \rfloor}{3} \right\rfloor, \left\lfloor \frac{\lfloor \log_{10} g \rfloor}{3} \right\rfloor\right) + 1 = 2.$$

  Thus, the radix is $x = t^{\ell} = 10^2$.
- Split f into $f_2 = 12$, $f_1 = 34$, $f_0 = 56$.
  Similarly, split g into $g_2 = 23$, $g_1 = 45,g_0 = 67$.
- Write f and g in polynomial form as:

$$\begin{aligned} f(x) &= f_0 + f_1 x + f_2 x^2, \\ g(x) &= g_0 + g_1 x + g_2 x^2. \end{aligned}$$

- Let $h(x)$ be the polynomial product of $f(x)$ and $g(x)$, then

$$h(x) = h_0 + h_1 x^1 + h_2 x^2 + h_3 x^3 + h_4 x^4$$

  where

$$\begin{aligned} h_0 &= f_0 g_0, \\ h_1 &= f_1 g_0 + f_0 g_1, \\ h_2 &= f_2 g_0 + f_1 g_1 + f_0 g_2, \\ h_3 &= f_2 g_0 + f_1 g_2, \\ h_4 &= f_2 g_2. \end{aligned}$$

- Step2: evaluation.
  Evaluate f and g at points: $0, \pm 1, 2$ and ∞. Note that $f(\infty)$ implicitly means $f(\infty)/\infty^2$.

$$\begin{aligned}
f(\ \ 0) &= f_0 + f_1(\ \ 0) + f_2(\ \ 0)^2 && = f_0 && = 56, \\
f(\ \ 1) &= f_0 + f_1(\ \ 1) + f_2(\ \ 1)^2 && = f_0 + \ \ f_1 + \ \ f_2 && = 102, \\
f(-1) &= f_0 + f_1(-1) + f_2(-1)^2 && = f_0 - \ \ f_1 + \ \ f_2 && = 34, \\
f(\ \ 2) &= f_0 + f_1(\ \ 2) + f_2(\ \ 2)^2 && = f_0 + 2f_1 + 4f_2 && = 172, \\
f(\infty) &= f_0 + f_1(\infty) + f_2(\infty)^2 && = f_2 && = 12, \\
g(\ \ 0) &= g_0 + g_1(\ \ 0) + g_2(\ \ 0)^2 && = g_0 && = 67, \\
g(\ \ 1) &= g_0 + g_1(\ \ 1) + g_2(\ \ 1)^2 && = g_0 + \ \ g_1 + \ \ g_2 && = 135, \\
g(-1) &= g_0 + g_1(-1) + g_2(-1)^2 && = g_0 - \ \ g_1 + \ \ g_2 && = 45, \\
g(\ \ 2) &= g_0 + g_1(\ \ 2) + g_2(\ \ 2)^2 && = g_0 + 2g_1 + 4g_2 && = 249, \\
g(\infty) &= g_0 + g_1(\infty) + g_2(\infty)^2 && = g_2 && = 23.
\end{aligned}$$

- Step3: multiplication.

$$\begin{aligned}
t_0 &= f(\ \ 0)g(\ \ 0) = (f_0)(g_0) && = 3752,\\
t_{1^+} &= f(\ \ 1)g(\ \ 1) = (f_0 + \ \ f_1 + \ \ f_2)(g_0 + \ \ g_1 + \ \ g_2) && = 13770,\\
t_{1^-} &= f(-1)g(-1) = (f_0 - \ \ f_1 + \ \ f_2)(g_0 - \ \ g_1 + \ \ g_2) && = 1530,\\
t_{2^+} &= f(\ \ 2)g(\ \ 2) = (f_0 + 2f_1 + 4f_2)(g_0 + 2g_1 + 4g_2) && = 42828,\\
t_\infty &= f(\infty)g(\infty) = (f_2)(g_2) && = 276.
\end{aligned}$$

- Step4: interpolation.

$$\begin{aligned}
t_0 &= h_0,\\
t_{1^+} &= h_0 + \ \ h_1 + \ \ h_2 + \ \ h_3 + \ \ \ \ h_4,\\
t_{1^-} &= h_0 - \ \ h_1 + \ \ h_2 - \ \ h_3 + \ \ \ \ h_4,\\
t_{2^+} &= h_0 + 2h_1 + 4h_2 + 8h_3 + 16h_4,\\
t_\infty &= h_4.
\end{aligned}$$

  Using these equalities to interpolate h_0, h_1, h_2, h_3 and h_4 yields

$$\begin{aligned}
h_0 &= t_0 && = 3752,\\
h_4 &= t_\infty && = 276,\\
h_2 &= (t_{1^+} + t_{1^-})/2 - h_0 - h_4 && = 3622,\\
h_3 &= (t_{2^+} - h_0 - 4h_2 - 16h_4 - t_{1^+} + t_{1^-})/6 && = 1322,\\
h_1 &= t_{1^+} - h_0 - h_2 - h_3 - h_4 && = 4798.
\end{aligned}$$

- Step5: recombination.
  Recall that h is expressed as

$$h(x) = h_0 + h_1x^1 + h_2x^2 + h_3x^3 + h_4x^4$$

  in radix $x = t^\ell = 10^2$. Thus, the product of $f \cdot g$ is

$$\begin{aligned}
&3752 + 4798(100) + 3622(100^2) + 1322(100^3) + 276(100^4)\\
&\qquad = 28958703552
\end{aligned}$$

  which equals $123456 \times 234567 = 28958703552$.

2.2 Elliptic-curve cryptography

This section introduces elliptic curves, gives different representations of these curves, and considers coordinate systems for efficient arithmetic on those curves. Although elliptic curves are usually defined by the Weierstrass equation, the same curves can be represented in different forms and have different curve equations. Extensive data on curve shapes and coordinate systems (including arithmetic formulas) is collected at [BLb].

The representations of the curve have effects on the performance of scalar multiplication, i.e., different curve shapes offer different advantages. This also applies to the choice of coordinate systems used for representing points on the curves. Therefore, choosing curve shapes and coordinate systems is also part of the optimization.

2.2.1 Curve shapes

The set of points on an elliptic curve forms a group, written additively throughout this thesis. All elliptic curves can be written in Weierstrass form. However, it is also possible to express elliptic curves in different forms by variable substitution.

Weierstrass curves

An elliptic curve E in Weierstrass form over a field K is given by the following equation

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where coefficients $a_1, a_2, a_3, a_4, a_6 \in K$. This equation is referred to as the *general Weierstrass equation* or the *long Weierstrass equation* for elliptic curves.

Theorem 2.1. *(Hasse) Let $E/\mathbb{F}_p$ be an elliptic curve defined over a finite field. Then*

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

The above Weierstrass equation can be further simplified depending on the underlying field. For example, if the characteristic of K is not 2 or 3, the Weierstrass equation can be simplified to

$$E : y^2 = x^3 + ax + b$$

where $a, b \in K$ and $4a^3+27b^2 \neq 0$. This equation is referred to as the *short Weierstrass equation*.

A curve is *supersingular* if there are no points of order p over $\mathbb{F}_p$ and any extension field.

Hessian curves

A Hessian form [Sma01] [JQ01] [FJ10] of an elliptic curve E defined over a field K is expressed by the following equation

$$E : x^3 + y^3 + 1 = cxy$$

where $c \in K$ and $c^3 \neq 1$.

Note that only curves with points of order 3 can be expressed in the (twisted) Hessian form.

Edwards curves

An Edwards form [Edw07] [BL07] of an elliptic curve E defined over a field K is expressed by the following equation

$$E : x^2 + y^2 = 1 + dx^2y^2$$

where $d \in K \backslash \{0, 1\}$.

A generalization of Edwards curves are called *twisted Edwards curves* [BBJ+08]. A twisted Edwards curve E defined over a field K is expressed by the following equation

$$E : ax^2 + y^2 = 1 + dx^2y^2$$

where $a, d \in K^*$ and $a \neq d$. Notice that an Edwards curve is a twisted Edwards curve with $a = 1$.

Note that over finite fields, only curves with order divisible by 4 can be expressed in the (twisted) Edwards form.

2.2.2 Coordinate systems

As previously mentioned, choices of coordinate systems affect the performance of scalar multiplication. This subsection describes four popular coordinate systems, namely, affine, projective, extended and Jacobian coordinates.

Affine coordinates

In affine coordinates, points on an elliptic curve are represented using only 2 coordinates x and y satisfying the (affine) equation. This is the most basic coordinate system in which a pair (x, y) specifies a unique point on the curve.

Projective coordinates

In projective coordinates, a point is an equivalence class of coordinate triples $(X : Y : Z)$ where at least one of the coordinates must be nonzero, i.e., it is not allowed to have $X = Y = Z = 0$, and $(X : Y : Z) = (cX : cY : cZ)$ for all $c \neq 0$. Representations of points in projective coordinates are not unique. In order to get the unique representation of a point, it has to be mapped to affine coordinates.

A map from affine coordinates to projective coordinates is $(x, y) \mapsto (\lambda x, \lambda y, \lambda)$ where λ is nonzero.

The map from projective coordinates to affine coordinates is $(X : Y : Z) \mapsto (X/Z, Y/Z)$ for $Z \neq 0$. For example, the projective form of short Weierstrass curves is given by

$$E : Y^2Z = X^3 + aXZ^2 + bZ^3.$$

Projective curves might have more points, namely, when $Z = 0$. For example, Weierstrass curves have the point $(0 : 1 : 0)$ which is the point at infinity.

The main advantage of using projective coordinates is that inversion(s), which is a very expensive operation, can be reduced to a single inversion at the end by accumulating intermediate inversions into the extra coordinate Z.

Extended coordinates

It is possible to append more coordinates, even more than just Z as being done in projective coordinates, to represent a point. An example of an extended coordinate system is to represent a point as $(X : Y : Z : T)$ where $T = XY/Z$. Hisil, Wong, Carter and

Dawson [HWCD08] introduced these extended coordinates and showed that point addition can be computed faster (requires fewer field operations) using these extended coordinates (mixed with projective coordinates) by accumulating some intermediate values in the extra coordinate T.

For example, extended coordinates for twisted Edwards curves satisfy the following equation

$$E : aX^2 + Y^2 = Z^2 + dT^2.$$

The map from affine coordinates to extended coordinates is $(x, y) \mapsto (\lambda x, \lambda y, \lambda, \lambda xy)$ where $\lambda \neq 0$. The inverse map from extended coordinates back to affine coordinates is $(X : Y : Z : T) \mapsto (X/Z, Y/Z)$.

Jacobian coordinates

It is also possible to scale the coordinates X and Y differently. One example of such scaling is a representation in Jacobian coordinates where X is scaled by Z^2 but Y is scaled by Z^3.

Jacobian coordinates for short Weierstrass curves satisfy the following equation

$$E : Y^2 = X^3 + aXZ^4 + bZ^6.$$

A map from affine coordinates to Jacobian coordinates is $(x, y) \mapsto (\lambda^3 x, \lambda^2 y, \lambda)$ where $\lambda \neq 0$, and a map from Jacobian coordinates back to affine coordinates is $(X : Y : Z) \mapsto (X/Z^2, Y/Z^3)$.

2.3 Scalar multiplication

Recall that the aim of the scalar multiplication is to compute $nP = P + P + \cdots + P$, i.e., the nth multiple of P, given an input point P on an elliptic curve and a (usually secret) scalar n. In many protocols, scalar multiplication has to be executed very frequently. Therefore, there have been a number of studies targeting optimizing scalar multiplication in order to improve the overall performance of elliptic-curve-based cryptosystems.

There are many levels of optimization to be considered and/or to be applied. For example, optimization at

(1) field level: how to perform field operations which may lead to reducing the number of arithmetic operations

(2) group level: how to perform group operations which may lead to reducing the number of field operations

(3) scalar multiplication level: how the scalar n is represented which may lead to reducing the number of point operations

Optimization at field level is generic and can also be applied outside the scope of elliptic curves. Thus, it has already been explained in Section 2.1. Optimization at group level and scalar multiplication level are related and very specific to elliptic curves. They are the main focus in this section.

This section explains the well-known and commonly used algorithms to compute scalar multiplication, namely, *double-and-add* (Section 2.3.1) and *Montgomery ladder* (Section 2.3.2). The input scalar n is often expressed in its usual binary representation, i.e., the 0-and-1 bit representation which is commonly used in computer architecture. However, it is also possible to express the scalar n using different representations such as the NAF representation (Section 2.3.3), or a double-base chain representation (Section 2.3.4).

2.3.1 Double-and-add

Given a (positive) scalar n and a point P on an elliptic curve, the double-and-add algorithm outputs $R = nP$. Like its name suggests, this algorithm performs doublings and additions depending on each bit of the binary representation of the scalar n. To be more precise, the double-and-add algorithm performs one doubling at each bit and performs addition if and only if that bit is 1. There are 2 directions to scan through the bits: left-to-right and right-to left. Throughout this thesis, left-to-right direction is used since it requires fewer registers.

The left-to-right variant of the double-and-add algorithm works by initializing the result R to the neutral element, i.e., $0P$. The result is accumulated in R by iterating through each bit of the binary representation from left to right. It doubles the current value R every bit. Only if the current bit is 1, the value P is added to R. The outline of the left-to-right double-and-add algorithm is presented in Algorithm 2.1.

Note that the number of point doublings is ℓ where ℓ is the bitlength of n. The number of point additions is the Hamming weight of n, i.e., the number of bits set $(n_i = 1)$ in the scalar n.

Algorithm 2.1 Double-and-add (left-to-right)

```
Input: $P, n = (n_{\ell-1} \dots n_0)_2$
Output: $R = nP$
R ← $0P$
for $i := \ell - 1$ down to 0 do
    $R \leftarrow 2R$
    if $n_i = 1$ then
        $R \leftarrow R + P$
return R
```

Note: “End” statements are implied by indentation, as in Python.

> **Example 2.4:** *Compute the scalar multiplication for a given point P and a scalar $n = 9$ using the left-to-right double-and-add algorithm.*
>
> The algorithm requires the scalar n to be expressed in a binary representation. Thus, n is rewritten in a binary representation as $(n_3n_2n_1n_0)_2 = 1001_2$. The algorithm starts by initializing R to $0P$, then iterates the loop from n_3 to n_0.
>
> - First iteration: $n_3 = 1$. R is updated to $2R = 2 \cdot 0P = 0P$. Because $n_3 = 1$,

this iteration performs another update (line 5) of R to $R+P = 0P+P = P$. The value stored in R by the end of this iteration is P.

- Second iteration: $n_2 = 0$. R is updated to $2R = 2 \cdot P = 2P$. This iteration does not perform another update (line 5) because $n_2 = 0$. Therefore, the value stored in R by the end of this iteration is $2P$.
- Third iteration: $n_1 = 0$. R is updated to $2R = 2 \cdot 2P = 4P$. Similar to the previous iteration, this iteration also does not perform another update (line 5) because $n_1 = 0$. Therefore, the value stored in R by the end of this iteration is $4P$.
- Fourth iteration: $n_0 = 1$. R is updated to $2R = 2 \cdot 4P = 8P$. Like the first iteration, because $n_0 = 1$ this iteration performs another update of R to $R+P = 8P+P = 9P$. The value stored in R by the end of this iteration is $9P$. This is the last iteration.
- The algorithm returns $9P$ as desired. Table 2.1 shows how the value R is updated in each iteration.

Table 2.1: Steps of computing $9P$ using double-and-add algorithm.

i		3	2	1	0
n_i		1	0	0	1
1: $R \leftarrow P$	$0P$	–	–	–	–
2: **for**: $i = \ell - 1$ **down to** 0 **do** 3: $R \leftarrow 2R$	–	$0P$	$2P$	$4P$	$8P$
4: **if** $n_i = 1$ **then** 5: $R \leftarrow R+P$	–	P	–	–	$9P$
R	$0P$	P	$2P$	$4P$	$9P$

Double-and-add-always. The pattern of whether each iteration performs only doubling or performs both doubling followed by addition may reveal the bit of the scalar n through side-channel attacks (see Section 2.5). Therefore, in some contexts where the scalar n is secret, it is suggested that operations performed in each iteration should be independent from that secret scalar n, i.e., whether the bit is 0 or 1 each iteration should perform the same operations. The easy way to cope with this problem is to perform doubling and always perform addition regardless of the bit being set or not. The algorithm that includes this modification is thus called *double-and-add-always*. The addition can be either a real addition or a dummy addition. If the bit is set, then the real addition is performed. Otherwise, the dummy addition is performed, i.e., addition by the neutral element such as $0P$.

The double-and-add-always algorithm is very similar to the double-and-add algorithm. The difference is that it may perform a dummy addition of $0P$ if the addition of $0P$ can be computed with the same sequence of operations as a regular addition. This works for Edwards and also for twisted Hessian curves which are introduced in

Chapter 3. That is, the algorithm starts by initializing the value R to $0P$. Then, each bit of the binary representation of n is scanned from left to right. (Similarly, there is also a variant of scanning from right to left. However, this thesis does not use that variant, thus the details are omitted.) In each iteration, the value R is doubled then followed by an addition. If the bit is set, P is added to R. Otherwise, $0P$ (neutral element) is added to R. The outline of the left-to-right double-and-add-always algorithm is presented in Algorithm 2.2.

Note that the numbers of point doublings and of point additions are both ℓ where ℓ is the bitlength of n. In terms of the number of point operations, the double-and-add-always performs worse than the previous double-and-add algorithm. However, the extra computation of the double-and-add-always provides an advantage of defending against side-channel attacks that distinguish the operations performed at each bit.

Algorithm 2.2 Double-and-add-always (left-to-right)

```
Input: $P, n = (n_{\ell-1} \dots n_0)_2$
Output: $R = nP$
R ← 0P
for $i := \ell - 1$ down to 0 do
    $R \leftarrow 2R$
    $R \leftarrow R + n_i P$
return R
```

Example 2.5: *Compute the scalar multiplication for a given point P and a scalar $n = 9$ using the left-to-right double-and-add-always algorithm.*

The algorithm requires the scalar n to be expressed in a binary representation. Thus, n is rewritten in a binary representation as $(n_3 n_2 n_1 n_0)_2 = 1001_2$. The algorithm starts by initializing R to $0P$, then iterates the loop from n_3 to n_0.

- First iteration: $n_3 = 1$. R is updated to $2R = 2 \cdot 0P = 0P$. Because $n_3 P = 1P = P$, this iteration also updates R to $R + P = 0P + P = P$. The value stored in R by the end of this iteration is P.

- Second iteration: $n_2 = 0$. R is updated to $2R = 2 \cdot P = 2P$. Since $n_2 P = 0P$, this iteration updates R to $R + 0P = 2P + 0P = P$. Therefore, the value stored in R by the end of this iteration is $2P$.

- Third iteration: $n_1 = 0$. R is updated to $2R = 2 \cdot 2P = 4P$. Similar to the previous iteration, this iteration updates R to $R + 0P = 4P + 0P = 4P$. Therefore, the value stored in R by the end of this iteration is $4P$.

- Fourth iteration: $n_0 = 1$. R is updated to $2R = 2 \cdot 4P = 8P$. Like the first iteration, because $n_0 P = 1P = P$, this iteration updates R to $R + P = 8P + P = 9P$. The value stored in R by the end of this iteration is $9P$. This is the last iteration.

- The algorithm returns $9P$ as desired. Table 2.2 shows how the value R is updated in each iteration.

Table 2.2: Steps of computing $9P$ using double-and-add-always algorithm.

i		3	2	1	0
n_i		1	0	0	1
1: $R \leftarrow 0P$	$0P$	–	–	–	–
2: **for**: $i = \ell - 1$ **down to** 0 **do**					
3: $R \leftarrow 2R$	–	$0P$	$2P$	$4P$	$8P$
4: $R \leftarrow R + n_i P$	–	P	$2P$	$4P$	$9P$
R	$0P$	P	$2P$	$4P$	$9P$

Windowing method. Recall that in the previously described algorithms, the scalar n is scanned only one bit at a time. One way to improve these algorithms is to scan many bits, say ω bits, at a time by doubling ω times and adding $(n_{i+\omega} \dots n_{i+1}) \cdot P$ instead of P, for precomputed $(n_{i+\omega} \dots n_{i+1}) \cdot P$. Notice that the number of point doublings stays the same, but the number of point additions decreases to approximately ℓ/ω where ℓ is the bitlength of n. This method of scanning ω bits at a time is called the *windowing method* where ω is referred to as the window *width*.

There are two main variants of the windowing method. The first is to always move the window by ω bits. The second is to move the window so that it starts with a bit that is set. The former method is called *fixed window* while the latter is called *sliding window*.

In the **fixed window method**, the algorithm starts by defining the window width ω, then rewriting the scalar n in radix 2^ω, i.e., $n = (c_{j-1}, \dots, c_0)_{2^\omega}$. Before entering the loop, R is initialized to $0P$. The loop iterates from c_{j-1} to c_0 at each iteration doubling R repeatedly ω times then retrieving the value $c_k P$ from a lookup table and adding it to R.

Note that the values $2P, 3P, \dots, (2^\omega - 1)P$ in the lookup table have to be precomputed. Note also that in the main computation, the number of point doublings is the same as in the previous two algorithms, i.e., ℓ where ℓ is the bitlength of the scalar n, but the number of point additions decreases to approximately ℓ/ω where ω is the window width. Algorithm 2.3 outlines the fixed window method.

Example 2.6: *Compute the scalar multiplication for a given point P and a scalar $n = 2345$ using the fixed window method with width $\omega = 5$.*

First, rewrite $n = 2345$ in radix 2^5. This can be done by writing 2345 in base 2 and then grouping 5 bits together from right to left as follows:

$$\begin{aligned} 2345 &= \underline{10} \quad \underline{01001} \quad \underline{01001}_2 \\ &= 299_{32} \end{aligned}$$

Before entering the loop, retrieve the value $c_k P$ from the lookup table where c_k

Algorithm 2.3 Fixed window method

```
Input: $P, n = (n_{\ell-1}, \dots, n_0)_2$
Output: $R = nP$
1: convert $n$ to radix $2^\omega$: $n = (c_{j-1}, \dots, c_0)_{2^\omega}$    ▷ Note: $\omega$ = window width
2: $R \leftarrow 0P$
3: for $k := j-1$ down to $0$ do
4:     $R \leftarrow 2^\omega R$
5:     $R \leftarrow R + c_k P$    ▷ Note: retrieve $c_k P$ from lookup table
6: return $R$
```

is the leftmost digit of n in radix 2^5 (which is 2 in this case) and initialize to R. Thus R has value $2P$.

- First iteration: $k = 1, c_k = c_1 = 9$. Doubling is performed on R for 5 times; R becomes $2^\omega R = 2^5 R = 32 \cdot 2P = 64P$. The value $c_1 = 9$ is retrieved from the lookup table and added to R; R becomes $R = 64P + 9P = 73P$.
- Second iteration: $k = 0, c_k = c_0 = 9$. Similar to the previous iteration, doubling is performed on R for 5 times; R becomes $2^5 R = 32 \cdot 73P = 2336P$. The value $c_0 = 9$ is retrieved from the lookup table and added to R; R becomes $R = 2336P + 9P = 2345P$. This is the last iteration.
- The algorithm returns $2345P$ as desired. Table 2.3 shows how the value R is updated at each iteration.

Table 2.3: Steps of computing $2345P$ using fixed window method.

	k	2	1	0
(base 2)	c_k	10	01001	01001
(base 32)		2	9	9
2: $R \leftarrow c_2 P$		$2P$	–	–
3: **for**: c_1 **to** c_0 **do**				
4: $R \leftarrow 2^\omega R$		–	$2^5(2P) = 64P$	$2^5(73P) = 2336P$
5: $R \leftarrow R + c_k P$		–	$64P + 9P = 73P$	$2336P + 9P = 2345P$
	R	$2P$	$73P$	$2345P$

In the **sliding window method**, the window starts at the leftmost bit and moves to the right until the leftmost of the ω bits is 1. The value to retrieve from the lookup table is the largest substring among those ω bits ending in a 1 on the right. Thus, the lookup table only needs to contains odd multiples of P. Algorithm 2.4 outlines the sliding window method.

Example 2.7: *Compute the scalar multiplication for a given point P and a scalar $n = 2345$ using the sliding window method with width $\omega = 5$.*

Algorithm 2.4 Sliding window method [ACD+05]

```
Input: $P, n = (n_{\ell-1} \dots n_0)_2, \omega \geq 1$, and precomputed values $3P, 5P, \dots, 2^{\omega} - 1P$
Output: $R = nP$
$R \leftarrow P$
$i \leftarrow \ell - 1$
while $i \geq 0$ do
    if $n_i = 0$ then
        $R \leftarrow 2R$
        $i \leftarrow i - 1$
    else
        $s \leftarrow \max(i - \omega + 1, 0)$
        while $n_s = 0$ do
            $s \leftarrow s + 1$
        for $h := 1$ to $i - s + 1$ do
            $R \leftarrow 2R$
        $u \leftarrow (n_i \dots n_s)_2$
        $R \leftarrow R + uP$                    ▷ Note: retrieve $uP$ from lookup table
        $i \leftarrow s - 1$
return $R$
```

The algorithm requires the scalar n to be expressed in a binary representation. Thus, n is rewritten in a binary representation as $(n_{11}n_{10}\dots n_0)_2 = 100100101001_2$.

To outline how the windows are moved and the actual values to be retrieved from the lookup table, the above bits are grouped as

$$\underline{1001}00\underline{101}00\underline{1}$$

where the underlined parts highlight the windows and value to lookup.

The first 5 bits (from the left) are $(n_{11}n_{10}n_9n_8n_7)_2 = 10010$. The leftmost bit is already 1, so there is no need to move this window. The rightmost bit is 0, so ignore that bit. Thus, the window to consider is $(n_{11}n_{10}n_9n_8)_2 = 1001_2 = 9$. For the first window, the value $9P$ is retrieved from the lookup table and assigned to R. Therefore, R becomes $9P$.

The next two bits n_7n_6 are 0. Thus, R is doubled two times and becomes $R = 2^2 \cdot 9P = 36P$.

The next 5 bits with the leftmost bit being 1 are $n_5n_4n_3n_2n_1 = 10100_2$. The two rightmost bits are 0, so these two bits are ignored. Thus, the window to consider is $(n_5n_4n_3)_2 = 101_2 = 5$. The value R is doubled three times. The value $5P$ is retrieved from the lookup table and added to R. Therefore, R becomes $2^3 \cdot 36P + 5P = 293P$.

The next two bits n_2n_1 are 0. Thus, R is doubled two times and becomes $R = 2^2 \cdot 293P = 1172P$.

> There is only one bit left which is $n_0 = 1$. R is doubled and P is added to it. Therefore, R becomes $2 \cdot 1172P + P = 2345P$.

2.3.2 Montgomery ladder

Another well-known and commonly used algorithm to compute scalar multiplication is the Montgomery ladder [Mon87]. This algorithm is very similar to the double-and-add algorithm in the sense that each bit of the input scalar n is scanned. It is especially close to the double-and-add-always variant because in each iteration (at each bit) both point doubling and point addition are performed. The differences between the Montgomery ladder and the double-and-add-always algorithm are that Montgomery ladder:

(1) requires 2 variables to keep intermediate values,
(2) performs addition first (i.e., prior to doubling),
(3) uses differential addition allowing a speed up,
(4) chooses which register to double according to whether the bit is set or not.

Because at each iteration, one addition and one doubling are always performed, this algorithm behaves regularly and thus offers some protection against timing side-channel attacks.

The algorithm starts by initializing points R_0 and R_1 to $0P$ and $1P$ respectively. Then the scalar is scanned from left to right. If the bit is not set, i.e., $n_i = 0$, then R_1 is replaced by the sum of R_0 and R_1 and R_0 is replaced by the double of R_0. On the other hand, if the bit is set, i.e., $n_i = 1$, then the sum of R_0 and R_1 is kept in R_0 and R_1 keeps the double of R_1. At the end of the last iteration, the result of nP is kept in variable R_0. Note that the difference between 2 variables R_0 and R_1 is always P. The outline of the Montgomery ladder is presented in Algorithm 2.5.

Algorithm 2.5 Montgomery ladder

```
Input: $P, n = (n_{\ell-1}, \dots, n_0)_2$
Output: $R_0 = nP$
R_0 ← 0P
R_1 ← 1P
for $i := \ell - 1$ to 0 do
    if $n_i = 0$ then
        $R_1 \leftarrow R_0 + R_1$
        $R_0 \leftarrow 2R_0$
    else
        $R_0 \leftarrow R_0 + R_1$
        $R_1 \leftarrow 2R_1$
return $R_0$
```

Example 2.8: *Compute scalar multiplication for a given point P and a scalar $n = 9$ using the Montgomery ladder.*

The algorithm requires the scalar n to be expressed in a binary representation. Thus, n is rewritten in a binary representation as $n = (n_3 n_2 n_1 n_0)_2 = 1001_2$. Before entering the loop, initialize R_0 to $0P$ and R_1 to $1P$.

- First iteration: $n_3 = 1$. Because the bit is set, the sum of two registers R_0 and R_1 is accumulated in R_0, then R_1 is doubled. That is, R_0 is updated to $R_0 = R_0 + R_1 = 0P + 1P = 1P$ and R_1 is updated to $R_1 = 2R_1 = 2 \cdot P = 2P$. Note that the difference of R_0 and R_1 is $|1P - 2P| = P$.

- Second iteration: $n_2 = 0$. Because this bit is not set, the sum of two registers R_0 and R_1 is accumulated in R_1, then R_0 is doubled. That is, R_1 is updated to $R_1 = R_0 + R_1 = 1P + 2P = 3P$ and R_0 is updated to $R_0 = 2R_0 = 2 \cdot P = 2P$. Note that the difference of R_0 and R_1 is $|2P - 3P| = P$.

- Third iteration: $n_1 = 0$. Similar to the previous iteration, this bit is not set. Therefore the sum of two registers R_0 and R_1 is accumulated in R_1, then R_0 is doubled. That is, R_1 is updated to $R_1 = R_0 + R_1 = 2P + 3P = 5P$ and R_0 is updated to $R_0 = 2R_0 = 2 \cdot 2P = 4P$. Note that the difference of R_0 and R_1 is $|4P - 5P| = P$.

- Fourth iteration: $n_0 = 1$. This iteration is similar to the first iteration since the bit is set. The sum of two registers R_0 and R_1 is accumulated in R_0, then R_1 is doubled. That is, R_0 is updated to $R_0 = R_0 + R_1 = 4P + 5P = 9P$ and R_1 is updated to $R_1 = 2R_1 = 2 \cdot 5P = 10P$. Note that the difference of R_0 and R_1 is $|9P - 10P| = P$. This is the last iteration.

- The result of scalar multiplication $nP = 9P$ is accumulated in R_0. The algorithm returns R_0. Table 2.4 shows how the value R is updated at each iteration.

Table 2.4: Steps of computing $9P$ using the Montgomery ladder.

i	3	2	1	0
n_i	1	0	0	1
1: $R_0 \leftarrow 0P$ 2: $R_1 \leftarrow 1P$	$(0P, 1P)$	–	–	–
3: **for**: n_3 **to** n_0 **do**				
4: **if** $n_i = 0$ **then**				
5: $R_1 \leftarrow R_0 + R_1$	–	$(1P, 3P)$	$(2P, 5P)$	–
6: $R_0 \leftarrow 2R_0$	–	$(2P, 3P)$	$(4P, 5P)$	–
7: **else**				
8: $R_0 \leftarrow R_0 + R_1$	$(1P, 1P)$	–	–	$(9P, 5P)$
9: $R_1 \leftarrow 2R_1$	$(1P, 2P)$	–	–	$(9P, 10P)$
(R_0, R_1)	$(1P, 2P)$	$(2P, 3P)$	$(4P, 5P)$	$(9P, 10P)$

2.3.3 Non-adjacent form

In situations where negation has negligible cost, and when addition and subtraction have essentially the same cost as each other, it might be better to express numbers using a *signed* representation, for example, non-adjacent form. A non-adjacent form, or NAF for short, is a signed-binary representation defined as

$$n = \sum_{i=0}^{\ell} d_i 2^i$$

where $d_i \in \{-1, 0, 1\}$ with the extra requirement $n_i n_{i+1} = 0$ for all $i \geq 0$, i.e., adjacent bits cannot be set at the same time. Due to this property, the Hamming weight of this representation is on average only $\approx \ell/3$ where ℓ is the bitlength of n.

In scalar multiplication (see Section 2.3), the Hamming weight on average reflects the number of point additions (assuming double-and-add algorithm without windowing method, see Section 2.3.1). In the context of elliptic-curve cryptography, negation almost comes for free. Therefore, representing numbers using NAF usually achieves better performance than using (unsigned) binary representation. Algorithm 2.6 outlines the conversion of a scalar n from binary to NAF representation.

Algorithm 2.6 NAF representation [ACD+05]

Input: A positive integer $n = (n_{\ell+1} n_\ell \dots n_0)_2$ with $n_{\ell+1} = n_\ell = 0$
Output: The signed-binary representation of n in non-adjacent form $(n'_\ell \dots n'_0)_{NAF}$

```
$c_0 \leftarrow 0$
for $i := 0$ to $\ell$ do
    $c_{i+1} \leftarrow \lfloor (c_i + n_i + n_{i+1})/2 \rfloor$
    $n'_i \leftarrow c_i + n_i - 2c_{i+1}$
return $(n'_\ell \dots n'_0)_{NAF}$
```

Example 2.9: *Convert* 123 *into NAF representation.*

By following Algorithm 2.6, Table 2.5 shows how each c_i and n'_i are computed. According to the table, the NAF representation of $n = 123 = (1111011)_2$ is $(n'_7, n'_6, \dots, n'_0)_{NAF} = (1, 0, 0, 0, 0, -1, 0, -1)_{NAF}$.

Table 2.5: Steps of converting 123 into NAF representation.

i	n_i	$c_i = \lfloor (c_{i-1} + n_{i-1} + n_i)/2 \rfloor$	$n'_i = c_i + n_i - 2c_{i+1}$
0	1	0	$-1 = 0 + 1 - 2(1)$
1	1	$1 = \lfloor (0+1+1)/2 \rfloor$	$0 = 1 + 1 - 2(1)$
2	0	$1 = \lfloor (1+1+0)/2 \rfloor$	$-1 = 1 + 0 - 2(1)$
3	1	$1 = \lfloor (1+0+1)/2 \rfloor$	$0 = 1 + 1 - 2(1)$
4	1	$1 = \lfloor (1+1+1)/2 \rfloor$	$0 = 1 + 1 - 2(1)$
5	1	$1 = \lfloor (1+1+1)/2 \rfloor$	$0 = 1 + 1 - 2(1)$
6	1	$1 = \lfloor (1+1+1)/2 \rfloor$	$0 = 1 + 1 - 2(1)$
7	0	$1 = \lfloor (1+1+0)/2 \rfloor$	$1 = 1 + 0 - 2(0)$
8	0	$0 = \lfloor (1+0+0)/2 \rfloor$	–

2.3.4 Double-base chain

Even though the binary representation is commonly used in computer science, it is not the only format to express numbers. In situations where addition is expensive, it might be better to express numbers using *multiple-base* representation, for example, double-base chain.

When using a single-base, e.g., binary representation or NAF representation, a number n is expressed as

$$\sum_{i=0}^{\ell-1} d_i 2^{a_i}$$

where ℓ is the bitlength of the number and d_i is chosen from a set of allowed coefficients. For example, $d_i \in \{0,1\}$ for the binary representation and $d \in \{-1,0,1\}$ for the NAF representation. When using a double-base binary-ternary representation, a number n is expressed as

$$\sum_i d_i 2^{a_i} 3^{b_i}$$

where $d_i \in \{-1,1\}$ or any set of allowed coefficients. This is called *double-base number system* or *DBNS* [DIM05].

If there are restrictions on the exponents such that $a_1 \geq a_2 \geq \cdots \geq a_\ell \geq 0$; and $b_1 \geq b_2 \cdots \geq b_\ell \geq 0$; and d_i are coefficients from a specific set S, then this is called a *double-base chain* or *DBC*.

The main advantage of double-base chains over the general DBNS is that it requires only a_1 doublings and b_1 triplings (plus some additions) to compute n since the chain can be rewritten using Horner's method [Hor19], for example, $2^5 3^4 + 2^4 3^2 - 2^2 3 + 1$ can be rewritten as $2^2 3(2^2 3(2 \cdot 3^2 + 1) - 1) + 1$.

Double-base chain representations are not unique even for a small set S. For example, using $S = \{-1, 0, 1\}$ the number 14 can be represented differently as:

$$\begin{aligned} 14 &= 2^4 - 2 \\ &= 2^3 3 - 2^2 3 + 2 \\ &= 2^2 3 + 2 \\ &= 2 \cdot 3^2 - 2^2 \\ &= 3^3 - 3^2 - 3 - 1 \end{aligned}$$

Notice that the representation in the second line $2^3 3 - 2^2 3 + 2$ is worse than the representation in the third line $2^2 3 + 2$ in terms of the number of operations. Therefore, it is desirable to find a "good" representation (even though different algorithms may define "good" differently). This subsection explains two algorithms which use different approaches to find double-base chains.

Greedy

One common method to find a double-base representation (not guaranteed to be a chain) is to use a greedy algorithm [DIM08]. This is done by finding the best approximation $z = 2^a 3^b$ of n (i.e., the z in this form closest to n) and subtracting z from n. This process is repeated until n reaches 0. The pairs (a, b), which correspond to the terms $2^a 3^b$, form a double-base representation constructing n. If the approximation z is greater than n, then the sign of the following term is flipped. The greedy method for computing double-base representation is outlined in Algorithm 2.7. Note that to extend this algorithm for finding double-base chain representations, the restrictions on the exponents must be applied. That is, when finding the best approximations of n of the form $z = 2^a 3^b$, the value a and b must not exceed the previous ones.

Algorithm 2.7 Greedy algorithm for finding double-base representation [DIM08]

```
Input: A positive integer $n$
Output: A sequence of triples $(s_i, a_i, b_i)_{i \geq 0}$ such that $n = \sum_i s_i 2^{a_i} 3^{b_i}$ with $s_i \in \{-1, 1\}$
    and $a_i, b_i \geq 0$
1: $c = []$
2: $s \leftarrow 1$
3: while $n \neq 0$ do
4:     Find the best approximation of $n$ of the form $z = 2^a 3^b$
5:     Append $(s, a, b)$ to $c$
6:     if $n < z$ then
7:         $s \leftarrow -s$
8:     $n \leftarrow |n - z|$
9: return $c$
```

Note: Different methods to "Find the best approximation of n of the form $z = 2^a 3^b$" (line 4) offer different efficiency. See, for example, [BI04].

Example 2.10: *Find a double-base representation of* 917 *using the greedy algorithm.*

- First iteration: the best approximation for $n = 917$ is $z = 2^5 3^3 = 864$. Therefore, $(1, 5, 3)$ is appended to the list. After this iteration n becomes $|n-z| = |917-864| = 53$.
- Second iteration: the best approximation for $n = 53$ is $z = 2 \cdot 3^3 = 54$. Therefore, $(1, 1, 3)$ is appended to the list. Since z is greater than n $(54 > 53)$, the sign s for the next term has to be switched. Thus, s is changed to $-s = -1$. After this iteration, n becomes $|n-z| = |53-54| = 1$.
- Third iteration: the best approximation for $n = 1$ is simply $z = 2^0 = 1$. Therefore, $(-1, 0, 0)$ is appended to the list. After this iteration n becomes $|n - z| = |1 - 1| = 0$. Since $n = 0$, this is the end of the while loop. The algorithm returns $(1, 5, 3), (1, 1, 3), (-1, 0, 0)$, which is a double-base representation to compute 917 since $917 = 2^5 3^3 + 2 \cdot 3^3 - 2^0 3^0$. This double-base representation is also a double-base *chain* representation because the exponents are in a non-increasing order.

Note that compared to the binary representation of 917 which is $917 = 2^9 + 2^8 + 2^7 + 2^4 + 2^2 + 2^0$, this double-base chain representation is shorter, i.e., has fewer terms (3 vs.6) and thus requires fewer additions. If “good” is defined to be “the shorter the chain the better”, then the above double-base chain for 917 is better than the single-base chain of the binary representation.

Tree-based

Another common method to find a double-base chain representation is to use a tree-based approach [DH08]. The algorithm starts by dividing out all powers of 2 and 3 from n, i.e., $n = n/2^{v_2(n)} 3^{v_3(n)}$ where $v_p(n) = \max(i \in \mathbb{N} : p^i | n)$. Then, the root node is initialized to the resulting integer.

The algorithm continues by building a tree using a breadth-first traversal and performing the following: (1) compute $n - 1$ and $n + 1$; (2) factor out all 2s and 3s; and (3) insert these two child nodes. Note that $2 \nmid n$ and $3 \nmid n$ means $n \equiv \pm 1 \bmod 6$. This implies that both $n - 1$ and $n + 1$ must be divisible by 2 and one of them by 3. The search continues until a leaf node with value ± 1 is reached.

To keep the size at each level of the tree small, this algorithm ignores leaf nodes with repeated values and introduces a parameter B where at most B nodes are kept at each level. Algorithm 2.8 outlines the tree-based approach to generating double-base chains.

Example 2.11: *Find a double-base chain representation of* 917 *using the tree-based approach.*

Since 917 is already equivalent to $-1 \bmod 6$, the root node is initialized to

Algorithm 2.8 Tree-based double-base chain search [DH08]

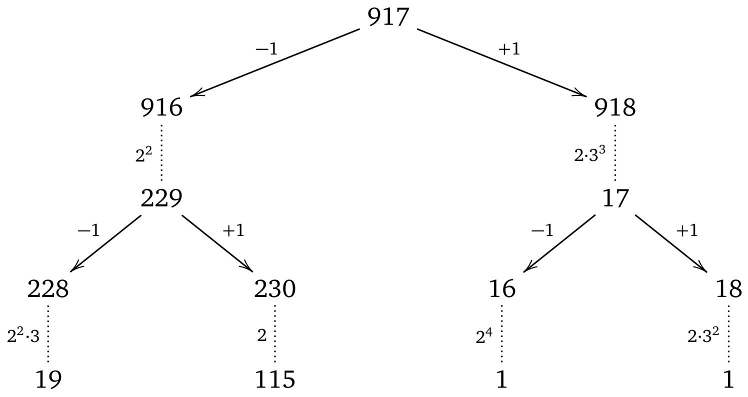
```
Input: An integer $n$ and a bound $B$
Output: A binary tree containing a double-base chain computing $n$
Set $t \leftarrow f(n)$                          ▷ $f(n) = n/(2^{\nu_2(n)}3^{\nu_3(n)})$
Initialize a binary tree $\mathscr{T}$ with root node $t$
repeat
    for each leaf node $m$ in $\mathscr{T}$ do
        Insert left child $f(m-1)$
        Insert right child $f(m+1)$
    Discard any redundant leaf nodes
    Discard all but the $B$ smallest leaf nodes
until a leaf node is equal to 1
return $\mathscr{T}$
```

917. Two nodes are computed as $917-1=916=2^2\cdot 229$ and $917+1=918=2\cdot 3^3\cdot 17$. These nodes 229 and 17 are then inserted as the left and right child nodes respectively. Current leaf nodes (in order of visiting) are: 229 and 17.

At node 229, two nodes are computed as $229-1=228=2^2\cdot 3\cdot 19$ and $229+1=230=2\cdot 115$. These nodes 19 and 115 are then inserted as the left and right child nodes respectively.

At node 17, two nodes are computed as $17-1=16=2^4\cdot 1$ and $17+1=18=2\cdot 3^2\cdot 1$. Since the node with value 1 is reached, this is the end of search.

The following diagram depicts the tree generated by this example.



By following the tree, $n = 2\cdot 3^3(2^4+1)-1$ and $2\cdot 3^3(2\cdot 3^2-1)-1$ are double-base chains for $n=917$.

Note that the algorithm stops at the first node with value ±1, this leads to a chain having as few additions as possible. Note also that this algorithm does not optimize the number of doublings and triplings.

2.4 Cryptographic pairings

Let $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ be cyclic groups of prime order r. A (cryptographic) pairing is a map

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$$

that satisfies the following conditions [Men09]:

1. *Bilinearity*: $e(aP, bQ) = e(P,Q)^{ab}$ for all $P \in \mathbb{G}_1, Q \in \mathbb{G}_2$ and all $a, b \in \mathbb{Z}$.

2. *Non-degeneracy*: for all $P \in \mathbb{G}_1$, there exists some $Q \in \mathbb{G}_2$ such that $e(P,Q) \neq 1$ where 1 is the neutral element in $\mathbb{G}_T$. Similarly, for all $Q \in \mathbb{G}_2$, there exists some $P \in \mathbb{G}_1$ such that $e(P,Q) \neq 1$.

3. *Computability*: e can be efficiently computed.

In cryptographic applications, generally $\mathbb{G}_1$ and $\mathbb{G}_2$ are groups constructed from elliptic curves, hence written additively, and $\mathbb{G}_T$ is a subgroup of the multiplicative group of a finite field.

Let p be a prime, let E be an elliptic curve defined over $\mathbb{F}_p$, and let r be the largest prime factor of $\#E(\mathbb{F}_p)$. The *embedding degree* (with respect to r) is defined to be the smallest positive integer k such that $r|(p^k - 1)$.

Notice that there are interactions between different groups in pairings, namely, $\mathbb{G}_1$ and $\mathbb{G}_2$ are elliptic curve groups while $\mathbb{G}_T$ is a finite field group. Pairing-based systems use the Discrete Logarithm Problem (DLP) in $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$. Bilinearlity (property (1) above) shows how the DLP in $\mathbb{G}_1$ or $\mathbb{G}_2$ translates to the DLP in $\mathbb{G}_T$. Therefore, all these three groups must stay secure against the underlying hardness assumption of the discrete logarithm problem, i.e., if attackers can break any of these groups, they can break pairings. The state of the art is that elliptic curve groups have a higher security level per bit than finite field groups. Thus, to achieve the same level of security, finite field groups ($\mathbb{G}_T$) must have a larger cardinality than elliptic curve groups ($\mathbb{G}_1$ and $\mathbb{G}_2$).

2.4.1 Pairing types

Elliptic curve pairings can be categorized into types by how $\mathbb{G}_1$ and $\mathbb{G}_2$ are defined. [GPS08]

- Type-1 pairing: $\mathbb{G}_1 = \mathbb{G}_2$.

- Type-2 pairing: $\mathbb{G}_1 \neq \mathbb{G}_2$ but there is an efficiently computable homomorphism $\phi : \mathbb{G}_2 \to \mathbb{G}_1$.

- Type-3-pairing: $\mathbb{G}_1 \neq \mathbb{G}_2$ and there are no efficiently computable homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$.

2.4.2 Twists of curves

Let E and E' be elliptic curves over $\mathbb{F}_q$, then E' is a twist of E if E and E' are isomorphic over some extension field of $\mathbb{F}_q$. More precisely, E' is a *degree-d twist* of E if they are isomorphic over an extension field of degree d and not over any smaller field. For short Weierstrass curves, a curve isomorphic to $E/\mathbb{F}_q : y^2 = x^3 + ax + b$ is of the form $E'/\mathbb{F}_q : y^2 = x^3 + a\omega^4 x + b\omega^6$ where $a\omega^4, b\omega^6 \in \mathbb{F}_q$.

Twists of curves in pairings make it possible to work in a subfield instead of the full extension field. If the degree d divides the embedding degree k, then using a degree-d twist allows us to work in the subfield $\mathbb{F}_{q^{k/d}}$ instead of the full extension field $\mathbb{F}_{q^k}$. This means that the larger the degree d is, the smaller subfield that can be. Therefore, it is desirable to have the degree d of the twist to be as high as possible. Because $a\omega^4$ and $b\omega^6$ must be in $\mathbb{F}_q$, the only posibilities for d are $d \in \{2, 3, 4, 6\}$. Moreover, a specific degree of twist is possible only with the following specified curve forms.

Quadractic twist : $d = 2$

This type of twist is possible for any elliptic curve. If $E/\mathbb{F}_q : y^2 = x^3 + ax + b$, then a quadratic twist is defined by $E'/\mathbb{F}_q : y^2 = x^3 + a\omega^4 x + b\omega^6$ for any ω satisfying $\omega \in \mathbb{F}_{q^2}$ and $\omega^2 \in \mathbb{F}_q$. The isomorphism $\Psi : E' \to E$ defined by $\Psi : (x', y') \mapsto (x'/\omega^2, y'/\omega^3)$ maps elements of $E'(\mathbb{F}_q)$ to elements of $E(\mathbb{F}_{q^2})$ whereas the restriction of Ψ^{-1} to $\Psi(E'(\mathbb{F}_q))$ reverses the map by mapping some elements of $E(\mathbb{F}_{q^2})$ to elements of $E'(\mathbb{F}_q)$.

Cubic twist : $d = 3$

This type of twist is only possible with elliptic curves having $a = 0$. If $E/\mathbb{F}_q : y^2 = x^3 + b$, then a cubic twist is defined by $E'/\mathbb{F}_q : y^2 = x^3 + b\omega^6$ for any ω satisfying $\omega^3 \in \mathbb{F}_q$ and $\omega \in \mathbb{F}_{q^3} \backslash \mathbb{F}_q$. The isomorphism $\Psi : E \to E'$ defined by $\Psi : (x', y') \mapsto (x'/\omega^2, y'/\omega^3)$ maps elements of $E'(\mathbb{F}_q)$ to elements of $E(\mathbb{F}_{q^3})$ whereas the restriction of Ψ^{-1} to $\Psi(E'(\mathbb{F}_q))$ reverses the map by mapping some elements of $E(\mathbb{F}_{q^3})$ to elements of $E'(\mathbb{F}_q)$.

Quartic twist : $d = 4$

This type of twist is only possible with elliptic curves having $b = 0$. If $E/\mathbb{F}_q : y^2 = x^3 + ax$, then a quartic twist is defined by $E'/\mathbb{F}_q : y^2 = x^3 + a\omega^4 x$ for any ω satisfying $\omega^4 \in \mathbb{F}_q$ and $\omega \in \mathbb{F}_{q^4} \backslash \mathbb{F}_{q^2}$. The isomorphism $\Psi : E \to E'$ defined by $\Psi : (x', y') \mapsto (x'/\omega^2, y'/\omega^3)$ maps elements of $E'(\mathbb{F}_q)$ to elements of $E(\mathbb{F}_{q^4})$ whereas the restriction of Ψ^{-1} to $\Psi(E'(\mathbb{F}_q))$ reverses the map by mapping some elements of $E(\mathbb{F}_{q^4})$ to elements of $E'(\mathbb{F}_q)$.

Sextic twist : $d = 6$

This type of twist is only possible with elliptic curves having $a = 0$. If $E/\mathbb{F}_q : y^2 = x^3 + b$, then a sextic twist is defined by $E'/\mathbb{F}_q : y^2 = x^3 + b\omega^6$ for any ω satisfying $\omega^6 \in \mathbb{F}_q$ and $\omega \in \mathbb{F}_{q^6} \backslash \mathbb{F}_{q^i}$ for $i \in \{2, 3\}$. The isomorphism $\Psi : E \rightarrow E'$ defined by $\Psi : (x', y') \mapsto (x'/\omega^2, y'/\omega^3)$ maps elements of $E'(\mathbb{F}_q)$ to elements of $E(\mathbb{F}_{q^6})$ whereas the restriction of Ψ^{-1} to $\Psi(E'(\mathbb{F}_q))$ reverses the map by mapping some elements of $E(\mathbb{F}_{q^6})$ to elements of $E'(\mathbb{F}_q)$.

2.4.3 Pairing-friendly curves

The goal is to compute pairings $e(P, Q)$ where P is a point in $E(\mathbb{F}_p)$ and Q is a point in $E(\mathbb{F}_{p^k})$. Since working over the extension field is quite expensive, it is preferable to have low extension degree. This also means that it is preferable to have small embedding degree because the embedding degree is directly related to the degree of the extension field. Note that the security of pairing-based cryptography (partially) relies on the largest prime order subgroup r and the embedding degree k such that p^k is big enough because of index calculus attacks. Therefore, it is desirable to have a large r and k. Having fulfilled these two requirements, elliptic curves with small embedding degree and large prime order subgroup are called *pairing-friendly curves*.

It is rather ambiguous when saying “small” embedding and “large” prime order subgroup. In [FST10], a more concrete definition of pairing-friendly curves is given. Let E be an elliptic curve defined over a finite field $\mathbb{F}_p$. The curve E is called a pairing-friendly curve if it satisfies the following properties:
(1) there is a prime $r \geq \sqrt{p}$ dividing $\#E(\mathbb{F}_p)$ and
(2) the embedding degree of E with respect to r is less than $\log_2(r)/8$.

In general, it is extremely rare that any randomly chosen elliptic curve would be pairing-friendly. Therefore, it requires specific methods to generate pairing-friendly curves. Algorithms to generate pairing-friendly curves are summarized in [FST10].

2.5 Side-channel attacks

In theory, it should suffice to base the difficulty of breaking cryptosystems on mathematically hard problems. However, this does not totally apply in practice due to *side-channel attacks* [Koc96] which try to discover secret information via physical measurements such as electromagnetic radiation, power consumption, run time, and noise. Examples of side-channel attacks include timing attacks and power attacks.

2.5.1 Constant time

The obvious solution to protect against timing attacks is to ensure that algorithms run in constant time regardless of secret input(s). Among previously explained algorithms (assuming all scalars have the same known bitlength):

- *double-and-add* (Algorithm 2.1) is non-constant time because whether to perform "if" condition depends on the secret scalar.
- *double-and-add-always* (Algorithm 2.2), on the other hand, is constant time because each iteration always performs the same operations, i.e., one doubling and one addition.
- *sliding window* (Algorithm 2.4) is non-constant time because how many bits to slide depends on the secret scalar.
- *fixed window* (Algorithm 2.3), in contrast, is constant time because it always performs the same number of additions regardless of the scalar (assuming that the addition of $0P$ is not free).
- *double-base chain* (Section 2.3.4) is non-constant time because the pattern of doubling, tripling and addition varies for different scalars.

2.5.2 Conditional branching

Besides having implementations run in constant time in order to prevent software side-channel attacks, the implementations should contain neither input-dependent branches nor input-dependent array indices. If implementations do contain conditional branches, the selection should be done via arithmetic.

For example, let c be a condition bit. If c is set, then s is assigned to t. Otherwise, t remains the same. This conditional branch can be represented using arithmetic instructions as

$$t = (t \cdot (1 - c)) + (s \cdot \ (c)\ ),$$

or logic instructions as

$$t = (t \wedge (c - 1)) \vee (s \wedge (-c)).$$

Using arithmetic instructions, if $c = 1$ then $1 - c$ becomes 0 which also makes the first term 0. Therefore, the value to assign to t depends on the second term which is $s \cdot c = s \cdot 1 = s$, meaning s is assigned to t as intended. On the other hand, if $c = 0$ then the second term becomes 0. Therefore, the value to assign to t depends on the first term which is $t \cdot (1 - c) = t \cdot 1 = t$, meaning t is assigned to t, i.e., t stays the same.

Using logic instructions, if $c = 1$ then $c - 1$ becomes 0 and $-c$ becomes -1, which correspond to all zero and all one respectively in bit representation. The first half $(t \wedge 0)$ means t is masked by all zero, i.e., $t \wedge 000\ldots0 = 0$. The second half $(s \wedge -1)$ means s is masked by all one, i.e., $s \wedge 111\ldots1 = s$. Therefore, $0 \vee s = s$, meaning s is assigned to t as intended. On the other hand, if $c = 0$ then $c - 1$ becomes -1 and $-c$ becomes 0. The first half $(t \wedge -1)$ means t is masked by all one, i.e., $t \wedge 111\ldots1 = t$. The second half $(s \wedge 0)$ means s is masked by all zero, i.e., $s \wedge 000\ldots0 = 0$. Therefore, $t \vee 0 = t$, meaning t is assigned to t or t stays the same.

2.5.3 Table lookup

If a lookup table is used, retrieving entries from the table should also be constant-time. This can be done by reading the entire table and selecting the particular entry. To select the entry, the concept of conditional branch selection via arithmetic (as explained in the previous subsection) should also be applied.

3

Twisted Hessian Curves

All of NIST's standard prime-field elliptic curves have cofactor 1. However, by now there is overwhelming evidence that cofactor 1 does not provide the best performance/security tradeoff for elliptic-curve cryptography. All of the latest speed records for ECC are set by curves with cofactor divisible by 2, with base fields $\mathbb{F}_p$ where p is a square, and with extra endomorphisms: Faz-Hernández–Longa–Sánchez [FHLS14] use a twisted Edwards GLS curve with cofactor 8 over $\mathbb{F}_p$ where $p = (2^{127} - 5997)^2$; Oliveira–López–Aranha–Rodríguez-Henríquez [OLARH13] use a GLV+GLS curve with cofactor 2 over $\mathbb{F}_p$ where $p = 2^{254}$; and Costello–Hisil–Smith [CHS14] use a Montgomery $\mathbb{Q}$-curve with cofactor 4 (and twist cofactor 8) over $\mathbb{F}_p$ where $p = (2^{127} - 1)^2$. Similarly, for "conservative" ECC over prime fields without extra endomorphisms, Bernstein [Ber06a] uses a Montgomery curve with cofactor 8 (and twist cofactor 4), and Bernstein–Duif–Lange–Schwabe–Yang [BDL+12] use an equivalent twisted Edwards curve.

The very fast Montgomery ladder for Montgomery curves [Mon87] was published at the dawn of ECC, and its speed always relied on a cofactor divisible by 4. However, for many years the benefit of such cofactors seemed limited to ladders for variable-base single-scalar multiplication. Cofactor 1 seemed slightly faster than cofactor 4 for signature generation and signature verification; NIST's curves were published in the context of a signature standard. Many years of investigations of addition formulas for a wide range of curve shapes (see, e.g., [CC86], [CMO98], [JQ01], [LS01], and [BJ03]) failed to produce stronger arguments for cofactors above 1 — until the advent [Edw07] and performance analysis [BL07] of Edwards curves.

Several papers have tried to exploit a different cofactor, namely 3, as follows. Hessian curves $x^3 + y^3 + 1 = dxy$, which always have points of order 3 over finite fields, have a very simple and symmetric addition law due to Sylvester. Chudnovsky–

Chudnovsky in [CC86] already observed that this law requires just 12**M** in projective coordinates. However, Hessian doublings were much slower than Jacobian-coordinate Weierstrass doublings; this slowdown outweighed the addition speedup, since (in most applications) doublings are much more frequent than additions. The best way to handle a curve with cofactor 3 was to forget about the points of order 3 and simply use the same formulas used for curves with cofactor 1.

This chapter shows how to use cofactor 3 to beat the best available results for cofactor 1. The results do not beat cofactor 4 but significantly narrow the gap.

Credits. The content of this chapter is based on the paper *"Twisted Hessian curves"* [BCKL15] which is a joint work with Daniel J. Bernstein, David Kohel and Tanja Lange. Note that "we" in this chapter is used to refer to the aforementioned authors.

Organization of this chapter. Section 3.1 describes approaches of this work and previous related results. Section 3.2 states definitions of elliptic curves and relations to twisted Hessian form. Section 3.3 reviews the standard addition laws. Section 3.4 shows how to generalize to the rotated addition law. Section 3.5 explains the conversion from Weierstrass to Hessian curves and vice versa. Section 3.6 presents new doubling and tripling formulas. Section 3.7 compares results to different curve shapes.

3.1 Literature reviews

We now review previous speeds and compare them to our speeds. We adopt the following rules to maximize comparability:

- For individual elliptic-curve operations we count multiplications and squarings. **M** is the cost of a multiplication, and **S** is the cost of a squaring. We do not count additions or subtractions. (Computer-verified operation counts for our formulas, including counts of additions and subtractions, appear in the latest update of EFD [BLb].)
- In summaries of scalar-multiplication performance we take $\mathbf{S} = 0.8\mathbf{M}$ as commonly used. Of course, squarings are much faster than multiplications in characteristic 2, but we emphasize the case of large characteristic.
- We also count multiplications by curve parameters: e.g., $\mathbf{m}_d$ is the cost of multiplying by d. We assume that curves are sensibly chosen with small d. In summaries we take $\mathbf{m}_d = 0$.
- We do not include the cost of final conversion to affine coordinates. We also assume that inversion is not fast enough to make intermediate inversions useful. Consequently the exact cost of inversion does not appear.
- We focus on the traditional case of variable-base single-scalar multiplication, in particular for average 256-bit scalars. Beware that this is only loosely correlated with other scalar-multiplication tasks. (Other tasks tend to rely more on additions, so the fast complete addition law for twisted Hessian curves should provide an even larger benefit compared to Weierstrass curves.)

Bernstein–Lange in [BL08] analyzed scalar-multiplication performance on several curve shapes and concluded, under these assumptions, that Weierstrass curves $y^2 = x^3 - 3x + b$ in Jacobian coordinates used 9.34**M** per bit on average, and that Hessian curves were slower. Bernstein–Birkner–Lange–Peters in [BBLP07] used double-base chains (doublings, triplings, and additions) to considerably speed up Hessian curves to 9.65**M** per bit and to slightly speed up Weierstrass curves to 9.29**M** per bit. Hisil in [Hış10, Table 6.4], without double-base chains, reported more than 10**M** per bit for Hessian curves.

Our new results are just 8.73**M** per bit. This means that one actually gains something by taking advantage of a point of order 3. The new speeds require a base field with $6 \neq 0$ and with fast multiplication by a primitive cube root of 1, such as a field of the form $\mathbb{F}_p[\omega]/(\omega^2 + \omega + 1)$ where $p \in 2 + 3\mathbb{Z}$. This quadratic field structure might seem to constrain the applicability of the results, but (1) GLS-curve and $\mathbb{Q}$-curve results already show that a quadratic field structure is desirable for performance; (2) there is also a fast primitive cube root of 1 in, e.g., the prime field $\mathbb{F}_p$ where $7p = 2^{298} + 2^{149} + 1$; (3) we do not lose much speed from more general fields (the cost of a tripling increases by 0.4**M**). Note that the 8.73**M** per bit does not use the speedups in (1). Our speedups can be combined with the speedups in (1) but we have not quantified the resulting performance.

3.1.1 Completeness, side channels, and precomputation

For a large fraction of curves, the formulas we use have a further benefit not reflected in the multiplication counts stated above: namely, the formulas are *complete*. This means that the formulas work for all curve points. The implementor does not have to waste any time checking for exceptional cases, and does not have to worry that an attacker can generate inputs that trigger exceptional cases: there are no exceptional cases. (For comparison, a *strongly unified* but incomplete addition law works for most additions and works for most doublings, but still has exceptional cases. The traditional addition law for Weierstrass curves is not even strongly unified: it consistently fails for doublings.)

Often completeness is used as part of a side-channel defense; see, e.g., [BL07, Section 8]. In this chapter we focus purely on speed: we do not limit attention to scalar-multiplication techniques that are safe inside applications that expose secret scalars to side-channel attacks. Note that scalars are public in many cryptographic protocols, such as signature verification, and also in many other elliptic-curve computations, such as the elliptic-curve method of integer factorization.

We also allow scalar-multiplication techniques that rely on scalar-dependent precomputation. This is reasonable for applications that reuse a single scalar many times. For example, in the context of signatures, the *signer* can carry out the precomputation and compress the results into the signature. The signer can also choose different techniques for different scalars: in particular, there are some scalars where our cofactor-3 techniques are even faster than cofactor 4. One can easily find, and we suggest choosing, curves of cofactor 12 that simultaneously allow the current cofactor-3 and cofactor-4 methods; these curves are also likely to be able to take advantage of any future improvements in cofactor-3 and cofactor-4 methods.

3.1.2 Tools and techniques

At a high level, we use a tree search for double-base chains, allowing windows and taking account of the costs of doublings, triplings, and additions. At a lower level, we use tripling formulas that take $6\mathbf{M}+6\mathbf{S}$, doubling formulas that take $6\mathbf{M}+2\mathbf{S}$, and addition formulas that take $11\mathbf{M}$; in this overview we ignore multiplications by constants. These formulas work in projective coordinates for Hessian curves.

Completeness relies on two further tools. First, we use a rotated addition law. Unlike the standard (Sylvester) addition law, the rotated addition law is strongly unified. In fact, the rotated addition law works in every case where the standard addition law fails; i.e., the two laws together form a complete system of addition laws. Second, we work more generally with twisted Hessian curves $ax^3+y^3+1=dxy$. If a is not a cube then the rotated addition law by itself is complete. The doubling formulas and tripling formulas are also complete, meaning that they have no exceptional cases. The generalization also provides more flexibility in finding curves with small parameters.

For comparison, Jacobian coordinates for Weierstrass curves $y^2=x^3-3x+b$ use $7\mathbf{M}+7\mathbf{S}$ for tripling, $3\mathbf{M}+5\mathbf{S}$ for doubling, and $11\mathbf{M}+5\mathbf{S}$ for addition. This saves $3(\mathbf{M}-\mathbf{S})$ in doubling but loses $\mathbf{M}+\mathbf{S}$ in tripling and loses $5\mathbf{S}$ in addition. Given these operation counts it is not a surprise that we beat Weierstrass curves.

$6\mathbf{M}+6\mathbf{S}$ triplings were achieved once before, namely by tripling-oriented Doche–Icart–Kohel curves [DIK06]. Those curves also offer $2\mathbf{M}+7\mathbf{S}$ doublings, competitive with our $6\mathbf{M}+2\mathbf{S}$. However, the best addition formulas known for those curves take $11\mathbf{M}+6\mathbf{S}$, even slower than Weierstrass curves.

As noted earlier, Edwards curves are still faster for average scalars, thanks to their particularly fast doublings and additions. However, we do beat Edwards curves for scalars that involve many triplings.

3.1.3 Priority dates

Hessian curves and the standard addition law are classical material. The rotated addition law, the fact that the rotated addition law is strongly unified, the concept of twisted Hessian curves, the generalization of the addition laws to twisted Hessian curves, the complete system of addition laws, and the completeness of the rotated addition law for non-cube a are all due to this chapter. We announced the essential details online in July 2009 (e.g., stating the completeness result in [Ber09b, page 40], and contributing a "twisted Hessian" section to EFD), but our twisted Hessian curves paper [BCKL15], matching this chapter, is our first formal publication of these results.

The speeds that we announced at that time for twisted Hessian curves were no better than known speeds for standard formulas for Hessian curves: $8\mathbf{M}+6\mathbf{S}$ for tripling, $6\mathbf{M}+3\mathbf{S}$ for doubling, and $12\mathbf{M}$ for addition. Followup work found better formulas for all of these operations. Almost all of those formulas are superseded by formulas that we now present; the only exception is that we use $11\mathbf{M}$ addition formulas [Hış10] from Hisil. See Table 3.1 for an overview.

Table 3.1: Costs of various formulas for Hessian curves in projective coordinates.

	T	S	2	3	>	Cost	Source
doubling			✓	✓	✓	$6\mathbf{M}+3\mathbf{S}\approx 8.4\mathbf{M}$	Chudnovsky–Chudnovsky [CC86]
	✓		✓	✓	✓	$6\mathbf{M}+3\mathbf{S}\approx 8.4\mathbf{M}$	our 2009 announcement
				✓	✓	$3\mathbf{M}+6\mathbf{S}\approx 7.8\mathbf{M}$	Hisil–Carter–Dawson [HCD07]
			✓	✓	✓	$7\mathbf{M}+1\mathbf{S}\approx 7.8\mathbf{M}$	Hisil–Carter–Dawson [HCD07]
	✓			✓	✓	$6\mathbf{M}+2\mathbf{S}\approx 7.6\mathbf{M}$	this chapter
addition		✓		✓	✓	$9\mathbf{M}+6\mathbf{S}\approx 13.8\mathbf{M}$	Hisil–Wong–Carter–Dawson [HWCD09]
			✓	✓	✓	$12\mathbf{M}=12.0\mathbf{M}$	Chudnovsky–Chudnovsky [CC86]
	✓	✓	✓	✓	✓	$12\mathbf{M}=12.0\mathbf{M}$	our 2009 announcement
	✓	✓		✓	✓	$11\mathbf{M}=11.0\mathbf{M}$	Hisil [Hış10]
tripling				✓	✓	$8\mathbf{M}+6\mathbf{S}\approx 12.8\mathbf{M}$	Hisil–Carter–Dawson [HCD07]
	✓			✓	✓	$8\mathbf{M}+6\mathbf{S}\approx 12.8\mathbf{M}$	our 2009 announcement
	✓		✓			$7\mathbf{M}+6\mathbf{S}\approx 11.8\mathbf{M}$	Farashahi–Joye [FJ10]
	✓			✓		$8\mathbf{M}+4\mathbf{S}\approx 11.2\mathbf{M}$	Farashahi–Wu–Zhao [FWZ12]
	✓			✓	✓	$8\mathbf{M}+4\mathbf{S}\approx 11.2\mathbf{M}$	Kohel [Koh15]
	✓		✓	✓	✓	$8\mathbf{M}+4\mathbf{S}\approx 11.2\mathbf{M}$	this chapter
	✓		✓		✓	$6\mathbf{M}+6\mathbf{S}\approx 10.8\mathbf{M}$	this chapter, assume fast primitive $\sqrt[3]{1}$

Note: Costs are sorted using the assumption $\mathbf{S}\approx 0.8\mathbf{M}$; note that $\mathbf{S}/\mathbf{M}$ is normally much smaller in characteristic 2. "T" means that the formula was stated for twisted Hessian curves, not just Hessian curves; all of the "T" formulas are complete for suitable curves. "S" means "strongly unified": an addition formula that also works for doubling. "2" means that the formula works in characteristic 2. "3" means that the formula works in characteristic 3. ">" means that the formula works in characteristic above 3.

Tripling: One of the followup papers [FJ10], by Farashahi–Joye, reported $7\mathbf{M}+6\mathbf{S}$ for twisted Hessian tripling, but only for characteristic 2. Another followup paper [FWZ12], by Farashahi–Wu–Zhao, reported 4 multiplications and 4 cubings, overall $8\mathbf{M}+4\mathbf{S}$, for Hessian tripling, but only for characteristic 3. Further followup work [Koh15], by Kohel, reported 4 multiplications and 4 cubings for twisted Hessian tripling in any odd characteristic. In Section 3.6 we generalize the approach of [Koh15] and show how a better specialization reduces cost to just 6 cubings, assuming that the field has a fast primitive cube root of 1.

Doubling: In Section 3.6 we present four doubling formulas, starting with $6\mathbf{M}+3\mathbf{S}$ and culminating with $6\mathbf{M}+2\mathbf{S}$. In the case $a=1$, the first formula was already well known before our work. Hisil, Carter and Dawson in [HCD07] had already introduced doubling formulas using $3\mathbf{M}+6\mathbf{S}$, and also introduced doubling formulas using $7\mathbf{M}+1\mathbf{S}$, using techniques that seem to be specific to small cube a such as $a=1$; see also [Hış10]. Our $6\mathbf{M}+2\mathbf{S}$ is better than $7\mathbf{M}+1\mathbf{S}$ if $\mathbf{S}<\mathbf{M}$, and is better than $3\mathbf{M}+6\mathbf{S}$ if $\mathbf{S}>0.75\mathbf{M}$.

At a higher level, double-base chains have been explored in several papers. The idea of a tree search for double-base chains was introduced by Doche and Habsieger in [DH08]. The tree search in [DH08] tries to minimize the number of additions used in a double-base chain, ignoring the cost of doublings and triplings; we do better by using the cost of doublings and triplings to adjust the weights of nodes in the tree.

3.2 Elliptic curves in twisted Hessian form

Definition 3.1. *Let K be a field. A* **projective twisted Hessian curve** *over K is a curve of the form $aX^3 + Y^3 + Z^3 = dXYZ$ in $\mathbb{P}^2$ with specified point $(0 : -1 : 1)$, where a, d are elements of K with $a(27a - d^3) \neq 0$.*

Theorem 3.1 below states that any projective twisted Hessian curve is an elliptic curve. The correponding affine curve $ax^3 + y^3 + 1 = dxy$ with specified point $(0, -1)$ is an **affine twisted Hessian curve**.

We state theorems for the projective curve, and allow the reader to deduce corresponding theorems for the affine curve. When we say "Let H be the twisted Hessian curve $aX^3 + Y^3 + Z^3 = dXYZ$ over K" we mean that a, d are elements of K, that $a(27a - d^3) \neq 0$, and that H is the projective twisted Hessian curve $aX^3 + Y^3 + Z^3 = dXYZ$ in $\mathbb{P}^2$ with specified point $(0 : -1 : 1)$. Some theorems need, and state, further assumptions such as $d \neq 0$.

The special case $a = 1$ of a twisted Hessian curve is simply a **Hessian curve**. The twisted Hessian curve $aX^3 + Y^3 + Z^3 = dXYZ$ is isomorphic to the Hessian curve $\bar{X}^3 + Y^3 + Z^3 = (d/a^{1/3})\bar{X}YZ$ over any extension of K containing a cube root $a^{1/3}$ of a: simply take $\bar{X} = a^{1/3}X$. Similarly, taking $\bar{X} = dX$ when $d \neq 0$ shows that the twisted Hessian curve for (a, d) is isomorphic to the twisted Hessian curve for $(a/d^3, 1)$; but we retain a and d as separate parameters to allow more curves with small parameters and thus with fast arithmetic.

Hessian curves have a long history, but twisted Hessian curves do not. The importance of twisted Hessian curves, beyond their extra generality, is that they have a complete addition law when a is not a cube. See Theorem 3.10 below.

3.2.1 Proof strategy: twisted Hessian curves as foundations

One can use the first isomorphism stated above to derive many features of twisted Hessian curves from corresponding well-known features of Hessian curves. We instead give direct proofs in the general case, meant as replacements for the older proofs in the special case: in other words, we propose starting with the theory of twisted Hessian curves rather than starting with the theory of Hessian curves. This reduces the total proof length: the extra cost of tracking a through the proofs is smaller than the extra cost of applying the isomorphism.

We do not claim that this tracking involves any particular difficulty. In one case the tracking has been done before: specifically, some of the nonsingularity computations in Theorem 3.1 are special cases of classical discriminant computations for ternary cubics $aX^3 + bY^3 + cZ^3 = dXYZ$. See, e.g., [Aro50] and [Cay81]. However, the classical computations were carried out in characteristic 0, and the range of validity of the computations is not always obvious. Many of the computations fail in characteristic 3, even though Theorem 3.1 is valid in characteristic 3. Since the complete proofs are straightforward we simply include them here.

Similarly, one can derive many features of twisted Hessian curves from corresponding well-known features of Weierstrass curves, but we instead give direct proofs. We do use Weierstrass curves inside Theorem 3.14, which proves a property of all elliptic curves having points of order 3.

3.2.2 Notes on definitions of Hessian curves

There are various superficial differences among the definitions of Hessian curves in the literature. First, often characteristic 3 is prohibited. For example, [Sma01] considers only base fields $\mathbb{F}_p$ with $p \in 2+3\mathbb{Z}$, and [JQ01] considers only characteristics larger than 3. Our main interest is in the case $p \in 1+3\mathbb{Z}$, and in any event we see no reason to restrict the characteristic in the definition.

Second, often constants are introduced into the parameter d. For example, [JQ01] defines a Hessian curve as $X^3+Y^3+Z^3=3dXYZ$, and the curve actually considered by Hesse in [Hes44, page 90, formula 54] was $X^3+Y^3+Z^3+6dXYZ=0$.

Third, the specified point is often taken as a point at infinity, specifically $(-1:1:0)$; see, e.g., [CC86]. We use an affine point $(0:-1:1)$ to allow completeness of the *affine* twisted Hessian curve rather than merely completeness of the *projective* twisted Hessian curve; if a is not a cube then there are no points at infinity for implementors to worry about. Converting addition laws (and twists and so on) between these two choices of neutral element is a trivial matter of permuting X, Y, Z.

3.2.3 Notes on definitions of elliptic curves

There are also various differences among the definitions of elliptic curves in the literature. The most specific definitions would say that Hessian curves are not elliptic curves: for example, Koblitz in [Kob98, page 117] defines elliptic curves to have long Weierstrass form. Obviously we do not use such restrictive definitions.

Two classical definitions that allow Hessian curves are as follows: (1) an elliptic curve is a nonsingular cubic curve in $\mathbb{P}^2$ with a specified point; (2) an elliptic curve is a nonsingular cubic curve in $\mathbb{P}^2$ with a specified *inflection* point. The importance of the inflection-point condition is that it allows the traditional geometric addition law: three distinct curve points on a line have sum 0; more generally, all curve points on a line, counted with multiplicity, have sum 0. If the specified point were not an inflection point then the addition law would be more complicated. See, e.g., [Hus03, Chapter 3, Theorem 1.2].

We take the first of these two definitions. The statement that any twisted Hessian curve H is elliptic (Theorem 3.1) thus means that H is a nonsingular cubic curve with a specified point. We prove separately (Theorem 3.2) that the specified point $(0:-1:1)$ is an inflection point.

These definitions are still not broad enough to allow, e.g., Edwards curves as elliptic curves. Edwards curves in $\mathbb{P}^2$ are singular and not cubic; the Arène–Lange–

Naehrig–Ritzenthaler geometric addition law [ALNR11] for Edwards curves is not the traditional geometric addition law; etc. “Elliptic curve” is often defined more broadly as “smooth projective genus-1 curve with a specified point”, but this leaves ambiguous whether a “projective curve” is a curve for which there *exists* an embedding into projective space or a curve *equipped with* an embedding into projective space. With the first notion, the concept of addition laws for a curve is ill-defined, as is any other concept that relies on choices of coordinates. The second notion does not admit, e.g., Edwards curves in $\mathbb{P}^1 \times \mathbb{P}^1$ as elliptic curves; it does allow Edwards curves in $\mathbb{P}^3$, but the switch from $\mathbb{P}^1 \times \mathbb{P}^1$ to $\mathbb{P}^3$ damages the performance of doublings, so this definition is not broad enough for a serious analysis of performance. We avoid further discussion of ways to define elliptic curves in more generality: all of our theorems are focused on twisted Hessian curves, and then the classical definitions suffice.

Theorem 3.1. *Let H be the twisted Hessian curve $aX^3 + Y^3 + Z^3 = dXYZ$ over a field K. Then H is an elliptic curve.*

Proof. $aX^3+Y^3+Z^3 = dXYZ$ is a cubic curve in $\mathbb{P}^2$, and $(0 : -1 : 1)$ is a point on the curve. What remains is to prove that this curve is nonsingular.

Recall that $a(27a-d^3) \neq 0$ by definition of twisted Hessian curves. A singularity $(X : Y : Z) \in \mathbb{P}^2$ of $aX^3 + Y^3 + Z^3 = dXYZ$ satisfies $3aX^2 = dYZ$, $3Y^2 = dXZ$, and $3Z^2 = dXY$. We will deduce $X = Y = Z = 0$, contradicting $(X : Y : Z) \in \mathbb{P}^2$.

Case 1: $3 \neq 0$ in K. Multiply to obtain $27aX^2Y^2Z^2 = d^3X^2Y^2Z^2$, i.e., $(27a - d^3)X^2Y^2Z^2 = 0$. By hypothesis $27a - d^3 \neq 0$, so $X^2Y^2Z^2 = 0$, so $X = 0$ or $Y = 0$ or $Z = 0$.

- Case 1.1: $X = 0$. Then $3Y^2 = 0$ and $3Z^2 = 0$ so $Y = 0$ and $Z = 0$ as claimed.
- Case 1.2: $Y = 0$. Then $3aX^2 = 0$ and $3Z^2 = 0$, and $a \neq 0$ by hypothesis, so $X = 0$ and $Z = 0$ as claimed.
- Case 1.3: $Z = 0$. Then $3aX^2 = 0$ and $3Y^2 = 0$, and again $a \neq 0$, so $X = 0$ and $Y = 0$ as claimed.

Case 2: $3 = 0$ in K. Then $dYZ = 0$ and $dXZ = 0$ and $dXY = 0$. By hypothesis $a(-d^3) \neq 0$, so $d \neq 0$, so at least two of the coordinates X, Y, Z are 0.

- Case 2.1: $X = Y = 0$. Then the curve equation $aX^3 + Y^3 + Z^3 = dXYZ$ forces $Z^3 = 0$ so $Z = 0$ as claimed.
- Case 2.2: $X = Z = 0$. Then the curve equation forces $Y^3 = 0$ so $Y = 0$ as claimed.
- Case 2.3: $Y = Z = 0$. Then the curve equation forces $aX^3 = 0$, and $a \neq 0$ by hypothesis, so $X = 0$ as claimed.

□

Theorem 3.2. *Let H be the twisted Hessian curve $aX^3 + Y^3 + Z^3 = dXYZ$ over a field K. Then $(0 : -1 : 1)$ is an inflection point on H.*

Proof. We claim that $(0 : -1 : 1)$ is the only point of intersection of the line $-3(Y + Z) = dX$ with the curve $aX^3+Y^3+Z^3 = dXYZ$ over any extension of K. Consequently, by Bézout’s theorem, this point has intersection multiplicity 3. (An alternative proof, involving essentially the same calculation, computes the multiplicity directly from its definition.)

To prove the claim, assume that $-3(Y+Z) = dX$ and $aX^3 + Y^3 + Z^3 = dXYZ$. Then $(27a - d^3)X^3 = 27aX^3 - (-3(Y+Z))^3 = 27(aX^3 + (Y+Z)^3) = 27(aX^3 + Y^3 + Z^3 + 3(Y+Z)YZ) = 27(dXYZ - dXYZ) = 0$ so $X^3 = 0$ so $X = 0$. Now $Y + Z = 0$: this follows from $-3(Y+Z) = dX = 0$ if $3 \neq 0$ in K, and it follows from $Y^3 + Z^3 = 0$ if $3 = 0$ in K. Thus $(X : Y : Z) = (0 : -1 : 1)$. □

3.3 The standard addition law

Theorem 3.4 states an addition law for twisted Hessian curves. We originally derived this addition law as follows:

- Start from Sylvester's addition law for $X^3 + Y^3 + Z^3 = dXYZ$. See, e.g., [CC86, page 425, equation 4.21i].
- Observe, as noted in [CC86], that the addition law is independent of d.
- Conclude that the addition law also works for $X^3 + Y^3 + Z^3 = (d/c)XYZ$, where c is a cube root of a.
- Permute X, Y, Z to our choice of neutral element.
- Replace X with cX.
- Rescale the outputs X_3, Y_3, Z_3 by a factor c.

The resulting polynomials X_3, Y_3, Z_3 are identical to Sylvester's addition law: they are independent of curve parameters, and in particular are independent of a. We refer to this addition law as the **standard addition law**. For reasons explained in Section 3.2, we prove Theorem 3.4 here by giving a direct proof of the standard addition law for the general case, rather than deriving the general case from the special case $a = 1$.

The standard addition law is never complete: it fails whenever $(X_2 : Y_2 : Z_2) = (X_1 : Y_1 : Z_1)$. More generally, it fails if and only if $(X_2 : Y_2 : Z_2) - (X_1 : Y_1 : Z_1)$ has the form $(0 : -\omega : 1)$ where $\omega^3 = 1$, or equivalently $(X_2 : Y_2 : Z_2) = (\omega^2 X_1 : \omega Y_1 : Z_1)$. See Theorem 3.11 for the equivalence, and Theorem 3.5 for the failure analysis.

A different way to analyze the failure cases, with somewhat less calculation, is as follows. First prove that $(X_2 : Y_2 : Z_2)$ has the form $(0 : -\omega : 1)$ if and only if the addition law fails to add the neutral element $(0 : -1 : 1)$ to $(X_2 : Y_2 : Z_2)$. Then use a theorem of Bosma and Lenstra [BHWL95, Theorem 2] stating that the set of failure cases of a degree-$(2,2)$ addition law for a cubic elliptic curve in $\mathbb{P}^2$ is a union of shifted diagonals $\Delta_S = \{(P_1, P_1 + S)\}$. The theorems in [BHWL95] are stated only for Weierstrass curves, but they are invariant under linear equivalence and thus also apply to twisted Hessian curves. See [Koh11] for a generalization to elliptic curves embedded in projective space of any dimension.

Theorems 3.7 and 3.10 below introduce a new addition law that (1) works for all doublings on any twisted Hessian curve and (2) is complete for any twisted Hessian curve with non-cube a.

Theorem 3.3. *Let H be the twisted Hessian curve $aX^3 + Y^3 + Z^3 = dXYZ$ over a field K. Let X_1, Y_1, Z_1 be elements of K such that $(X_1 : Y_1 : Z_1) \in H(K)$. Then $-(X_1 : Y_1 : Z_1) = (X_1 : Z_1 : Y_1)$.*

Proof. Recall that the specified neutral element of the curve is $(0 : -1 : 1)$.

Case 1: $(X_1 : Y_1 : Z_1) \neq (X_1 : Z_1 : Y_1)$. Then $X_1(Y + Z) = X(Y_1 + Z_1)$ is a line in $\mathbb{P}^2$: if all its coefficients $-Y_1 - Z_1, X_1, X_1$ are 0 then $(X_1 : Y_1 : Z_1) = (0 : -1 : 1) = (X_1 : Z_1 : Y_1)$, contradiction. This line intersects the curve at the distinct points $(0 : -1 : 1)$, $(X_1 : Y_1 : Z_1)$, and $(X_1 : Z_1 : Y_1)$. Hence $-(X_1 : Y_1 : Z_1) = (X_1 : Z_1 : Y_1)$.

Case 2: $(X_1 : Y_1 : Z_1) = (X_1 : Z_1 : Y_1)$ and $X_1 \neq 0$. Again $(X_1 : Y_1 : Z_1) \neq (0 : -1 : 1)$, and again $X_1(Y + Z) = X(Y_1 + Z_1)$ is a line. This line intersects the curve at both $(0 : -1 : 1)$ and $(X_1 : Y_1 : Z_1)$, and we show in a moment that it is the tangent to the curve at $(X_1 : Y_1 : Z_1)$. Hence $-(X_1 : Y_1 : Z_1) = (X_1 : Y_1 : Z_1) = (X_1 : Z_1 : Y_1)$.

For the tangent calculation we take coordinates $y = Y/X$ and $z = Z/X$. The curve is then $a + y^3 + z^3 = dyz$; the point P_1 is $(y_1, z_1) = (Y_1/X_1, Z_1/X_1)$, which by hypothesis satisfies $y_1 = z_1$; and the line is $y + z = y_1 + z_1$. The curve is symmetric between y and z, so its slope at $(y_1, z_1) = (z_1, y_1)$ must be -1, which is the same as the slope of the line.

Case 3: $(X_1 : Y_1 : Z_1) = (X_1 : Z_1 : Y_1)$ and $X_1 = 0$. Then $Y_1^3 + Z_1^3 = 0$ by the curve equation so $Y_1 = \lambda Z_1$ for some λ with $\lambda^3 = -1$; but $(Y_1 : Z_1) = (Z_1 : Y_1)$ implies $\lambda = 1/\lambda$, so $\lambda = -1$, so $(X_1 : Y_1 : Z_1) = (0 : -1 : 1)$. Hence $-(X_1 : Y_1 : Z_1) = (0 : -1 : 1) = (0 : 1 : -1) = (X_1 : Z_1 : Y_1)$. □

Theorem 3.4. *Let H be the twisted Hessian curve $aX^3 + Y^3 + Z^3 = dXYZ$ over a field K. Let $X_1, Y_1, Z_1, X_2, Y_2, Z_2$ be elements of K such that $(X_1 : Y_1 : Z_1), (X_2 : Y_2 : Z_2) \in H(K)$. Define*

$$X_3 = X_1^2 Y_2 Z_2 - X_2^2 Y_1 Z_1,$$
$$Y_3 = Z_1^2 X_2 Y_2 - Z_2^2 X_1 Y_1,$$
$$Z_3 = Y_1^2 X_2 Z_2 - Y_2^2 X_1 Z_1.$$

If $(X_3, Y_3, Z_3) \neq (0, 0, 0)$ then $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2) = (X_3 : Y_3 : Z_3)$.

Proof. The polynomial identity

$$\begin{aligned}
&aX_3^3 + Y_3^3 + Z_3^3 - dX_3Y_3Z_3 \\
&= (X_1^3Y_2^3Z_2^3 + Y_1^3X_2^3Z_2^3 + Z_1^3X_2^3Y_2^3 - 3X_1Y_1Z_1X_2^2Y_2^2Z_2^2)(aX_1^3 + Y_1^3 + Z_1^3 - dX_1Y_1Z_1) \\
&- (X_2^3Y_1^3Z_1^3 + Y_2^3X_1^3Z_1^3 + Z_2^3X_1^3Y_1^3 - 3X_2Y_2Z_2X_1^2Y_1^2Z_1^2)(aX_2^3 + Y_2^3 + Z_2^3 - dX_2Y_2Z_2)
\end{aligned}$$

implies that $(X_3 : Y_3 : Z_3) \in H(K)$. The rest of the proof uses the chord-and-tangent definition of addition to show that $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2) = (X_3 : Y_3 : Z_3)$.

If $(X_1 : Y_1 : Z_1) = (X_2 : Y_2 : Z_2)$ then $(X_3, Y_3, Z_3) = (0, 0, 0)$, contradiction. Assume from now on that $(X_1 : Y_1 : Z_1) \neq (X_2 : Y_2 : Z_2)$.

The line through $(X_1 : Y_1 : Z_1)$ and $(X_2 : Y_2 : Z_2)$ is $(Z_1Y_2 - Z_2Y_1)X + (X_1Z_2 - X_2Z_1)Y + (X_2Y_1 - X_1Y_2)Z = 0$. The polynomial identity

$$(Z_1Y_2 - Z_2Y_1)X_3 + (X_1Z_2 - X_2Z_1)Z_3 + (X_2Y_1 - X_1Y_2)Y_3 = 0$$

shows that $(X_3 : Z_3 : Y_3)$ is also on this line.

One would now like to conclude that $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2) = -(X_3 : Z_3 : Y_3)$, so $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2) = (X_3 : Y_3 : Z_3)$ by Theorem 3.3. The only difficulty is that $(X_3 : Z_3 : Y_3)$ might be the same as $(X_1 : Y_1 : Z_1)$ or $(X_2 : Y_2 : Z_2)$; the rest of the proof consists of verifying that, in these two cases, the line is the tangent to the curve at $(X_3 : Z_3 : Y_3)$.

We use two other easy polynomial identities. First, $X_1Y_2Y_3+Y_1Z_2X_3+Z_1X_2Z_3 = 0$. Second, $aX_1X_2X_3+Z_1Z_2Y_3+Y_1Y_2Z_3 = (aX_1^3+Y_1^3+Z_1^3)X_2Y_2Z_2-(aX_2^3+Y_2^3+Z_2^3)X_1Y_1Z_1$. The curve equations for $(X_1 : Y_1 : Z_1)$ and $(X_2 : Y_2 : Z_2)$ then imply $aX_1X_2X_3+Z_1Z_2Y_3+Y_1Y_2Z_3 = 0$.

Case 1: $(X_3 : Z_3 : Y_3) = (X_1 : Y_1 : Z_1)$. The two identities above then imply $X_1Y_2Z_1 + Y_1Z_2X_1 + Z_1X_2Y_1 = 0$ and $aX_1^2X_2 + Z_1^2Z_2 + Y_1^2Y_2 = 0$ respectively. Our line is $(Z_1Y_2 - Z_2Y_1)X + (X_1Z_2 - X_2Z_1)Y + (X_2Y_1 - X_1Y_2)Z = 0$, while the tangent to the curve at $(X_1 : Y_1 : Z_1)$ is $(3aX_1^2 - dY_1Z_1)X + (3Y_1^2 - dX_1Z_1)Y + (3Z_1^2 - dX_1Y_1)Z = 0$. To see that these lines are the same, observe that the cross product

$$\begin{pmatrix} (3Y_1^2 - dX_1Z_1)(X_2Y_1 - X_1Y_2) - (3Z_1^2 - dX_1Y_1)(X_1Z_2 - X_2Z_1) \\ (3Z_1^2 - dX_1Y_1)(Z_1Y_2 - Z_2Y_1) - (3aX_1^2 - dY_1Z_1)(X_2Y_1 - X_1Y_2) \\ (3aX_1^2 - dY_1Z_1)(X_1Z_2 - X_2Z_1) - (3Y_1^2 - dX_1Z_1)(Z_1Y_2 - Z_2Y_1) \end{pmatrix}$$

is exactly

$$\begin{pmatrix} 3X_2 & -3X_1 & dX_1 \\ 3Y_2 & -3Y_1 & dY_1 \\ 3Z_2 & -3Z_1 & dZ_1 \end{pmatrix} \begin{pmatrix} aX_1^3 + Y_1^3 + Z_1^3 - dX_1Y_1Z_1 \\ aX_1^2X_2 + Z_1^2Z_2 + Y_1^2Y_2 \\ X_1Y_2Z_1 + Y_1Z_2X_1 + Z_1X_2Y_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Case 2: $(X_3 : Z_3 : Y_3) = (X_2 : Y_2 : Z_2)$. Exchanging $(X_1 : Y_1 : Z_1)$ with $(X_2 : Y_2 : Z_2)$ replaces (X_3, Y_3, Z_3) with $(-X_3, -Y_3, -Z_3)$ and moves to case 1. □

Theorem 3.5. *In the situation of Theorem 3.4,* $(X_3, Y_3, Z_3) = (0, 0, 0)$ *if and only if* $(X_2 : Y_2 : Z_2) = (\omega^2X_1 : \omega Y_1 : Z_1)$ *for some* $\omega \in K$ *with* $\omega^3 = 1$.

Proof. If $(X_2 : Y_2 : Z_2) = (\omega^2X_1 : \omega Y_1 : Z_1)$ and $\omega^3 = 1$ then (X_3, Y_3, Z_3) is proportional to $(X_1^2\omega Y_1Z_1 - \omega^4X_1^2Y_1Z_1, Z_1^2\omega^2X_1\omega Y_1 - Z_1^2X_1Y_1, Y_1^2\omega^2X_1Z_1 - \omega^2Y_1^2X_1Z_1) = (0, 0, 0)$.

Conversely, assume that $(X_3, Y_3, Z_3) = (0, 0, 0)$. Then $X_1^2Y_2Z_2 = X_2^2Y_1Z_1$, $Z_1^2X_2Y_2 = Z_2^2X_1Y_1$, and $Y_1^2X_2Z_2 = Y_2^2X_1Z_1$.

If $X_1 = 0$ then $Y_1^3 + Z_1^3 = 0$ by the curve equation, so $Y_1 \neq 0$ and $Z_1 \neq 0$. Write $\lambda_1 = Y_1/Z_1$; then $(X_1 : Y_1 : Z_1) = (0 : \lambda_1 : 1)$ and $\lambda_1^3 = -1$. Furthermore $X_2^2Y_1Z_1 = 0$ so $X_2 = 0$ so $(X_2 : Y_2 : Z_2) = (0 : \lambda_2 : 1)$ where $\lambda_2^3 = -1$. Define $\omega = \lambda_2/\lambda_1$; then $\omega^3 = \lambda_2^3/\lambda_1^3 = 1$ and $(X_2 : Y_2 : Z_2) = (0 : \lambda_2 : 1) = (0 : \omega\lambda_1 : 1) = (\omega^2X_1 : \omega Y_1 : Z_1)$.

If $X_2 = 0$ then similarly $X_1 = 0$. Assume from now on that $X_1 \neq 0$ and $X_2 \neq 0$. Write $y_1 = Y_1/X_1$, $z_1 = Z_1/X_1$, $y_2 = Y_2/X_2$, and $z_2 = Z_2/X_2$. Rewrite the three equations $X_3 = 0$, $Y_3 = 0$, and $Z_3 = 0$ as $y_2z_2 = y_1z_1$, $z_1^2y_2 = z_2^2y_1$, and $y_1^2z_2 = y_2^2z_1$. The first two equations imply $z_1^3y_1 = z_1^2y_2z_2 = z_2^3y_1$, so $(z_1^3 - z_2^3)y_1 = 0$; the first and third equations imply $y_1^3z_1 = y_1^2y_2z_2 = y_2^3z_1$, so $(y_1^3 - y_2^3)z_1 = 0$.

If $y_1 = 0$ then $z_1^2 y_2 = 0$ by the second equation. The curve equation $a + y_1^3 + z_1^3 = dy_1z_1$ forces $a + z_1^3 = 0$ so $z_1 \neq 0$; hence $y_2 = 0$. The curve equation $a + y_2^3 + z_2^3 = dy_2z_2$ similarly forces $a + z_2^3 = 0$ so $z_2^3 = z_1^3$. Write $\omega = z_2/z_1$; then $\omega^3 = 1$ and $(X_2 : Y_2 : Z_2) = (1 : y_2 : z_2) = (1 : 0 : z_2) = (1 : 0 : \omega z_1) = (\omega^2 : \omega y_1 : z_1) = (\omega^2 X_1 : \omega Y_1 : Z_1)$.

If $z_1 = 0$ then similar logic applies. Assume from now on that $y_1 \neq 0$ and $z_1 \neq 0$. Then $z_1^3 = z_2^3$ and $y_1^3 = y_2^3$. Write $\omega = y_1/y_2$; then $\omega^3 = 1$. The equation $X_3 = 0$ forces $\omega = z_2/z_1$. Hence $(X_2 : Y_2 : Z_2) = (1 : y_2 : z_2) = (1 : \omega^{-1} y_1 : \omega z_1) = (\omega^2 X_1 : \omega Y_1 : Z_1)$. □

3.4 The rotated addition law

Theorem 3.7 states a new addition law for twisted Hessian curves. This addition law is obtained as follows:

- Subtract $(1 : -c : 0)$ from one input, using Theorem 3.6, where c is a cube root of a.
- Use the standard addition law in Theorem 3.4.
- Add $(1 : -c : 0)$ to the output, using Theorem 3.6 again.

The formulas in Theorem 3.6 are linear, so the resulting addition law has the same bidegree as the standard addition law. This is an example of what Bernstein and Lange in [BL11, Section 8] call **rotation** of an addition law.

This rotated addition law is new, even in the case $a = 1$. Unlike the standard addition law, the rotated addition law works for doublings. Specializing the rotated addition law to doublings, and further to $a = 1$, produces exactly the Joye–Quisquater doubling formula from [JQ01, Proposition 2]. Even better, the rotated addition law is complete when a is not a cube; see Theorem 3.10 below.

Theorem 3.12 states that the standard addition law and the rotated addition law form a complete system of addition laws for any twisted Hessian curve: any pair of input points can be added by at least one of the two laws. This system is vastly simpler than the Bosma–Lenstra complete system [BHWL95] of addition laws for Weierstrass curves, and arguably even simpler than the Bernstein–Lange complete system [BL11] of addition laws for twisted Edwards curves: each output coordinate here is a difference of just two degree-$(2, 2)$ monomials, as in [BL11], but here there are just three output coordinates while in [BL11] there were four.

One can easily rotate the addition law again (or, equivalently, exchange the two inputs) to obtain a third addition law with the same features as the second addition law. One can also prove that these three addition laws are a basis for the space of degree-$(2, 2)$ addition laws for H: it is easy to see that the laws are linearly independent, and Bosma and Lenstra showed in [BHWL95, Section 4] that the whole space has dimension 3.

Theorem 3.6. *Let H be the twisted Hessian curve $aX^3 + Y^3 + Z^3 = dXYZ$ over a field K. Assume that $c \in K$ satisfies $c^3 = a$. Then $(1 : -c : 0) \in H(K)$. Furthermore, if X_1, Y_1, Z_1*

are elements of K such that $(X_1 : Y_1 : Z_1) \in H(K)$, then $(X_1 : Y_1 : Z_1) + (1 : -c : 0) = (Y_1 : cZ_1 : c^2X_1)$.

Proof. First $a(1)^3 + (-c)^3 + (0)^3 = 0$ so $(1 : -c : 0) \in H(K)$.

Case 1: $Z_1 \neq 0$. Write $(X_2, Y_2, Z_2) = (1, -c, 0)$, and define (X_3, Y_3, Z_3) as in Theorem 3.4. Then $X_3 = -Y_1Z_1$, $Y_3 = -cZ_1^2$, and $Z_3 = -c^2X_1Z_1$, so $(X_3 : Y_3 : Z_3) = (Y_1 : cZ_1 : c^2X_1)$, so $(X_1 : Y_1 : Z_1) + (1 : -c : 0) = (Y_1 : cZ_1 : c^2X_1)$ by Theorem 3.4.

Case 2: $Z_1 = 0$. (Note that Theorem 3.4 is not useful in this case, since it defines $(X_3, Y_3, Z_3) = (0, 0, 0)$.) Then $aX_1^3 + Y_1^3 = 0$ by the curve equation, so $X_1 \neq 0$ and $Y_1 \neq 0$. Write $\omega = Y_1/(-cX_1)$; then $\omega^3 = Y_1^3/(-aX_1^3) = 1$, and $(X_1 : Y_1 : Z_1) = (1 : -\omega c : 0)$.

- Case 2.1: $\omega \neq 1$. The line $Z = 0$ intersects the curve at the three distinct points $(1 : -c : 0)$, $(1 : -\omega c : 0)$, and $(1 : -\omega^{-1}c : 0)$, so $(1 : -c : 0) + (1 : -\omega c : 0) = -(1 : -\omega^{-1}c : 0) = (1 : 0 : -\omega^{-1}c) = (-\omega c : 0 : c^2) = (Y_1 : cZ_1 : c^2X_1)$ by Theorem 3.3.
- Case 2.2: $\omega = 1$, i.e., $(X_1 : Y_1 : Z_1) = (1 : -c : 0)$. The line $3c^2X + 3cY + dZ = 0$ intersects the curve at $(1 : -c : 0)$. We will see in a moment that it has no other intersection points. Consequently $3(1 : -c : 0) = 0$; i.e., $(X_1 : Y_1 : Z_1) + (1 : -c : 0) = 2(1 : -c : 0) = -(1 : -c : 0) = (1 : 0 : -c) = (-c : 0 : c^2) = (Y_1 : cZ_1 : c^2X_1)$ by Theorem 3.3.

We finish by showing that the only intersection is $(1 : -c : 0)$. Assume that $3c^2X + 3cY + dZ = 0$ and $aX^3 + Y^3 + Z^3 = dXYZ$. Then $-dZ = 3c(cX + Y)$, but also $(cX+Y)^3 = aX^3 + Y^3 + 3c^2X^2Y + 3cXY^2 = -Z^3$, so $-d^3Z^3 = 27a(cX+Y)^3 = -27aZ^3$. By hypothesis $27a \neq d^3$, so $Z^3 = 0$, so $Z = 0$, so $cX + Y = 0$, so $(X : Y : Z) = (1 : -c : 0)$. □

Theorem 3.7. *Let H be the twisted Hessian curve $aX^3 + Y^3 + Z^3 = dXYZ$ over a field K. Let $X_1, Y_1, Z_1, X_2, Y_2, Z_2$ be elements of K such that $(X_1 : Y_1 : Z_1), (X_2 : Y_2 : Z_2) \in H(K)$. Define*

$$\begin{aligned} X_3' &= Z_2^2X_1Z_1 - Y_1^2X_2Y_2, \\ Y_3' &= Y_2^2Y_1Z_1 - aX_1^2X_2Z_2, \\ Z_3' &= aX_2^2X_1Y_1 - Z_1^2Y_2Z_2. \end{aligned}$$

If $(X_3', Y_3', Z_3') \neq (0, 0, 0)$ then $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2) = (X_3' : Y_3' : Z_3')$.

Proof. Fix a field extension K' of K containing a cube root c of a. Replace K, X_1, Y_1, Z_1 with K', Z_1, c^2X_1, cY_1 respectively throughout Theorem 3.4. This replaces X_3, Y_3, Z_3 with $-Z_3', -c^2X_3', -cY_3'$ respectively. Hence $(Z_1 : c^2X_1 : cY_1) + (X_2 : Y_2 : Z_2) = (Z_3' : c^2X_3' : cY_3')$ if $(X_3', Y_3', Z_3') \neq (0, 0, 0)$.

Now add $(1 : -c : 0)$ to both sides. Theorem 3.6 implies $(1 : -c : 0) + (Z_1 : c^2X_1 : cY_1) = (c^2X_1 : c^2Y_1 : c^2Z_1) = (X_1 : Y_1 : Z_1)$ and similarly $(1 : -c : 0) + (Z_3' : c^2X_3' : cY_3') = (X_3' : Y_3' : Z_3')$. Hence $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2) = (X_3' : Y_3' : Z_3')$ if $(X_3', Y_3', Z_3') \neq (0, 0, 0)$. □

Theorem 3.8. *In the situation of Theorem 3.7, $(X_3', Y_3', Z_3') = (0, 0, 0)$ if and only if $(X_2 : Y_2 : Z_2) = (Z_1 : \gamma^2X_1 : \gamma Y_1)$ for some $\gamma \in K$ with $\gamma^3 = a$.*

Proof. Fix a field extension K' of K containing a cube root c of a. Replace K, X_1, Y_1, Z_1 with K', Z_1, c^2X_1, cY_1 respectively throughout Theorem 3.4 and Theorem 3.5 to see that $(-Z_3', -c^2X_3', -cY_3') = (0,0,0)$ if and only if $(X_2 : Y_2 : Z_2) = (\omega^2 Z_1 : \omega c^2 X_1 : cY_1)$ for some $\omega \in K'$ with $\omega^3 = 1$.

If $(X_2 : Y_2 : Z_2) = (Z_1 : \gamma^2 X_1 : \gamma Y_1)$ for some $\gamma \in K$ with $\gamma^3 = a$ then this condition is satisfied by the ratio $\omega = \gamma/c \in K'$ so $(X_3', Y_3', Z_3') = (0,0,0)$.

Conversely, if $(X_3', Y_3', Z_3') = (0,0,0)$ then $(X_2 : Y_2 : Z_2) = (\omega^2 Z_1 : \omega c^2 X_1 : cY_1)$ for some $\omega \in K'$ with $\omega^3 = 1$, so $(X_2 : Y_2 : Z_2) = (Z_1 : \gamma^2 X_1 : \gamma Y_1)$ where $\gamma = c\omega$. To see that $\gamma \in K$, note that at least two of X_1, Y_1, Z_1 are nonzero. If X_1, Y_1 are nonzero then Y_2, Z_2 are nonzero and $(\gamma^2 X_1)/(\gamma Y_1) = Y_2/Z_2$ so $\gamma = (Y_2/Z_2)(Y_1/X_1) \in K$. If Y_1, Z_1 are nonzero then X_2, Z_2 are nonzero and $(\gamma Y_1)/Z_1 = Z_2/X_2$ so $\gamma = (Z_2/X_2)(Z_1/Y_1) \in K$. If X_1, Z_1 are nonzero then X_2, Y_2 are nonzero and $(\gamma^2 X_1)/Z_1 = Y_2/X_2$ so $\gamma^2 = (Y_2/X_2)(Z_1/X_1) \in K$; but also $\gamma^3 = c^3 = a \in K$, so $\gamma = a/\gamma^2 \in K$. ☐

Theorem 3.9. *In the situation of Theorem 3.7,* $(X_3', Y_3', Z_3') \neq (0,0,0)$ *if* $(X_2 : Y_2 : Z_2) = (X_1 : Y_1 : Z_1)$.

Proof. Suppose $(X_3', Y_3', Z_3') = (0,0,0)$. Then $(X_2, Y_2, Z_2) = (Z_1, \gamma^2 X_1, \gamma Y_1)$ for some $\gamma \in K$ with $\gamma^3 = a$ by Theorem 3.8, so $(X_2 : Y_2 : Z_2) + (1 : -\gamma : 0) = (\gamma^2 X_1 : \gamma^2 Y_1 : \gamma^2 Z_1) = (X_1 : Y_1 : Z_1)$ by Theorem 3.6. Subtract $(X_2 : Y_2 : Z_2) = (X_1 : Y_1 : Z_1)$ to obtain $(1 : -\gamma : 0) = (0 : -1 : 1)$, contradiction.

Alternative proof, showing more directly that $Y_3' \neq 0$ or $Z_3' \neq 0$: Write (X_2, Y_2, Z_2) as $(\lambda X_1, \lambda Y_1, \lambda Z_1)$ for some $\lambda \neq 0$. Then $Y_3' = \lambda^2 Z_1 (Y_1^3 - aX_1^3)$ and $Z_3' = \lambda^2 Y_1 (aX_1^3 - Z_1^3)$.

Case 1: $Y_1 = 0$. Then $aX_1^3 = -Z_1^3$ by the curve equation, so $Y_3' = -\lambda^2 Z_1^4$. If $Y_3' = 0$ then $Z_1 = 0$ so $aX_1^3 = 0$ so $X_1 = 0$ so $(X_1, Y_1, Z_1) = (0,0,0)$, contradiction. Hence $Y_3' \neq 0$.

Case 2: $Z_1 = 0$. Then $aX_1^3 = -Y_1^3$ by the curve equation, so $Z_3' = -\lambda^2 Y_1^4$. If $Z_3' = 0$ then $Y_1 = 0$ so $aX_1^3 = 0$ so $X_1 = 0$ so $(X_1, Y_1, Z_1) = (0,0,0)$, contradiction. Hence $Z_3' \neq 0$.

Case 3: $Y_1 \neq 0$ and $Z_1 \neq 0$. If $Y_3' = 0$ and $Z_3' = 0$ then $aX_1^3 = Y_1^3$ and $aX_1^3 = Z_1^3$; in particular $X_1 \neq 0$, so $3aX_1^3 = dX_1 Y_1 Z_1$ by the curve equation, so $27a^3 X_1^9 = d^3 X_1^3 Y_1^3 Z_1^3 = d^3 a^2 X_1^9$, so $27a = d^3$, contradiction. Hence $Y_3' \neq 0$ or $Z_3' \neq 0$. ☐

Theorem 3.10. *In the situation of Theorem 3.7, assume that a is not a cube in K. Then* $(X_3', Y_3', Z_3') \neq (0,0,0)$ *and* $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2) = (X_3' : Y_3' : Z_3')$.

Proof. By hypothesis no $\gamma \in K$ satisfies $\gamma^3 = a$. By Theorem 3.8, $(X_3', Y_3', Z_3') \neq (0,0,0)$. By Theorem 3.7, $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2) = (X_3' : Y_3' : Z_3')$.

We also give a second, more direct, proof that $Z_3' \neq 0$. The curve equation forces $Z_1 \neq 0$ and $Z_2 \neq 0$. Write $x_1 = X_1/Z_1$, $y_1 = Y_1/Z_1$, $x_2 = X_2/Z_2$, and $y_2 = Y_2/Z_2$. Suppose that $Z_3' = 0$, i.e., $y_2 = ax_1 y_1 x_2^2$. Eliminate y_2 in the curve equation $ax_2^3 + y_2^3 + 1 = dx_2 y_2$ to obtain $ax_2^3 + (ax_1 y_1 x_2^2)^3 + 1 = dax_1 y_1 x_2^3$. Use the curve equation at (x_1, y_1) to eliminate d and rewrite $(ax_1 y_1 x_2^2)^3 = -ax_2^3 - 1 + ax_2^3(ax_1^3 + y_1^3 + 1) = ax_2^3(ax_1^3 + y_1^3) - 1$ which factors as $(a^2 x_1^3 x_2^3 - 1)(ax_2^3 y_1^3 - 1) = 0$, implying that a is a cube in K. ☐

Theorem 3.11. *Let H be the twisted Hessian curve $aX^3+Y^3+Z^3=dXYZ$ over a field K. Assume that $\omega\in K$ satisfies $\omega^3=1$. Then $(0:-\omega:1)\in H(K)$. Furthermore, if X_1,Y_1,Z_1 are elements of K such that $(X_1:Y_1:Z_1)\in H(K)$, then $(X_1:Y_1:Z_1)+(0:-\omega:1)=(\omega^2X_1:\omega Y_1:Z_1)$.*

Proof. Take $(X_2,Y_2,Z_2)=(0,-\omega,1)$ in Theorem 3.4 to obtain $(X_3,Y_3,Z_3)=(-\omega X_1^2, -X_1Y_1,-\omega^2X_1Z_1)$. If $X_1\neq 0$ then $(X_3,Y_3,Z_3)\neq(0,0,0)$ and $(X_1:Y_1:Z_1)+(0:-\omega:1)=(X_3:Y_3:Z_3)=(\omega^2X_1:\omega Y_1:Z_1)$.

Also take $(X_2,Y_2,Z_2)=(0,-\omega,1)$ in Theorem 3.7 to obtain $(X_3',Y_3',Z_3')=(X_1Z_1, \omega^2Y_1Z_1,\omega Z_1^2)$. If $Z_1\neq 0$ then $(X_3',Y_3',Z_3')\neq(0,0,0)$ and $(X_1:Y_1:Z_1)+(0:-\omega:1)=(X_3':Y_3':Z_3')=(\omega^2X_1:\omega Y_1:Z_1)$.

At least one of X_1,Z_1 must be nonzero, so at least one of these cases applies. □

Theorem 3.12. *Let H be the twisted Hessian curve $aX^3+Y^3+Z^3=dXYZ$ over a field K. Let X_1,Y_1,Z_1,X_2,Y_2,Z_2 be elements of K such that $(X_1:Y_1:Z_1),(X_2:Y_2:Z_2)\in H(K)$. Define (X_3,Y_3,Z_3) as in Theorem 3.4, and (X_3',Y_3',Z_3') as in Theorem 3.7. Then $(X_3,Y_3,Z_3)\neq(0,0,0)$ or $(X_3',Y_3',Z_3')\neq(0,0,0)$.*

Proof. Suppose that $(X_3,Y_3,Z_3)=(0,0,0)$ and $(X_3',Y_3',Z_3')=(0,0,0)$. Then $(X_2:Y_2:Z_2)=(\omega^2X_1:\omega Y_1:Z_1)$ for some $\omega\in K$ with $\omega^3=1$ by Theorem 3.5, so $(X_2:Y_2:Z_2)=(X_1:Y_1:Z_1)+(0:-\omega:1)$ by Theorem 3.11. Furthermore $(X_2:Y_2:Z_2)=(Z_1:\gamma^2X_1:\gamma Y_1)$ for some $\gamma\in K$ with $\gamma^3=a$ by Theorem 3.8, so $(X_2:Y_2:Z_2)=(X_1:Y_1:Z_1)-(1:-\gamma:0)$ by Theorem 3.6. Hence $(0:-\omega:1)=-(1:-\gamma:0)=(1:0:-\gamma)$, contradiction. □

3.5 Points of order 3

Each projective twisted Hessian curve over $\mathbb{F}_p$ has a rational point of order 3. See Theorem 3.13. In particular, for $p\in 1+3\mathbb{Z}$, the point $(0:-\omega:1)$ is a rational point of order 3, where ω is a primitive cube root of 1 in $\mathbb{F}_p$.

Conversely, if $p\in 1+3\mathbb{Z}$, then each elliptic curve over $\mathbb{F}_p$ with a point P_3 of order 3 is isomorphic to a twisted Hessian curve via an isomorphism that takes P_3 to $(0:-\omega:1)$. We prove this converse in two steps:

- Over any field, each elliptic curve with a point P_3 of order 3 is isomorphic to a curve of the form $y^2+dxy+ay=x^3$, where $a(27a-d^3)\neq 0$, via an isomorphism taking P_3 to $(0,0)$. This is a standard fact; see, e.g., [DL05, Section 13.1.5.b]. To keep this chapter self-contained we include a proof as Theorem 3.14. We refer to $y^2+dxy+ay=x^3$ as a **triangular curve** because its Newton polygon is a triangle of minimum area (equivalently, minimum number of boundary lattice points) among all Newton polygons of Weierstrass curves.

- Over a field with a primitive cube root ω of 1, this triangular curve is isomorphic to the twisted Hessian curve $(d^3-27a)X^3+Y^3+Z^3=3dXYZ$ via an isomorphism that takes $(0,0)$ to $(0:-\omega:1)$. See Theorem 3.15.

Furthermore, over any field, this triangular curve is 3-isogenous to the twisted Hessian curve $aX^3+Y^3+Z^3=dXYZ$, provided that $d\neq 0$. See Theorem 3.16. This gives an alternate proof, for $d\neq 0$, that $aX^3+Y^3+Z^3=dXYZ$ has a point of order 3 over $\mathbb{F}_p$: the triangular curve $y^2+dxy+ay=x^3$ has a point of order 3, namely $(0,0)$, so its group order over $\mathbb{F}_p$ is a multiple of 3; the isogenous twisted Hessian curve $aX^3+Y^3+Z^3=dXYZ$ has the same group order, and therefore also a point of order 3. This isogeny also leads to extremely fast tripling formulas; see Section 3.6.

For comparison: Over a field where all elements are cubes, such as a field $\mathbb{F}_p$ with $p\in 2+3\mathbb{Z}$, Smart in [Sma01, Section 3] states an isomorphism from the triangular curve to a Hessian curve, taking $(0,0)$ to the point $(-1:0:1)$ of order 3 (modulo permutation of coordinates to put the neutral element at infinity). We instead emphasize the case $p\in 1+3\mathbb{Z}$ since this is the case that allows completeness.

Theorem 3.13. *Let H be the twisted Hessian curve $aX^3+Y^3+Z^3=dXYZ$ over a finite field K. Then $H(K)$ has a point of order* 3.

Proof. Case 1: $\#K\in 1+3\mathbb{Z}$. There is a primitive cube root ω of 1 in K. The point $(0:-\omega:1)$ is in $H(K)$ by Theorem 3.11, is nonzero since $\omega\neq 1$, satisfies $2(0:-\omega:1)=(0:-\omega^2:1)=(0:1:-\omega)$ by Theorem 3.11, and satisfies $-(0:-\omega:1)=(0:1:-\omega)$ by Theorem 3.3, so it is a point of order 3.

Case 2: $\#K\notin 1+3\mathbb{Z}$. There is a cube root c of a in K. The point $(1:-c:0)$ is in $H(K)$ by Theorem 3.6, is visibly nonzero, satisfies $2(1:-c:0)=(-c:0:c^2)=(1:0:-c)$ by Theorem 3.6, and satisfies $-(1:-c:0)=(1:0:-c)$ by Theorem 3.3, so it is a point of order 3. ☐

Theorem 3.14. *Let E be an elliptic curve over a field K. Assume that $E(K)$ has a point P_3 of order* 3. *Then there exist a,d,ϕ such that $a,d\in K$; $a(27a-d^3)\neq 0$; ϕ is an isomorphism from E to the triangular curve $y^2+dxy+ay=x^3$; and $\phi(P_3)=(0,0)$.*

Proof. Write E in long Weierstrass form $v^2+e_1uv+e_3v=u^3+e_2u^2+e_4u+e_6$. The point P_3 is not the neutral element so it is affine, say (u_3,v_3).

Substitute $u=x+u_3$ and $v=t+v_3$ to obtain an isomorphic curve C in long Weierstrass form $t^2+c_1xt+c_3t=x^3+c_2x^2+c_4x+c_6$. This isomorphism takes P_3 to the point $(0,0)$. This point has order 3, so the tangent line to C at $(0,0)$ intersects the curve at that point with multiplicity 3, so it does not intersect the point at infinity, so it is not vertical; i.e., it has the form $t=\lambda x$ for some $\lambda\in K$.

Substitute $y=t-\lambda x$ to obtain an isomorphic curve A in long Weierstrass form $y^2+a_1xy+a_3y=x^3+a_2x^2+a_4x+a_6$. This isomorphism preserves $(0,0)$, and now the line $y=0$ intersects A at $(0,0)$ with multiplicity 3. Hence $a_2=a_4=a_6=0$; i.e., the curve is $y^2+a_1xy+a_3y=x^3$. Write $d=a_1$ and $a=a_3$.

The discriminant of this curve is $a^3(d^3-27a)$ so $a\neq 0$ and $27a-d^3\neq 0$. More explicitly, if $a=0$ then $(0,0)$ is singular; if $d^3=27a$ and $3=0$ in K then $(-(a^2/4)^{1/3},-a/2)$ is singular; if $d^3=27a$ and $3\neq 0$ in K then $(-d^2/9,a)$ is singular. ☐

Theorem 3.15. *Let a,d be elements of a field K such that $a(27a-d^3)\neq 0$. Let ω be an element of K with $\omega^3=1$ and $\omega\neq 1$. Let E be the triangular curve $VW(V+$*

$dU + aW) = U^3$. Then there is an isomorphism ϕ from E to the twisted Hessian curve $(d^3-27a)X^3+Y^3+Z^3 = 3dXYZ$, defined by $\phi(U : V : W) = (X : Y : Z)$ where $X = U$, $Y = \omega(V + dU + aW) - \omega^2 V - aW$, $Z = \omega^2(V + dU + aW) - \omega V - aW$. Furthermore $\phi(0 : 0 : 1) = (0 : -\omega : 1)$.

Proof. Note that $3 \neq 0$ in K: otherwise $(\omega - 1)^3 = \omega^3 - 1 = 0$ so $\omega - 1 = 0$, contradiction.

Write H for the curve $a'X^3 + Y^3 + Z^3 = d'XYZ$, where $a' = d^3 - 27a$ and $d' = 3d$. Then $a'(27a' - (d')^3) = (d^3 - 27a)(27(d^3 - 27a) - 27d^3) = 27^2 a(27a - d^3) \neq 0$, so H is a twisted Hessian curve over K.

The identity $a'X^3 + Y^3 + Z^3 - d'XYZ = 27a(VW(V + dU + aW) - U^3)$ in the ring $\mathbb{Z}[a, d, U, V, W, \omega]/(\omega^2 + \omega + 1)$ shows that ϕ maps E to H.

The map ϕ is invertible on $\mathbb{P}^2$: specifically, $\phi^{-1}(X : Y : Z) = (U : V : W)$ where $U = X$, $V = -(dX + \omega Y + \omega^2 Z)/3$, and $W = -(dX + Y + Z)/(3a)$. The same identity shows that ϕ^{-1} maps H to E.

Hence ϕ is an isomorphism of curves from H to E. To see that it is an isomorphism of elliptic curves, observe that it maps the neutral element of E to the neutral element of H: specifically, $\phi(0 : 1 : 0) = (0 : \omega - \omega^2 : \omega^2 - \omega) = (0 : -1 : 1)$.

Finally $\phi(0 : 0 : 1) = (0 : \omega a - a : \omega^2 a - a) = (0 : \omega - 1 : \omega^2 - 1) = (0 : -\omega : 1)$. □

Theorem 3.16. *Let H be the twisted Hessian curve $aX^3 + Y^3 + Z^3 = dXYZ$ over a field K. Assume that $d \neq 0$. Let E be the triangular curve $VW(V + dU + aW) = U^3$. Then there is an isogeny ι from H to E defined by $\iota(X : Y : Z) = (-XYZ : Y^3 : X^3)$; there is an isogeny ι' from E to H defined by*

$$\begin{aligned}&\iota'(U : V : W)\\&= \left(\frac{R^3+S^3+V^3-3RSV}{d} : RS^2+SV^2+VR^2-3RSV : RV^2+SR^2+VS^2-3RSV\right)\end{aligned}$$

where $Q = dU$, $R = aW$, and $S = -(V + Q + R)$; and $\iota'(\iota(P)) = 3P$ for each point P on H.

Proof. If $U = -XYZ$, $V = Y^3$, and $W = X^3$ then $VW(V + dU + aW) - U^3 = X^3Y^3(aX^3 + Y^3 + Z^3 - dXYZ)$. Hence ι is a rational map from H to E. The neutral element $(0 : -1 : 1)$ of H maps to the neutral element $(0 : 1 : 0)$ of E, so ι is an isogeny from H to E. Note that ι is defined everywhere on H: each point $(X : Y : Z)$ on H has $X \neq 0$ or $Y \neq 0$, so $(-XYZ, Y^3, X^3) \neq (0, 0, 0)$.

If $Q = dU$, $R = aW$, $S = -(V + Q + R)$, $X = (R^3 + S^3 + V^3 - 3RSV)/d$, $Y = RS^2 + SV^2 + VR^2 - 3RSV$, and $Z = RV^2 + SR^2 + VS^2 - 3RSV$ then the following identities hold:

$$\begin{aligned}&aX^3 + Y^3 + Z^3 - dXYZ\\&\quad = a(Q^2 + 3QR + 3R^2 + 3QV + 3VR + 3V^2)^3(VW(V + dU + aW) - U^3);\\&a(R + S + V)^3 - d^3RSV = ad^3(VW(V + dU + aW) - U^3);\\&dX + 3Y + 3Z = (R + S + V)^3 - 27RSV.\end{aligned}$$

The first identity implies that ι' is a rational map from E to H. The neutral element $(0:1:0)$ of E maps to the neutral element $(0:-1:1)$ of H, so ι' is an isogeny from E to H. The remaining identities imply that ι' is defined everywhere on E. Indeed, if $(X,Y,Z)=(0,0,0)$ then $a(R+S+V)^3-d^3RSV=0$ and $(R+S+V)^3-27RSV=dX+3Y+3Z=0$ so $(d^3-27a)RSV=0$, implying $R=0$ or $S=0$ or $V=0$. If $R=0$ then $0=Y=SV^2$ so $S=0$ or $V=0$; if $S=0$ then $0=Y=VR^2$ so $V=0$ or $R=0$; if $V=0$ then $0=Y=RS^2$ so $R=0$ or $S=0$. In all cases at least two of R,S,V are 0, but also $R+S+V=0$, so all three are 0. This implies $W=0$, $Q=0$, and $U=0$, contradicting $(U:V:W)\in\mathbb{P}^2$.

What remains is to prove that $\iota'\circ\iota$ is tripling on H. Take a point $(X_1:Y_1:Z_1)$ on H. Define $(X_2,Y_2,Z_2)=((Z_1^3-Y_1^3)X_1,(Y_1^3-aX_1^3)Z_1,(aX_1^3-Z_1^3)Y_1)$; then $(X_2:Y_2:Z_2)=2(X_1:Y_1:Z_1)$ by Theorem 3.7 and Theorem 3.8. Define (X_3,Y_3,Z_3) and (X_3',Y_3',Z_3') as in Theorem 3.4 and Theorem 3.7 respectively. Define $(U,V,W)=(-X_1Y_1Z_1,Y_1^3,X_1^3)$; then $(U:V:W)=\iota(X_1:Y_1:Z_1)$. Define $Q=dU$, $R=aW$, $S=-(V+Q+R)$, $X=(R^3+S^3+V^3-3RSV)/d$, $Y=RS^2+SV^2+VR^2-3RSV$, and $Z=RV^2+SR^2+VS^2-3RSV$; then $\iota'(\iota(X_1:Y_1:Z_1))=(X:Y:Z)$. Write C for the polynomial $aX_1^3+Y_1^3+Z_1^3-dX_1Y_1Z_1$.

Case 1: $X_1\neq 0$. The identities

$$\begin{aligned}
X_3&=X_1(-X+C(2aX_1^3+2Y_1^3-Z_1^3-dX_1Y_1Z_1)X_1Y_1Z_1),\\
Y_3&=X_1(-Y+C(a^2X_1^6-adX_1^4Y_1Z_1-aX_1^3Z_1^3+4aX_1^3Y_1^3-Y_1^6)),\\
Z_3&=X_1(-Z+C(-a^2X_1^6-dX_1Y_1^4Z_1-Y_1^3Z_1^3+4aX_1^3Y_1^3+Y_1^6))
\end{aligned}$$

show that $(X_3:Y_3:Z_3)=(X:Y:Z)$. In particular, $(X_3,Y_3,Z_3)\neq(0,0,0)$, so $3(X_1:Y_1:Z_1)=(X_3:Y_3:Z_3)$ by Theorem 3.4, so $3(X_1:Y_1:Z_1)=(X:Y:Z)$.

Case 2: $Y_1\neq 0$. The identities

$$\begin{aligned}
X_3'&=Y_1(X-C(2aX_1^3+2Y_1^3-Z_1^3-dX_1Y_1Z_1)X_1Y_1Z_1),\\
Y_3'&=Y_1(Y-C(a^2X_1^6-adX_1^4Y_1Z_1-aX_1^3Z_1^3+4aX_1^3Y_1^3-Y_1^6)),\\
Z_3'&=Y_1(Z-C(-a^2X_1^6-dX_1Y_1^4Z_1-Y_1^3Z_1^3+4aX_1^3Y_1^3+Y_1^6))
\end{aligned}$$

show that $(X_3':Y_3':Z_3')=(X:Y:Z)$. In particular, $(X_3',Y_3',Z_3')\neq(0,0,0)$, so $3(X_1:Y_1:Z_1)=(X_3':Y_3':Z_3')$ by Theorem 3.7, so $3(X_1:Y_1:Z_1)=(X:Y:Z)$.

At least one of X_1 and Y_1 must be nonzero, so at least one of these cases applies.

□

3.6 Cost of point operations

This section analyzes the cost of various formulas for arithmetic on twisted Hessian curves. Input and output points are assumed to be represented in projective coordinates $(X:Y:Z)$.

All of the formulas in this section are complete when a is not a cube. In particular, the addition formulas use the rotated addition law (Theorem 3.7) rather than the standard addition law (Theorem 3.4). Switching back to the standard addition law is a straightforward rotation exercise and saves $1\mathbf{m}_a$ in addition, at the expense of completeness. If incomplete formulas are acceptable then one can achieve the same savings in the rotated addition law by taking $a = 1$, although this would force somewhat larger constants in doublings and triplings.

3.6.1 Addition

The following formulas compute addition $(X_3 : Y_3 : Z_3) = (X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2)$ in $12\mathbf{M} + 1\mathbf{m}_a$.

$$\begin{aligned} &A = X_1 \cdot Z_2; && B = Z_1 \cdot Z_2; && C = Y_1 \cdot X_2; \\ &D = Y_1 \cdot Y_2; && E = Z_1 \cdot Y_2; && F = aX_1 \cdot X_2; \\ &X_3 = A \cdot B - C \cdot D; && Y_3 = D \cdot E - F \cdot A; && Z_3 = F \cdot C - B \cdot E. \end{aligned}$$

Mixed addition, computing $(X_3 : Y_3 : Z_3) = (X_1 : Y_1 : Z_1) + (X_2 : Y_2 : 1)$, takes only $10\mathbf{M} + 1\mathbf{m}_a$: eliminate the two multiplications by Z_2 in the above formulas.

In followup work, Hisil has saved $1\mathbf{M}$ as follows, achieving $11\mathbf{M}+1\mathbf{m}_a$ for addition (and $9\mathbf{M} + 1\mathbf{m}_a$ for mixed addition), assuming $2 \neq 0$ in the field:

$$\begin{aligned} &A = X_1 \cdot Z_2; && B = Z_1 \cdot Z_2; && C = Y_1 \cdot X_2; \\ &D = Y_1 \cdot Y_2; && E = Z_1 \cdot Y_2; && F = aX_1 \cdot X_2; \\ &G = (D + B) \cdot (A - C); && H = (D - B) \cdot (A + C); && J = (D + F) \cdot (A - E); \\ &K = (D - F) \cdot (A + E); && L = (B - F) \cdot (C + E); && X_3 = G - H; \\ &Y_3 = K - J; && Z_3 = J + K - G - H - 2L. \end{aligned}$$

Theorem 3.10 shows that all of these formulas are complete if a is not a cube. In particular, these formulas can be used to compute doublings. This is one way to reduce side-channel leakage in twisted Hessian coordinates. However, faster doublings are feasible as we show below.

3.6.2 Doubling

Each of the following formulas is a complete doubling formula, i.e., correctly doubles all curve points, whether or not a is a cube. To see this, substitute $(X_2, Y_2, Z_2) = (X_1, Y_1, Z_1)$ in Theorem 3.7, and observe that the resulting vector (X_3', Y_3', Z_3') is, up to sign (and scaling by a power of 2 for the formulas labeled as requiring $2 \neq 0$), the same as the vector (X_3, Y_3, Z_3) computed here. Recall that Theorem 3.7 is always usable for doublings by Theorem 3.9.

The first doubling formulas use $6\mathbf{M} + 3\mathbf{S} + 1\mathbf{m}_a$. Note that the formulas compute the squares of all input values as a step towards cubing them. They are not used individually, so the formulas would benefit from dedicated cubings.

$$\begin{aligned} &A = X_1^2; && B = Y_1^2; && C = Z_1^2; && D = A \cdot X_1; \\ &E = B \cdot Y_1; && F = C \cdot Z_1; && G = aD && X_3 = X_1 \cdot (E - F); \\ &Y_3 = Z_1 \cdot (G - E); && Z_3 = Y_1 \cdot (F - G). \end{aligned}$$

The second doubling formulas require $2 \neq 0$ in the field and require the field to contain an element i with $i^2 = -1$. These formulas use $8\mathbf{M} + 1\mathbf{m}_i + 1\mathbf{m}_d$.

$$\begin{array}{lll} J = iZ_1; & P = Y_1 \cdot Z_1; & A = (Y_1 - J) \cdot (Y_1 + J); \\ C = (A - P) \cdot (Y_1 + Z_1); & D = (A + P) \cdot (Z_1 - Y_1); & E = 3C - 2dX_1 \cdot P; \\ X_3 = -2X_1 \cdot D; & Y_3 = (D - E) \cdot Z_1; & Z_3 = (D + E) \cdot Y_1. \end{array}$$

The third doubling formulas eliminate the multiplication by i, further improve cost to $7\mathbf{M} + 1\mathbf{S} + 1\mathbf{m}_d$, and eliminate the requirement for the field to contain i, although they still require $2 \neq 0$ in the field.

$$\begin{array}{lll} P = Y_1 \cdot Z_1; & Q = 2P; & R = Y_1 + Z_1; \\ A = R^2 - P; & C = (A - Q) \cdot R; & D = A \cdot (Z_1 - Y_1); \\ E = 3C - dX_1 \cdot Q; & X_3 = -2X_1 \cdot D; & Y_3 = (D - E) \cdot Z_1; \\ Z_3 = (D + E) \cdot Y_1. & & \end{array}$$

The fourth doubling formulas, also requiring $2 \neq 0$ in the field, improve cost even more, to $6\mathbf{M} + 2\mathbf{S} + 1\mathbf{m}_d$.

$$\begin{array}{lll} R = Y_1 + Z_1; & S = Y_1 - Z_1; & T = R^2; \\ U = S^2; & V = T + 3U; & W = 3T + U; \\ C = R \cdot V; & D = S \cdot W; & E = 3C - dX_1 \cdot (W - V); \\ X_3 = -2X_1 \cdot D; & Y_3 = (D + E) \cdot Z_1; & Z_3 = (D - E) \cdot Y_1. \end{array}$$

In most situations the fastest approach is to choose small d and use the fourth doubling formulas. Characteristic 3 typically has fast cubings, making the first doubling formulas faster. Characteristic 2 allows only the first doubling formulas.

3.6.3 Tripling

Assume that $d \neq 0$. The 3-isogenies in Theorem 3.16 then lead to efficient tripling formulas that compute $(X_3 : Y_3 : Z_3) = 3(X_1 : Y_1 : Z_1)$ significantly faster than a doubling followed by an addition. This is useful in, e.g., scalar multiplications using double-base chains; see Section 3.7.

Specifically, define

$$\begin{array}{ll} U = -X_1Y_1Z_1; & Q = dU = -dX_1Y_1Z_1; \\ V = Y_1^3; & W = X_1^3; \\ R = aW = aX_1^3; & S = -(V + Q + R) = -(Y_1^3 - dX_1Y_1Z_1 + aX_1^3) = Z_1^3; \\ X_3 = (R^3 + S^3 + V^3 - 3RSV)/d; & Y_3 = RS^2 + SV^2 + VR^2 - 3RSV; \\ Z_3 = RV^2 + SR^2 + VS^2 - 3RSV. & \end{array}$$

Then the isogenies ι and ι' in Theorem 3.16 satisfy $\iota(X_1 : Y_1 : Z_1) = (U : V : W)$ and $3(X_1 : Y_1 : Z_1) = \iota'(U : V : W) = (X_3 : Y_3 : Z_3)$. All tripling formulas that we consider begin by computing $R = aX_1^3$, $V = Y_1^3$, and $S = Z_1^3$ with three cubings (normally $3\mathbf{M} + 3\mathbf{S}$, except for fields supporting faster cubing) and then compute X_3, Y_3, Z_3 from R, S, V. Note that computing S as Z_1^3 is faster than computing U as $-X_1Y_1Z_1$, and there does not seem to be any benefit in computing U or $Q = dU$.

The following straightforward formulas compute X_3, Y_3, Z_3 from R, S, V in $5\mathbf{M} + 3\mathbf{S} + \mathbf{m}_{1/d}$, assuming $2 \neq 0$ in the field, where $\mathbf{m}_{1/d}$ means the cost of multiplying by the curve parameter $1/d$:

$$\begin{aligned} A &= (R-V)^2; & B &= (R-S)^2;\\ C &= (V-S)^2; & D &= A+C;\\ E &= A+B; & X_3 &= (1/d)(R+V+S)\cdot(B+D);\\ Y_3 &= 2RC - V\cdot(C-E); & Z_3 &= 2VB - R\cdot(B-D). \end{aligned}$$

The total cost for tripling this way is $8\mathbf{M} + 6\mathbf{S} + \mathbf{m}_a + \mathbf{m}_{1/d}$. For the case $a = 1$ the same cost had been achieved by Hisil, Carter, and Dawson in [HCD07]. One can of course scale X_3, Y_3, Z_3 by a factor of d, replacing $\mathbf{m}_{1/d}$ with $2\mathbf{m}_d$.

Here is a technique to produce faster formulas, building upon the structure used in the proofs in Section 3.5. Start with the polynomial identity

$$\begin{aligned} &(\alpha R + \beta S + \gamma V)(\alpha S + \beta V + \gamma R)(\alpha V + \beta R + \gamma S)\\ &\quad = \alpha\beta\gamma d X_3 + (\alpha\beta^2{+}\beta\gamma^2{+}\gamma\alpha^2)Y_3 + (\beta\alpha^2{+}\gamma\beta^2{+}\alpha\gamma^2)Z_3 + (\alpha{+}\beta{+}\gamma)^3 RSV. \end{aligned}$$

Specialize this identity to three choices of constants (α, β, γ), and use the curve equation $d^3RSV = a(R+S+V)^3$ appearing in the proof of Theorem 3.16, to obtain four linear equations for dX_3, Y_3, Z_3, RSV. If the constants are sensibly chosen then the equations are independent.

We now give three examples of this technique. First: Taking $(\alpha, \beta, \gamma) = (1, 1, 1)$ gives $(R+S+V)^3 = dX_3 + 3Y_3 + 3Z_3 + 27RSV$, as already used in the proof of Theorem 3.16. Taking $(\alpha, \beta, \gamma) = (1, -1, 0)$ gives $(R-S)(S-V)(V-R) = Y_3 - Z_3$, and taking $(\alpha, \beta, \gamma) = (1, 1, 0)$ gives $(R+S)(S+V)(V+R) = Y_3 + Z_3 + 8RSV$. These equations, together with $a(R+S+V)^3 = d^3RSV$, are linearly independent except in characteristic 2: we have

$$\begin{aligned} dX_3 &= (1 - 3a/d^3)(R{+}S{+}V)^3 - 3(R{+}S)(S{+}V)(V{+}R),\\ 2Y_3 &= (R{+}S)(S{+}V)(V{+}R) + (R{-}S)(S{-}V)(V{-}R) - 8(a/d^3)(R{+}S{+}V)^3,\\ 2Z_3 &= (R{+}S)(S{+}V)(V{+}R) - (R{-}S)(S{-}V)(V{-}R) - 8(a/d^3)(R{+}S{+}V)^3. \end{aligned}$$

Computing $(2X_3, 2Y_3, 2Z_3)$ from these formulas takes one cubing for $(R+S+V)^3$, $2\mathbf{M}$ for $(R+S)(S+V)(V+R)$, $2\mathbf{M}$ for $(R-S)(S-V)(V-R)$, one multiplication by a/d^3 (or, alternatively, a multiplication of $R+S+V$ by $1/d$ and a subsequent multiplication by a), one multiplication by $1/d$, and several additions, for a total cost of $8\mathbf{M} + 4\mathbf{S} + \mathbf{m}_a + \mathbf{m}_{a/d^3} + \mathbf{m}_{1/d}$; i.e., $8\mathbf{M} + 4\mathbf{S}$ when both a and $1/d$ are chosen to be small. As noted in the introduction, this result is due to Kohel [Koh15], as a followup to our preliminary announcements of results in this chapter.

Second example: For characteristic 2 one must take at least one vector (α,β,γ) outside $\mathbb{F}_2^3$, creating more multiplications by constants. The overall cost is still 8**M**+4**S** if all constants are chosen to be small and $(1,1,1)$ is used as an (α,β,γ).

Third example: Assume that the base field K is $\mathbb{F}_p[\omega]/(\omega^2+\omega+1)$ where $p \in 2+3\mathbb{Z}$, or more generally has any primitive cube root ω of 1 for which multiplications by ω are fast. Now take the vectors $(\alpha,\beta,\gamma)=(1,\omega^i,\omega^{2i})$ and observe that the left side of the above identity is always a cube:

$$(R+\omega S+\omega^2 V)^3 = dX_3+3\omega^2 Y_3+3\omega Z_3,$$
$$(R+\omega^2 S+\omega V)^3 = dX_3+3\omega Y_3+3\omega^2 Z_3.$$

These equations and $(1-27a/d^3)(R+S+V)^3 = dX_3+3Y_3+3Z_3$ are linearly independent; the matrix of coefficients of $dX_3, 3Y_3, 3Z_3$ is a Fourier matrix. We apply the inverse Fourier matrix to obtain $dX_3, 3Y_3, 3Z_3$ with a few more multiplications by ω. Overall this tripling algorithm costs just 6 cubings, i.e., 6**M** + 6**S**.

One way to understand the appearance of the Fourier matrix here is to observe that the polynomial $dX_3+3Y_3t+3Z_3t^2+9(1+t+t^2)RSV$ is the cube of $V+St+Rt^2$ modulo t^3-1. We compute the cube of $V+St+Rt^2$ separately modulo $t-1$, $t-\omega$, and $t-\omega^2$.

3.7 Cost of scalar multiplication

This section analyzes the cost of scalar multiplication using twisted Hessian curves. In particular, this section explains how we obtained a cost of just 8.73**M** per bit for average 256-bit scalars.

Since our new twisted-Hessian formulas provide very fast tripling and reasonably fast doubling, the results of [BBLP07] suggest that it will be fastest to represent scalars using $\{2,3\}$-double-base chains. Scalar multiplication then involves not only doubling and addition but also tripling. A well-known advantage of double-base representations is that the number of additions is smaller than in the binary representation.

We use a newer algorithm to generate double-base chains, shown in Algorithm 3.1. This algorithm is an improved version of the basic "tree-based" algorithm proposed and analyzed by Doche and Habsieger in [DH08] (see Algorithm 2.8).

Recall that in the basic algorithm, n is computed recursively from either $(n-1)/(2^{\cdots}3^{\cdots})$ or $(n+1)/(2^{\cdots}3^{\cdots})$, where the exponents of 2 and 3 are chosen to be as large as possible. The algorithm explores the branching tree of possibilities in breadth-first fashion until it reaches $n=1$. To limit time and memory usage, the algorithm keeps only the smallest B nodes at each level. We chose $B=200$.

We use an extension to this algorithm mentioned but not analyzed in [DH08]. The extension uses not just $n-1$ and $n+1$, but all $n-c$ where c is in a precomputed set (including both positive and negative values). We include the cost of precomputing this set. We chose 21 different possibilities for the precomputed set, namely the 21 sets listed in [BBLP07].

We change the way to add new nodes as follows:

- n has *one* child node $n/2$ if n is divisible by 2;
- otherwise, n has *one* child node $n/3$ if n is divisible by 3;
- otherwise, n has *several* child nodes $n-c$, one for each $c \in S$.

We improve the algorithm by continuing to search the tree until we have found C chains, rather than stopping with the first chain; we then take the lowest-cost chain. We chose $C = 200$.

We further improve the algorithm by taking the lowest-weight B nodes at each level instead of the smallest B nodes at each level; here "weight" takes account not just of smallness but also of the cost of operations used to reach the node. More precisely, we define "weight" as cost $+ 8 \cdot \log_2(n)$.

Algorithm 3.1 Modified tree-search

Input: An integer n, precomputation set S, and bounds B and C
Output: A double-base chain computing n

```
for each precomputation set $S$ do
    counter ← 0
    Initialize a tree $T$ with root node $n$
    while (counter < $C$) do
        for each leaf node $m$ in $T$ do
            if $m$ divisible by 2 then
                Insert child ← $f_2(m)$                          ▷ $f_2(m) = m/2^{v_2(m)}$
                if $f_2(m)$ equals 1 then
                    counter ← counter +1
            else if $m$ divisible by 3 then
                Insert child ← $f_3(m)$                          ▷ $f_3(m) = m/3^{v_3(m)}$
                if $f_3(m)$ equals 1 then
                    counter ← counter +1
            else
                for each element $c$ in precomputation set $S$ do
                    if $m - c > 0$ then
                        Insert child ← $f(m - c)$
                        if $m - c$ equals 1 then
                            counter ← counter + 1
        Discard all but the $B$ smallest weight leaf nodes
return The smallest cost chain
```

We ran this algorithm for 10000 random 256-bit scalars, i.e., integers between 2^{255} and $2^{256}-1$, using as input the costs of twisted Hessian operations. The average cost of the resulting chain was 8.73**M** per bit, with standard deviation 0.06**M** per bit.

To more precisely assess the advantage of cofactor 3 over cofactor 1, we carried out a larger series of experiments for smaller scalars, comparing the cost of twisted Hessian curves to the cost of short Weierstrass curves $y^2 = x^3 - 3x + b$ in Jacobian coordinates. Specifically, for each ℓ from 2 through 16, we constructed double-base chains for all ℓ-bit integers; for each ℓ from 17 through 64, we constructed double-base chains for 1000 randomly chosen ℓ-bit integers. The top of Figure 3.1 plots pairs (x, y) where x is the cost to multiply by n on a twisted Hessian curve and $x + y$ is the cost to multiply by the same integer n on a Weierstrass curve; i.e., switching from Weierstrass to twisted Hessian saves $y\mathbf{M}$. We reduced the number of dots plotted in this figure to avoid excessive PDF file sizes and display times, but a full plot is similar. Dots along the x-axis represent integers with the same cost for both curve shapes. Different colors are used for different bitlengths ℓ.

We have generated similar plots for some other pairs of curve shapes. For example, the bottom of Figure 3.1 shows that Edwards is faster than Hessian for most values of n. In some cases, such as Hessian vs. tripling-oriented Doche–Icart–Kohel curves, the plots are concentrated much more narrowly around a line, since these curve shapes favor similar integers that use many triplings; the line has a positive slope, i.e., Hessian is faster.

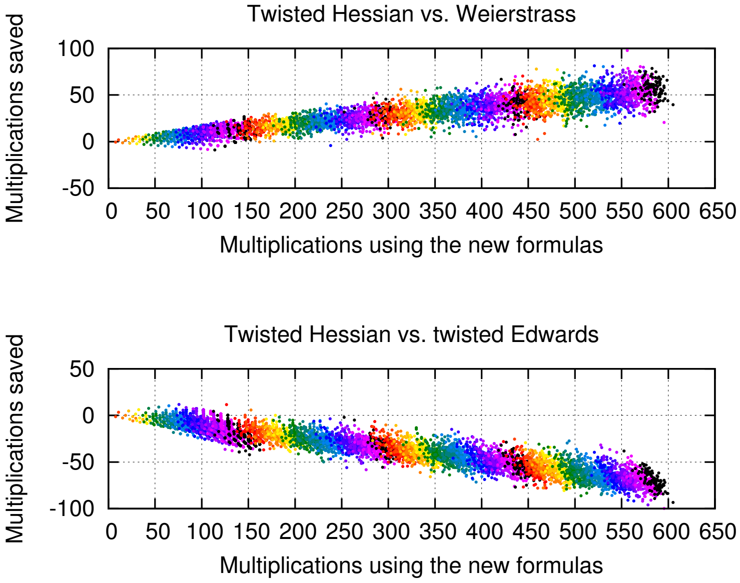


Figure 3.1: Top: Points (x, y) for 100 randomly sampled ℓ-bit integers n for each $\ell \in \{2, 3, \ldots, 64\}$. Here $x\mathbf{M}$ are used to compute $P \mapsto nP$ on a twisted Hessian curve in projective coordinates; $(x + y)\mathbf{M}$ are used to compute $P \mapsto nP$ on a Weierstrass curve $y^2 = x^3 - 3x + b$ in Jacobian coordinates; and the color is a function of ℓ. Bottom: Similar, but using a twisted Edwards curve rather than a Weierstrass curve.

4

Double-Base Scalar Multiplication

Elliptic-curve computations have many applications ranging from the elliptic-curve method for factorization (ECM) to elliptic-curve cryptography. The most important elliptic-curve computation is scalar multiplication: computing the nth multiple nP of a curve point P. For example, in ECDH, n is Alice's secret key, P is Bob's public key, and nP is a secret shared between Alice and Bob.

There is an extensive literature proposing and analyzing scalar-multiplication algorithms that decompose the map $P \mapsto nP$ into field operations in various ways. Some speedups rely on algebraically "special" curves, but this chapter restricts attention to the types of curves that arise in ECM and in conservative ECC: large characteristic, no subfields, no extra endomorphisms, etc. Even with this restriction, there is a remarkable range of ideas for speeding up scalar multiplication.

It is standard to compare these algorithms by counting field multiplications **M**, counting each field squaring **S** as 0.8**M**, and disregarding the overhead of field additions, subtractions, and multiplications by small constants. (Of course, the most promising algorithms are then analyzed in more detail in implementation papers for various platforms, accounting for all overheads.) Since 2008, the record number of field multiplications has been held by an algorithm of Hisil, Wong, Carter, and Dawson [HWCD08]; for an average 256-bit scalar n, this algorithm computes $P \mapsto nP$ using 7.62**M** per bit.

This chapter does better by presenting an algorithm that, for an average 256-bit scalar n, computes $P \mapsto nP$ using just 7.47**M** per bit. Similarly, for double-scalar multiplication, we use just 8.80**M** per bit, where previous techniques used 9.34**M** per bit.

Notice that the new 7.47**M** and 8.80**M** per bit, like the old 7.62**M** and 9.34**M** per bit, are averages over n; the time depends slightly on n. These algorithms leak some information about n through total timing, and much more information through higher-bandwidth side channels, so they should be avoided in contexts where n is secret. However, there are many environments in which scalars are not secret: examples include ECC signature verification and ECM. This chapter presents new algorithms for single- and double-scalar multiplication, covering these important applications.

Notice also that these operation counts ignore the cost of converting n into a multiply-by-n algorithm. The conversion cost is significant for the new algorithm. This does not matter if n is used many times, but it can be a bottleneck in scenarios where n is not used many times. To address this issue, this chapter introduces a fast new conversion method, explained in more detail below; this reduces the number of times that n has to be reused to justify applying the conversion. Note also that, in the context of signature verification, this conversion can be carried out by the *signer* and included in the signature as an encoding of n. This helps ECC signatures in applications where so far RSA was preferred due to the faster signature verification speeds (with small public RSA exponent).

Credits. The content of this chapter is based on the paper *"Double-base scalar multiplication revisited"* [BCL17] on ePrint which is a joint work with Daniel J. Bernstein and Tanja Lange. Note that "we" in this chapter is used to refer to the aforementioned authors.

Organization of this chapter. Section 4.1 recalls double-base and double-base chain representations. Section 4.2 presents new tripling formulas. Section 4.3 recasts the search for an optimal double-base chain as the search for a shortest path in an explicit DAG. Section 4.4 constructs the second explicit DAG, with a rectangular structure that simplifies computations; using this DAG to find the optimal double-base chain for n takes $(\log n)^{3+o(1)}$ bit operations. Section 4.5 shows how to use smaller integers to represent nodes in the same graph, reducing $(\log n)^{3+o(1)}$ to $(\log n)^{2.5+o(1)}$. Section 4.6 extends these ideas to generate optimal double-base chains for double-scalar multiplication. Section 4.7 compares new results to previous results.

4.1 Double-base representations

One way to speed up scalar multiplication is to reduce the number of point operations, such as point additions and point doublings. For a simple double-and-add algorithm, using the binary representation of n, the number of point doublings is approximately the bitlength of n and the number of point additions is the Hamming weight of n, i.e., the number of bits set in n. Signed digits, windows, and sliding windows write n in a more general binary representation $\sum_{i=1}^{\ell} d_i 2^{a_i}$ with d in a coefficient set, and reduce the number of point additions to a small fraction of the bitlength of n.

4.1.1 Double-base chains

Even though the binary representation is widely and commonly used, it is not the only way to express integers. A double-base representation, specifically a $\{2,3\}$-

representation, writes n as $\sum_{i=1}^{\ell} d_i 2^{a_i} 3^{b_i}$, where $a_1 \geq a_2 \geq \cdots \geq a_\ell \geq 0$; $b_1 \geq b_2 \geq \cdots \geq b_\ell \geq 0$; and d_i is restricted to a specified set S, such as $\{1\}$ or $\{-1, 1\}$. This representation is easy to view as a double-base chain to compute nP: a chain of additions, doublings, and triplings to compute nP, where each addition is of some $d_i P$. Double-base chains were introduced by Dimitrov, Imbert, and Mishra in [DIM05] for the case $S = \{-1, 1\}$, and were generalized to any S by Doche and Imbert in [DI06].

Representing an integer as a double-base chain allows various tradeoffs between doublings and triplings. This representation of n is not unique; selecting the fastest chain for n can save time. As a concrete example, it is worth to comment that the computation $2^{12}(2^3+1)-1$ used in [BK12] for ECM computations can be improved to $2^{12}3^2-1$, provided that two triplings are faster than three doublings and an addition.

For *some* elliptic-curve coordinate systems, the ratio between tripling cost and doubling cost is fairly close to $\log_2 3 \approx 1.58$. It is then easy to see that the total cost of doublings and triplings combined does not vary much across double-base chains, and the literature shows that within this large set one can find double-base chains with considerably fewer additions than single-base chains, saving time in scalar multiplication.

For example, twisted Hessian curves [BCKL15] (see also Chapter 3) cost 7.6**M** for doubling, 11.2**M** for tripling (or 10.8**M** for special fields with fast primitive cube roots of 1), and 11**M** for addition. A pure doubling chain costs $7.6(\log_2 n)\mathbf{M}$ for doublings, and a pure tripling chain costs $11.2(\log_3 n)\mathbf{M} \approx 7.1(\log_2 n)\mathbf{M}$ for triplings. Many different double-base chains have costs between $7.1(\log_2 n)\mathbf{M}$ and $7.6(\log_2 n)\mathbf{M}$ for doublings and triplings, while they vary much more in the costs of additions. The speeds reported in [BCKL15] use double-base chains, and are the best speeds known for scalar multiplication *for curves with cofactor 3*.

However, the situation is quite different for twisted Edwards curves. Compared to twisted Hessian curves, twisted Edwards curves cost more for tripling, less for doubling, and less for addition. The higher tripling-to-doubling cost ratio (close to 2) means that trading doublings for triplings generally loses speed, and the higher tripling-to-addition cost ratio (around 1.5) means that the disadvantage of extra triplings easily outweighs the advantage of reducing the number of additions.

The literature since 2007 has consistently indicated that the best speeds for scalar multiplication are obtained from (1) taking a curve expressible as a twisted Edwards curve and (2) using single-base chains. These choices are fully compatible with the needs of applications such as ECM and ECC; see, for example, [BK12] and the previous Edwards-ECM papers cited there. Bernstein, Birkner, Lange and Peters [BBLP07] obtained double-base speedups for some coordinate systems but obtained their best speeds from single-base Edwards. Subsequent improvements in double-base techniques have not been able to compete with single-base Edwards; for example, the double-base twisted Hessian speeds reported in [BCKL15] (see also Chapter 3), above 8**M** per bit, are not competitive with the earlier single-base Edwards speed from [HWCD08], just 7.62**M** per bit.

The new speed, 7.47**M** per bit, marks the first time that double-base chains have broken through the single-base barrier. This speed relies on double-base chains, more specifically optimal double-base chains (see below), and also relies on new Edwards tripling formulas that will be introduced in this chapter. These tripling formulas use just $9\mathbf{M} + 3\mathbf{S}$, i.e., $11.4\mathbf{M}$, saving $1\mathbf{S}$ compared to the best previous results. This is almost as fast as the tripling speeds from tripling-oriented Doche–Icart–Kohel [DIK06] and twisted Hessian [BCKL15], and has the advantage of being combined with very fast doublings and additions.

4.1.2 Converting n to a chain

Because there are so many choices of double-base chains for n (with any particular S), finding the optimal chain for n is not trivial (even for that S). Doche and Habsieger [DH08], improving on chain length compared to previous papers, proposed a tree-based algorithm to compute double-base chains (see Algorithm 2.8). It does not seem to have been noticed that the algorithm of [DH08], without any pruning, finds an optimal double-base chain in time polynomial in n: the tree has only polynomially many levels, and there are only polynomially many possible nodes at each level. A recent paper by Capuñay and Thériault [CT15] presents an algorithm with an explicit $(\log n)^{4+o(1)}$ time bound to find an optimal double-base chain for n, assuming $S \subseteq \{-1, 1\}$.

This chapter explains a new observation that finding an optimal double-base chain is equivalent to a shortest-path computation in an explicit directed acyclic graph (DAG) with $O(\omega(\log n)^2)$ nodes, assuming $S \subseteq \{-\omega, \ldots, -1, 0, 1, \ldots, \omega\}$; in particular, $(\log n)^{2+o(1)}$ nodes when $\omega \in (\log n)^{o(1)}$. Two construction methods for such graphs are presented, and it is shown how one of the graphs allows optimized arithmetic, reducing the total chain-construction time to just $(\log n)^{2.5+o(1)}$. For comparison, scalar multiplication takes time $(\log n)^{3+o(1)}$ using schoolbook techniques for field arithmetic, time $(\log n)^{2.58\ldots+o(1)}$ using Karatsuba, or time $(\log n)^{2+o(1)}$ using FFTs.

4.2 Faster point tripling

This section presents faster point tripling formulas for twisted Edwards curves $ax^2 + y^2 = 1 + dx^2y^2$ in both projective and extended coordinates. We also show how to minimize the cost when mixing additions, doublings and triplings in different coordinate systems.

4.2.1 Projective coordinates

Recall that projective coordinates $(X : Y : Z)$, for nonzero Z, represent $(x, y) = (X/Z, Y/Z)$. These formulas are faster by one squaring than the previously best tripling formulas in projective coordinates [BBLP07], which in turn are faster than performing point doubling followed by point addition.

Here are the formulas for computing $(X_3 : Y_3 : Z_3) = 3(X_1 : Y_1 : Z_1)$, i.e., point tripling in projective coordinates. These formulas cost $9\mathbf{M} + 3\mathbf{S} + 1\mathbf{m}_a + 2\mathbf{m}_2 + 7\mathbf{a}$, where as before $\mathbf{M}$ denotes field multiplication, $\mathbf{S}$ denotes field squaring, $\mathbf{m}_a$ denotes field multiplication by curve parameter a, $\mathbf{m}_2$ denotes field multiplication by constant 2, and **a** denotes a general field addition.

$$\begin{array}{llll}
U = aX_1^2; & V = Y_1^2; & A = V + U; & B = 2(2Z_1^2 - A); \\
C = U \cdot B; & D = V \cdot B; & G = A \cdot (V - U); & E = G - D; \\
F = G + C; & X_3 = X_1 \cdot (D + G) \cdot E; & Y_3 = Y_1 \cdot (C - G) \cdot F; & Z_3 = Z_1 \cdot E \cdot F.
\end{array}$$

For affine inputs where $Z_1 = 1$, computing $(X_3 : Y_3 : Z_3) = 3(X_1 : Y_1 : 1)$ costs only $8\mathbf{M} + 2\mathbf{S} + 1\mathbf{m}_a + 1\mathbf{m}_2 + 7\mathbf{a}$. That is, we save $1\mathbf{M} + 1\mathbf{S} + 1\mathbf{m}_2$ by ignoring multiplication and squaring of Z_1.

4.2.2 Extended coordinates

We also present similar tripling formulas in extended coordinates. Recall that extended coordinates $(X : Y : Z : T)$ represent $(x, y) = (X/Z, Y/Z)$, like projective coordinates, but also have an extra coordinate $T = XY/Z$. The importance of extended coordinates is that addition of points in extended coordinates is faster than addition of points in projective coordinates.

The following formulas compute $(X_3 : Y_3 : Z_3 : T_3) = 3(X_1 : Y_1 : Z_1)$, i.e., point tripling in extended coordinates. These formulas cost $11\mathbf{M} + 3\mathbf{S} + 1\mathbf{m}_a + 2\mathbf{m}_2 + 7\mathbf{a}$; in other words, it costs $2\mathbf{M}$ extra to compute the coordinate T.

$$\begin{array}{llll}
U = aX_1^2; & V = Y_1^2; & A = V + U; & B = 2(2Z_1^2 - A); \\
C = U \cdot B; & D = V \cdot B; & G = A \cdot (V - U); & E = G - D; \\
F = G + C; & H = X_1 \cdot (D + G); & J = Y_1 \cdot (C - G); & K = Z_1 \cdot E; \\
L = Z_1 \cdot F; & X_3 = H \cdot K; & Y_3 = J \cdot L; & Z_3 = K \cdot L; \\
T_3 = H \cdot J. & & &
\end{array}$$

For affine inputs where $Z_1 = 1$, computing $(X_3 : Y_3 : Z_3 : T_3) = 3(X_1 : Y_1 : 1 : T_1)$ costs only $9\mathbf{M} + 2\mathbf{S} + 1\mathbf{m}_a + 1\mathbf{m}_2 + 7\mathbf{a}$. That is, we save $2\mathbf{M} + 1\mathbf{S} + 1\mathbf{m}_2$ by ignoring multiplication and squaring of Z_1.

Note that the input for these formulas is projective $(X_1 : Y_1 : Z_1)$; to triple an extended $(X_1 : Y_1 : Z_1 : T_1)$ we simply discard the extra T_1 input. We could instead compute $T_3 = T_1 \cdot (D + G) \cdot (C - G)$ and skip computing one of K and L but this would not save any multiplications.

4.2.3 Sequencing point operations

Point doubling in extended coordinates [BLb] also takes projective input, and costs only $1\mathbf{M}$ extra to compute the extra T output. The best known single-base chains compute a series of doublings in projective coordinates, with the final doubling producing extended coordinates; and then an addition, again producing projective coordinates for the next series of doublings.

In the double-base context, because triplings cost $2\mathbf{M}$ extra to produce extended coordinates while doublings cost only $1\mathbf{M}$ extra, we suggest replacing DBL-TPL-ADD with the equivalent TPL-DBL-ADD. More generally, the conversion from projective to extended coordinates should be performed after point doubling and not tripling (if possible). A good sequence of point operations and coordinate systems is as follows: For every nonzero term, first compute point tripling(s) in projective coordinates; then compute point doubling(s) in projective coordinates, finishing with one doubling leading to extended coordinates; finally, compute the addition taking both input points in extended coordinates and outputting the result in projective coordinates.

We still triple into extended coordinates if a term does not include any doublings (e.g., computing $(3^6+1)P$): i.e., compute point tripling(s) in projective coordinates, finishing with one tripling leading to extended coordinates; finally, as before, compute the addition taking both input points in extended coordinates and outputting the result in projective coordinates.

4.2.4 Cost of point operations

Table 4.1 summarizes the costs for point operations for twisted Edwards curves. Note that the tripling formulas presented in Subsection 4.2.1 and 4.2.2 are for twisted Edwards curves for any curve parameter a; the table assumes $a=-1$ to include the point-addition speedup from [HWCD08]. The rest of the chapter builds fast scalar multiplication on top of the point operations summarized in this table.

Table 4.1: Cost of point operations for twisted Edwards curves with $a=-1$.

Operations	Coordinate systems	Cost
Mixed Addition	$\mathscr{E}+\mathscr{A}\to\mathscr{P}$	6M ≈ 6.0M
Addition	$\mathscr{E}+\mathscr{E}\to\mathscr{P}$	7M ≈ 7.0M
Doubling	$\mathscr{P}\to\mathscr{P}$	3M+4S ≈ 6.2M
Doubling	$\mathscr{P}\to\mathscr{E}$	4M+4S ≈ 7.2M
Tripling	$\mathscr{P}\to\mathscr{P}$	9M+3S ≈ 11.4M
Tripling	$\mathscr{P}\to\mathscr{E}$	11M+3S ≈ 13.4M
Doubling + Mixed Addition	$\mathscr{P}\to\mathscr{E}$; $\mathscr{E}+\mathscr{A}\to\mathscr{P}$	10M+4S ≈ 13.2M
Doubling + Addition	$\mathscr{P}\to\mathscr{E}$; $\mathscr{E}+\mathscr{E}\to\mathscr{P}$	11M+4S ≈ 14.2M
Tripling + Mixed Addition	$\mathscr{P}\to\mathscr{E}$; $\mathscr{E}+\mathscr{A}\to\mathscr{P}$	17M+3S ≈ 19.4M
Tripling + Addition	$\mathscr{P}\to\mathscr{E}$; $\mathscr{E}+\mathscr{E}\to\mathscr{P}$	18M+3S ≈ 20.4M

Note: We use symbols $\mathscr{A}$ for (extended) affine coordinates $(X:Y:1:T)$; $\mathscr{P}$ for projective coordinates $(X:Y:Z)$; and $\mathscr{E}$ for extended coordinates $(X:Y:Z:T)$.

4.3 Graph-based approach

This section shows how to view double-base chains for n as paths from n to 0 in an explicit DAG. If weights are assigned properly to the edges of the DAG then the weight of a path is the same as the cost of the double-base chain. Finding the lowest-cost double-base chain for n is therefore the same as finding the lowest-cost ("shortest") path from n to 0. Dijkstra's algorithm [Dij59] finds this path in time $(\log n)^{O(1)}$.

4.3.1 Double-base chains

We formally define a double-base chain as a finite sequence of operations, where each operation is either "×2+c" for some integer c or "×3+c" for some integer c. The integers c are allowed to be negative, and when c is negative we abbreviate "+c" as "—$|c|$"; e.g., the operation "×3+—7" is abbreviated "×3—7"; c is also allowed to be 0.

A double-base chain represents a computation of various multiples nP of a group element P. This computation starts from $0P = 0$, converts each "×2+c" into a doubling $Q \mapsto 2Q$ followed by an addition $Q \mapsto Q + cP$, and converts each "×3+c" into a tripling $Q \mapsto 3Q$ followed by an addition $Q \mapsto Q + cP$, after an initial computation of all relevant multiples cP. For example, the chain

$$(\text{“×2+1”}, \text{“×3+0”}, \text{“×3+0”}, \text{“×2+1”}, \text{“×2+0”})$$

computes successively $0P, 1P, 3P, 9P, 19P, 38P$.

Formally, given a double-base chain $(o_1, o_2, \ldots, o_\ell)$, define a sequence of integers $(n_0, n_1, n_2, \ldots, n_\ell)$ as follows: $n_0 = 0$; if $o_i =$ "×2+c" then $n_i = 2n_{i-1} + c$; if $o_i =$ "×3+c" then $n_i = 3n_{i-1} + c$. This is the sequence of **intermediate results** for the chain, and the chain is a **chain for n_ℓ**. Evidently one can compute $n_\ell P$ from $0P$ using one doubling and one addition of cP for each "×2+c" in the chain, and one tripling and one addition of cP for each "×3+c" in the chain. Note that the sequence of intermediate results does not determine the chain, and does not even determine the cost of the chain.

4.3.2 Restrictions on additions

Several variations in the definition of a double-base chain appear in the literature. Often the differences are not made explicit. We now introduce terminology to describe these variations.

Some definitions allow double-base chains to carry out two additions in a row, with no intervening doublings or triplings. Our double-base chains are **reduced**, in the sense that "+c", "+d" is merged into "+$(c + d)$".

Obviously some limit needs to be placed on the set of c for the concept of double-base chains to be meaningful: for example, the chain ("×2+1", "×2+314157") computes $314159P$ with two additions and an intermediate doubling but begs the question of how the summand $314157P$ was computed. Some papers require "+c" to have $c \in \{-1, 0, 1\}$, while other papers allow $c \in S$ for a larger set S of integers.

We consider the general case, and define an **S-chain** as a chain for which each "+c" has $c \in S$. We require the set S here to contain 0. We focus on sets S for which it is easy to see the cost of computing cP for all $c \in S$, such as the set $S = \{-\omega, \ldots, -2, -1, 0, 1, 2, \ldots, \omega\}$. Subtracting cP is as easy as adding cP for elliptic curves (in typical curve shapes), so we focus on sets S that are closed under negation, but our framework also allows nonnegative sets S; this means that the distinction between addition chains and addition-subtraction chains is subsumed by the distinction between different sets S.

A double-base chain $(o_1, o_2, \ldots, o_\ell)$ is **increasing** if the sequence of intermediate results $(n_0, n_1, n_2, \ldots, n_\ell)$ has $n_0 < n_1 < n_2 < \cdots < n_\ell$. For example, the chain ("×2+5","×2+ − 1") is increasing since $0 < 5 < 9$; but ("×2+1","×2+ − 1" is not increasing. Any $n_i > -\min S$ (with $i < \ell$) automatically has $n_{i+1} > n_i$, so allowing non-increasing chains for positive integers n cannot affect anything beyond initial computations of integers bounded by $-\min S$.

A double-base chain is **greedy** if each intermediate result that is a multiple of 2 or 3 (or both) is obtained by either "×2+0" or "×3+0". This is a more serious limitation on the set of available chains.

4.3.3 The DAG

Fix a finite set S of integers with $0 \in S$. Define an infinite directed graph D as follows. There is a node n for each nonnegative integer n. For each $c \in S$ and each nonnegative integer n, there is an edge $2n + c \to n$ with label "×2+c" if $2n + c > n$, and there is an edge $3n + c \to n$ with label "×3+c" if $3n + c > n$.

Each edge points from a larger nonnegative integer to a smaller nonnegative integer, so this graph D is acyclic, and the set of nodes reachable from any particular n is finite. Theorem 4.1 states that this set forms a directed acyclic graph containing at most $O((\log n)^2)$ nodes for any particular S.

Theorem 4.1. *Assume that* $S \subseteq \{-\omega, \ldots, -1, 0, 1, \ldots, \omega\}$. *Let n be a positive integer. Then there are at most* $(2\omega+1)(\lfloor \log_2 n + 1 \rfloor \lfloor \log_3 n + 1 \rfloor + 1)$ *nodes in D reachable from n.*

Proof. First step: Show that each node v reachable in exactly s steps from n has the form $n/(2^a 3^b) + d$ for some integers a, b and some rational number d with $a \ge 0$, $b \ge 0$, $|d| \le \omega$, and $a + b = s$.

Induct on s. If $s = 0$ then v must be n, so $v = n/(2^a 3^b) + d$ with $(a, b, d) = (0, 0, 0)$. If $s \ge 1$ then there is an edge $u \to v$ for some node u reachable in exactly $s - 1$ steps from n. By the inductive hypothesis, u has the form $n/(2^a 3^b) + d$ with $a \ge 0$, $b \ge 0$, $|d| \le \omega$, and $a + b = s - 1$.

If the edge has label "×2+c" then $u = 2v + c$ so $v = (u - c)/2 = n/(2^{a+1} 3^b) + (d - c)/2$; and $c \in S$ so $|c| \le \omega$ so $|(d - c)/2| \le \omega$. Hence v has the correct form. Similarly, if the edge has label "×3+c" then $u = 3v + c$ so $v = (u - c)/3 = n/(2^a 3^{b+1}) + (d - c)/3$, and $|(d - c)/3| \le (2/3)\omega \le \omega$. This completes the proof of the first step.

Second step: We count the nodes v with $a \le \log_2 n$ and $b \le \log_3 n$. There are at most $\lfloor \log_2 n + 1 \rfloor$ possibilities for a, and at most $\lfloor \log_3 n + 1 \rfloor$ possibilities for b. Any particular (a, b) limits v to the interval $[n/(2^a 3^b) - \omega, n/(2^a 3^b) + \omega]$, which contains at most $2\omega + 1$ integers.

Third step: We count the remaining nodes v. Here $a > \log_2 n$ so $2^a > n$, or $b > \log_3 n$ so $3^b > n$, or both; in any case $n/(2^a 3^b) < 1$ so $|v| < 1 + |d| < 1 + \omega$; i.e.,

$v \in \{-\omega, \ldots, \omega\}$. This limits v to at most $2\omega + 1$ possibilities across all of the possible pairs (a, b). □

Theorem 4.2 states a straightforward correspondence between the set of paths in D from n to 0 and the set of increasing double-base S-chains for n. The correspondence simply reads the edge labels in reverse order.

Theorem 4.2. *Let n be a nonnegative integer. If $(e_\ell, \ldots, e_1)$ is a path from n to 0 in D with labels $(o_\ell, \ldots, o_1)$ then $(o_1, \ldots, o_\ell)$ is an increasing double-base S-chain for n. Conversely, every increasing double-base S-chain for n has this form.*

Proof. Each o_i is an edge label in D, which by definition of D has the form "$\times 2{+}c$" or "$\times 3{+}c$", so $C = (o_1, \ldots, o_\ell)$ is a double-base chain; what remains is to show that it is an increasing S-chain for n.

Specifically, say o_i = "$\times t_i{+}c_i$". Define $(n_0, n_1, \ldots, n_\ell)$ as the corresponding sequence of intermediate results; then $n_0 = 0$, and $n_i = t_i n_{i-1} + c_i$ for $i \in \{1, \ldots, \ell\}$.

We now show by induction on i that e_i is an edge from n_i to n_{i-1}. If $i = 1$ then by hypothesis of the theorem $e_i = e_1$ is an edge to $0 = n_0 = n_{i-1}$. If $i > 1$ then e_{i-1} is an edge from n_{i-1} by the inductive hypothesis so e_i is an edge to n_{i-1}. For any i, e_i is an edge to n_{i-1}. The label o_i = "$\times t_i{+}c_i$" then implies that e_i is an edge from $t_i n_{i-1} + c_i$, i.e., from n_i, as claimed.

In particular, e_ℓ is an edge from n_ℓ, but also e_ℓ is an edge from n by hypothesis of the theorem, so $n = n_\ell$. Hence C is a chain for n. Furthermore, C is an S-chain since each $c_i \in S$ by definition of D, and C is increasing since each edge decreases by definition of D.

Conversely, take any increasing double-base S-chain C for n. Write C as $(o_1, \ldots, o_\ell)$, write o_i as "$\times t_i{+}c_i$", and define $(n_0, n_1, \ldots, n_\ell)$ as the corresponding sequence of intermediate results. Then D has an edge e_i with label o_i from n_i to n_{i-1}, so $(e_\ell, \ldots, e_1)$ is a path from $n_\ell = n$ to 0 in D. □

4.3.4 Chain cost and path cost

Theorem 4.2 suggests the following simple strategy to find an optimal increasing double-base S-chain for n: use Dijkstra's algorithm to find a shortest path from n to 0 in D. This takes time polynomial in $\log n$ if ω is polynomially bounded: the number of nodes visited is polynomial by Theorem 4.1, and it is easy to see that constructing all of the outgoing edges from a node takes polynomial time.

Dijkstra's algorithm requires each edge to be assigned a positive weight, and requires the cost of a path to be defined as the sum of edge weights. This strategy therefore requires the cost of a chain to be defined "locally": the cost of doubling nP and adding cP must not depend on any context other than (n, c), and similarly for tripling. This limitation is not a problem for, e.g., accounting for free additions of $0P$; accounting for a free initial doubling of $0P$; accounting for lower-cost addition of P, i.e., $c = 1$, if P is kept in affine coordinates while other multiples of P are kept in projective or extended coordinates; accounting for the cost of tripling into extended coordinates (see Section 4.2); etc.

One can also handle non-increasing chains by allowing negative integers and dropping the conditions $2n + c > n$ and $3n + c > n$. This allows cycles in D, not a problem for typical breadth-first shortest-path algorithms building a shortest-path tree; see, e.g., [CLRS09]. For simplicity we focus on acyclic graphs in this chapter.

Example 4.1: *Apply DAG to find double-base chains for* $n = 17$.

Figure 4.1 shows the subset of D reachable from $n = 17$ when $S = \{-1, 0, 1\}$. We omit the 0 node from the figure. We replace the remaining edge labels with costs according to Table 4.1, namely, 11.4 for tripling, 6.2 for doubling, 7 for mixed addition.

The choice $S = \{-1, 0, 1\}$ means that each even node t has two outgoing edges: one for $t/2$ and one for $(t + c)/3$ for a unique c, because exactly one of $t, t+1, t-1$ is divisible by 3. Each odd node t has three outgoing edges: one for $(t - 1)/2$, one for $(t + 1)/2$, and one for $(t + c)/3$ for a unique c. For example, 8 is reached from 17 as $(17 - 1)/2$ costing one addition and one doubling; 6 is reached as $(17 + 1)/3$ costing one addition and one tripling. There are two edges between 5 and 2, namely, one corresponding to $2 = (5 - 1)/2$ and one (obviously worse) corresponding to $2 = (5 + 1)/3$.

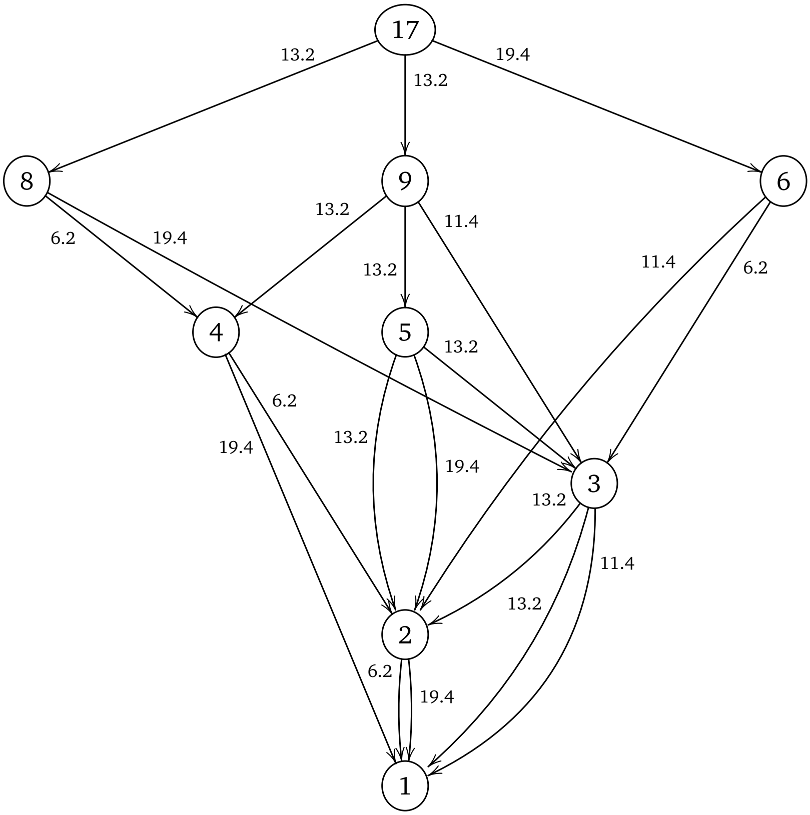


Figure 4.1: DAG with path cost to finding double-base chains for $n = 17$.

4.3.5 The DAG approach vs. the tree approach

The Doche–Habsieger tree-based algorithm summarized in Chapter 2.3.4 (see also Algorithm 2.8) considers only some special greedy chains: when it sees an even node it insists on dividing by 2, prohibiting nontrivial additions, and similarly when it sees a multiple of 3 it insists on dividing by 3; when it sees a number that is neither a multiple of 2 or of 3 it uses an addition. This reduces the number of nodes by roughly a factor 3, but we have found many examples where the best greedy chain is more expensive than the best chain.

Each possible intermediate result appears exactly once in the DAG in this section, but can appear at many different tree levels in the tree-based algorithm. The tree has $\Theta(\log n)$ levels, and a simple heuristic suggests that an intermediate result will, on average, appear on $\Theta((\log n)^{0.5})$ of these levels. The tree-based algorithm repeatedly considers the same edges out of the same coefficient, while the DAG avoids this redundant work. Pruning the tree reduces the cost of the tree-based algorithm but compromises the quality of the resulting chains.

4.4 Rectangular DAG-based approach

The DAG that we introduced in Section 4.3 reduces the target integer n to various smaller integers, each obtained by subtracting an element of S and dividing by 2 or 3. It repeats this process to obtain smaller and smaller integers t, stopping at 0.

In this section we build a slightly more complicated DAG with a three-dimensional structure. Each node in this new DAG has the form (a, b, t), where a is the number of doublings used in paths from n to t, and b is the number of triplings used in paths from n to t.

One advantage of this DAG is that it is now very easy to locate nodes in a simple three-dimensional array, rather than incurring the costs of the associative arrays typically used inside Dijkstra's algorithm. The point is that for t between $n/(2^a 3^b) - \omega$ and $n/(2^a 3^b) + \omega$, as in the proof of Theorem 4.1, we store information about node (a, b, t) at index $(a, b, t - \lfloor n/(2^a 3^b) - \omega \rfloor)$ in the array.

Another advantage of this DAG is that we no longer incur the costs of maintaining a list of nodes to visit inside Dijkstra's algorithm. Define the "position" of the node (a, b, t) as (a, b): then each doubling edge is from position (a, b) to position $(a+1, b)$, and each tripling edge is from position (a, b) to position $(a, b + 1)$. There are many easy ways to topologically sort the nodes, for example by sweeping through positions in increasing order of $a + b$.

The disadvantage of this DAG is that a single t can now appear multiple times at different positions (a, b). However, this disadvantage is limited: it can occur only when there are near-collisions among the values $n/(2^a 3^b)$, something that is quite rare except for the extreme case of small values. We show that the DAG has $(\log n)^{2+o(1)}$ nodes (assuming $\omega \in (\log n)^{o(1)}$), like the DAG in Section 4.3, and thus a total of $(\log n)^{3+o(1)}$ bits in all of the nodes.

We obtain an algorithm to find shortest paths in this DAG, and thus optimal double-base chains, using time just $(\log n)^{3+o(1)}$. Section 4.5 explains how to do even better, reducing the time to $(\log n)^{2.5+o(1)}$ by using reduced representatives for almost all of the integers t.

4.4.1 The three-dimensional DAG

Fix a positive integer ω. Fix a subset $S \subseteq \{-\omega, \dots, -1, 0, 1, \dots, \omega\}$ with $0 \in S$. Fix a positive integer n. Define a finite directed acyclic graph R_n as follows.

There is a node (a, b, v) for each integer $a \in \{0, 1, \dots, \lfloor \log_2 n \rfloor + 1\}$, each integer $b \in \{0, 1, \dots, \lfloor \log_3 n \rfloor + 1\}$, and each integer v within ω of $n/(2^a 3^b)$. Note that not all nodes will be reachable from n in general.

If (a, b, v) and $(a+1, b, u)$ are nodes, $v > u$, and $v = 2u + c$ with $c \in S$, then there is an edge $(a, b, v) \to (a + 1, b, u)$ with label "$\times 2+c$".

Similarly, if (a, b, v) and $(a, b + 1, u)$ are nodes, $v > u$, and $v = 3u + c$ with $c \in S$, then there is an edge $(a, b, v) \to (a, b + 1, u)$ with label "$\times 3+c$".

Theorem 4.3, analogously to Theorem 4.1, says that R_n does not have many nodes. Theorem 4.4, analogously to Theorem 4.2, says that paths in R_n from $(0,0,n)$ to $(\ldots,\ldots,0)$ correspond to double-base chains for n.

Theorem 4.3. *There are at most* $(2\omega+1)(\lfloor\log_2 n+2\rfloor\lfloor\log_3 n+2\rfloor)$ *nodes in* R_n.

Proof. There are $\lfloor\log_2 n+2\rfloor$ choices of a and $\lfloor\log_3 n+2\rfloor$ choices of b. For each (a,b), there are at most $2\omega+1$ integers v within ω of $n/(2^a3^b)$. ☐

Theorem 4.4. *Let n be a positive integer. If $(e_\ell,\ldots,e_1)$ is a path from $(0,0,n)$ to $(a,b,0)$ in R_n with labels $(o_\ell,\ldots,o_1)$ then $(o_1,\ldots,o_\ell)$ is an increasing double-base S-chain for n with at most $\lfloor\log_2 n\rfloor+1$ doublings and at most $\lfloor\log_3 n\rfloor+1$ triplings. Conversely, every increasing double-base S-chain for n with at most $\lfloor\log_2 n\rfloor+1$ doublings and at most $\lfloor\log_3 n\rfloor+1$ triplings has this form.*

Proof. Given a path from $(0,0,n)$ to $(a,b,0)$ in R_n, remove the first two components of each node to obtain a path from n to 0 in D. This path has the same labels $(o_\ell,\ldots,o_1)$, so $(o_1,\ldots,o_\ell)$ is an increasing double-base S-chain for n by Theorem 4.2. It has at most $\lfloor\log_2 n\rfloor+1$ doublings since each doubling increases the first component within $\{0,1,\ldots,\lfloor\log_2 n\rfloor+1\}$, and similarly has at most $\lfloor\log_3 n\rfloor+1$ triplings.

Conversely, given an increasing double-base S-chain for n, construct the corresponding path in D by Theorem 4.2. Insert two extra components into each node to count the number of doublings and triplings. If the chain has at most $\lfloor\log_2 n\rfloor+1$ doublings and at most $\lfloor\log_3 n\rfloor+1$ triplings then these components are contained in $\{0,1,\ldots,\lfloor\log_2 n\rfloor+1\}$ and $\{0,1,\ldots,\lfloor\log_3 n\rfloor+1\}$ respectively, producing a path in R_n from $(0,0,n)$ to $(a,b,0)$. ☐

Example 4.2: *Compute double-base chains for $n=17$ using the 3D-DAG approach.*

In this example, we also use the set of coefficients $S=\{-1,0,1\}$ and operation costs according to Table 4.1, namely, tripling = 11.4, doubling = 6.2, doubling followed by addition and/or subtraction by one = 13.2, and tripling followed by addition and/or subtraction by one = 19.4.

Figure 4.2 illustrates R_{17}, Only nodes reachable from $(0,0,n)$ are included, nodes $(\ldots,\ldots,0)$ are omitted. The rectangular plane shows positions (a,b).

The algorithm starts by initializing $(0,0,17)$.

- At $(0,0,17)$: since 17 is odd, there are 3 outgoing edges to:
    - $(1,0,8)$ having cost $13.2=0+13.2$.
    - $(1,0,9)$ having cost $13.2=0+13.2$.
    - $(0,1,6)$ having cost $19.4=0+19.4$.
- At $(1,0,8)$: since 8 is even, there are 2 outgoing edges to:
    - $(2,0,4)$ having cost $19.4=13.2+6.2$.

  - $(1,1,3)$ having cost $32.6 = 13.2 + 19.4$.
- At $(1,0,9)$: since 9 is odd, there are 3 outgoing edges to:
  - $(2,0,4)$ having cost $26.4 = 13.2+13.2$. This node already exists, and the new cost is more expensive than the previous one. Therefore, the previous cheaper cost 19.4 remains.
  - $(2,0,5)$ having cost $26.4 = 13.2 + 13.2$.
  - $(1,1,3)$ having cost $24.6 = 13.2+11.4$. This node also already exists, but the new cost is cheaper than the previous one. Thus, we update the cost at this node.
- At $(0,1,6)$: since 6 is even, there are 2 outgoing edges to:
  - $(1,1,3)$ having cost $25.6 = 19.4 + 6.2$. However, this is not cheaper than the previous cost. Therefore, no update at this node.
  - $(0,2,2)$ having cost $30.8 = 19.4 + 11.4$.
- At $(2,0,5)$: since 5 is odd, there are 3 outgoing edges to:
  - $(3,0,2)$ having cost $39.6 = 26.4 + 13.2$.
  - $(3,0,3)$ having cost $39.6 = 26.4 + 13.2$.
  - $(2,1,2)$ having cost $45.8 = 26.4 + 19.4$.
- At $(2,0,4)$: since 4 is even, there are 2 outgoing edges to:
  - $(3,0,2)$ having cost $25.6 = 19.4 + 6.2$. This is cheaper than before, so update the cost.
  - $(2,1,1)$ having cost $38.8 = 19.4 + 19.4$.
- At $(1,1,3)$: since 3 is odd, there are 3 outgoing edges to:
  - $(2,1,1)$ having new cost $37.8 = 24.6 + 13.2$.
  - $(2,1,2)$ having new cost $37.8 = 24.6 + 13.2$.
  - $(1,2,1)$ having cost $36 = 24.6 + 11.4$.
- At $(0,2,2)$: since 2 is even, there are 2 outgoing edges to:
  - $(1,2,1)$ having no cost update.
  - $(0,3,1)$ having cost $50.2 = 30.8 + 19.4$.
- At $(3,0,3)$: since 3 is odd, there are 3 outgoing edges to:
  - $(4,0,1)$ having cost $52.8 = 39.6 + 13.2$.
  - $(4,0,2)$ having cost $52.8 = 39.6 + 13.2$.

  - $(3,1,1)$ having cost $51 = 39.6 + 11.4$.

- At $(3,0,2)$: since 2 is even, there are 2 outgoing edges to:
  - $(4,0,1)$ having new cost $31.8 = 25.6 + 6.2$.
  - $(3,1,1)$ having new cost $45 = 25.6 + 19.4$.
- At $(2,1,2)$: since 2 is even, there are 2 outgoing edges to:
  - $(3,1,1)$ having new cost $44 = 37.8 + 6.2$.
  - $(2,2,1)$ having cost $57.2 = 37.8 + 19.4$.
- At $(2,1,1)$, $(1,2,1)$ and $(0,3,1)$, since $1 \in S$, continue to next node.
- At $(4,0,2)$, there are 2 outgoing edges to:
  - $(5,0,1)$ having cost $59 = 52.8 + 6.2$.
  - $(4,1,1)$ having cost $72.2 = 52.8 + 19.4$.
- At $(4,0,1)$, $(3,1,1)$, $(2,2,1)$, $(5,0,1)$ and $(4,1,1)$, since $1 \in S$, continue to next node.

Once all nodes are visited, by selecting the path with minimum cost, we obtain the optimal $\{2,3\}$-chain, namely $2^4 + 1$, having cost 31.8.

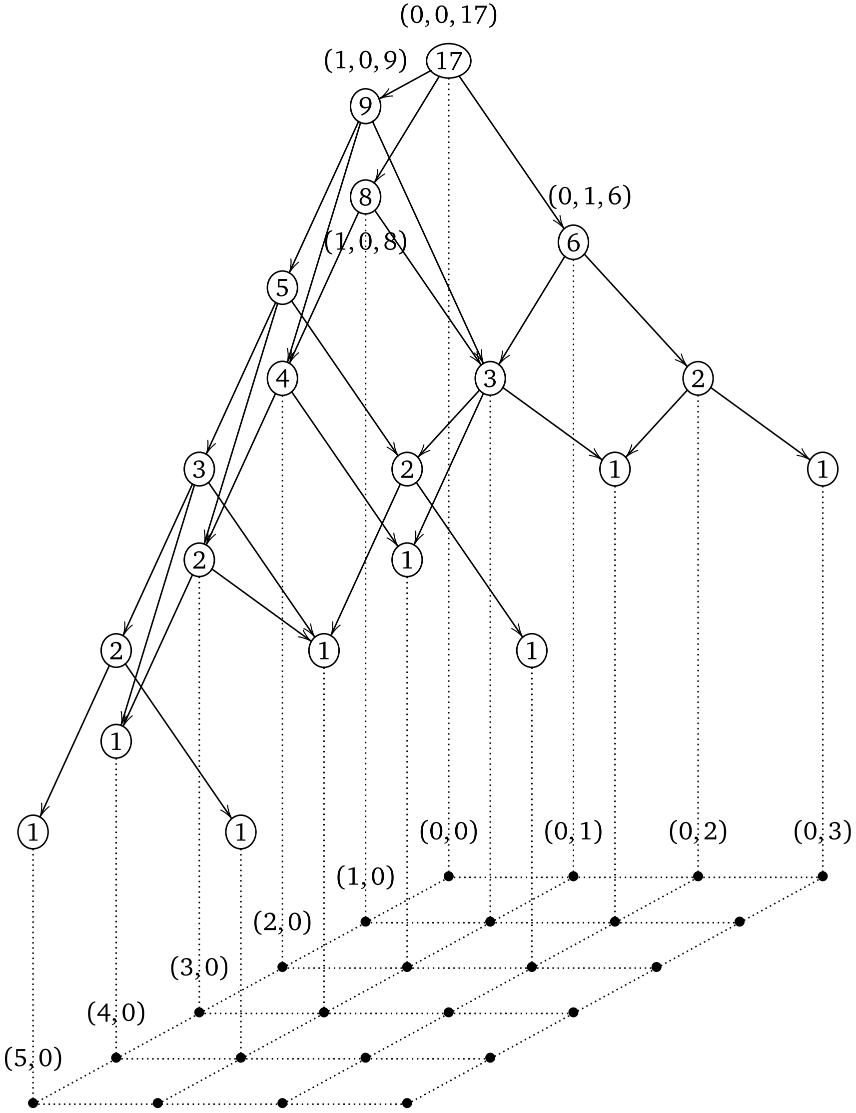


Figure 4.2: 3D-DAG finding double-base chains for $n = 17$.

4.4.2 Cost analysis

We now analyze the performance of shortest-path computation using this DAG, taking account of the cost of handling multiprecision integers such as n. A Python script suitable for carrying out experiments appears in Section 4.8.1.

Recall that information about node (a, b, t) is stored in a three-dimensional array at index $(a, b, t - \lfloor n/(2^a 3^b) - \omega \rfloor)$. We keep a table of these base values $\lfloor n/(2^a 3^b) - \omega \rfloor$. Building this table takes one division by 2 or 3 at each position (a, b). Each division input has $O(\log n)$ bits, and dividing by 2 or 3 takes time linear in the number of bits; the total time is $O((\log n)^3)$.

The information stored for (a, b, t) is the minimum cost of a path from $(0, 0, n)$ to (a, b, t). We optionally store the nearest edge label for a minimum-cost path, to simplify output of the path, but this can also be efficiently reconstructed afterwards from the table of costs.

We sweep through positions (a, b) in topological order, starting from $(0, 0, n)$. At each node (a, b, t) with $t > 0$, for each $s \in S$ with $2 \mid (t - s)$, we update the cost stored for $(a + 1, b, (t - s)/2)$; and, for each $s \in S$ with $3 \mid (t - s)$, we update the cost stored for $(a, b + 1, (t - s)/3)$. An easy optimization is to stop with any $t \in S$, rather than just $t = 0$.

There are $O(\omega(\log n)^2)$ nodes, each with $O(\omega)$ outgoing edges, for a total of $O(\omega^2(\log n)^2)$ update steps. Each update step takes time $O(\log n)$, for total time $O(\omega^2(\log n)^3)$.

4.5 Reduced rectangular DAG-based approach

Recall that the shortest-path computation in Section 4.4 takes time $(\log n)^{3+o(1)}$, assuming $\omega \in (\log n)^{o(1)}$. This section explains how to reduce the exponent from $3 + o(1)$ to $2.5 + o(1)$.

4.5.1 Multiprecision arithmetic as a bottleneck

There are $O(\omega(\log n)^2)$ nodes in the graph R_n; there are $O(\omega)$ edges out of each node; there are $O(1)$ operations at each edge. The reason for an extra factor of $\log n$ in the time complexity is that the operations are multiprecision arithmetic operations on integers that often have as many bits as n. We now look more closely at where such large integers are actually used.

Writing down a position (a, b) takes only $O(\log \log n)$ bits. Writing down a cost also takes only $O(\log \log n)$ bits. We are assuming here, as in most analyses of Dijkstra's algorithm, that edge weights are integers in $O(1)$; for example, each edge weight in Figure 4.1 is (aside from a scaling by 0.1) one of the three integers $62, 132, 184$, so any s-step path has weight at most $184s$.

Writing down an element of S, or an array index $t - \lfloor n/(2^a 3^b) - \omega \rfloor$, takes only $O(\log(2\omega + 1))$ bits. However, both t and the precomputed $\lfloor n/(2^a 3^b) - \omega \rfloor$ are usually much larger, on average $\Theta(\log n)$ bits. Several arithmetic operations are forced to work with these large integers: for example, writing down the first t at a position (a, b), computing $t - s$ for the first $s \in S$, and checking whether $t - s$ is divisible by 3.

4.5.2 Reduced representatives for large numbers

This section saves time by replacing almost all of the integers t with reduced representatives. Each of these representatives occupies only $(\log n)^{0.5+o(1)}$ bits, for a total of $(\log n)^{2.5+o(1)}$ bits. There are occasional "boundary nodes" that each use $(\log n)^{1+o(1)}$ bits, but there are only $(\log n)^{1.5+o(1)}$ of these nodes, again a total of $(\log n)^{2.5+o(1)}$ bits. Arithmetic becomes correspondingly less expensive.

Specifically, (a, b, t) is a **boundary node** if a is a multiple of α or b is a multiple of β or both. Here α and β are positive integers, parameters for the algorithm. For example, the solid lines in Figure 4.3 connect the boundary nodes for $\alpha = \beta = 2$. We will choose α and β as $(\log n)^{0.5+o(1)}$, giving the above-mentioned $(\log n)^{1.5+o(1)}$ boundary nodes.

These boundaries partition the DAG into **subgraphs**. Each subgraph has $O(\omega\alpha\beta)$ nodes at $(\alpha+1)(\beta+1)$ positions covering an area $\alpha \times \beta$ within the space of positions. Specifically, subgraph (q, r) covers all positions (a, b) where $a \in \{q\alpha, q\alpha+1, \ldots, q\alpha+\alpha\}$ and $b \in \{r\beta, r\beta+1, \ldots, r\beta+\beta\}$. Some subgraphs are truncated because they stretch beyond the maximum positions (a, b) allowed in R_n; one can, if desired, avoid this truncation by redefining R_n so that the maximum a is a multiple of α and the maximum b is a multiple of β.

To save time in handling a node (a, b, t) at the top position $(a, b) = (q\alpha, r\beta)$ in subgraph (q, r), we work with the remainder $t \bmod 2^\alpha 3^\beta$. More generally, to save time in handling a node (a, b, t) at position $(a, b) = (q\alpha + i, r\beta + j)$ in the subgraph, we work with the remainder $t \bmod 2^{\alpha-i}3^{\beta-j}$. Each of these remainders is below $2^\alpha 3^\beta$, and therefore occupies only $(\log n)^{0.5+o(1)}$ bits when α and β are $(\log n)^{0.5+o(1)}$.

The critical point here is that the remainder $t \bmod 2^\alpha 3^\beta$ is enough information to see whether $t-s$ is divisible by 2 or 3. This remainder also reveals the next remainder $((t-s)/2) \bmod 2^{\alpha-1}3^\beta$ if $t-s$ is divisible by 2, and $((t-s)/3) \bmod 2^\alpha 3^{\beta-1}$ if $t-s$ is divisible by 3. There is no need to take a detour through writing down the much larger integer $t-s$; we instead carry out a computation on a much smaller number $(t-s) \bmod 2^\alpha 3^\beta$. Continuing in the same way allows as many as α divisions by 2 and as many as β divisions by 3, reaching the bottom boundary of the subgraph. We reconstruct the original nodes at this boundary and then continue to the next subgraph.

Note that remainders do *not* easily allow testing whether t is 0. These tests are used in Section 4.4, but only as a constant-factor optimization to save time in enumerating some cost-0 edges. What is important is testing divisibility of $t-s$ by 2 or 3. Similarly, there is no reason to limit attention to increasing chains.

We have found it simplest to allow negative remainders inside each subgraph, skipping all intermediate mod operations. For example, if $t \bmod 2^\alpha 3^\beta$ happens to be 0 and $s = 2$, then we write down $(0-2)/2 = -1$ rather than $2^{\alpha-1}3^\beta - 1$. The integer operations inside each subgraph then consist of divisions by 2, divisions by 3, and subtractions of elements of S. To reconstruct a node at the bottom boundary of the subgraph, we first merge the sequence of operations into a single subtract-and-divide operation, and then apply this subtract-and-divide operation to the top node t. For example, we merge "subtract 1, divide by 2, subtract 1, divide by 2" into "subtract

3, divide by 4" and then directly compute $(t-3)/4$, without computing the large intermediate result $(t-1)/2$. Note that arbitrary divisions of $O(\log n)$-bit numbers take time $(\log n)^{1+o(1)}$, as shown by Cook in [Coo66, pages 81–86], using Simpson's division method from [Sim40, page 81] on top of Toom's multiplication method from [Too63].

In the case $\alpha = \beta = 2$ depicted in Figure 4.3, we begin by computing the small integer $n \bmod 2^2 3^2$ at the top of the first subgraph. This is enough information to reconstruct the pattern of edges involving as many as $\alpha = 2$ divisions by 2 and as many as $\beta = 2$ divisions by 3. The boundary nodes involving 2 divisions by 2 are marked $\circ$ and $\oplus$. These boundary nodes have values close to $n/2^2$, $n/(2^2 3)$, and $n/(2^2 3^2)$; we reconstruct these values, reduce them again, and continue with the next subgraph down and to the left. Similarly, the boundary nodes involving 2 divisions by 3 are marked $+$ and $\oplus$, with values close to $n/3^2$, $n/(2 \cdot 3^2)$, and $n/(2^2 3^2)$; we reconstruct these values, reduce them again, and continue with the next subgraph down and to the right. The fourth subgraph similarly begins by reconstructing its boundary nodes, marked $\oplus$ and $\diamond$.

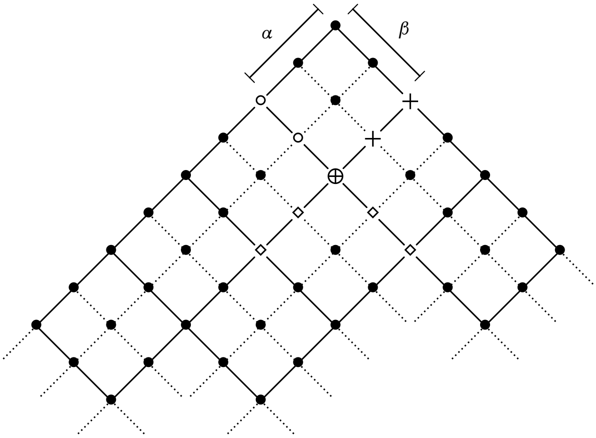


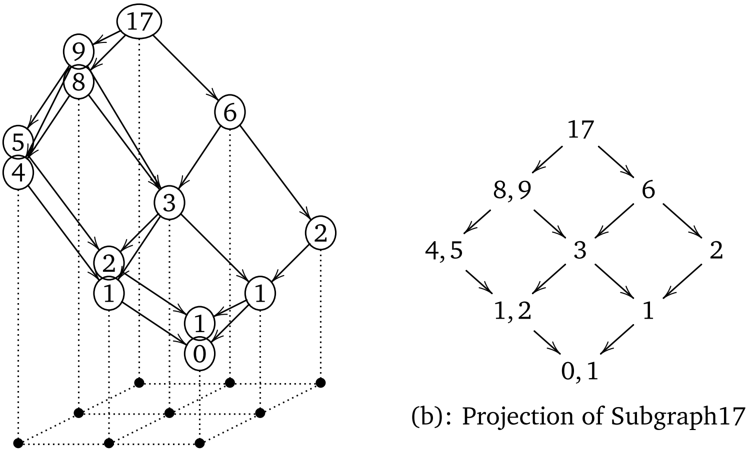
Figure 4.3: Graph division where $\alpha = \beta = 2$.

A Python script suitable for experiments appears in Section 4.8.2.

Example 4.3: *Compute double-base chains for* $n = 917$ *using the rectangular DAG-based approach.*

In this example, we take $\alpha = \beta = 2$. We also use the set of coefficients $S = \{-1, 0, 1\}$ and follow the same cost model as in previous examples.

Since $917 \equiv 17 \bmod 2^2 3^2$, we compute a subgraph of size 2×2 starting with 17. We refer to this subgraph as Subgraph17. The result of this computation is depicted using a 3D graph in Figure 4.3(a) and using a projection in Figure 4.3(b).



(a): 3D graph of Subgraph17

(b): Projection of Subgraph17

Figure 4.4: A subgraph of $17 \equiv 917 \bmod 2^2 3^2$.

To reconstruct nodes from the original graph at the bottom boundary of the subgraph, we start at the root, or more generally at any known node in the original graph, and then follow a path to the boundary. Consider, for example, the node $(2,0,4)$ in Figure 4.4. This 4 was (optimally) computed as $(17-1)/2/2$, i.e., as $(17-1)/4$, so we compute $(917-1)/4 = 229$, obtaining the corresponding node $(2,0,229)$ in the original graph. Similarly, the 5 in $(2,0,5)$ was (optimally) computed as $((17+1)/2+1)/2 = (17+3)/4$, so we compute $(917+3)/4 = 230$, obtaining the corresponding node $(2,0,230)$ in the original graph. There are several ways to speed this up further, such as computing 230 as $229+5-4$, but computing each node separately is adequate for our $(\log n)^{2.5+o(1)}$ result.

Once we have recomputed all nodes in the original graph at the bottom boundary of the subgraph, we move to the next subgraph, for example replacing 229 with $229 \bmod 2^2 3^2 = 13$ and replacing 230 with $230 \bmod 2^2 3^2 = 14$. Figure 4.5 shows the values provided as input (left) and obtained as output (right) in the next subgraph to the bottom left. Similarly, Figure 4.6 shows the values provided as input (left) and obtained as output (right) in the next subgraph to the bottom right.

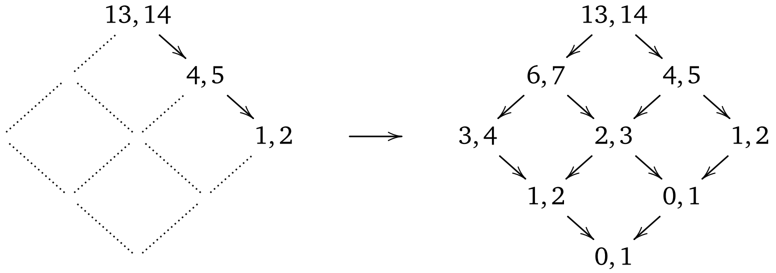


Figure 4.5: Right boundary and subgraph computation.

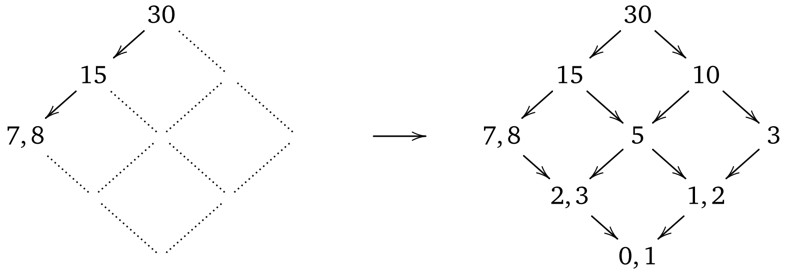


Figure 4.6: Left boundary and subgraph computation.

These procedures of computing intermediate and boundary nodes repeat, eventually covering all subgraphs of the original graph, although one can save a constant factor by skipping computations of some subgraphs. For example, Figure 4.7 shows the fourth subgraph computation. Reconstruction shows that the bottom 0, 1 in this subgraph corresponds to 0, 1 in the original graph, so at this point we have found complete chains to compute 917, and there is no need to explore further subgraphs below these 0, 1 nodes.

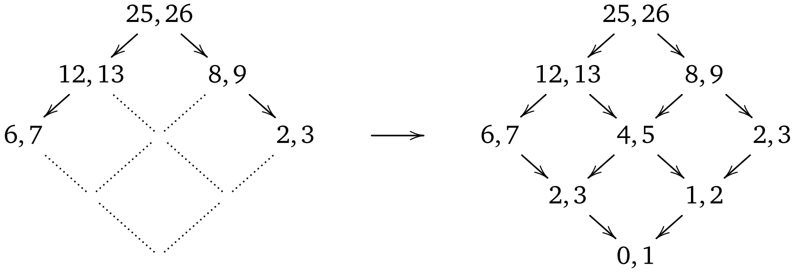


Figure 4.7: Both boundaries and subgraph computation.

4.6 Double-base double-scalar multiplication

All previously explained graph-based approaches to finding double-base chains for single-scalar multiplication can easily be extended to finding double-base chains for double-scalar multiplication. In this section, we explain how to extend the reduced rectangular DAG-based approach to finding double-base chains for double-scalar multiplication.

Recall that inputs for double-scalar multiplication are two scalars (which we will call n_1 and n_2), and two points on an elliptic curve (which we will call P and Q). Given these inputs, the algorithm returns a double-base chain computing $n_1P + n_2Q$, where n_1 and n_2 are expressed as $(n_1, n_2) = \sum_{i=1}^{\ell}(c_i, d_i)2^{a_i}3^{b_i}$ and pairs (c_i, d_i) are chosen from a precomputed set S.

Note that the precomputed set S for double-scalar multiplication is defined differently from the single-scalar case. Recall that in the latter case, each member in the set S is a *single* integer which is the coefficient in a double-base chain representation as defined in Chapter 2.3.4. In the former case, each member in the set S is a *pair* of integers (c_i, d_i) where c_i and d_i are coefficients in a double-base chain representation of n_1 and n_2 respectively.

If $d_i = 0$ this means that the precomputed point depends only on P, e.g., $(2, 0)$, $(3, 0)$, $(4, 0)$, $(5, 0)$ correspond to $2P$, $3P$, $4P$, $5P$ respectively. Similarly, if $c_i = 0$ this means that the precomputed point depends only on Q, e.g., $(0, 2)$, $(0, 3)$, $(0, 4)$, $(0, 5)$ correspond to $2Q$, $3Q$, $4Q$, $5Q$ respectively. If both c_i and d_i are nonzero, this means that the precomputed points depend on both P and Q, e.g., $(1, 1)$, $(1, -1)$, $(2, 1)$, $(2, -1)$, $(1, 2)$, $(1, -2)$ correspond to $P + Q$, $P - Q$, $2P + Q$, $2P - Q$, $P + 2Q$, $P - 2Q$ respectively.

For example, $S = \pm\{(0, 0), (1, 0), (0, 1), (1, 1), (1, -1)\}$ or equivalently $S = \{(0, 0),$ $(1, 0)$, $(-1, 0)$, $(0, 1)$, $(0, -1)$, $(1, 1)$, $(-1, -1)$, $(1, -1)$, $(-1, 1)\}$ means that we allow addition of 0 (no addition), $P, -P, Q, -Q, (P + Q), (-P - Q), (P - Q)$, and $(-P + Q)$ in double-base chains. Note that in this case, S requires space to store 4 points (since negation can be computed on the fly) but it costs only 2 point additions for precomputation, namely, $P + Q$ and $P - Q$.

The algorithm for generating double-base chains for double-scalar multiplication starts by computing $t_1 \equiv n_1 \bmod 2^{\alpha}3^{\beta}$ and $t_2 \equiv n_2 \bmod 2^{\alpha}3^{\beta}$, and then initializes the root node $t_{0,0}$ to the pairs (t_1, t_2), i.e., the reduced representation of (n_1, n_2). For each pair (t_r, t_s) at each node $t_{i,j}$ where $0 \le i \le \alpha$ and $0 \le j \le \beta$, we follow a subgraph computation similar to that explained in Section 4.5, namely,

- If $t_1' = (t_r - c)/2$ and $t_2' = (t_s - d)/2$ are integers, where (c, d) is from the set S, then insert this pair (t_1', t_2') to the node $t_{i+1,j}$ if it does not exist yet or update the cost if cheaper.

- If $t_1'' = (t_r - c)/3$ and $t_2'' = (t_s - d)/3$ are integers, where (c, d) is from the set S, then insert this pair (t_1'', t_2'') to the node $t_{i,j+1}$ if it does not exist yet or update the cost if cheaper.

Once all pairs (t_r, t_s) at all nodes $t_{i,j}$ are visited, apply the boundary node computation. The algorithm continues by repeating subgraph and boundary node computation until pairs $(c, d) \in S$ are reached in a subgraph for which the values of (t_r, t_s) at the root node are less than $2^\alpha 3^\beta$ as integers, i.e., without performing modular reduction (see example below). Notice that the concepts of reduced representative, boundary nodes, and subgraphs are the same as explained in Section 4.5.

Example 4.4: *Compute double-base chains for* $n_1 = 83$ *and* $n_2 = 125$.

This example also uses $\alpha = \beta = 2$ and follows the same cost model as in previous examples, but the set of coefficients is $S = \pm\{(0,0),(1,0),(0,1),(1,1),(1,-1)\}$.

Since $83 \equiv 11 \bmod 2^2 3^2$ and $125 \equiv 17 \bmod 2^2 3^2$, we compute a subgraph of size 2×2 starting with $(11, 17)$. We refer to this subgraph as Subgraph1117. Figure 4.8 depicts the result of this computation.

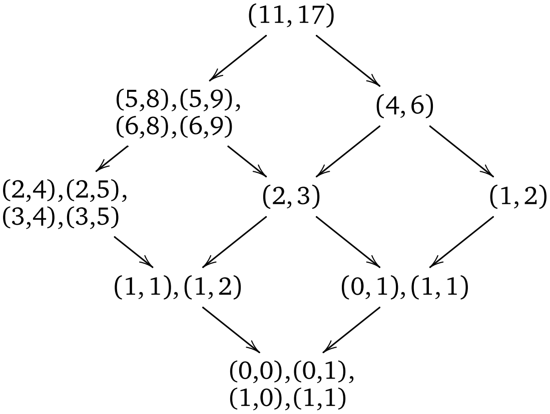


Figure 4.8: Computation of Subgraph1117.

To compute the next subgraphs, we have to recompute all pairs of boundary nodes as if they were computed from the original integers (no modular reduction by $2^2 3^2$). For example, the pair $(2, 4)$ of the leftmost node, are computed as $2 = ((11-1)/2-1)/2$ and $4 = ((17-1)/2)/2$. Therefore, these numbers map to $((83-1)/2-1)/2 = 20$ and $((125-1)/2)/2 = 31$.

Apply similar recomputations for all pairs along the bottom left boundary nodes: (2,5), (3,4), (3,5) map to (20,32), (21,31), (21,32); (1,1), (1,2) map to (7,10), (7,11); and (0,0), (0,1), (1,0), (1,1) map to (2,3), (2,4), (3,3), (3,4) respectively. We refer to this subgraph as Subgraph2031. Figure 4.9 depicts the result of the boundary node recomputation and the subgraph computation of that subgraph.

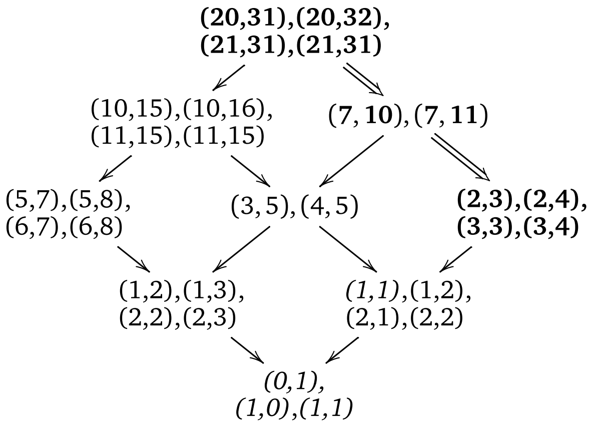


Figure 4.9: Computation of Subgraph2031, the bottom left to Subgraph1117. Note that pairs from Subgraph1117 are labeled with bold face and doubled arrows; pairs in S are labeled in italics.

Similar recomputations are also applied to all pairs along the bottom right boundary nodes of Subgraph1117. We refer to this subgraph as Subgraph0914. Figure 4.10 depicts the result of the boundary node recomputation and the subgraph computation of that subgraph.

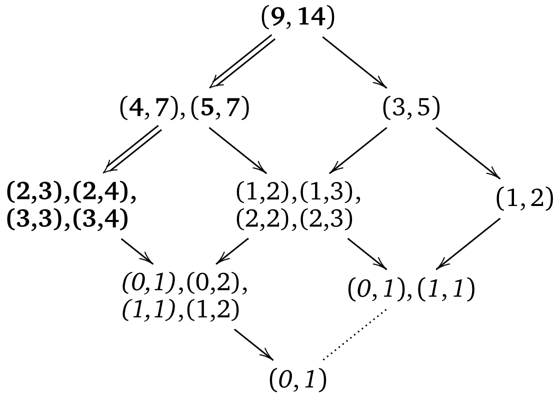


Figure 4.10: Computation of Subgraph0914, the bottom right to Subgraph1117. Note that pairs from Subgraph1117 are labeled with bold face and doubled arrows; pairs in S are labeled in italics; dotted line means no computations.

Notice that we do not need to compute the subgraph below Subgraph1117 (i.e., the bottom right of Subgraph2031 and bottom left of Subgraph0914) because (1) the root nodes of Subgraph2031 and Subgraph0914 are already less than $2^2 3^2$, meaning that the reduced representations are the same as without performing modular reduction; (2) both bottom left and bottom right boundaries reach pairs $(c,d) \in S$, i.e., both left and right child nodes of $(2,3),(2,4),(3,3)$, $(3,4)$ reach pairs in S.

We continue recomputing the bottom left boundary of Subgraph2031. We refer to this subgraph as Subgraph0507. Figure 4.11 depicts the result of this computation. We also continue recomputing the bottom right boundary of Subgraph0914. We refer to this subgraph as Subgraph0102. Figure 4.12 depicts the result of this computation. We do not perform further subgraph computation because Subgraph0507 and Subgraph0102 reach pairs $(c,d) \in S$. An overview of subgraphs is depicted in Figure 4.13.

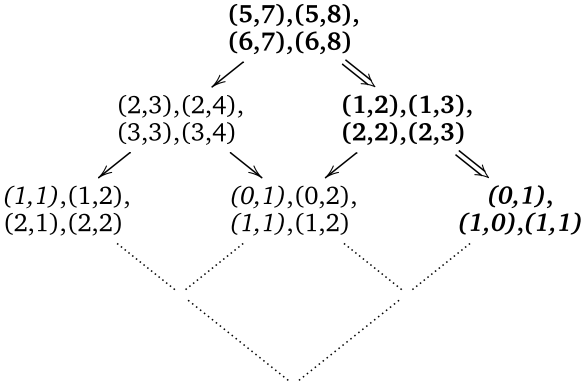


Figure 4.11: Computation of Subgraph0507, the bottom left to Subgraph2031. Note that pairs from Subgraph2031 are labeled with bold face and doubled arrows; pairs in S are labeled in italics; dotted lines mean no computations.

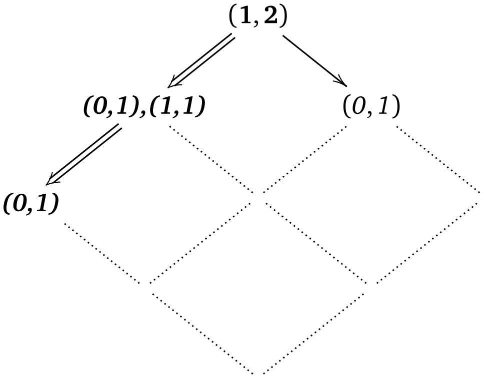


Figure 4.12: Computation of Subgraph0102, the bottom right to Subgraph0914. Note that pairs from Subgraph0914 are labeled with bold face and doubled arrows; pairs in S are labeled in italics; dotted lines mean no computations.

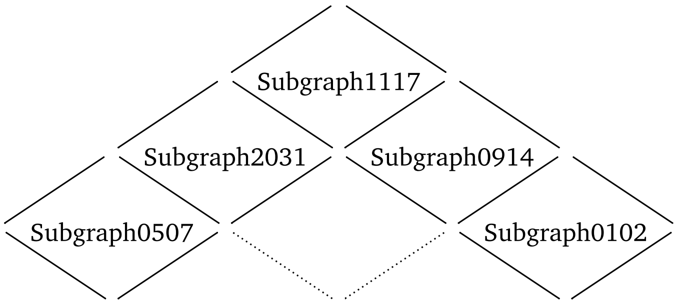


Figure 4.13: Overview of subgraphs. Note that dotted lines mean no computation for that subgraph.

4.7 Results and comparisons

We conducted several experiments to measure the performance of our algorithms for single-scalar and double-scalar multiplications where in each case we compared to both single-base and double-base algorithms, namely, we considered single-base single-scalar, double-base single-scalar, single-base double-scalar, and double-base double-scalar multiplications. These experiments were designed to separately measure the performance due to the new tripling formulas, the performance due to the graph-based approach to generate double-base chains, and the overall performance of combining these two. Comparisons among various algorithms and previous results are also presented.

In all experiments we used at least 1000 randomly chosen integers between 0 and $2^\ell - 1$. The average number of multiplications observed in these experiments, and the average divided by ℓ, are rounded to a limited number of digits after the decimal point and reported below as "Mults" and "Mults/ℓ". The rounding means that dividing "Mults" by ℓ does not exactly produce "Mults/ℓ".

4.7.1 Overall performance

To illustrate the overall performance when using the new tripling formulas together with optimal double-base chains, we performed experiments on many different sets of coefficients focusing on 256-bit scalars. In these experiments, we used the traditional $\mathbf{S} = 0.8\mathbf{M}$ and $\mathbf{m}_a = \mathbf{m}_2 = \mathbf{a} = 0\mathbf{M}$.

We used mixed coordinate systems: projective coordinates as input to doubling and tripling mixed with extended coordinates as input to addition. Each "×2+c" and "×3+c" thus produces projective coordinates as output but uses extended coordinates internally if $c \neq 0$. We take P in affine coordinates but compute further multiples cP in extended coordinates: an inversion to convert to affine coordinates is not worthwhile.

Recall the operation costs shown in Table 4.1: a doubling without an addition costs $3\mathbf{M} + 4\mathbf{S}$; a tripling without an addition costs $9\mathbf{M} + 3\mathbf{S}$; a doubling followed by a mixed addition of an extended point costs $11\mathbf{M}+4\mathbf{S}$ ($4\mathbf{M}+4\mathbf{S}$ for the doubling producing extended coordinates, and $7\mathbf{M}$ for the mixed addition producing projective coordinates); a doubling followed by a mixed addition of an affine point costs $10\mathbf{M}+4\mathbf{S}$; a tripling followed by a mixed addition of an extended point costs $18\mathbf{M}+3\mathbf{S}$; a tripling followed by a mixed addition of an affine point costs $17\mathbf{M}+3\mathbf{S}$.

For double-base *single*-scalar multiplication, we considered many sets of coefficients, including those in [BBLP07]. Table 4.2 shows the results from the four best coefficient sets S, along with $S = \{-1,0,1\}$ for comparison. Mults/ℓ denotes the number of field multiplications per bit, including the cost of initial cP computations and the cost of the subsequent double-base chain. The best result is $7.47\mathbf{M}$ per bit to compute scalar multiplication. A comparison to previous single-scalar multiplication results is presented in Section 4.7.5 and Table 4.9.

Table 4.2: New results for double-base single-scalar multiplication for 256-bit numbers.

Mults	Mults/ℓ	S	Pre cost	Table size
1994.84	7.79233	$\pm\{0,1\}$	–	1
1915.82	7.48369	$\pm\{0,1,2,4,5,7,11,13\}$	$44.4\mathbf{M}$	7
1914.77	7.47959	$\pm\{0,1,5,7,11,13,17,19,23,25\}$	$76.4\mathbf{M}$	9
1913.14	7.47320	$\pm\{0,1,5,7,11,13,17,19\}$	$60.4\mathbf{M}$	7
1912.91	**7.47229**	$\pm\{0,1,2,4,5,7,11,13,17,19\}$	$60.4\mathbf{M}$	9

Note: "Pre cost" denotes the precomputation cost. This cost is already included in the first column.

For double-base *double*-scalar multiplication, we considered all sets of coefficients as in [DKS09], namely, $S_1 = \pm\{(0,0),(1,0),(0,1),(1,1),(1,-1)\}$ by precomputing $P+Q$ and $P-Q$, $S_5 = S_1 \cup \pm\{(5,0),(0,5)\}$ by precomputing $5P$ and $5Q$ in addition to S_1, $S_7 = S_5 \cup \pm\{(7,0),(0,7)\}$ by precomputing $7P$ and $7Q$ in addition to S_5, and $S_{5^2} = S_5 \cup \pm\{(1,5),(1,-5),(5,1),(5,-1),(5,5),(5,-5)\}$ by precomputing $P+5Q, P-5Q, 5P+Q, 5P-Q, 5P+5Q$ and $5P-5Q$ extra to S_5. We also considered sets of coefficients that we label $S_{5a}, S_{5b}, S_{5c}, S_{5d}, S_{5e}, S_{7a}, S_{7b}$, S_{7c}, S_{best1} (see below). Sets S_{5*} extend S_5 to include more precomputation points. Similarly, sets S_{7*} extend S_7 to include more precomputation points. Set S_{best1} is derived from the best precomputation set of the double-base single-scalar to work for double-base double-scalar multiplication. Table 4.3 summarizes precomputation sets for double-base double-scalar multiplication.

The results from the four best coefficient sets S are shown in Table 4.4 along with the result using $S_1 = \pm\{(0,0),(1,0),(0,1),(1,1),(1,-1)\}$ for comparison. Note that these costs already include the cost of precomputation. The best result is $2250.76\mathbf{M}$ using set S_{5e} with table size 16. A comparison to previous double-scalar multiplication results is presented in Section 4.7.6 and Table 4.10.

Table 4.3: Precomputation sets for double-base double-scalar multiplication.

Set S	Precomputation	Pre cost	|T|
$S_1 = \pm\{(0,0),(1,0),(0,1),(1,1),(1,-1)\}$	$P+Q, P-Q$	12.0**M**	4
$S_5 = S_1 \cup \pm\{(5,0),(0,5)\}$	$5P, 5Q$ (and S_1)	52.8**M**	6
$S_7 = S_5 \cup \pm\{(7,0),(0,7)\}$	$7P, 7Q$ (and S_5)	68.8**M**	8
$S_{5^2} = S_5 \cup \pm\{(1,5),(1,-5),(5,1),(5,-1),(5,5),(5,-5)\}$	$P+5Q, P-5Q, 5P+Q, 5P-Q, 5P+5Q, 5P-5Q$ (and S_5)	96.8**M**	12
$S_{5a} = S_5 \cup \pm\{(2,0),(0,2),(4,0),(0,4)\}$	$2P, 2Q, 4P, 4Q$ (and S_5)	52.8**M**	10
$S_{5b} = S_{5a} \cup \pm\{(1,2),(1,-2),(2,1),(2,-1),(1,4),(1,-4),(4,1),(4,-1),(1,5),(1,-5),(5,1),(5,-1)\}$	$P+2Q, P-2Q, 2P+Q, 2P-Q, P+4Q, P-4Q, 4P+Q, 4P-Q, P+5Q, P-5Q, 5P+Q, 5P-Q$ (and S_{5a})	136.8**M**	22
$S_{5c} = S_{5a} \cup \pm\{(2,2),(2,-2),(4,4),(4,-4),(5,5),(5,-5)\}$	$2P+2Q, 2P-2Q, 4P+4Q, 4P-4Q, 5P+5Q, 5P-5Q$ (and S_{5a})	100.8**M**	16
$S_{5d} = S_{5a} \cup \pm\{(2,4),(2,-4),(4,2),(4,-2),(2,5),(2,-5),(5,2),(5,-2),(4,5),(4,-5),(5,4),(5,-4)\}$	$2P+4Q, 2P-4Q, 4P+2Q, 4P-2Q, 2P+5Q, 2P-5Q, 5P+2Q, 5P-2Q, 4P+5Q, 4P-5Q, 5P+4Q, 5P-4Q$ (and S_{5a})	148.8**M**	22
$S_{5e} = S_{5a} \cup \pm\{(1,5),(1,-5),(5,1),(5,-1),(5,5),(5,-5)\}$	$P+5Q, P-5Q, 5P+Q, 5P-Q, 5P+5Q, 5P-5Q$ (and S_{5a})	96.8**M**	16
$S_{7a} = S_{5a} \cup \pm\{(7,0),(0,7)\}$	$7P, 7Q$ (and S_{5a})	68.8**M**	12
$S_{7b} = S_{5b} \cup \pm\{(7,0),(0,7),(1,7),(1,-7),(7,1),(7,-1)$	$7P, 7Q, P+7Q, P-7Q, 7P+Q, 7P-Q$ (and S_{5b})	180.8**M**	28
$S_{7c} = S_{5c} \cup \pm\{(7,0),(0,7),(7,7),(7,-7)$	$7P, 7Q, 7P+7Q, 7P-7Q$ (and S_{5c})	132.8**M**	20
$S_{\text{best1}} = S_1 \cup \pm\{(2,0),(0,2),(4,0),(0,4),(5,0),(0,5),(7,0),(0,7),(11,0),(0,11),(13,0),(0,13),(17,0),(0,17),(19,0),(0,19)\}$	$2P, 4P, 5P, 7P, 11P, 13P, 17P, 19P, 2Q, 4Q, 5Q, 7Q, 11Q, 13Q, 17Q, 19Q$ (and S_1)	132.8**M**	20

Note: “Pre cost” denotes the precomputation cost. “|T|” denotes the table size.

Table 4.4: New results for double-base double-scalar multiplication for 256-bit numbers.

Mults	Mults/ℓ	S	Table size
2351.86	9.18695	S_1	4
2266.32	8.85281	S_{5d}	22
2264.67	8.84637	S_{5b}	22
2251.70	8.79570	S_{5^2}	12
2250.76	**8.79203**	S_{5e}	16

4.7.2 Impact of curve shapes, tripling formulas, S/M ratios

We also conducted experiments to observe the impact of our algorithms performed on other curve shapes of interest, including traditional $a = -3$ short Weierstrass curves in Jacobian coordinates and twisted Hessian curves in projective coordinates. We considered several coefficient sets but, for each curve shape, present only the coefficient set producing the best results for that curve shape. In order to explicitly distinguish the gain due to the new tripling formulas from the gain due to the optimal double-base chain generation algorithm, we also considered twisted Edwards curves with the old tripling formulas. We also varied the ratio of costs between field squaring and field multiplication, considering $\mathbf{S} = 1\mathbf{M}$, $\mathbf{S} = 0.8\mathbf{M}$, and $\mathbf{S} = 0.67\mathbf{M}$. All of these experiments used double-base single-scalar multiplication with 256-bit scalars. Table 4.5 shows this comparison expressed in the number of field multiplications per bit. Rows with "(new)" show results for the new double-base chains.

Table 4.5: Costs on different curve shapes, tripling formulas, and **S**/**M** ratios for double-base single-scalar for 256-bit numbers.

Curve shape		**S**/**M** ratio		
		1	0.8	0.67
Jacobian-3	[BL08]	-	9.34297	-
	(new)	10.20950	9.12516	8.39722
Hessian	[BCKL15]	-	8.77382	-
	(new)	9.16351	8.52279	8.09017
Twisted Edwards	[HWCD08]	8.40625	7.62109	-
	(new)	8.27195	7.52247	7.01979
Twisted Edwards	(new formulas)	**8.20036**	**7.47415**	**6.97923**

4.7.3 Tree-based vs. graph-based for single scalars

To compare the gain due to the optimal double-base chain generation algorithms with the previous (non-optimal) heuristic algorithms, we present the cost to perform scalar multiplication together with the cost to generate double-base chains. The former is measured by counting the number of field multiplications needed to perform scalar multiplication. The latter is measured by counting the number of nodes generated during the double-base chain generation of each algorithm, namely, the tree-based [DH08] (for double-base single-scalar), tree-JBT [DKS09] (for double-base double-scalar), DAG-based (DAG, Section 4.3), rectangular DAG-based (rDAG, Section 4.4), and reduced rectangular DAG-based (rrDAG, Section 4.5). Note that all our DAG-based algorithms are applicable for both double-base single-scalar and double-base double-scalar.

The number of nodes together with the operation cost per node reflects the time required to run these algorithms. We categorize nodes into 2 types, ones with operation cost $\log_2 n$ and ones with operation cost $\log_2 2^\alpha 3^\beta$. In tree-based, tree-JBT, DAG-based and rDAG-based algorithms, only the first type appears. In rrDAG-based, both types appear; nodes with $\log_2 2^\alpha 3^\beta$ operation cost are intermediate nodes while nodes with $\log_2 n$ operation cost are boundary nodes. The cost to generate double-base chains is computed using the number of nodes and operation cost per node.

We emphasize that the tree-based and tree-JBT algorithms *do not* produce optimal chains while DAG-based, rDAG-based and rrDAG-based algorithms *do* produce optimal chains. Beware that these costs have somewhat high variance, and this variance is visible despite the number of experiments that we carried out.

Table 4.6 displays the results of these experiments for double-base *single*-scalar multiplication using 256-bit scalars, assuming $\mathbf{S} = 0.8\mathbf{M}$. We varied the bound B used for tree pruning, i.e., the maximum number of nodes kept for each level in the tree-based approach, by using $B = 10^2$, $B = 10^3$, $B = 10^4$, and $B = 10^5$; costs increase as B increases, until B is large enough to eliminate the pruning. For the rrDAG, we set the size of subgraphs to be $\alpha = \lfloor (\log_2 n)^{0.5} \rceil$ and $\beta = \lfloor (\log_3 n)^{0.5} \rceil$. These experiments focus on the case where no precomputation is allowed, i.e., $S = \{-1, 0, 1\}$, for comparability to previous work using this S.

Table 4.6: Tree-based vs. graph-based algorithms for double-base single-scalar multiplication for 256-bit numbers.

Method		Mults	Mults/ℓ	Optimal	Nodes	Cost
Tree	$B = 10^2$	2035.56	7.95141	no	6072+ 0	1554432
	$B = 10^3$	2034.46	7.94711	no	41948+ 0	10738688
	$B = 10^4$	2034.46	7.94711	no	171739+ 0	43965184
	$B = 10^5$	2034.46	7.94711	no	173206+ 0	44340736
Graph	DAG	1994.83	7.79230	yes	10449+ 0	2674944
	rDAG	1994.83	7.79230	yes	40845+ 0	10456320
	rrDAG	1994.83	7.79230	yes	4607+38106	2574244

These results suggest that in double-base single-scalar multiplication only the most extreme pruning produces results faster with the tree-based method than DAG, but trees produce suboptimal chains. It depends on the application, e.g., on how often the chain will be used, whether the extra effort is justified. It is interesting to see that even for large pruning bounds B the tree-based algorithm does not reach optimal chains.

The results clearly show the obvious savings of rrDAG over rDAG in the cost of computing the chains and the less obvious benefit of rrDAG over DAG.

4.7.4 Tree-based vs. graph-based for double scalars

Table 4.7 displays the results of analogous experiments for *double*-scalar multiplication. For the rrDAG, we set the size of subgraphs to be $\alpha = \lfloor(\log_2 n)^{0.5}\rceil$ and $\beta = \lfloor(\log_3 n)^{0.5}\rceil$. Because we focus on counting the number of nodes generated during the search for the chain and not on finding the best precomputation set, these experiments focus on the simple case where no precomputation is allowed, i.e., $S = \pm\{(0,0),(1,0),(0,1)\}$. Extensive comparisons of the cost to evaluate double-base double-scalar multiplication where precomputation is allowed can be found in Table 4.10.

Table 4.7: Tree-based vs. graph-based for double-base double-scalar multiplication for 256-bit numbers.

Method	Mults	Mults/ℓ	Optimal	Nodes	Cost
Tree-JBT	2392.17	9.34441	no	34339+0	8790841
DAG	2335.97	9.12488	yes	40309+0	10319079
rDAG	2335.97	9.12488	yes	80193+0	20529550
rrDAG	2335.97	9.12488	yes	11286+93267	6303312

These results suggest that in double-base double-scalar multiplication the cost to generate optimal double-base chains using rrDAG is less than using the tree-JBT approach. Moreover, the non-optimal chains are indeed worse than the optimal ones. This means that our new rrDAG algorithm achieves a better performance in both the time to generate double-base chains and the time to evaluate those chains.

4.7.5 Single-base vs. double-base for single scalars

We also implemented a conventional signed-sliding-window double-and-add algorithm for computing single-base single-scalar and single-base double-scalar multiplication. We used the fastest formulas available, namely, twisted Edwards with parameter $a = -1$, together with the state-of-the-art technique of using mixed coordinate systems. We ran the experiment using various precomputation sets, namely, the 21 sets listed in [BBLP07], and 6 other sets: $\pm\{0,1,2,4,5,7\}$, $\pm\{0,1,2,4,5,7,11,13,17,19,23,25\}$, $\pm\{0,1,2,4,5\}$, $\pm\{0,1,2,4,5,7,11\}$, $\pm\{0,1,2,4,5,7,11,13\}$, $\pm\{0,1,2,4,5,7,11,13,17,19\}$.

Note that the set $\pm\{0,1,5,7\}$ was considered in [BBLP07]. The reason we also considered the set $\pm\{0,1,2,4,5,7\}$ is that in order to compute $5P$ we need to compute $2P$ and $4P$. Therefore, we included $2P$ and $4P$ in the precomputed set. Similar reasons apply to the other 5 extra sets.

Table 4.8 shows results from the best precomputation sets for single-scalar multiplication. We also include the case of no precomputation, i.e., $S = \{-1,0,1\}$. The best result is $7.57\mathbf{M}$ per bit to compute single-base single-scalar multiplication.

We compare our results to previous works. Comparison for single-scalar multiplication is shown in Table 4.9. If no precomputation is allowed, Doche [Doc14] reported $2092.60\mathbf{M} \approx 8.17\ell\mathbf{M}$ using "near optimal controlled" method. Our results

Table 4.8: New results for single-base single-scalar multiplication for 256-bit numbers.

Mults	Mults/ℓ	S	Table size
2170.39	8.47808	$\pm\{0,1\}$	1
1947.55	7.60762	$\pm\{0,1,3,5,7,\ldots,31\}$	16
1939.85	7.57752	$\pm\{0,1,3,5,7,\ldots,17\}$	9
1939.02	7.57428	$\pm\{0,1,3,5,7,\ldots,25\}$	13
1938.57	**7.57252**	$\pm\{0,1,3,5,7,\ldots,21\}$	11

show that our algorithm requires only 1994.84**M** $\approx 7.79\ell$**M**. For the case that precomputation is allowed, Bernstein and Lange [BL08] reported 2038.7**M** $\approx 7.96\ell$**M** using $S = \pm\{0,1,3,5,\ldots,17\}$ with inverted Edwards coordinates. Hisil, Wong, Carter and Dawson [HWCD08] then reported 1950.60**M** $\approx 7.62\ell$**M** using “4-NAF” with mixed (projective/extended) Edwards coordinates and the $a = -1$ addition speedup. Table 4.8 shows that replacing 4-NAF with $\pm\{0,1,3,5,\ldots,21\}$ does better, needing $\approx 7.57\ell$**M**. This is further beaten by our best result for double-base chains, only 1912.91**M** $\approx 7.47\ell$**M**.

Table 4.9: Comparison of single-scalar multiplication to previous works.

B	Mults		Mults/ℓ	S	|T|
d	2092.60	[Doc14]	8.17422	$\pm\{0,1\}$	1
d	**1994.84**	(new)	**7.79233**	$\pm\{0,1\}$	1
s	1950.60	[HWCD08]	7.61953	$\pm\{0,1,3,5,7,9,11,13,15\}$	8
s	1938.57	(new)	7.57252	$\pm\{0,1,3,5,7,\ldots,21\}$	11
d	1913.14	(new)	7.47320	$\pm\{0,1,5,7,11,13,17,19\}$	7
d	**1912.91**	(new)	**7.47229**	$\pm\{0,1,2,4,5,7,11,13,17,19\}$	9

Note: “B” in the first column denotes the base used where “d” means double-base and “s” means single-base. “|T|” in the last column denotes the table size.

4.7.6 Single-base vs. double-base for double scalars

Table 4.10 shows an analogous comparison for double-scalar multiplication. The results show that single-base signed-sliding-window methods use fewer multiplications than double-base Tree-JBT (with old tripling formulas) for double-scalar multiplication. Our rrDAG algorithms, even without precomputations, use fewer multiplications than signed-sliding-window methods with the best precomputation. This means that our algorithm is less costly and at the same time requires less space for lookup tables.

The Tree-JBT and JSF costs in Table 4.10 are copied from [DKS09] which obtained by implicitly converting precomputed points into affine coordinates. However, these costs *do not* include the cost of conversion. Therefore, the total cost to perform scalar multiplication using Tree-JBT and JSF would be higher than the costs shown in the table. The other costs in Table 4.10 are the *total cost* to perform scalar multiplication.

Table 4.10: Comparison of double-scalar multiplication to previous works.

B	Method		|T|	192	256	320	384	448	512
single	s.sld$_{\omega=2}$	[OS02]	16	2355	3140	3925	4710	5495	6280
	s.sld$_{\omega=3}$	[OS02]	34	2286	3049	3811	4573	5335	6097
	inter$_{\omega=4}$	[OS02]	20	2295	3060	3825	4590	5356	6121
	inter$_{\omega=5}$	[OS02]	44	2294	3058	3823	4587	5352	6116
	JSF	[DKS09]	4	2044	2722	3401	4062	4758	5436
double	RHBTJF	[ADI11]	2	1997	2661	3326	3990	4654	5319
	Tree-JBT	[DKS09]	4	1953	2602	3248	3896	4545	5197
	RHBTJF + new tpl	(new)	2	1946	2594	3241	3889	4536	5183
	Tree-JBT$_5$	[DKS09]	6	1920	2543	3168	3792	4414	5042
	HBTJF	[ADI11]	14	1914	2530	3145	3761	4376	4992
	Tree-JBT$_7$	[DKS09]	8	1907	2521	3137	3753	4365	4980
	Tree-JBT$_{52}$	[DKS09]	12	1890	2485	3079	3677	4270	4862
	HBTJF + new tpl	(new)	14	1859	2456	3053	3650	4247	4844
single	slide$_{\omega=3}$		4	1884	2494	3103	3715	4324	4935
	slide$_{\omega=4}$		8	1838	2424	3009	3595	4181	4767
	slide$_{\omega=5}$		16	1836	2391	2944	3500	4055	4610
	slide$_{\omega=6}$		32	1931	2478	3016	3550	4084	4617
double	Tree-JBT + new tpl	(new)	4	1848	2467	3074	3695	4314	4919
	Tree-JBT$_5$ + new tpl	(new)	6	1823	2408	3010	3592	4182	4792
	Tree-JBT$_7$ + new tpl	(new)	8	1800	2394	2973	3554	4142	4727
	Tree-JBT$_{52}$ + new tpl	(new)	12	1787	2354	2918	3489	4056	4621
	rrDAG	(new)	4	1768	2352	2937	3521	4105	4690
	rrDAG$_5$	(new)	6	1748	2315	2883	3450	4018	4585
	rrDAG$_7$	(new)	8	1734	2292	2851	3410	3969	4527
	rrDAG$_{52}$	(new)	12	1709	2252	2794	3337	3879	4422

Note: “B” in the first column specifies the base used. Abbreviations in the second column: “s.sld” means simutaneous sliding (precompute $c_i P + d_i Q$), “inter” means interleaving (precompute multiples of P and Q individually, i.e., $c_i P$ and $d_i Q$), “slide” means sliding window, and “new tpl” means using our new tripling formulas. “|T|” in the third column denotes the table size. The rest of the columns: 192, 245, 320, 384, 448, and 512 denote the bitlength.

The results attributed to [OS02] are quoted from [OS02] for 160-bit integers; we extrapolated linearly to larger bitlengths. The sliding-window results and rrDAG results in Table 4.10 are from our own experiments.

To summarize, our new double-base chains are (for 256-bit scalars) more than 6% better than all previous results for double-scalar multiplication, and more than 10% better than all previous double-base results for double-scalar multiplication.

4.8 Implementations

This section presents python scipts for the retangular DAG-based algorithm (Section 4.8.1) and the reduced rectangular DAG-based algorithm (Section 4.8.2).

4.8.1 Code for the rectangular DAG-based algorithm

```
import math

# cost of point arithmetic on twisted Edwards #
mad = 7          # mixed addition (Z = 1) need to +1 in case tpl -> mad
add = 8          # general addition       need to +1 in case tpl -> add
dbl = 3 + 0.8*4  # doubling
tpl = 9 + 0.8*3  # tripling

w = 1    # max coefficient (d_i)
s = []   # precomputation
for i in range (w,-(w+1),-1):
        s.append(i)

def basic(n,s):

 amax = int(math.log(n,2) + w + 0.5) + 1  # max power of 2
 bmax = int(math.log(n,3) + w + 0.5) + 1  # max power of 3
 wmax = 2*w + 1                           # max element at (i,j)

 # init table #
 table = []
 for i in range (amax+1):
  table.append([])
  for j in range (bmax+1):
   table[i].append([])
   for k in range (wmax+1):
    table[i][j].append([])

 table[0][0][0] = [n,0,[0,0,0]]  # init root
 # table[i][j][k] = [data1, data2, data3]
 # data1 = curr n
 # data2 = cost to reach this curr n,
 # data3 = [div 2 or 3, amount add, prev k index]

 result = []

 for i in range (amax):   # for each power of 2
  for j in range (bmax):  # for each power of 3
   base2 = -1
   base3 = -1
   for k in range (wmax):  # for each element at vertex (i,j)
    if table[i][j][k] == []: continue
    t = table[i][j][k][0]

    if t in s and not(t == 0):  # reach known chain integer
     if len(result) == 0 or table[i][j][k][1] < result[0][1][1]:
      result.insert(0,[[i,j,k],table[i][j][k]])  # add result
     continue

    for ic in range (len(s)):  # for each coefficient
     c = s[ic]
     if (t-c)%2 == 0:
      curr2 = (t-c)/2
      if base2 == -1: base2 = curr2
      idx2 = curr2 - base2
      cost = table[i][j][k][1] + dbl
      if abs(c) == 1: cost += mad
      elif abs(c) > 0: cost += add
      if table[i+1][j][idx2] == [] or table[i+1][j][idx2][1] > cost:
      # update if new number or less cost
       table[i+1][j][idx2] = [(t-c)/2,cost,[2,c,k]]

     if (t-c)%3 == 0:
      curr3 = (t-c)/3
```

```
      if base3 == -1:
       if table[i][j+1][0] == []:
        base3 = curr3
       else:
        base3 = table[i][j+1][0][0]
      idx3 = curr3 - base3
      cost = table[i][j][k][1] + tpl
      if abs(c) == 1: cost += mad + 1   # p2e
      elif abs(c) > 0: cost += add + 1  # p2e
      if table[i][j+1][idx3] == [] or table[i][j+1][idx3][1] > cost:
      # update if new number or less cost
       table[i][j+1][idx3] = [(t-c)/3,cost,[3,c,k]]
 return table,result
```

4.8.2 Code for the reduced rectangular DAG-based algorithm

```
import math

# cost of point arithmetic on twisted Edwards #
mad = 7          # mixed addition (Z = 1) need to +1 in case tpl -> mad
add = 8          # general addition       need to +1 in case tpl -> mad
dbl = 3 + 0.8*4  # doubling
tpl = 9 + 0.8*3  # tripling

w = 1   # max coefficient (d_i)
s = []  # precomputation
for i in range (w,-(w+1),-1):
        s.append(i)

wmax = 2*w + 1  # max element at (i,j)

amod = 16
bmod = 13

def getchain(table,x,rev):
 chain = []
 strchain = ""  # sequence of (mul,add) e.g., (2,0) (3,0) (3,-1) = 3*(3*(2-0)-0)-1 = 17
 [i,j,k] = x[0]
 t = x[1][0]     # value
 d = x[1][2][0]  # dbl or tpl
 a = x[1][2][1]  # add
 while i > 0 or j > 0:
  k = table[i][j][k][2][2]
  if d == 2: i -= 1
  elif d == 3: j -= 1
  strchain += "(" + str(d) + "," + str(a) + ") "
  if rev: chain.insert(0,[d,a])
  else: chain.append([d,a])
  d = table[i][j][k][2][0]
  a = table[i][j][k][2][1]
  t = table[i][j][k][0]
 return chain

def boundary3(table,idxa,idxb,n,mincost):
 i = 0
 init = False
 for j in range (bmod+1):
  for k in range (wmax):

   if idxa*amod+i >= len(table): continue
   elif idxb*bmod+j >= len(table[idxa*amod+i]): continue
   elif k >= len(table[idxa*amod+i][idxb*bmod+j]): continue
```

```
   if table[idxa*amod+i][idxb*bmod+j][k] == []: continue
   if table[idxa*amod+i][idxb*bmod+j][k][1] > mincost[0]: continue

   nd = [[idxa*amod+i,idxb*bmod+j,k],table[idxa*amod+i][idxb*bmod+j][k]]
    # [starting coordinate, table at that coordinate]
   ch = getchain(table,nd,True)
    # chain from that coordinate back to the original n
   t = n
   for x in ch:
    t -= x[1]
    t /= x[0]
   if not init:
    nb = t
    t %= 2**amod * 3**bmod
    nm = t
    table[idxa*amod+i][idxb*bmod+j][k] = [t,nd[1][1],nd[1][2]]
    init = True
   else:
    tt = t % (2**amod * 3**bmod)
    table[idxa*amod+i][idxb*bmod+j][k] = [nm+(t-nb),nd[1][1],nd[1][2]]
  if not init: continue
  nb/=3; nm/=3

def boundary2(table,idxa,idxb,n,mincost):
 j = 0
 init = False
 for i in range (amod+1):
  for k in range (wmax):

   if idxa*amod+i >= len(table): continue
   elif idxb*bmod+j >= len(table[idxa*amod+i]): continue
   elif k >= len(table[idxa*amod+i][idxb*bmod+j]): continue

   if table[idxa*amod+i][idxb*bmod+j][k] == []: continue
   if table[idxa*amod+i][idxb*bmod+j][k][1] > mincost[0]: continue

   nd = [[idxa*amod+i,idxb*bmod+j,k],table[idxa*amod+i][idxb*bmod+j][k]]
    # [starting coordinate, table at that coordinate]
   ch = getchain(table,nd,True)
    # chain from that coordinate back to the original n
   t = n
   for x in ch:
    t -= x[1]
    t /= x[0]
   if not init:
    nb = t
    t %= 2**amod * 3**bmod
    nm = t
    table[idxa*amod+i][idxb*bmod+j][k] = [t,nd[1][1],nd[1][2]]
    init = True
   else:
    tt = t % (2**amod * 3**bmod)
    table[idxa*amod+i][idxb*bmod+j][k] = [nm+(t-nb),nd[1][1],nd[1][2]]
  if not init: continue
  nb/=2; nm/=2

def subgraph(table,idxa,idxb,n,check,result,mincost):
 for i in range (amod+1):   # for each power of 2
  for j in range (bmod+1):  # for each power of 3
   base2 = -1
   base3 = -1
   for k in range (wmax):   # for each element at vertex (i,j)

    if idxa*amod+i >= len(table): continue
    elif idxb*bmod+j >= len(table[idxa*amod+i]): continue
    elif k >= len(table[idxa*amod+i][idxb*bmod+j]): continue
```

```
    if table[idxa*amod+i][idxb*bmod+j][k] == []: continue
    t = table[idxa*amod+i][idxb*bmod+j][k][0]

    if table[idxa*amod+i][idxb*bmod+j][k][1] > mincost[0]: continue

    # check condition #
    if check:
     nd = [[idxa*amod+i,idxb*bmod+j,k],table[idxa*amod+i][idxb*bmod+j][k]]
     ch = getchain(table,nd,True)
     m = n
     for x in ch:
      m -= x[1]
      m /= x[0]
     if m == 1:        # reach known chain integer
      if len(result) == 0 or table[idxa*amod+i][idxb*bmod+j][k][1] < result[0][1][1]:
       result.insert(0,[[idxa*amod+i,idxb*bmod+j,k],table[idxa*amod+i][idxb*bmod+j][k]])
       # add result
       mincost[0] = result[0][1][1]
      continue

    for ic in range (len(s)):  # for each coefficient
     c = s[ic]

     if i < amod and (t-c)%2 == 0:
      curr2 = (t-c)/2
      if base2 == -1: base2 = curr2
      idx2 = curr2 - base2
      cost = table[idxa*amod+i][idxb*bmod+j][k][1] + dbl
      if abs(c) == 1: cost += mad
      elif abs(c) > 0: cost += add
      if table[idxa*amod+i+1][idxb*bmod+j][idx2] == [] or \
       table[idxa*amod+i+1][idxb*bmod+j][idx2][1] > cost:
       # update if new number or less cost
       table[idxa*amod+i+1][idxb*bmod+j][idx2] = [(t-c)/2,cost,[2,c,k]]

     if j < bmod and (t-c)%3 == 0 and not (idxa>0 and i==0):
      curr3 = (t-c)/3
      if base3 == -1:
       if table[idxa*amod+i][idxb*bmod+j+1][0] == []:
        base3 = curr3
       else:
        base3 = table[idxa*amod+i][idxb*bmod+j+1][0][0]
      idx3 = (k+ic)/3
      cost = table[idxa*amod+i][idxb*bmod+j][k][1] + tpl
      if abs(c) == 1: cost += mad + 1   # p2e
      elif abs(c) > 0: cost += add + 1  # p2e
      if table[idxa*amod+i][idxb*bmod+j+1][idx3] == [] or \
       table[idxa*amod+i][idxb*bmod+j+1][idx3][1] > cost:
       # update if new number or less cost
       table[idxa*amod+i][idxb*bmod+j+1][idx3] = [(t-c)/3,cost,[3,c,k]]

def advance(n,s):

 mincost = [n]

 amax = int(math.log(n,2) + w + 0.5) + 1  # max power of 2
 bmax = int(math.log(n,3) + w + 0.5) + 1  # max power of 3

 # init table #
 table = []
 for i in range (amax+1):
  table.append([])
  for j in range (bmax+1):
   table[i].append([])
   for k in range (wmax+1):
    table[i][j].append([])
```

```
table[0][0][0] = [n%(2**amod * 3**bmod),0,[0,0,0]]  # init root
# table[i][j][k] = [data1, data2, data3]
# data1 = curr n
# data2 = cost to reach this curr n,
# data3 = [div 2 or 3, amount add, prev k index]

result = []

for idxJ in range (bmax/bmod):
 for idxI in range (amax/amod):
  if idxI > 0:
   boundary3(table,idxI,idxJ,n,mincost)
  if idxJ > 0:
   boundary2(table,idxI,idxJ,n,mincost)
  subgraph(table,idxI,idxJ,n,True,result,mincost)
return table,result
```

5

Manipulating Curve Standards

This chapter analyzes how hard it is for an attacker who knows a secret vulnerablity in a family of curves, to bring one of these curves to widespread use. The chapter is written from the perspective of the attacker. The attacker wants to stop secure encryption and might argue as follows.

More and more Internet traffic is encrypted. This poses a threat to our society as it limits the ability of government agencies to monitor Internet communication for the prevention of terrorism and globalized crime. For example, an increasing number of servers use *Transport Layer Security* (TLS) as default (not only for transmissions that contain passwords or payment information) and also most modern chat applications encrypt all communication. This increases the cost of protecting society as it becomes necessary to collect the required information at the end points, i.e., either the servers or the clients. This requires agencies to either convince the service providers to make the demanded information available or to deploy a back door on the client system respectively. Both actions are much more expensive for the agencies than collecting unprotected information from the transmission wire. Fortunately, under reasonable assumptions, it is feasible for agencies to fool users into deploying cryptographic systems that the users believe are secure but that the agencies are able to break.

Elliptic-curve cryptography (ECC) has a reputation for high security and has become increasingly popular. For definiteness this chapter considers the elliptic-curve Diffie–Hellman (ECDH) key-exchange protocol and specifically “ephemeral ECDH”, which has a reputation of being the best way to achieve forward secrecy. The literature models ephemeral ECDH as the following protocol $\mathrm{ECDH}_{E,P}$, Diffie–Hellman key exchange using a point P on an elliptic curve E:

1. Alice generates a private integer a and sends the ath multiple of P on E.

2. Bob generates a private integer b and sends bP.
3. Alice computes abP as the ath multiple of bP.
4. Bob computes abP as the bth multiple of aP.
5. Alice and Bob encrypt data using a secret key derived from abP.

There are various published attacks showing that this protocol is breakable for many elliptic curves E, no matter how strong the encryption is. See Section 5.1 for details. However, there are also many curves E for which the public literature does not indicate any security problems. Similar comments apply to, e.g., elliptic-curve signatures.

This model begs the question of where the curve E comes from. The standard answer is that a central authority generates a curve for the public (while advertising the resulting benefits for security and performance).[1] This does not mean that the public will accept arbitrary curves; the main objective in this chapter is to analyze the security consequences of various possibilities for what the public will accept. The general picture is that Alice, Bob, and the central authority Jerry are actually participating in the following three-party protocol ECDH_A, where A is a function determining the public acceptability of a standard curve:

−1. Jerry generates a curve E, a point P, auxiliary data S with $A(E,P,S) = 1$. (The "seeds" for the NIST curves are examples of S; see Section 5.3.)

0. Alice and Bob verify that $A(E,P,S) = 1$.

1. Alice generates a private integer a and sends aP.

2. Bob generates a private integer b and sends bP.

3. Alice computes abP as the ath multiple of bP.

4. Bob computes abP as the bth multiple of aP.

5. Alice and Bob encrypt data using a secret key derived from abP.

This chapter analyzes the consequences of Jerry cooperating with an eavesdropper Eve to break the encryption used by Alice and Bob. The central question is how Jerry can use his curve-selection flexibility to minimize the attack cost.

Obviously the cost c_A of breaking ECDH_A depends on A, the same way that the cost $c_{E,P}$ of breaking $\text{ECDH}_{E,P}$ depends on E and P. One might think that, to evaluate c_A, one simply has to check what the public literature says about $c_{E,P}$, and then minimize $c_{E,P}$ over all (E,P,S) with $A(E,P,S) = 1$. The reality is more complicated, for three reasons:

---

[1] See, e.g., ANSI X9.62 [X999] ("public key cryptography for the financial services industry"), IEEE P-1363 [IEE00], SECG [Cer00b], NIST FIPS 186 [NIS00], ANSI X9.63 [X901], Brainpool [Bra05], NSA Suite B [NSA05], and ANSSI FRP256V1 [ANS11]. Note that this chapter is not a historical review of which standards have been sabotaged and which have not; it is a sabotage cost assessment and a guide for manipulating future standards.

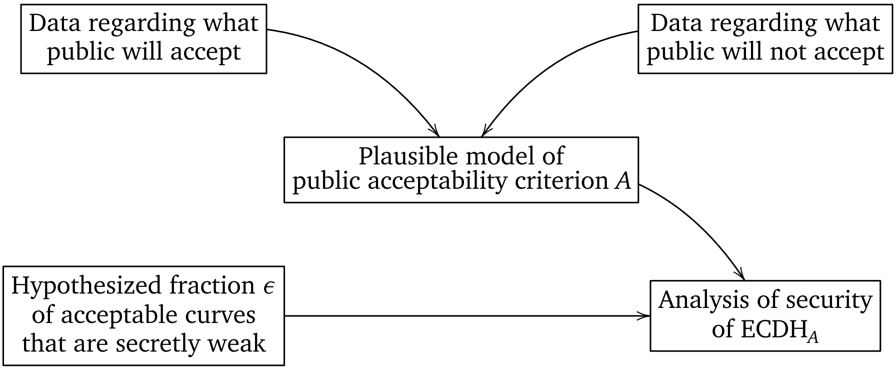


Figure 5.1: Data flow in this chapter. The available data regarding public acceptability is stratified into five different models of the public acceptability criterion A, considered in Sections 5.2, 5.3, 5.4, 5.5, and 5.6 respectively, with five different shapes of the auxiliary curve data S. The security of each A is analyzed for variable ϵ.

1. There may be vulnerabilities not known to the public: curves E for which $c_{E,P}$ is smaller than indicated by the best public attacks. The starting assumption of this work is that Jerry and Eve are secretly aware of a vulnerability that applies to a fraction ϵ of all curves that the public believes to be secure. The obvious strategy for Jerry is to standardize a vulnerable curve.
2. Some choices of A limit the number of curves E for which there *exists* suitable auxiliary data S. If $1/\epsilon$ is much larger than this limit then Jerry cannot expect any vulnerable (E, P, S) to have $A(E, P, S) = 1$. This chapter show that, fortunately for Jerry, this limit is much larger than the public thinks it is. See Sections 5.4 and 5.5.
3. Other choices of A do not limit the number of vulnerable E for which S *exists* but nevertheless complicate Jerry's task of *finding* a vulnerable (E, P, S) with $A(E, P, S) = 1$. See Section 5.3 for analysis of the cost of this computation.

If Jerry succeeds in finding a vulnerable (E, P, S) with $A(E, P, S) = 1$, then Eve simply exploits the vulnerability, obtaining access to the information that Alice and Bob have encrypted for transmission.

Of course, this could require considerable computation for Eve, depending on the details of the secret vulnerability. The goal of this chapter is not to evaluate the cost of Eve's computation, but rather to evaluate the impact of A and ϵ upon the cost of Jerry's computation.

For this evaluation it is adequate to use simplified models of secret vulnerabilities. This chapter specifies various artificial curve criteria that have no connection to vulnerabilities but that are satisfied by (E, P, S) with probability ϵ for various sizes of ϵ, then evaluates how difficult it is for Jerry to find (E, P, S) that satisfy these criteria and that have $A(E, P, S) = 1$.

The possibilities that this chapter analyzes for A are models built from data regarding what the public will accept. See Figure 5.1 for the data flow. Consider, for example, the following data: the public has accepted without complaint the constants $\sin(1), \sin(2), \ldots, \sin(64)$ in MD5, the constants $\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{10}$ in SHA-1, the constants $\sqrt[3]{2}, \sqrt[3]{3}, \sqrt[3]{5}, \sqrt[3]{7}$ in SHA-2, the constant $(1+\sqrt{5})/2$ in RC5, the constant $e = \exp(1)$ in Brainpool, the constant $1/\pi$ in ARIA, etc. All of these constants are listed in [Wik15a] as examples of "nothing up my sleeve numbers". Extrapolating from this data, it is predictable that the public would accept, e.g., the constant $\cos(1)$ used in the example curve BADA55-VPR-224 in Section 5.4. Enumerating a complete list of acceptable constants would require more systematic psychological experiments, so this chapter has chosen a conservative acceptability function A in Section 5.4 that allows just 17 constants and their reciprocals.

The reader might object to the specification of ECDH_A as implicitly assuming that the party sabotaging curve choices is the same as the party issuing curve standards to be used by Alice and Bob. In reality, these two parties are different, and having the first party exercise sufficient control over the second party is often a delicate exercise in finesse. See, for example, [Kel03] and [CFN$^+$14].

Credits. The content of this chapter is based on full version of the paper *"How to manipulate curve standards: a white paper for the black hat"* [BCC$^+$15] which is a joint work with Daniel J. Bernstein, Tung Chou, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen and Christine van Vredendaal. Note that "we" in this chapter refers to these authors playing the role of advisors to Jerry.

Organization. Section 5.1 reviews the curve attacks known to the public and analyzes the probability that a curve resists these attacks; this probability has an obvious impact on the cost of generating curves. Section 5.2, as a warm-up, shows how to manipulate curve choices when A merely checks for these public vulnerabilities. Section 5.3 shows how to manipulate "verifiably random" curve choices obtained by hashing seeds. Section 5.4 shows how to manipulate "verifiably pseudorandom" curve choices obtained by hashing "nothing-up-my-sleeves numbers". Section 5.5 shows how to manipulate "minimal" curve choices. Section 5.6 shows how to manipulate "the fastest curve".

5.1 Security analyses

One obstacle Jerry has to face in deploying his backdoored elliptic curves is that public researchers have raised awareness of certain weaknesses an elliptic curve may have. Once sufficient awareness of a weakness has been raised, many standardization committees will feel compelled to mention that weakness in their standards. This in turn may alert the targeted users, i.e., the general public: some users will check what standards say regarding the properties that an elliptic curve should have or should not have.

The good thing about standards is that there are so many to choose from. Standards evaluating or claiming the security of various elliptic curves include ANSI X9.62 (1999) [X999], IEEE standard P1363 (2000) [IEE00], Certicom SEC 1 v1 (2000)

[Cer00a], Certicom SEC 2 v1 (2000) [Cer00b], NIST FIPS 186-2 (2000) [NIS00], ANSI X9.63 (2001) [X901], Brainpool (2005) [Bra05], NSA Suite B (2005) [NSA05], Certicom SEC 1 v2 (2009) [Cer09], Certicom SEC 2 v2 (2010) [Cer10], OSCCA SM2 (2010) [OSC10a, OSC10b], ANSSI FRP256V1 (2011) [ANS11], and NIST FIPS 186-4 (2013) [NIS13].

These standards vary in many details, and also demonstrate important variations in public acceptability criteria, an issue explored in depth later in this chapter.

Unfortunately for Jerry, some public criteria have become so widely known that all of the above standards agree upon them. Jerry's curves need to satisfy these criteria. This means not only that Jerry will be unable to use these public attacks as back doors, but also that Jerry will have to bear these criteria in mind when searching for a vulnerable curve. Perhaps the vulnerability known secretly to Jerry does not occur in curves that satisfy the public criteria; on the other hand, perhaps this vulnerability occurs *more* frequently in curves that satisfy the public criteria than in general curves. The chance ϵ of a curve being vulnerable is defined relative to the curves that the public will accept.

This section reviews these standard criteria for "secure" curves, along with attacks known to the public. Jerry must be careful, when designing and justifying his curve, to avoid revealing attacks outside this list; such attacks are not known to the public.

Let E be an elliptic curve defined over a large prime field $\mathbb{F}_p$. One can always write E in the form $y^2 = x^3 + ax + b$. Most curve standards choose $a = -3$ for efficiency reasons. Practically all curves have low-degree isogenies to curves with $a = -3$, so this choice does not affect security.

Write $|E(\mathbb{F}_p)|$ for the number of points on E defined over $\mathbb{F}_p$, and write $|E(\mathbb{F}_p)|$ as $p + 1 - t$. Hasse's theorem (see, e.g., [Sil09]) states that $|E(\mathbb{F}_p)|$ is in the "Hasse interval" $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$; i.e., t is between $-2\sqrt{p}$ and $2\sqrt{p}$.

Define r as the largest prime factor of $|E(\mathbb{F}_p)|$ and the cofactor h as $|E(\mathbb{F}_p)|/r$. Let P be a point on E of order r.

5.1.1 ECDLP security criteria

Elliptic curve cryptography is based on the believed hardness of the *elliptic-curve discrete-logarithm problem* (ECDLP), i.e., the belief that it is computationally infeasible to find a scalar n satisfying $Q = nP$ given a random multiple Q of P on E. The state-of-the-art public algorithm for solving the ECDLP is Pollard's rho method (with negation), which on average requires approximately $0.886\sqrt{r}$ point additions. Most publications require the value r to be large; for example, the SafeCurves web page [BL14b] requires that $0.886\sqrt{r} > 2^{100}$.

Some standards put upper limits on the cofactor h, but the limits vary. FIPS 186-2 [NIS00, page 24] claims that "for efficiency reasons, it is desirable to take the cofactor to be as small as possible"; the 2000 version of SEC 1 [Cer00a, page 17] required $h \leq 4$; but the 2009 version of SEC 1 [Cer09, pages 22 and 78] claims that there are efficiency benefits to "some special curves with cofactor larger than four" and thus requires merely $h \leq 2^{\alpha/8}$ for security level 2^{α}.

Another security parameter is the *complex-multiplication field discriminant* (CM field discriminant) which is defined as $D = (t^2 - 4p)/s^2$ if $(t^2 - 4p)/s^2 \equiv 1 \pmod 4$ or otherwise $D = 4(t^2 - 4p)/s^2$, where t is defined as $p + 1 - |E(\mathbb{F}_p)|$ and s^2 is the largest square dividing $t^2 - 4p$. One standard, Brainpool, requires $|D|$ to be large (by requiring a related quantity, the "class number", to be large). However, other standards do not constrain D, there are various ECC papers choosing curves where D is small, and the only published attacks related to the size of D are some improvements to Pollard's rho method on a few curves.

All standards prohibit efficient *additive and multiplicative transfers*. An additive transfer reduces the ECDLP to an easy DLP in the additive group of $\mathbb{F}_p$; this transfer is applicable when r equals p. A degree-k multiplicative transfer reduces the ECDLP to the DLP in the multiplicative group of $\mathbb{F}_{p^k}$ where the problem can be solved efficiently using index calculus if the *embedding degree* k is not too large; this transfer is applicable when r divides $p^k - 1$. All standards prohibit $r = p$, r dividing $p - 1$, r dividing $p + 1$, and r dividing various larger $p^k - 1$; the exact limit on k varies from one standard to another.

5.1.2 ECC security vs. ECDLP security

The most extensive public list of requirements is on the SafeCurves web page [BL14b]. SafeCurves covers hardness of ECDLP, generally imposing more stringent constraints than the standards listed in Section 5.1.1; for example, SafeCurves requires the discriminant D of the CM field to satisfy $|D| > 2^{100}$ and requires the order of p modulo r, i.e., the embedding degree, to be at least $(r-1)/100$. SafeCurves also covers the general security of ECC, i.e., the security of ECC implementations.

For example, if an implementor of NIST P-224 ECDH uses the side-channel-protected scalar-multiplication algorithm recommended by Brier and Joye [BJ02], reuses an ECDH key for more than a few sessions,[2] and fails to perform a moderately expensive input validation that has no impact on normal usage,[3] then a *twist attack* finds the user's secret key using approximately 2^{58} elliptic-curve additions. See [BL14b] for details. SafeCurves prohibits curves with low *twist security*, such as NIST P-224.

Twist-security requires the twist E' of the original curve E to be secure. If $|E(\mathbb{F}_p)| = p + 1 - t$ then $|E'(\mathbb{F}_p)| = p + 1 + t$. Define r' as the largest prime factor of $p + 1 + t$. SafeCurves requires $0.886\sqrt{r'} > 2^{100}$ to prevent Pollard's rho method; $r' \neq p$ to prevent additive transfers; and p having order at least $(r' - 1)/100$ modulo r' to prevent multiplicative transfers. SafeCurves also requires various "combined attacks" to be difficult; this is automatic when cofactors are very small, i.e., when $(p+1-t)/r$ and $(p + 1 + t)/r'$ are very small integers.

---

[2] [CFN⁺14, Section 4.2] reports that Microsoft's SChannel automatically reuses "ephemeral" keys "for two hours".

[3] A very recent paper [JSS15] reports complete breaks of the ECC implementations in Bouncy Castle and Java Crypto Extension, precisely because those implementations fail to validate input points.

5.1.3 Probability analysis

This subsection analyzes the probability of random curves passing the public criteria described above. The important quantities for Jerry are (1) this probability, (2) the probability ϵ that curves passing the public criteria are secretly vulnerable to Jerry's attack, and (3) the number of curves that Jerry can afford and is allowed to try, depending on various optimizations and constraints discussed in subsequent sections. Combining these quantities produces Jerry's overall success chance at creating a vulnerable curve that passes the public criteria.

We begin by analyzing how many random curves have small cofactors. As illustrations we consider cofactors $h = 1$, $h = 2$, and $h = 4$. Note that for large enough (at least 224 bits) prime p, curves with these cofactors automatically satisfy the requirement $0.886\sqrt{r} > 2^{100}$; in other words, requiring a curve to have a small cofactor supersedes requiring a curve to meet minimal public requirements for security against Pollard's rho method.

Let $\pi(x)$ be the number of primes $p \le x$, and let $\pi(S)$ be the number of primes p in a set S. The prime-number theorem states that the ratio between $\pi(x)$ and $x/\log x$ converges to 1 as $x \to \infty$, where log is the natural logarithm. Explicit bounds such as [RS62] are not sufficient to count the number of primes in a short interval $I = [x - y, x]$, but there is nevertheless ample experimental evidence that $\pi(I)$ is very close to $y/\log x$ when y is larger than $\sqrt{x}$.

The number of integers in I of the form $r, 2r$, or $4r$, where r is prime, is the same as the total number of primes in the intervals $I_1 = [x-y, x]$, $I_2 = [(x-y)/2, x/2]$ and $I_4 = [(x-y)/4, x/4]$, namely

$$\pi(I_1) + \pi(I_2) + \pi(I_4) \approx \frac{y}{\log x} + \frac{y/2}{\log(x/2)} + \frac{y/4}{\log(x/4)} = \sum_{h\in\{1,2,4\}} \frac{y/h}{\log(x/h)}.$$

Take $x = p + 1 + 2\sqrt{p}$ and $y = 4\sqrt{p}$ to see that the number of such integers in the Hasse interval is approximately $\sum_{h\in\{1,2,4\}}(4\sqrt{p}/h)/(\log((p+1+2\sqrt{p})/h))$. The total number of integers in the Hasse interval is almost exactly $4\sqrt{p}$, so the chance of an integer in the interval having the form r, $2r$, or $4r$ is approximately

$$\sum_{h\in\{1,2,4\}} \frac{1}{h\log((p+1+2\sqrt{p})/h)}.$$

This does not imply, however, that the same approximation is valid for the number of points on a random elliptic curve. It is known, for example, that the number of points on an elliptic curve is odd with probability almost exactly 1/3, not 1/2; this suggests that the number is prime less often than a uniformly distributed random integer in the Hasse interval would be.

A further difficulty is that we need to know not merely the probability that the cofactor h is small, but the joint probability that both h and $h' = (p + 1 + t)/r'$ are small. Even if one disregards the subtleties in the distribution of $p+1-t$, one should not expect (e.g.) the probability that $p + 1 - t$ is prime to be independent of the probability that $p+1+t$ is prime: for example, if one quantity is odd then the other is also odd.

Galbraith and McKee in [GM00, Conjecture B] formulated a precise conjecture for the probability of any particular h (called "k" there). Perhaps the techniques of [GM00] can be extended to formulate a conjecture for the probability of any particular pair (h,h'). However, no such conjectures appear to have been formulated yet, let alone tested.

To collect facts we performed the following experiment: take $p = 2^{224} - 2^{96} + 1$ (the NIST P-224 prime, which is also used in the following sections), and count the number of points on 1000000 curves. Specifically, we took the curves $y^2 = x^3 - 3x + 1$ through $y^2 = x^3 - 3x + 1000001$, skipping the non-elliptic curve $y^2 = x^3 - 3x + 2$. It is conceivable that the small coefficients make these curves behave nonrandomly, but the same type of nonrandomness appears naturally in Section 5.5, so this is a relevant experiment. Furthermore, the simple description makes the experiment easy to reproduce.

Within this sample we found probability 0.003705 of $h = 1$, probability 0.002859 of $h = 2$, and probability 0.002372 of $h = 4$, with total $0.008936 \approx 2^{-7}$. We also found, unsurprisingly, practically identical probabilities for the twist cofactor: probability 0.003748 of $h' = 1$, probability 0.002902 of $h' = 2$, and probability 0.002376 of $h' = 4$, with total 0.009026.

In our sample we found probability 0.000049 that simultaneously $h \in \{1,2,4\}$ and $h' \in \{1,2,4\}$. This provides reasonable confidence, although not a guarantee, that the events $h \in \{1,2,4\}$ and $h' \in \{1,2,4\}$ are statistically dependent: independence would mean that the joint event would occur with probability approximately 0.000081, so a sample of size 1000000 would contain ≤ 49 such curves with probability under 0.0001.

We found probability $0.000032 \approx 2^{-15}$ of $h = h' = 1$. Our best estimate, with the caveat of considerable error bars, is therefore that Jerry must try about 2^{15} curves before finding one with $h = h' = 1$. If Jerry is free to neglect twist security, searching only for $h = 1$, then the probability jumps by two orders of magnitude to about 2^{-8}. If Jerry is allowed to take any $h \in \{1,2,4\}$ then the probability is about 2^{-7}.

These probabilities are not noticeably affected by the SafeCurves requirements regarding the CM discriminant, additive transfers, and multiplicative transfers. Specifically, random curves have a large CM field discriminant, practically always meeting the SafeCurves CM criterion; none of our experiments found a CM field discriminant below 2^{100}. We also found, unsurprisingly, no curves with $r = p$. As for multiplicative transfers: Luca, Mireles, and Shparlinski gave a theoretical estimate [LMS04] for the probability that for a sufficiently large prime number p and a positive integer K with $\log K = O(\log\log p)$ a randomly chosen elliptic curve $E(\mathbb{F}_p)$ has embedding degree $k \leq K$; this result shows that curves with small embedding degree are very rare. The SafeCurves bound $K = (r-1)/100$ is not within the range of applicability of their theorem, but experimentally we found that about 99% of all curves had a high embedding degree $\geq K$.

5.2 Manipulating curves

Here we target users with minimal acceptability criteria: i.e., we assume that $A(E, P, S)$ checks only the public security criteria for (E, P) described in Section 5.1. The auxiliary data S might be used to communicate, e.g., a precomputed $|E(\mathbb{F}_p)|$ to be verified by the user, but is not used to constrain the choice of (E, P). Curves that pass the acceptability criteria are safe against known attacks, but have no protection against Jerry's secret vulnerability.

5.2.1 Curves without public justification

Here are two examples of standard curves distributed without any justification for how they were chosen. These examples suggest that there are many ECC users who do in fact have minimal acceptability criteria.

As a first example, we look at the FRP256V1 standard [ANS11] published in 2011 by the Agence nationale de la sécurité des systèmes d'information (ANSSI). This curve is $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, where

$$\begin{aligned} b &= \texttt{0xEE353FCA5428A9300D4ABA754A44C00FDFEC0C9AE4B1A1803075ED967B7BB73F}, \\ p &= \texttt{0xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03}. \end{aligned}$$

Another example is a curve published by the Office of State Commercial Cryptography Administration (OSCCA) in China along with the SM2 algorithms in 2010 (cf. [OSC10b, OSC10a]). The curve is of the same form as the ANSSI one with

$$\begin{aligned} b &= \texttt{0x28E9FA9E9D9F5E344D5A9E4BCF6509A7F39789F515AB8F92DDBCBD414D940E93}, \\ p &= \texttt{0xFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFF}. \end{aligned}$$

Each curve E is also accompanied by a point P. The curves meet the ECDLP requirements[4] reviewed in Section 5.1. The only further data provided with these curves is data that could also have been computed efficiently by users from the above information. Nothing in the curve documentation suggests any verification that would have further limited the choice of curves.

5.2.2 The attack

The attack is straightforward. Since the only things that users check are the public security criteria, Jerry can continue generating curves for a fixed p (either randomly or not) that satisfy the public criteria until he gets one that is vulnerable to his secret attack. Alternatively, Jerry can generate curves vulnerable to his secret attack and check them against the public security criteria. Every attack (publicly) known so far allows efficient computation of vulnerable curves, so it seems likely that the same will be true for Jerry's secret vulnerability. After finding a vulnerable curve, Jerry simply publishes it.

---

[4] But not the SafeCurves requirements. Specifically, FRP256V1 has twist security 2^{79}, and the OSCCA curve has twist security 2^{96}.

Of course, Jerry's vulnerability must not depend on any properties excluded by the public security criteria, and there must be enough vulnerable curves. Enumerating 2^7 vulnerable curves over $\mathbb{F}_p$ is likely to be sufficient if Jerry can ignore twist-security, and enumerating 2^{15} vulnerable curves over $\mathbb{F}_p$ is likely to be sufficient even if twist-security is required. See Section 5.1.

Even if Jerry's curves are less frequent, Jerry can increase his chances by also varying the prime p. To simplify our analysis we do not take advantage of this flexibility in this section: we assume that Jerry is forced to reuse a particular standard prime such as a NIST prime or the ANSSI prime. We emphasize, however, that the standard security requirements do not seriously scrutinize the choice of prime, and existing standards vary in their choices of primes. Any allowed variability in p would also improve the attack in Section 5.4, and we do vary p in Section 5.5.

5.2.3 Implementation

We implemented the attack to demonstrate that it is really feasible in practice. In our implementation the same setting as above is adopted and even made more restrictive: the resulting curve should be of the form $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, where p is the same as for the ANSSI curve. The public security criteria we consider are all the standard ECDLP security criteria plus twist security, and we further require that both cofactors are 1.

We use a highly structured parameter b as an artificial model of a secret vulnerability. We show that we can construct a curve with such a b that passes all the public criteria. In reality, Jerry would select a curve with a secret vulnerability rather than a curve with our artificial model of a vulnerability.

Our attack is implemented using the Sage computer algebra system [S+15]. We took

$$b = \texttt{0x5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55EC}$$

as the start value and incremented b until we found a curve that meets the public security criteria. This corresponds to Jerry iteratively checking whether curves that are vulnerable to the secret attack fulfill the public criteria.

As a result we found a desired curve, which we call BADA55-R-256 , with

$$b = \texttt{0x}\underline{\texttt{5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA5}}\texttt{A57}$$

after 1131 increments within 78 minutes on a single core of an AMD CPU.[5] See Figure 5.2.

One can easily check using a computer-algebra system that the curve does meet all the public criteria. It is thus clear that users who only verify public security criteria can be very easily attacked.

---

[5] Note that a *lucky* attacker starting from
$b = \texttt{0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA}$
is able to find the following secure parameter already within 43 minutes after only 622 increments:
$b = \texttt{0x}\underline{\texttt{BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55B}}\texttt{D48}$.

```
p = 0xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03 # ANSSI prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

A = p-3 # standard -3 modulo p
B = 0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BD48
if secure(A,B):
  print "p",hex(p).upper()
  print "A",hex(A).upper()
  print "B",hex(B).upper()

# output:
# p F1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03
# A F1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C00
# B BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BD48
```

Figure 5.2: A procedure to generate the new BADA55-R-256 curve.

5.3 Manipulating seeds

Section 5.2 deals with the easiest case for Jerry that the users are satisfied verifying public security criteria. However, some audiences might demand justifications for the curve choices. In this section, we consider users who are suspicious that the curve parameters might be maliciously chosen to enable a secret attack. Empirically many users are satisfied if they get a *hash verification routine* as justification; see, e.g., ANSI X9.62 [X999], IEEE P1363 [IEE00], SEC 2 [Cer10], or NIST FIPS 186-2 [NIS00]. Hash verification routines mean that Jerry cannot use a very small set of vulnerable curves, but we will show below that he has good chances to get vulnerable curves deployed if they are just somewhat more common.

5.3.1 Hash verification routine

As the name implies, a hash verification routine involves a cryptographic hash function. The inputs to the routine are the curve parameters and a seed that is published along with the curve. Usually the seed is hashed to compute a curve parameter or point coordinate. The ways of computing the parameters differ but the public justification is that these bind the curve parameters to the hash value, making them hard to manipulate since the hash function is preimage resistant[6]. In addition the user verifies a set of public security criteria. We focus on the obstacle that Jerry faces and call curves that can be verified with such routines *verifiably hashed curves*.

We do not recommend the phrase “verifiably hashed”: it is better to claim that the curves are totally random (even though this is not what is being verified) and that these curves could not possibly be manipulated (even though an attacker is in fact

---

[6] If Jerry has a back door in the hash function this situation is no different than in Section 5.2, so we will not assume this.

quite free to manipulate them). For example, ANSI X9.62 [X999, page 31] speaks of "selecting an elliptic curve verifiably at random"; SEC 2 [Cer10, copy and paste: page 8 and page 18] claims that "verifiably random parameters offer some additional conservative features" and "that the parameters cannot be predetermined". NIST uses the term "pseudo-random curves".

Below we recall the curve verification routine for the NIST P-curves. The routine is specified in NIST FIPS 186-2 [NIS00]. Each NIST P-curve is of the form $y^2 = x^3 - 3x + b$ over a prime field $\mathbb{F}_p$ and is published with a seed s. The hash function SHA-1 is denoted as `SHA1`; recall that SHA-1 produces a 160-bit hash value. The bitlength of p is denoted by m. We use $\text{bin}(i)$ to denote the 20-byte big-endian representation of some integer i and use $\text{int}(j)$ to denote the integer with binary expansion j. For given parameters b, p, and s, the verification routine is:

1. Let $z \leftarrow \text{int}(s)$. Compute $h_i \leftarrow \texttt{SHA1}(s_i)$ for $0 \leq i \leq v$, where $s_i \leftarrow \text{bin}((z + i) \bmod 2^{160})$ and $v = \lfloor (m-1)/160 \rfloor$.

2. Let h be the rightmost $m-1$ bits of $h_0 || h_1 || \cdots || h_v$. Let $c \leftarrow \text{int}(h)$.

3. Verify that $b^2 c = -27$ in $\mathbb{F}_p$.

To generate a verifiably hashed curve one starts with a seed and then follows the same steps 1 and 2 as above. Instead of step 3 one tries to solve for b given c; this succeeds for about 50% of all choices for s. The public perception is that this process is repeated with fresh seeds until the first resulting curve satisfies all public security criteria.

5.3.2 Acceptability criteria

One might think that the public acceptability criteria are defined by the NIST verification routine stated above: i.e., $A(E, P, s) = 1$ if and only if (E, P) passes the public security criteria from Section 5.1 and (E, s) passes the verification routine stated above with seed s and E defined as $y^2 = x^3 - 3x + b$.

However, the public acceptability criteria are not actually so strict. P1363 allows $y^2 = x^3 + ax + b$ without the requirement $a = -3$. P1363 does require $b^2 c = a^3$ where c is a hash as above, but neither P1363 nor NIST gives a justification for the relation $b^2 c = a^3$, and it is clear that the public will accept different relations. For example, the Brainpool curves (see Section 5.4) use the simpler relations $a = g$ and $b = h$ where g and h are separate hashes. One can equivalently view the Brainpool curves as following the P1363 procedure but using a different hash for c, namely computing c as g^3/h^2 where again g and h are separate hashes. Furthermore, even though NIST and Brainpool both use SHA-1, SHA-1 is not the only acceptable hash function; for example, Jerry can easily argue that SHA-1 is outdated and should be replaced by SHA-2 or SHA-3.

We do not claim that the public would accept *any* relation, or that the public would accept *any* choice of "hash function", allowing Jerry just as much freedom as in Section 5.2. The exact boundaries of public acceptability are complicated and not immediately obvious. We have determined approximations to these boundaries by extrapolating from existing data; see, e.g., Section 5.4.

5.3.3 The attack

Jerry begins the attack by defining a public hash verification routine. As explained above, Jerry has some flexibility to modify this routine. This flexibility is not *necessary* for the rest of the attack in this section (for example, Jerry can use exactly the NIST verification routine) but a more favorable routine does improve the *efficiency* of the attack. Our cost analysis below makes a particularly efficient choice of routine.

Jerry then tries one seed after another until finding a seed for which the verifiably hashed curve (1) passes the public security criteria but (2) is subject to his secret vulnerability. Jerry publishes this seed and the resulting curve, pretending that the seed was the first random seed that passed the public security criteria.

5.3.4 Optimizing the attack

Assume that the curves vulnerable to Jerry's secret attack are randomly distributed over the curves satisfying the public security criteria. Then the success probability that a seed leads to a suitable curve is the probability that a curve is vulnerable to the secret attack times the probability that a curve satisfies the public security criteria. Depending on which condition is easier to check Jerry runs many hash computations to compute candidate b's, checks them for the easier criterion and only checks the surviving choices for the other criterion. The hash computations and security checks for each seed are independent from other seeds; thus, this procedure can be parallelized with an arbitrary number of parallel computing instances.

We generated a family of curves to show the power of this method and highlight the computing power of hardware accelerators (such as GPUs or Xeon Phis). We began by defining our own curve verification routine and implementing the corresponding secret generation routine. The hash function we use is Keccak with 256-bit output instead of SHA-1. The hash value is $c = \text{int}(\text{Keccak}(s))$, and the relation is simply $b = c$ in $\mathbb{F}_p$. All choices are easily justified: Keccak is the winner of the SHA-3 competition and much more secure than SHA-1; using a hash function with a long output removes the weird order of hashed components and similarly $b = c$ is as simple and unsuspicious as it can get. In reality, however, these choices greatly benefit the attack: the GPUs efficiently search through many seeds in parallel, one single computation of Keccak has a much easier data flow than in the method above, and having b computed without any expensive number-theoretic computation (such as square roots) means that the curve can be tested already on the GPUs and only the fraction that satisfies the first test is passed on to the next stage. Of course, for a real vulnerability we would have to add the cost of checking for that vulnerability, but minimizing overhead is still useful.

Except for the differences stated above, we followed closely the setting of the NIST P-curves. The target is to generate curves of the form $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, and we consider 3 choices of p: the NIST P-224, P-256, and P-384 primes. (For P-384 we

```
import binascii
import simplesha3
hash = simplesha3.keccakc512 # SHA-3 winner with 256-bit output

p = 2^224 - 2^96 + 1 # standard NIST P-224 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def str2int(seed):
  return Integer(seed.encode("hex"),16)

A = p-3
S = "3CC520E9434349DF680A8F4BCADDA648D693B2907B216EE55CB4853DB68F9165"
B = str2int(hash(binascii.unhexlify(S))) # verifiably random
if secure(A,B):
  print "p",hex(p).upper()
  print "A",hex(A).upper()
  print "B",hex(B).upper()

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001
# A FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFE
# B BADA55ECFD9CA54C0738B8A6FB8CF4CCF84E916D83D6DA1B78B622351E11AB4E
```

Figure 5.3: A procedure to generate the new “verifiably random” BADA55-VR-224 curve. Since the hash output is more than 256 bits, we implicitly reduce it modulo p.

switched to Keccak with 384-bit output.) As a placeholder “vulnerability” we define E to be vulnerable if b starts with the hex-string `BADA55EC`. This fixes 8 hex digits, i.e., it simulates a 1-out-of-2^{32} attack. In addition we require that the curves meet the standard ECDLP criteria plus twist security and have both cofactors equal to 1.

5.3.5 Implementation

Our implementation uses NVIDIA’s CUDA framework for parallel programming on GPUs. A high-end GPU today allows several thousand threads to run in parallel, though at a frequency slightly lower than high-end CPUs. We let each thread start with its own random seed. The threads then hash the seeds in parallel. After hashing, each thread outputs the hash value if it starts with the hex-string `BADA55EC`. To restart, each seed is simply increased by 1, so no new source of randomness is required. Checking whether outputs from GPUs also satisfy the public security criteria is done by running a Sage [S+15] script on CPUs. Since only 1 out of 2^{32} curves has the desired pattern, the CPU computation is totally hidden by GPU computation. Longer strings, corresponding to less likely vulnerabilities, make GPUs even more powerful for our attack scheme.

In the end we found 3 “vulnerable” verifiably hashed curves: BADA55-VR-224, BADA55-VR-256, and BADA55-VR-384, each corresponding to one of the three NIST P-curves. See Figures 5.3, 5.4, and 5.5. Of course, as in Section 5.2, Jerry would use

```
import binascii
import simplesha3
hash = simplesha3.keccakc512 # SHA-3 winner with 256-bit output

p = 2^256 - 2^224 + 2^192 + 2^96 - 1 # standard NIST P-256 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def str2int(seed):
  return Integer(seed.encode("hex"),16)

A = p-3
S = "3ADCC48E36F1D1926701417F101A75F000118A739D4686E77278325A825AA3C6"
B = str2int(hash(binascii.unhexlify(S))) # verifiably random
if secure(A,B):
  print "p",hex(p).upper()
  print "A",hex(A).upper()
  print "B",hex(B).upper()

# output:
# p FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
# A FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
# B BADA55ECD8BBEAD3ADD6C534F92197DEB47FCEB9BE7E0E702A8D1DD56B5D0B0C
```

Figure 5.4: A procedure to generate the new “verifiably random” BADA55-VR-256 curve.

a secret vulnerability rather than our artificial “vulnerability”.

As an example, BADA55-VR-256 was found within 7 hours, using a cluster of 41 NVIDIA GTX780 GPUs (`http://blog.cr.yp.to/20140602-saber.html`). Each GPU is able to carry out 170 million 256-bit-output Keccak hashes in a second. Most of the instructions are bitwise logic instructions. On average each core performs 0.58 bitwise logic instructions per cycle while the theoretical maximum throughput is 0.83. We have two explanations for the gap: first, each thread uses many registers, which makes the number of active warps too small to fully hide the instruction latency; second, there is not quite enough instruction-level parallelism to fully utilize the cores in this GPU architecture. We also tested our implementation on K10 GPUs. Each of them carries out only 61 million hashes per second. This is likely to be caused by register spilling: the K10 GPUs have only 63 registers per thread instead of the 255 registers of the GTX780. Using a sufficient amount of computing power easily allows Jerry to deal with secret vulnerabilities that have smaller probabilities of occurrence than 2^{-32}.

5.4 Manipulating nothing-up-my-sleeve numbers

In 1999, M. Scott complained about the choice of unexplained seeds for the NIST curves [Sco99] and concluded “Do they want to be distrusted?”. In the same vein

```
import binascii
import simplesha3
hash = simplesha3.keccakc768 # SHA-3 winner with 384-bit output

p = 2^384 - 2^128 - 2^96 + 2^32 - 1 # standard NIST P-384 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def str2int(seed):
  return Integer(seed.encode("hex"),16)

A = p-3
S = "CA9EBD338A9EE0E6862FD329062ABC06A793575A1C744F0EC24503A525F5D06E"
B = str2int(hash(binascii.unhexlify(S))) # verifiably random
if secure(A,B):
  print "p",hex(p).upper()
  print "A",hex(A).upper()
  print "B",hex(B).upper()

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFF000000000000
#   0000FFFFFFFF
# A FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFF000000000000
#   0000FFFFFFFC
# B BADA55EC3BE2AD1F9EEEA5881ECF95BBF3AC392526F01D4CD13E684C63A17CC4D5F271642AD838991138
#   17A61006413D
```

Figure 5.5: A procedure to generate the new "verifiably random" BADA55-VR-384 curve.

the German ECC Brainpool consortium expressed skepticism [Bra05, Introduction] and suggested using natural constants in place of random seeds. They coined the term "verifiably pseudorandom" for this method of generating seeds. Others speak of "nothing-up-my-sleeves numbers". We comment that "nothing-up-my-sleeves numbers" also appear in other areas of cryptography and can be manipulated in similar ways, but this chapter focuses on manipulation of elliptic curves.

5.4.1 The Brainpool procedure

Brainpool requires that "curves shall be generated in a pseudo-random manner using seeds that are generated in a systematic and comprehensive way". Brainpool produces each curve coefficient by hashing a seed extracted from the bits of $e = \exp(1)$. This first curve cannot be expected to meet Brainpool's security criteria, so Brainpool counts systematically upwards from this initial seed until finding a curve that does meet the security criteria. Brainpool uses a similar procedure to generate primes.

We have written a Sage implementation, emphasizing simplicity and clarity, of the prime-generation and curve-generation procedures specified in the Brainpool standard [Bra05, Section 5]. For example, Figure 5.6 (designed to be shown to the public) uses Brainpool's procedure to generate a 224-bit curve. The output consists

```
import hashlib   # for a PRNG, SHA-1 is standard and sufficiently secure
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20   # 160-bit size for seed, determined by SHA-1 output size

# 224-bit prime p produced by very similar procedure, shown in separate file
p = 0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
k = GF(p); R.<x> = k[]

def secure(A,B):
  if k(B).is_square(): return False
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return "".join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode("hex"),16)

def update(seed):  # add 1 to seed, viewed as integer
  return int2str(str2int(seed) + 1,len(seed))

def fullhash(seed):
  return str2int(hash(seed) + hash(update(seed))) % 2^223

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

nums = real2str(exp(1)/16,7*seedbytes) # enough bits for all curve sizes
S = nums[2*seedbytes:3*seedbytes] # previous bytes are used for 160 and 192
while True:
  A = fullhash(S)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  S = update(S)
  B = fullhash(S)
  if not secure(A,B): S = update(S); continue
  print "p",hex(p).upper()
  print "A",hex(A).upper()
  print "B",hex(B).upper()
  break

# output:
# p D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
# A 2B98B906DC245F2916C03A2F953EA9AE565C3253E8AEC4BFE84C659E
# B 68AEC4BFE84C659EBB8B81DC39355A2EBFA3870D98976FA2F17D2D8D
```

Figure 5.6: An implementation of the Brainpool standard procedure [Bra05, Section 5] to generate a 224-bit curve.

of the following “verifiably pseudorandom” integers p, a, b defining an elliptic curve $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$:

$$p = \texttt{0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF}$$

$$a = \texttt{0x2B98B906DC245F2916C03A2F953EA9AE565C3253E}\underline{\texttt{8AEC4BFE84C659E}}$$

$$b = \texttt{0x6}\underline{\texttt{8AEC4BFE84C659E}}\texttt{BB8B81DC39355A2EBFA3870D98976FA2F17D2D8D}$$

We have added underlines to point out a collision of substrings, obviously quite different from what one expects in “pseudorandom” strings.

What happened here is that the Brainpool procedure generates each of a and

b as truncations of concatenations of various hash outputs (since the selected hash function, SHA-1, produces only 160-bit outputs), and there was a collision in the hash inputs. Specifically, Brainpool uses the same seed-increment function for three purposes: searching for a suitable a; moving from a to b; and moving within the concatenations. The first hash used in the concatenation for a was fed through this increment function to obtain the second hash, and was fed through the same increment function to obtain the first hash used in the concatenation for b, producing the overlap visible above.

A reader who checks the Brainpool standard [Bra05] will find that the 224-bit curve listed there does not have the same (a, b), and does not have this overlap. The reason for this is that, the 224-bit standard Brainpool curve was not actually produced by the standard Brainpool procedure. In fact, although the reader will find overlaps in the standard 192-bit, 256-bit, 384-bit, and 512-bit Brainpool curves, *none* of the standard Brainpool curves below 512 bits were produced by the standard Brainpool procedure. In the case of the 160-bit, 224-bit, 320-bit, and 384-bit Brainpool curves, one can immediately demonstrate this discrepancy by observing that the gap listed between "seed A" and "seed B" in [Bra05, Section 11] is larger than 1, while the standard procedure always produces a gap of exactly 1.

A procedure that actually *does* generate the Brainpool curves appeared a few years later in the Brainpool RFC [LM10] and is reimplemented in Figure 5.7. We now explain how Figures 5.6 and 5.7 differ:

- The procedure in [LM10] assigns seeds to an $(a^*ab^*b)^*$ pattern. It tries consecutive seeds for a until finding that $-3/a$ is a 4th power, then tries further seeds for b until finding that b is not a square, then checks whether the resulting curve meets Brainpool's security criteria. If this fails, it goes back to trying further seeds for a etc.

- The original procedure in [Bra05] assigns seeds to an $(a^*ab)^*$ pattern. It tries consecutive seeds for a until finding that $-3/a$ is a 4th power, then uses the next seed for b, then checks whether b is a non-square and whether the curve meets Brainpool's security criteria. If this fails, it goes back to trying further seeds for a etc.

Figure 5.8 and Figure 5.9 show our implementation of the procedure from [LM10] for all output sizes, including both Brainpool prime generation and Brainpool curve generation. The subroutine `secure` in this implementation also includes an "early abort" (using "division polynomials"), improving performance by an order of magnitude without changing the output; Figure 5.6 omits this speedup for simplicity. Our implementations also skip checking a few security criteria that have negligible probability of failing, such as having large CM field discriminant (see Section 5.1); these criteria are trivially verified after the fact.

We were surprised to discover the failure of the Brainpool standard procedure to generate the Brainpool standard curves. We have not found this failure discussed, or even mentioned, anywhere in the Brainpool RFCs or on the Brainpool web pages. We have also not found any updates or errata to the Brainpool standard after [Bra05].

```
import hashlib   # for a PRNG, SHA-1 is standard and sufficiently secure
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20   # 160-bit size for seed, determined by SHA-1 output size

# 224-bit prime p produced by very similar procedure, shown in separate file
p = 0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
k = GF(p); R.<x> = k[]

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return "".join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode("hex"),16)

def update(seed):  # add 1 to seed, viewed as integer
  return int2str(str2int(seed) + 1,len(seed))

def fullhash(seed):
  return str2int(hash(seed) + hash(update(seed))) % 2^223

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

nums = real2str(exp(1)/16,7*seedbytes) # enough bits for all curve sizes
S = nums[2*seedbytes:3*seedbytes] # previous bytes are used for 160 and 192
while True:
  A = fullhash(S)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  while True:
    S = update(S)
    B = fullhash(S)
    if not k(B).is_square(): break
  if not secure(A,B): S = update(S); continue
  print "p",hex(p).upper()
  print "A",hex(A).upper()
  print "B",hex(B).upper()
  break

# output:
# p D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
# A 68A5E62CA9CE6C1C299803A6C1530B514E182AD8B0042A59CAD29F43
# B 2580F63CCFE44138870713B1A92369E33E2135D266DBB372386C400B
```

Figure 5.7: An implementation of a procedure that actually generates the `brainpool224r1` curve.

One would expect that having a “verifiably pseudorandom” curve not actually produced by the specified procedure would draw more public attention, unless the public never actually tried verifying the curves. We do not explore this line of thought further: we make the worst-case assumption that future curves will be verified by the public.

The Brainpool standard also includes the following statement [Bra05, page 2]: “It is envisioned to provide additional curves on a regular basis for users who wish to change curve parameters regularly, cf. Annex H2 of [X9.62], paragraph ‘Elliptic curve domain parameter cryptoperiod considerations’.” However, the procedure for

```
import sys

import hashlib   # for a PRNG, SHA-1 is standard and sufficiently secure
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20   # 160-bit size for seed, determined by SHA-1 output size

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return "".join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode("hex"),16)

def update(seed):  # add 1 to seed, viewed as integer
  return int2str(str2int(seed) + 1,len(seed))

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

sizes = [160,192,224,256,320,384,512]
S = real2str(pi/16,len(sizes)*seedbytes)
primeseeds = [S[i:i+seedbytes] for i in range(0,len(S),seedbytes)]
S = real2str(exp(1)/16,len(sizes)*seedbytes)
curveseeds = [S[i:i+seedbytes] for i in range(0,len(S),seedbytes)]

for j in range(len(sizes)):
  L,S = sizes[j],primeseeds[j]
  v = (L-1)//160

  def fullhash(seed,bits):
    h = hash(seed)
    for i in range(v): seed = update(seed); h += hash(seed)
    return str2int(h) % 2^bits
```

Figure 5.8: Part 1 of 2: A complete procedure to generate the Brainpool standard curves. Continued in Figure 5.9.

generating further “verifiably pseudorandom” curves is not discussed. One possibility is to continue the original procedure past the first (a,b) pair, but this makes new curves more and more expensive to verify. Another possibility is to replace e by a different natural constant.

5.4.2 The BADA55-VPR-224 procedure

We now present a new verifiably pseudorandom 224-bit curve, BADA55-VPR-224. BADA55-VPR-224 uses the standard NIST P-224 prime, i.e., $p = 2^{224} - 2^{96} + 1$.

To avoid Brainpool’s complications of concatenating hash outputs, we upgrade from the deprecated SHA-1 hash function to the state-of-the-art maximum-security SHA3-512 hash function. We also upgrade to requiring maximum twist security: i.e., both the cofactor and the twist cofactor are required to be 1.

Brainpool already generates seeds using $\exp(1) = e$ and generates primes using $\arctan(1) = \pi/4$, and MD5 already uses $\sin(1)$, so we use $\cos(1)$. We eliminate Brainpool’s contrived, complicated search pattern for a: we simply count upwards, trying every seed for a, until finding the first secure (a,b). The full 160-bit seed for a is the 32-bit counter followed by $\cos(1)$. We complement this seed to obtain the seed for b, ensuring maximal difference between the two seeds. Figure 5.10 is a Sage script

```
while True:
  p = fullhash(S,L)
  while not (p % 4 == 3 and p.is_prime()): p += 1
  if 2^(L-1) - 1 < p and p < 2^L: break
  S = update(S)

k = GF(p)
R.<x> = k[]

def secure(A,B):
  E = EllipticCurve([k(A),k(B)])
  for q in [2,3,5,7]:
    # quick check whether q divides n, without computing n
    for r,e in E.division_polynomial(q).roots():
      if E.is_x_coord(r): return False
  n = E.cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

S = curveseeds[j]
while True:
  A = fullhash(S,L-1)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  while True:
    S = update(S)
    B = fullhash(S,L-1)
    if not k(B).is_square(): break
  if not secure(A,B): S = update(S); continue
  print "p",hex(p).upper()
  print "A",hex(A).upper()
  print "B",hex(B).upper()
  sys.stdout.flush()
  break
```

Figure 5.9: Part 2 of 2: A complete procedure to generate the Brainpool standard curves. Continued from Figure 5.8.

implementing the BADA55-VPR-224 generation procedure. This procedure is simpler and more natural than the Brainpool procedure in Figure 5.7. Here is the resulting curve:

$$\begin{aligned} a &= \texttt{0x7144BA12CE8A0C3BEFA053ED}\underline{\texttt{BADA55}}\texttt{5A42391AC64F052376E041C7D4AF23195E} \\ &\quad \texttt{BD8D83625321D452E8A0C3BB0A048A26115704E45DCEB346A9F4BD9741D14D49}, \\ b &= \texttt{0x5C32EC7FC48CE1802D9B70DBC3FA574EAF015FCE4E99B43EBE3468D6EFB2276B} \\ &\quad \texttt{A3669AFF6FFC0F4C6AE4AE2E5D74C3C0AF97DCE17147688DDA89E734B56944A2}. \end{aligned}$$

5.4.3 How BADA55-VPR-224 was generated: exploring the space of acceptable procedures

The surprising collision of Brainpool substrings had an easy explanation: two hashes in the Brainpool procedure were visibly given the same input. The surprising appearance of the 24-bit string `BADA55` in a above has no such easy explanation. There are 128 hexadecimal digits in a, so one expects this substring to appear anywhere within a with probability $123/2^{24} \approx 2^{-17}$.

```
import simplesha3
hash = simplesha3.sha3512   # SHA-3 standard at maximum security level

p = 2^224 - 2^96 + 1   # standard NIST P-224 prime
k = GF(p)
seedbytes = 20   # standard 160-bit size for seed

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return "".join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode("hex"),16)

def complement(seed):   # change all bits, eliminating Brainpool-type collisions
  return "".join([chr(255-ord(s)) for s in seed])

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(RealField(8*bytes)(seed)*256^bytes),bytes)

sizeofint = 4   # number of bytes in a 32-bit integer
nums = real2str(cos(1),seedbytes - sizeofint)
for counter in xrange(0,256^sizeofint):
  S = int2str(counter,sizeofint) + nums
  T = complement(S)
  A = str2int(hash(S))
  B = str2int(hash(T))
  if secure(A,B):
    print "p",hex(p).upper()
    print "A",hex(A).upper()
    print "B",hex(B).upper()
    break

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001
# A 7144BA12CE8A0C3BEFA053EDBADA555A42391FC64F052376E041C7D4AF23195EBD8D83625321D452E8A0
#   C3BB0A048A26115704E45DCEB346A9F4BD9741D14D49
# B 5C32EC7FC48CE1802D9B70DBC3FA574EAF015FCE4E99B43EBE3468D6EFB2276BA3669AFF6FFC0F4C6AE4
#   AE2E5D74C3C0AF97DCE17147688DDA89E734B56944A2
```

Figure 5.10: A procedure to generate the new "verifiably pseudorandom" BADA55-VPR-224 curve. Compare Figure 5.7.

The actual explanation is as follows. We decided in advance that we would force `BADA55` to appear somewhere in *a* as our artificial model of a "vulnerability". We then identified millions of natural-sounding "verifiably pseudorandom" procedures, and enumerated (using a few hours on our cluster) approximately 2^{20} of these procedures. The space of "verifiably pseudorandom" procedures has many dimensions analyzed below, such as the choice of hash function, the length of the input seed, the update function between seeds, and the initial constant for deriving the seed: i.e., each procedure is defined by a combination of hash function, seed length, etc. The exact number of choices available in any particular dimension is relatively unimportant; what is important is the exponential effect from combining many dimensions.

Since 2^{20} is far above 2^{17}, it is unsurprising that our "vulnerability" appeared in quite a few of these procedures. We selected one of those procedures and presented it as Section 5.4.2 as an example of what could be shown to the public. See Figure 5.11 for another example[7] of such a procedure, generating a BADA55-VPR2-224 curve, starting from e instead of cos(1). We could have easily chosen a more restrictive "vulnerability".

The structure of this attack means that Jerry can use the same attack to target a real vulnerability that has probability 2^{-17}, or (with reasonable success chance) even 2^{-20}, perhaps even reusing our database of curves.

In this section we do not manipulate the choice of prime, the choice of curve shape, the choice of cofactor criterion, etc. Taking advantage of this flexibility (see Section 5.5) would increase the number of natural-sounding Brainpool-like procedures above 2^{30}.

Our experience is that Alice and Bob, when faced with a *single* procedure such as Section 5.4.2 (or Section 5.4.1), find it extremely difficult to envision the entire space of possible procedures (they typically see just a few dimensions of flexibility), and find it inconceivable that the space could have size as large as 2^{20}, never mind 2^{30}. This is obviously a helpful phenomenon for Jerry.

5.4.4 Manipulating bit-extraction procedures

Consider the problem of extracting a fixed-length string of bits from (e.g.) the constant $e = \exp(1) = 2.71828\ldots = (10.10110111\ldots)_2$. Here are several plausible options for the starting bit position:

- Start with the most significant bit: i.e., take bits of e at bit positions 2^1, 2^0, 2^{-1}, 2^{-2}, etc.
- Start immediately after the binary point: i.e., take bits of e at bit positions 2^{-1}, 2^{-2}, etc. For some constants this is identical to the first option: consider, e.g., the first MD5 constant $\sin(1) = 0.84\ldots$.
- Start with the most significant *nibble*: i.e., take bits of e at bit positions 2^3, 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, etc.
- Start with the most significant *byte*: i.e., take bits of e at bit positions 2^7, 2^6, 2^5, etc.
- Start with the byte at position 0. In the case of e this is the same as the fourth option. In the case of sin(1) this means prepending 8 zero bits to the fourth option.

---

[7] Presenting two examples with the same string `BADA55` gives the reader of this chapter some assurance that we did, in fact, choose this string in advance. Otherwise we could have tried to fool the reader as follows: generate a relatively small number of curves, search for an interesting-sounding string in the results, write the previous sections of this chapter to target that string (rather than `BADA55`), and pretend that we had chosen this string in advance.

```
import simplesha3   # Keccak, the SHA-3 winner
hash = simplesha3.keccakc1024   # maximum security level: 512-bit output
seedbytes = 64   # maximum-security 512-bit seed, same size as output

p = 2^224 - 2^96 + 1   # standard NIST P-224 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def int2str(seed,bytes):   # standard little-endian encoding of integer seed
  return "".join([chr((seed//256^i)%256) for i in range(bytes)])

def str2int(seed):
  return sum([ord(seed[i])*256^i for i in range(len(seed))])

def rotate(seed):  # rotate seed by 1 bit, eliminating Brainpool-like collisions
  x = str2int(seed)
  x = 2*x + (x >> (8*len(seed)-1))
  return int2str(x,len(seed))

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

counterbytes = 3   # minimum number of bytes needed to guarantee success
nums = real2str(exp(1)/4,seedbytes - counterbytes)
for counter in xrange(0,256^counterbytes):
  S = int2str(counter,counterbytes) + nums
  R = rotate(S)
  A = str2int(hash(R))
  B = str2int(hash(S))
  if secure(A,B):
    print "p",hex(p).upper()
    print "A",hex(A).upper()
    print "B",hex(B).upper()
    break

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001
# A 8F0FF20E1E3CF4905D492E04110683948BFC236790BBB59E6E6B33F24F348ED2E16C64EE79F9FD27E9A3
#   67FF6415B41189E4FB6BADA555455DC44C4F87011EEF
# B E85067A95547E30661C854A43ED80F36289043FFC73DA78A97E37FB96A2717009088656B948865A660FF
#   3959330D8A1CA1E4DE31B7B7D496A4CDE555E57D05C
```

Figure 5.11: A procedure to generate the new “verifiably pseudorandom” BADA55-VPR2-224 curve. Compare Figure 5.10.

These options can be viewed as using different maps from real numbers x to real numbers y with $0 \le y < 1$: the first map takes x to $|x|/2^{\lfloor \log_2 |x| \rfloor}$, the second map takes x to $x - \lfloor x \rfloor$, the third map takes x to $|x|/16^{\lfloor \log_{16} |x| \rfloor}$, etc. Brainpool used the third of these options, describing it as using “the hexadecimal representation” of e. Jerry can use similarly brief descriptions for any of the options without drawing the public’s attention to the existence of other options. We implemented the first, second, and fourth options; for an average constant this produced slightly more than 2 distinct possibilities for real numbers y.

Jerry can easily get away with extracting an ℓ-bit integer from y by truncation (i.e., $\lfloor 2^\ell y \rfloor$) or by rounding (i.e., $\lceil 2^\ell y \rfloor$). Jerry can defend truncation (which has fundamentally lower accuracy) as simpler, and can defend rounding as being quite standard in mathematics and the physical sciences. We implemented both options, gaining a further factor of 1.5.

Actually, Brainpool uses the bit position indicated above only for the low-security 160-bit Brainpool curve. As shown in Figure 5.8, Brainpool shifts to subsequent bits of e for the 192-bit curve, then to further bits for the 224-bit curve, etc. Brainpool uses 160 bits for each curve (see below), so the seed for the 256-bit curve is shifted by 480 bits. This number 480 depends on how many lower security levels are allocated and on exactly how many bits are allocated to those seeds. A further option, pointed out in [Mer14] by Merkle (Brainpool RFC co-author), is to reverse the order of curve sizes; the number 480 then depends on how many *higher* security levels are allocated. Yet another option is to put curve sizes in claimed order of usage. We did not implement any of the options described in this paragraph.

5.4.5 Manipulating choices of hash functions

The latest (July 2013) revision of the NIST ECDSA standard [NIS13, Section 6.1.1] specifically requires that “the security strength of a hash function used [for curve generation] **shall** meet or exceed the security strength associated with the bit length”. The original NIST curves are exempted from this rule by [NIS13, footnote 2], but this rule prohibits SHA-1 for (e.g.) new 224-bit curves. On the other hand, a more recent Brainpool-related curve-selection document [Mer14] states that “For a PRNG, SHA-1 was (and still is) sufficiently secure.”

There are at least 10 plausible options for standard hash functions used to generate (e.g.) 256-bit curves:

- SHA-1. “We follow the Brainpool standard. What matters is preimage resistance, and SHA-1 still provides more than 2^{128} preimage resistance.”
- SHA-256. “The trusted, widely deployed SHA-256 standard.”
- SHA-384. “SHA-2 at the security level required to handle both sizes of Suite B curves.”
- SHA-512. “The maximum-security SHA-512 standard.”
- SHA-512/256. “NIST’s standard wide-pipe hash function.”
- SHA3-256. “The state-of-the-art SHA-3 standard at a 2^{128} security level.”
- SHA3-384. “The state-of-the-art SHA-3 standard, at the security level required to handle both sizes of Suite B curves.”
- SHA3-512. “The maximum-security state-of-the-art SHA3-512 standard.”
- SHAKE128. “The state-of-the-art SHA-3 standard at a 2^{128} security level, providing flexible output sizes.”

- SHAKE256. “The state-of-the-art SHA-3 standard at a 2^{256} security level, providing flexible output sizes.”

There are also several non-NIST hash functions with longer track records than SHA-3. Any of RIPEMD-128, RIPEMD-160, RIPEMD-256, RIPEMD-320, Tiger, Tiger/128, Tiger/160, and Whirlpool would have been easily justifiable as a choice of hash function before 2006. MD5 and all versions of Haval would have been similarly justifiable before 2004.

Since we targeted a 224-bit curve we had even more standard NIST hash-function options. For simplicity we implemented just 10 hash-function options, namely the following variants of Keccak, the SHA-3 competition winner: Keccak-224, Keccak-256, Keccak-384, Keccak-512, “default” Keccak (“capacity” $c = 576$, 128 output bytes), Keccak-128 (capacity $c = 256$, 168 output bytes), SHA3-224 (which has different input padding from Keccak-224, changing the output), SHA3-256, SHA3-384, and SHA3-512. All of these Keccak/SHA-3 choices can be implemented efficiently with a single code base and variable input parameters.

5.4.6 Manipulating counter sizes

The simplest way to obtain a 160-bit “verifiably pseudorandom” output with SHA-1 is to hash the empty string. Curve generation needs many more outputs (since most curves do not pass the public security criteria), but the simplest way to obtain 2^β “verifiably pseudorandom” outputs is to hash all β-bit inputs.

Hash-function implementations are often limited to byte-aligned inputs, so it is natural to restrict β to a multiple of 8. If each output has chance 2^{-15} of producing an acceptable curve then $\beta = 16$ finds an acceptable curve with chance nearly 90% (“this is retroactively justified by our successfully finding a curve”); $\beta = 24$ fails with negligible probability (“we chose the smallest β for which the probability of failure was negligible”); $\beta = 32$ is easily justified by reference to 32-bit machines; $\beta = 64$ is easily justified by reference to 64-bit machines.

Obviously Brainpool takes a more complicated approach, using bits of some natural constant to further “randomize” its outputs. The standard way to randomize a hash is to concatenate the randomness (e.g., bits of e) with the input being hashed (the counter). Brainpool instead *adds* the randomness to the input being hashed. The Brainpool choice is not secure as a general-purpose randomized hash, although these security problems are of no relevance to curve generation. There is no evidence of public objections to Brainpool’s use of addition here (and to the overall complication introduced by the extra randomization), so there is also no reason to think that the public would object to the more standard concatenation approach.

Overall there are 13 plausible possibilities here: the 4 choices of β above, with the counter on the left of the randomness; the 4 choices of β above, with the counter on the right of the randomness; the counter being added to the randomness; and 4 further possibilities in which the randomness is partitioned into an initial value for a counter (for the top bits) and the remaining seed (for the bottom bits). We implemented the first 9 of these 13 possibilities.

5.4.7 Manipulating hash input sizes

ANSI X9.62 requires $\geq$160 input bits for its hash input. One way for Jerry to advertise a long input is that it allows many people to randomly generate curves with a low risk of collision. For example, Jerry can advertise

- a 160-bit input as allowing 2^{64} curves with only a 2^{-32} risk of collision;
- a 256-bit input as allowing 2^{64} curves with only a 2^{-128} risk of collision; or
- a 384-bit input as allowing 2^{128} curves with only a 2^{-128} risk of collision.

All of these numbers sound perfectly natural. Of course, what Jerry is actually producing is a single standard for many people to use, so multiple-curve collision probabilities are of no relevance, but Jerry can simply say that the input length was chosen for "compatibility" with having users generate their own curves.

Jerry can advertise longer input lengths as providing "curve coverage". A 512-bit input will cover a large fraction of curves, even for primes as large as 512 bits. A 1024-bit input is practically guaranteed to cover all curves, and to produce probabilities indistinguishable from uniform. Jerry can also advertise, as input length, the "natural input block length of the hash function".

We implemented all 6 possibilities listed above. We gained a further factor of 2 by storing the seed (and counter) in big-endian format ("standard network byte order") or little-endian format ("standard CPU byte order").

5.4.8 Manipulating the (a, b) hash pattern

It should be obvious from Section 5.4.1 that there are many degrees of freedom in the details of how a and b are generated: how to distribute seeds between a and b; whether to require $-3/a$ to be a 4th power in $\mathbb{F}_p$; whether to require b to be a non-square in $\mathbb{F}_p$; whether to concatenate hash outputs from left to right or right to left; exactly how many bits to truncate hash outputs to (Brainpool uses one bit fewer than the prime; Jerry can argue for the same length as the prime "for coverage", or more bits "for indistinguishability"); whether to truncate to rightmost bits (as in Brainpool) or leftmost bits (as in various NIST requirements; see [NIS13]); et al.

For simplicity we eliminated the concatenation and truncation, always using a hash function long enough for the target 224-bit prime. We also eliminated the options regarding squares etc. We implemented a total of just 8 choices here. These choices vary in (1) whether to allocate seeds primarily to a or primarily to b and (2) how to obtain the alternate seed (e.g., the seed for a) from the primary seed (e.g., the seed for b): plausible options include complement, rotate 1 byte left, rotate 1 byte right, and four standard versions of 1-bit rotations.

5.4.9 Manipulating natural constants

As noted in the introduction of this chapter, the public has accepted dozens of "natural" constants in various cryptographic functions, and sometimes reciprocals of those constants, without complaint. Our implementation started with just 17 natural constants: π, e, Euler gamma, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, $\sqrt{7}$, $\log(2)$, $(1+\sqrt{5})/2$, $\zeta(3)$, $\zeta(5)$, $\sin(1)$, $\sin(2)$, $\cos(1)$, $\cos(2)$, $\tan(1)$, and $\tan(2)$. We gained an extra factor of almost 2 by including reciprocals.

Jerry could be creative and use previously unused numbers such as numbers derived from some historical document or newspaper, personal information of, e.g., arbitrary celebrities in an arbitrary order, arbitrary collections of natural or physical constants and even a combination of several sources. For example, NewDES [Wik15b] derives its S-Box from the United States Declaration of Independence. If the public accepts numbers with such flimsy justifications as “nothing-up-my-sleeves numbers” then Jerry obviously has as much flexibility as in Section 5.3. We make the worst-case assumption that the public is not quite as easily fooled, and similarly that the public would not accept $703e^{(\sqrt[8]{30}+4\pi)/9\sin(\sqrt[3]{16})}$ as a “nothing-up-my-sleeves number”.

5.4.10 Implementation

Any combination of the above manipulations defines a “systematic” curve-generation procedure. This procedure outputs the first curve parameters (using the specified update function) that result in a “secure” curve according to the public security tests. However, performing all public security tests for each set of parameters considered by each procedure is very costly. Instead, we split the attack into two steps:

1. For a given procedure f_i we iterate over the seeds $s_{i,k}$ using the specific update function of f_i. We check each parameter candidate from seed $s_{i,k}$ for our secret `BADA55` vulnerability. After a certain number of update steps the probability that we passed valid, secure parameters is very high; thus, we discard the procedure and start over with another one. If we find a candidate exhibiting the vulnerability, we perform the public security tests on this particular candidate. If the `BADA55` candidate passes, we proceed to step 2.

2. We perform the whole public procedure f_i starting with seed $s_{i,0}$ and check whether there is any valid parameter set passing the public security checks already before the `BADA55` parameters are reached. If there is such an earlier parameter set, we return to step 1 with the next procedure f_{i+1}.

The largest workload in our attack scenario is step 2, the re-checking for earlier safe curve parameters before `BADA55` candidates. The public security tests are not well suited for GPU parallelization; the first step of the attack procedure is relatively cheap and a GPU parallelization of this step does not have a remarkable impact on the overall runtime. Therefore, we implemented the whole attack only for the CPUs of the Saber cluster and left the GPUs idle.

We initially chose 8000 as the limit for the update counter to have a very good chance that the first secure twist-secure curve starting from the seed is the curve with our vulnerability. For example, BADA55-VPR-224 was found with counter just 184, and there was only a tiny risk of a smaller counter producing a secure twist-secure curve (which we checked later, in the second step). In total $\approx 2^{33}$ curves were covered by this limited computation; more than 2^{18} were secure and twist-secure. We then pushed the 8000 limit higher, performing more computation and finding more curves. This gradually increased the risk of the counter not being minimal, something that we would have had to address by the techniques of Section 5.5; but this issue still did not affect, e.g., BADA55-VPR2-224, which was found with counter 28025.

5.5 Manipulating minimality

Instead of supporting “verifiably pseudorandom” curves as in Section 5.4, some researchers have advocated choosing “verifiably deterministic” curves.

Both approaches involve specifying a “systematic” procedure that outputs a curve. The difference is that in a “verifiably pseudorandom” curve the curve coefficient is the output of a hash function for the *first hash input* that meets specified curve criteria, while a “verifiably deterministic” curve uses the *first curve coefficient* that meets specified curve criteria. Typically the curve uses a “verifiably deterministic” prime, which is the *first prime* that meets specified prime criteria.

Eliminating the hash function and hash input makes life harder for Jerry: it eliminates the main techniques that we used in previous sections to manipulate curve choices. However, as we explain in detail in this section, Jerry still has many degrees of freedom. Jerry can manipulate the concept of “first curve coefficient”, can manipulate the concept of “first prime”, can manipulate the curve criteria, and can manipulate the prime criteria, with public justifications claiming that the selected criteria provide convenience, ease of implementation, speed of implementation, and security.

In Section 5.4 and before we did not manipulate the choice of prime: we obtained a satisfactory level of flexibility in other ways. In this section, the choice of prime is an important component of Jerry’s flexibility. It should be clear to the reader that the techniques in this section to manipulate the prime, the curve criteria, etc. can be backported to the setting of Section 5.4, adding to the flexibility there.

We briefly review a recent proposal that fits into this category and then proceed to work out how much flexibility is left for Jerry.

5.5.1 NUMS curves

In early 2014, Bos, Costello, Longa, and Naehrig [BCLN15] proposed 13 Weierstrass and 13 Edwards curves, spread over 3 different security levels. Each curve was generated following a deterministic procedure (similar to the procedure proposed in [BHKL13]). Given that there are up to 10 different procedures per security level we cannot review all of them here but [BCLN15] is a treasure trove of arguments to justify different prime and curve properties and we will use this to our benefit below.

The same authors together with Black proposed a set of 6 of these curves as an Internet-Draft [BBC+14] referring to these curves as “Nothing Up My Sleeve (NUMS) Curves”. Note that this does not match the common use of “nothing up my sleeves”; see, e.g., the Wikipedia page [Wik15a]. These curves are claimed in [LC14] to have "independently-verifiable provenance", as if they were not subject to any possible manipulation; and are claimed in [BBC+15] to be selected "without any hidden parameters, reliance on randomness or any other processes offering opportunities for manipulation of the resulting curves". What we analyze in this section is the extent to which Jerry can manipulate the resulting curves.

5.5.2 Choice of security level

Jerry may propose curves aiming for multiple security levels. To quote the Brainpool-curves RFC [LM10] "The level of security provided by symmetric ciphers and hash functions used in conjunction with the elliptic curve domain parameters specified in this RFC should roughly match or exceed the level provided by the domain parameters." Table 1 in that document justifies security levels of 80, 96, 112, 128, 160, 192, and 256 bits.

5.5.3 Choice of prime

There are several parts to choosing a prime once the security level is fixed.

Choice of prime size

For a fixed security level α it should take about 2^α operations to break the DLP. The definition of "operation" leaves some flexibility. The choices for the bitlength r of the prime are:

- Exactly 2α, see e.g., [BCLN15].
- Exactly $2\alpha-1$, see e.g., [BCLN15].
- Exactly $2\alpha-2$, see e.g., [BCLN15].
- Exactly $2\alpha+1$ to make up for the loss of $\sqrt{\pi/4}$ in the Pollard-rho complexity.
- Exactly $2\alpha+2$ to *really* make up for the loss of $\sqrt{\pi/4}$ in the Pollard-rho complexity.

  $\vdots$
- Exactly $2\alpha+\beta$ to make up for the loss through precomputations for multi-target attacks.
- Exactly $2\alpha-3$ to make arithmetic easier and because each elliptic-curve operation takes at least 3 bit operations.
- Exactly $2\alpha-4$ to make arithmetic easier and because each elliptic-curve operation takes at least 4 bit operations.

  $\vdots$

- Exactly $2\alpha - \gamma$ to make arithmetic easier and because each elliptic-curve operation takes at least $2^{\gamma/2}$ bit operations.

These statements provide generic justifications for 8 options (actually even more, but we take a power of 2 to simplify). In the next two steps we show how to select different primes for each of these requirements. If the resulting p has additional beneficial properties these generic arguments might not be necessary, but they might be required if a competing (and by some measure superior) proposal can be excluded on the basis of not following the same selection criterion. If Jerry wants to highlight such benefits in his prime choice he may point to fast reduction or fast multiplication in a particular redundant representation with optimal limb size.

Choice of prime shape

The choices for the prime shape are:

- A random prime. This might seem somewhat hard to justify outside the scope of the previous section because arithmetic in $\mathbb{F}_p$ becomes slower. However, members of the ECC Brainpool working group have published several helpful arguments [LMSS14]. The most useful one is that random primes mean that the blinding factor in randomizing scalars against differential side-channel attacks can be chosen smaller.

- A pseudo-Mersenne prime, i.e., a prime of the shape $2^u \pm c$. The most common choice is to take c to be the smallest integer for a given u which leads to a prime because this makes reduction modulo the prime faster. (To reduce modulo $2^u \pm c$, divide by 2^u and add $\mp c$ times the dividend to the remainder.) See, e.g., [BCLN15]. Once u is fixed there are two choices for the two signs.

- A Solinas prime, i.e., a prime of the form $2^u \pm 2^v \pm 1$ as chosen for the Suite B curves [NSA05]. Also for these primes speed of modular reduction is the common argument. The difference $u - v$ is commonly chosen to be a multiple of the word size. Jerry can easily argue for multiples of 32 and 64. We skip this option in our count because it is partially subsumed in the following one.

- A "Montgomery-friendly" prime, i.e., a prime of the form $2^{u-v}(2^v - c) \pm 1$. These curves speed up reductions if elements in $\mathbb{F}_p$ are represented in Montgomery representation, $u - v$ is a multiple of the word size and c is less than the word size. Common word sizes are 32 and 64, giving two choices here. We ignore the flexibility of the ± because that determines p modulo 4, which is considered separately.

There are of course infinitely many random primes; in order to keep the number of options reasonable we take 4 as an approximation of how many prime shapes can be easily justified, making this a total of 8 options.

Choice of prime congruence

Jerry can get an additional bit of freedom by choosing whether to require $p \equiv 1 \pmod 4$ or to require $p \equiv 3 \pmod 4$. A common justification for the latter is that computations of square roots are particularly fast which could be useful for compression of points, see, e.g., [Bra05, BCLN15]. (In fact one can also compute square roots efficiently for $p \equiv 1 \pmod 4$, in particular for $p \equiv 5 \pmod 8$.) To instead justify $p \equiv 1 \pmod 4$, Jerry can point to various benefits of having $\sqrt{-1}$ in the field: for example, twisted Edwards curves are fastest when $a = -1$, but completeness for $a = -1$ requires $p \equiv 1 \pmod 4$.

If Jerry chooses twisted Hessian curves he can justify restricting to $p \equiv 1 \pmod 3$ to obtain complete curve arithmetic.

5.5.4 Choice of ordering of field elements

The following curve shapes each have one free parameter. It is easy to justify choosing this parameter as the smallest parameter under some side conditions. Here smallest can be chosen to mean smallest in $\mathbb{N}$ or as the smallest power of some fixed generator g of $\mathbb{F}_p^*$. The second option is used in, e.g., a recent ANSSI curve-selection document [FPRE15, Section 2.6.2]: "we define … g as the smallest generator of the multiplicative group … We then iterate over … $b = g^n$ for $n = 1, \ldots,$ until a suitable curve is found." Each choice below can be filled with these two options.

5.5.5 Choice of curve shape and cofactor requirement

Jerry can justify the following curve shapes:

1. Weierstrass curves, the most general curve shape. The usual choice is $y^2 = x^3 - 3x + b$, leaving one parameter b free. For simplicity we do not discuss the possibility of choosing values other than -3.

2. Edwards curves, the speed leader in fixed-base scalar multiplication offering complete addition laws. The usual choices are $ax^2 + y^2 = 1 + dx^2y^2$, for $a \in \{\pm 1\}$, leaving one parameter d free. The group order of an Edwards curve is divisible by 4.

3. Montgomery curves, the speed leader for variable-base scalar multiplication and the simplest to implement correctly. The usual choices are $y^2 = x^3 + Ax^2 + x$, leaving one parameter A free. The group order of a Montgomery curve is divisible by 4.

4. Hessian curves, a cubic curve shape with complete addition laws (for twisted Hessian). The usual choices are $ax^3 + y^3 + 1 = dxy$, where a is a small noncube, leaving one parameter d free. The group order of a Hessian curve is divisible by 3, making twisted Hessian curves the curves with the smallest cofactor while having complete addition.

The following choices depend on the chosen curve shape, hence we consider them separately.

Weierstrass curves

Most standards expect the point format to be (x, y) on Weierstrass curves. Even when computations want to use the faster Edwards and Hessian formulas, Jerry can easily justify specifying the curve in Weierstrass form. This also ensures backwards compatibility with existing implementations that can only use the Weierstrass form.

The following are examples of justifiable choices for the cofactor h of the curve:

- Require cofactor exactly 1, as in Suite B and Brainpool.
- Require cofactor exactly 2, the minimum cofactor that allows the techniques of [BHKL13] to transmit curve points as uniform random binary strings for censorship circumvention.
- Require cofactor exactly 3, the minimum cofactor that allows Hessian arithmetic.
- Require cofactor exactly 4, the minimum cofactor that allows Edwards arithmetic.
- Require cofactor exactly 12, the minimum cofactor that allows both Hessian arithmetic and Edwards arithmetic.
- Take the first curve having cofactor below $2^{\alpha/8}$. This cofactor limit is standardized in [Cer10] and [NIS13]. (This cofactor will almost always be larger than 12.)
- Take the first curve having cofactor below $2^{\alpha/8}$ and a multiple of 3.
- Take the first curve having cofactor below $2^{\alpha/8}$ and a multiple of 4.
- Take the first curve having cofactor below $2^{\alpha/8}$ and a multiple of 12.
- Replace “cofactor below $2^{\alpha/8}$” with the SafeCurves requirement of a largest prime factor above 2^{200}.

On average these choices produce slightly more than 8 options; the last few options sometimes coincide.

The curve is defined as $y^2 = x^3 - 3x + b$ where b is minimal under the chosen criterion. Changing from positive b to negative b changes from a curve to its twist if $p \equiv 3 \pmod 4$, and (as illustrated by additive transfers) this change does not necessarily preserve security. However, this option makes only a small difference in our final total, so for simplicity we skip it.

Hessian curves

A curve given in Hessian form (and chosen minimal there) can be required to have minimal cofactor, minimal cofactor while being compatible with Edwards form, cofactor smaller than $2^{\alpha/8}$, or largest prime factor larger than 2^u. This leads to 8 options considering positive and negative values of d. Of course other restrictions on the cofactor are possible.

Edwards curves

For Edwards curves we need to split up the consideration further:

Edwards curves with $p \equiv 3 \pmod 4$

Curves with $a = -1$ are attractive for speed but are not complete in this case. Nevertheless [BCLN15] argues for this option, so we have additionally the choice between aiming for a complete or an $a = -1$ curve.

A curve given in (twisted) Edwards form (and chosen minimal there) can be required to have minimal cofactor, minimal cofactor while being compatible with Hessian form, cofactor smaller than $2^{\alpha/8}$, or largest prime factor larger than 2^u (and the latter in combination with Hessian if desired). This leads to at least 8 choices considering completeness; for minimal cofactors [BCLN15] shows that minimal choices for positive and negative values of d are not independent. To stay on the safe side we count these as 8 options only.

Edwards curves with $p \equiv 1 \pmod 4$

The curves $x^2 + y^2 = 1 + dx^2y^2$ and $-x^2 + y^2 = 1 - dx^2y^2$ are isomorphic because -1 is a square, hence taking the smallest positive value for d finds the same curve as taking the smallest negative value for the other sign of a. Jerry can however insist or not insist on completeness.

Because $2p+2 \equiv 4 \pmod 8$ one of the curve and its twist will have order divisible by 8 while the other one has remainder 4 modulo 8. Jerry can require cofactor 4, as the minimal cofactor, or cofactor 8 if he chooses the twist with minimal cofactor as well and is concerned that protocols will only multiply by the cofactor of the curve rather than by that of the twist. The other options are the same as above. Again, to stay on the safe side, we count this as 8 options only.

Montgomery curves

There is a significant overlap between choosing the smallest Edwards curve and the smallest Montgomery curve. In order to ease counting and avoid overcounting we omit further Montgomery options.

Summary of curve choice

We have shown that Jerry can argue for $8+8+8=24$ options.

5.5.6 Choice of twist security

We make the worst-case assumption, as discussed in Section 5.1, that future standards will be required to include twist security. However, Jerry can play twist security to his advantage in changing the details of the twist-security requirements. Here are three obvious choices:

- Choose the cofactor of the twist as small as possible. Justification: This offers maximal protection.
- Choose the cofactor of the twist to be secure under the SEC recommendation, i.e., $h' < 2^{\alpha/8}$. Justification: This is considered secure enough for the main curve, so it is certainly enough for the twist.
- Choose the curve such that the curve passes the SafeCurves requirement of 2^{100} security against twist attacks. Justification: Attacks on the twist cannot use Pollard rho but need to do a brute-force search in the subgroups. The SafeCurves requirement captures the actual hardness of the attack.

Jerry can easily justify changes to the bound of 2^{100} by pointing to a higher security level or reducing it because the computations in the brute-force part are more expensive. We do not use this flexibility in the counting.

5.5.7 Choice of global vs. local curves

Jerry can take the first prime (satisfying some criteria), and then, for that prime, take the first curve coefficients (satisfying some criteria). Alternatively, Jerry can take the first possible curve coefficients, and then, for those curve coefficients, take the first prime. These two options are practically guaranteed to produce different curves. For example, in the Weierstrass case, Jerry can take the curve $y^2=x^3-3x+1$, and then search for the first prime p so that this curve over $\mathbb{F}_p$ satisfies the requirements on cofactor and twist security. If Jerry instead takes $y^2 = x^3 - 3x + g$ as in [FPRE15, Section 2.6.2], p must also meet the requirement that g be primitive in $\mathbb{F}_p$.

In mathematical terminology, the second option specifies a curve over a "global field" such as the rationals $\mathbb{Q}$, and then reduces the curve modulo suitable primes. This approach is particularly attractive when presented as a family of curves, all derived from the same global curve.

5.5.8 More choices

Brainpool [Bra05] requires that the number of points on the curve is less than p but also presents an argument for the opposite choice:

> To avoid overruns in implementations we require that $\#E(GF(p)) < p$. In connection with digital signature schemes some authors propose to use $q > p$ for security reasons, but the attacks described e.g. in [BRS] appear infeasible in a thoroughly designed PKI.

So Jerry can choose to insist on $p < |E(\mathbb{F}_p)|$ or on $p > |E(\mathbb{F}_p)|$.

5.5.9 Overall count

We have shown that Jerry can easily argue for 4 (security level) ·8 (prime size) ·8 (prime shape) ·2 (congruence) ·2 (definition of first) ·24 (curve choice) ·3 (twist conditions) ·2 (global/local) ·2 ($p \lessgtr |E(\mathbb{F}_p)|$) $= 294912$ choices.

5.6 Manipulating security criteria

A recent trend is to introduce top performance as a selection requirement. This means that Alice and Bob accept only *the fastest curve*, as demonstrated by benchmarks across a range of platforms. The most widely known example of this approach is Bernstein's Curve25519, the curve $y^2 = x^3 + 486662x^2 + x$ modulo the particularly efficient prime $2^{255} - 19$, which over the past ten years has set speed records for conservative ECC on space-constrained ASICs, Xilinx FPGAs, 8-bit AVR microcontrollers, 16-bit MSP430X microcontrollers, 32-bit ARM Cortex-M0 microcontrollers, larger 32-bit ARM smartphone processors, the Cell processor, NVIDIA and AMD GPUs, and several generations of 32-bit and 64-bit Intel and AMD CPUs, using implementations from 23 authors. See [Ber06a, GT07, CS09, BDL+12, BS12, LM13, MC14, SG14, Cho15a, DHH+15, HSSW15].

The annoyance for Jerry in this scenario is that, in order to make a case for his curve, he needs to present implementions of the curve arithmetic on a variety of devices, showing that his curve is fastest across platforms. Jerry could try to falsify his speed reports, but it is increasingly common for the public to demand verifiable benchmarks using open-source software.

Jerry can hope that some platforms will favor one curve while other platforms will favor another curve; Jerry can then use arguments for a "reasonable" weighting of platforms as a mechanism to choose one curve or the other. However, it seems difficult to outperform Curve25519 even on *one* platform. The prime $2^{255} - 19$ is particularly efficient, as is the Montgomery curve shape $y^2 = x^3 + 486662x^2 + x$. The same curve is also expressible as a complete Edwards curve, allowing fast additions without the overhead of checking for exceptional cases. Twist security removes the overhead of checking for invalid inputs. Replacing 486662 with a larger curve coefficient produces identical performance on many platforms but loses a measurable amount of performance on some platforms, violating the "top performance" requirement.

In Section 5.5, Jerry was free to, e.g., claim that $p \equiv 3 \pmod 4$ provides "simple square-root computations" and thus replace $2^{255} - 19$ with $2^{255} - 765$; claim that "compatibility" requires curves of the form $y^2 = x^3 - 3x + b$; etc. The new difficulty in this section is that Jerry is facing "top performance" advocates who reject $2^{255} - 765$

as not providing top performance; who reject $y^2 = x^3 - 3x + b$ as not providing top performance; etc.

Fortunately, Jerry still has some flexibility in defining what security requirements to take into account. Taking "the fastest curve" actually means taking the fastest curve *meeting specified security requirements*, and the list of security requirements is a target of manipulation.

Most importantly, Jerry can argue for any size of r. However, if there is a faster curve with a larger r satisfying the same criteria, then Jerry's curve will be rejected. Furthermore, if Jerry's curve is only marginally larger than a significantly faster curve, then Jerry will have to argue that a tiny difference in security levels (e.g., one curve broken with 0.7× or 0.5× as much effort as another) is meaningful, or else the top-performance advocates will insist on the significantly faster curve.

The choice of prime has the biggest impact on speed and closely rules the size of r. For pseudo-Mersenne primes larger than 2^{224} the only possibly competitive ones are: $2^{226}-5, 2^{228}+3, 2^{233}-3, 2^{235}-15, 2^{243}-9, 2^{251}-9, 2^{255}-19, 2^{263}+9, 2^{266}-3, 2^{273}+5, 2^{285}-9, 2^{291}-19, 2^{292}+13, 2^{295}+9, 2^{301}+27, 2^{308}+27, 2^{310}+15, 2^{317}+9, 2^{319}+9, 2^{320}+27, 2^{321}-9, 2^{327}+9, 2^{328}+15, 2^{336}-3, 2^{341}+5, 2^{342}+15, 2^{359}+23, 2^{369}-25, 2^{379}-19, 2^{390}+3, 2^{395}+29, 2^{401}-31, 2^{409}+29, 2^{414}-17, 2^{438}+25, 2^{444}-17, 2^{452}-3, 2^{456}+21, 2^{465}+29, 2^{468}-17, 2^{488}-17, 2^{489}-21, 2^{492}+21, 2^{495}-31, 2^{508}+15, 2^{521}-1$. Preliminary implementation work shows that the Mersenne prime $2^{521}-1$ has such efficient reduction that it outperforms, e.g., the prime $2^{512}-569$ from [BCLN15]; perhaps it even outperforms primes below 2^{500}. We would expect implementation work to also show, e.g., that $2^{319}+9$ is significantly faster than $2^{320}+27$, and Jerry will have a hard time arguing for $2^{320}+27$ on security grounds. Considering other classes of primes, such as Montgomery-friendly primes, might identify as many as 100 possibly competitive primes, but it is safe to estimate that fewer than 80 of these primes will satisfy the top-performance fanatics, and further implementation work is likely to reduce the list even more. Note that in this section, unlike other sections, we take a count that is optimistic for Jerry.

Beyond the choice of prime, Jerry can use different choices of security criteria. However, most of the flexibility in Section 5.5 consists of speed claims, compatibility claims, etc., few of which can be sold as security criteria. Jerry *can* use the different twist conditions, the choice whether $p < |E(\mathbb{F}_p)|$ or $p > |E(\mathbb{F}_p)|$, and possibly two choices of cofactor requirements. Jerry can also choose to require completeness as a security criterion, but this does not affect curve choice in this section: the complete formulas for twisted Hessian and Edwards curves are *faster* than the incomplete formulas for Weierstrass curves. The bottom line is that multiplying fewer than 80 primes by 12 choices of security criteria produces fewer than 960 curves. The main difficulty in pinpointing an exact number is carrying out detailed implementation work for each prime; we leave this to future work.

6

Curve41417: Karatsuba Revisited

This chapter introduces new ECDH software for a standard ARM Cortex-A8 CPU. This software is faster than the fastest ECDH option (`secp160r1`) in the latest version of OpenSSL (version 1.0.2-beta1, released 24 February 2014).

This performance bar was already reached in one previous paper, “NEON crypto” by Bernstein and Schwabe [BS12], implementing Bernstein’s Curve25519 [Ber06a] elliptic curve. The difference is that the software introduced in this chapter now reaches the same performance bar at a much higher security level, implementing a very strong new “Curve41417” elliptic curve introduced informally by Bernstein and Lange in [BL13b, page 12] and introduced formally in “Curve41417: Karatsuba revisited” paper [BCL14], matching this chapter.

The performance does not indicate that Curve41417 is as fast as Curve25519. It sugests that Curve41417 is fast enough for applications and provides a much higher security level than Curve25519. This chapter addresses the scalability challenges that appear at higher security levels.

Hyperelliptic-curve DH has also recently reached this performance bar for the Cortex-A8: the HECDH implementation in [BCLS14] is even faster than Curve25519. However, the performance benefits of hyperelliptic curves are specific to DH, as admitted in [BCLS14], while elliptic curves are easily adapted to other important applications such as signatures. More importantly, the 128-bit hyperelliptic curve used in [BCLS14] came from a massive computation by Gaudry and Schost in [GS12a], using more than 1000000 hours of CPU time. Finding a similar curve at a higher security level would be extraordinarily difficult.

Credits. The content of this chapter is based on the paper *“Curve41417: Karatsuba revisited”* [BCL14] which is a joint work with Daniel J. Bernstein and Tanja Lange. Note that “we” in this chapter is used to refer to the aforementioned authors.

Organization of this chapter. Section 6.1 reviews the performance of prime-field ECC software and emphasizes the need for curves at a high security level. Section 6.2 presents the design of Curve41417. Section 6.3 describes elliptic-curve arithmetic used in the introduced software. Section 6.4 explains how to perform fast multiplication using Karatsuba's method. Section 6.5 summarizes techniques behind the utilization of vector units.

6.1 Introduction

The Cortex-A8 contains a large integer-multiplication unit that multiplies 32-bit words to produce 64-bit results. Of course, there are CPUs with even larger multipliers, and CPUs (and FPGAs) with smaller multipliers, but 32-bit multipliers have been a popular choice for many years and seem likely to remain in widespread use in embedded systems for many years to come. We focus on the Cortex-A8 for the same reasons as [BS12] and [BCLS14, Section 5].

6.1.1 Karatsuba's method in prime-field ECC software

The conventional approach in ECC software is to take advantage of 32-bit multipliers by splitting, e.g., 160-bit prime-field elements into 5 words to be multiplied, or 256-bit prime-field elements into 8 words to be multiplied. Karatsuba's method [KO63, Theorem 2] is well known to be useful for binary fields, and is occasionally also considered for prime-field ECC software, but is practically always dismissed as having too much overhead: one Karatsuba level saves 25% of the integer-multiply instructions, but this is outweighed by the cost of many extra additions. (Of course, this comparison is biased by the availability of a large multiplier and relatively little area spent on adders, but this is how mass-market CPUs have always been designed.)

It should be obvious that scaling to larger and larger input sizes will eventually reach a cutoff where one Karatsuba level is useful: the overhead is linear in the size, while the 25% savings is quadratic in the size. But the conventional wisdom is that this cutoff is far beyond ECC sizes, so one would not expect that aiming for high-security ECC would reach this cutoff. The heavily optimized GMP multiprecision library [Gra], which includes automated searches for optimal cutoffs, does not switch over from schoolbook multiplication to one Karatsuba level on the Cortex-A8 until it reaches 832-bit inputs. A recent RSA performance analysis by Bos, Montgomery, Shumow, and Zaverucha [BMSZ13] avoided all use of Karatsuba's method even for 1024-bit modular multiplication.

We use *two* Karatsuba levels. There is a synergy between two design choices here: (1) we use Karatsuba's method; (2) we use a radix smaller than the CPU word size.

The conventional choice for b-bit CPUs is to use radix 2^b, minimizing the number of words that need to be multiplied. See, for example, the recent DH software from [Ham12], [LS12b], [BCHL13a], [FHLS14], and [CHS14]. However, a corner of the DH literature uses a smaller radix, with the goal of delaying carries, the same way that hardware multipliers typically use carry-save adders. See, for example, [BS12] and [BCLS14].

This corner of the literature does not seem to have exploited the fact that Karatsuba's method benefits heavily from a smaller radix. With radix 2^b, the extra additions in Karatsuba's method are add-with-carry chains. With a smaller radix, the extra additions in Karatsuba's method are independent additions without carries. Even on CPUs where add-with-carry is as cheap as add, having independent operations creates tremendous extra flexibility in register allocation, instruction scheduling, and vectorization.

Conversely, a smaller radix benefits from Karatsuba's method, especially as the security level increases. Reducing a radix from, e.g., 2^{32} to 2^{26} means that instead of w words one now needs $(32/26)w$ words and thus, without Karatsuba's method, $(32/26)^2w^2 \approx 1.5w^2$ multiplications instead of w^2 multiplications; this means that the benefits of eliminating carries have to be compared to the loss of $0.5w^2$ multiplications. Karatsuba's method moves the number of multiplications down to a smaller scale, improving this tradeoff.

6.1.2 Choice of prime and choice of curve

The standard NIST elliptic curves [NIS00] use primes p designed to allow easy computation of $x \bmod p$ in radix 2^{32}. For example, the popular NIST P-256 curve uses $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$, and at a higher security level NIST P-384 uses $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$.

We leave a gap between our radix and 2^{32} to speed up multiplications, as explained above, but this makes computation of $x \bmod p$ quite painful for the NIST primes p. The NIST primes are also suitable for a much smaller radix, namely 2^{16}, but that radix would make our multiplications considerably slower.

The Curve25519 prime, $2^{255} - 19$, is much less sensitive to the choice of radix, but our objective is to provide as much security as possible subject to a specified performance requirement, and in particular more security than Curve25519. An initial performance estimate indicated that a carefully designed curve of 384 bits or larger could meet our performance requirement, but we found very few 384-bit curves in the literature, and all of them have obvious performance problems.

We therefore designed a prime and curve from scratch. This also allowed us to take advantage of state-of-the-art curve shapes, while meeting stringent security criteria that are flunked by the NIST curves. See Section 6.2.

The prime we ended up with, namely $p = 2^{414} - 17$, has many attractive features from a performance perspective. It is extremely close to a power of 2. The difference 17 has just two bits set, allowing $2^{414}x \bmod p$ to be computed as $16x + x$ with a single shift-and-add operation. The exponent 414 is divisible by $9, 18, 23, 46$ and the exponent 416 (for $4p$) is divisible by $8, 13, 16, 26, 32, 52$, allowing easy choices of integer radix suitable with low overhead for practically any size of multiplier. A field element is easily transmitted in 32-bit words with under 1% wasted space ($13 \cdot 32 = 416$), while still allowing two extra bits for extensions, such as a bit typically used in encoding a compressed curve point.

Table 6.1: Prime-field ECC timings from `openssl speed ecdh` on two Cortex-A8 devices.

Curve	i.MX515 op/s	Cycles	Sitara op/s	Cycles
`secp160r1`	379.2	≈ 2.1 million	468.1	≈ 2.1 million
`nistp192`	274.3	≈ 2.9 million	350.9	≈ 2.8 million
`nistp224`	200.4	≈ 4.0 million	257.6	≈ 3.9 million
`nistp256`	201.1	≈ 4.0 million	258.7	≈ 3.9 million
`nistp384`	60.1	≈13.3 million	75.9	≈13.2 million
`nistp521`	26.9	≈29.7 million	33.7	≈29.7 million

Warning: `openssl speed ecdh` reports "operations per second" as the reciprocal of average seconds per operation without indicating standard deviation or other stability metrics. "i.MX515 op/s" column is reported by OpenSSL 1.0.2-beta1 compiled with gcc 4.4.3 on a Hercules eCafe laptop (`h4mx515e`) with a 2009 Freescale i.MX515 CPU running at 800MHz. "Sitara op/s" column is reported by OpenSSL 1.0.2-beta1 compiled with gcc 4.7.3 on a BeagleBone Black development board (`bblack`) with a 2012 TI Sitara XAM3359AZCZ100 CPU running at 1000MHz. The "cycles" columns translate "op/s" into CPU cycles per operation.

For our software we decided to use a slightly harder, but slightly more efficient, non-integer radix, namely $2^{414/16} = 2^{25.875}$. We split 414-bit prime-field elements into 16 words, use one Karatsuba level to reduce 16-word multiplication to three 8-word multiplications, and use another Karatsuba level to reduce each 8-word multiplication to three 4-word multiplications. See Section 6.4 for details of our multiplication strategy, and Section 6.5 for the extra challenges created by vectorization.

6.1.3 Expected scalability

As a measurement of the conventional scaling of ECC performance to higher security levels, we compiled OpenSSL 1.0.2-beta1 on two Cortex-A8 devices and ran `openssl speed ecdh`. The prime-field results are shown in Table 6.1. We also checked that (as expected) the prime-field results were faster than the binary-field results at each security level; the binary-field results are not shown here. The fastest OpenSSL cycle count was 2.1 million cycles for `secp160r1` (2^{80} security).

The following back-of-the-envelope calculation suggests that moving from 256 bits to 384 bits increases costs by a factor of $1.5^3 = 3.375$: each multiplication input is longer by a factor of 1.5, increasing the multiplication cost by a factor of 1.5^2; and the scalar in ECDH is 1.5× longer. The actual ratios between `nistp256` and `nistp384` in the table are close to this. The slowdown factor for `nistp521` is about 7.5, noticeably better than $(521/256)^3 \approx 8.4$, presumably because of the simpler prime shape used in P-521. The speedup factor for smaller curves is considerably worse than this calculation would suggest; presumably this reflects OpenSSL function-call overheads that become troublesome for smaller integers.

We also checked the eBACS [BLa] benchmarking site for Cortex-A8 results. The only results faster than 2.1 million cycles were 0.46 million cycles (i.MX515) and 0.50 million cycles (Sitara) for the Curve25519 (2^{125} security) implementation from [BS12]. The paper [BCLS14] reports better speeds, just 0.27 million Cortex-A8 cycles for HECDH; but scaling HECDH to higher security levels is very difficult, as mentioned earlier. The paper [BCHL13b] reports 0.77 million Cortex-A8 cycles for 2^{103} security using a different type of curve, evidently not competitive.

The same type of back-of-the-envelope calculation suggests that moving from Curve25519 up to Curve41417 would cost a factor of 4.3, increasing 0.50 million Sitara Cortex-A8 cycles to 2.15 million cycles. We do considerably better than this; see below.

6.1.4 Performance results

We tried our Curve41417 software on the same two Cortex-A8 machines shown in Table 6.1. On the FreeScale i.MX515 (`h4mx515e`) our software uses just 1648409 cycles (median; quartiles 1646391 and 1662710). On the TI Sitara (`bblack`) our software uses just 1775804 cycles (median; quartiles 1774878 and 1782850). These figures are for a complete scalar-multiplication operation, including unpacking a point from network format, precomputation, main computation, final inversion, and converting the result back to network format. We emphasize that our curve choice has security level above 2^{200}, and that the software is free of data-dependent branches and data-dependent array indices.

These speeds are, despite their very high security level, considerably faster than the 2.1 million cycles for the fastest ECDH in OpenSSL. These speeds are also considerably faster than the 2.15 million cycles predicted above by extrapolation from Curve25519. This chapter explains the design and implementation choices that led to this performance.

As a followup to our initial Curve41417 announcement, Hamburg announced a similar, slightly larger, curve “Ed448-Goldilocks”. Hamburg’s most recent performance report [Ham14] says 3.6 million Cortex-A9 cycles for Ed448-Goldilocks, compared to 4.4 million Cortex-A9 cycles for the OpenSSL 1.0.1 implementation of NIST P-256. There are several reasons that it is difficult to extrapolate from these results: the Cortex-A9 is not the same as the Cortex-A8; Hamburg’s Ed448-Goldilocks software is not vectorized; and OpenSSL 1.0.1 was missing some NIST P-256 speedups that appear in the most recent version of OpenSSL.

6.1.5 Is high security useful?

Most papers today consider security levels between 2^{80} and 2^{128}. The adequacy of 2^{80} is frequently a subject of dispute. There is general consensus that well-funded attackers and botnets can already perform 2^{80} operations; most HTTPS web sites have now switched from RSA-1024 (2^{80} security) to RSA-2048 (2^{112} security) or 256-bit ECC (2^{128} security). On the other hand, there are also many papers continuing to study 2^{80} security and stating that 2^{80} is ample protection for low-value targets.

The adequacy of 2^{128} is rarely a subject of dispute. It is easy to see that 2^{128} is far beyond any computation feasible today. Choosing 2^{128} is so common in the current literature that papers studying a 2^{128} security level rarely bother to justify this choice.

One can therefore reasonably ask whether there is any reason to go beyond 2^{128} security, and in particular whether we are accomplishing anything useful by going beyond 2^{200} security. We give five answers to this question, in what we consider to be increasing order of importance.

First, cryptographic primitives need time to be reviewed before they are standardized and deployed in embedded systems, so designers of cryptographic primitives today should be considering embedded systems designed at least 10 years from now. Some of those systems will have a lifetime of 30 years, and at the end of that lifetime could still be encrypting data that — even if recorded by an attacker — should remain confidential for another 30 years, i.e., 70 years from now.

Today's mass-market GPUs perform approximately 2^{58} floating-point operations per year per watt. If computation becomes a factor of 10 more efficient each decade then mass-market chips in 70 years will perform approximately 2^{81} floating-point operations per year per watt. Carrying out a 1-year computation on the same scale as 2^{128} floating-point operations will thus require just 2^{47} watts. For comparison, the Earth's surface receives 2^{56} watts from the Sun.

We do not mean to suggest that typical cryptographic applications should worry about such large attacks. But we also see value in designing cryptographic systems that are not broken by such large attacks.

Second, even though many researchers have studied the security of ECC and expressed confidence in the security of prime-field ECC, there is still the possibility of an algorithmic breakthrough that considerably reduces the amount of computation required to break ECC. By moving to a much higher security level we are providing a security margin against unexpected attack improvements.

For comparison, since 2013 the security of small-characteristic multiplicative-group discrete logarithms has dropped dramatically. A very recent paper [GKZ14] reports 2^{59} security for a system previously thought to provide 2^{128} security. We do not mean to suggest that this is a threat to prime-field ECC (there are clear barriers between small characteristic and prime fields, and more importantly between multiplicative groups and ECC) but it does illustrate the general principle that attack cost can suddenly drop.

Third, sometimes cryptographic protocols are not as secure as the underlying cryptographic primitives. Often there is a security proof putting a bound on the gap, but usually the security proofs are not "tight". In particular, many ECC protocols are not guaranteed to provide 2^{128} security using 256-bit curves, even assuming the standard security conjectures for ECDLP on those curves. Achieving a 2^{128} guarantee requires taking larger curves. We thank an anonymous referee for pointing out this argument.

Fourth, we suggest that the right question is not how efficiently a particular security level can be achieved, but rather how much security can be provided subject to the performance requirements set by the users. Of course, a typical cryptographic system also relies on block ciphers, hash functions, etc., and if those are breakable in time 2^{128} then the attacker does not have to bother breaking a 414-bit elliptic curve;

but AES-256 costs only 40% more than AES-128, and standard hashes also provide high-security options. It is natural for research into high-performance ECC to similarly provide high-security options for users who can afford those options.

The normal reason for users to reject high-security options is not that the users dislike high security, but rather that the high-security options are too slow. If a user rejects OpenSSL's `nistp384` in favor of `secp160r1`, probably the reason is that the user's performance budget does not allow 13.3 million cycles, while it does allow 2.1 million cycles. Unless there are severe bandwidth constraints, the user will be happier with Curve41417, which provides much higher security within the same performance budget.

Fifth, there are at least some users already demanding cryptography beyond a 2^{128} security level. For example, NSA's Suite B allows NIST P-256 for Secret information, but for Top Secret information it requires NIST P-384, SHA-384, and AES-256. This project began when Silent Circle requested a non-NIST curve to replace NIST P-384; we realized that we could design a curve that simultaneously provided better performance and better security. Silent Circle is now using Curve41417 by default.

6.2 Design of Curve41417

The IEEE standard P1363 [IEE00] and the Brainpool recommendations [Bra05] specify procedures to generate secure elliptic curves. Research has identified several other properties a secure curve should satisfy. A recent collection of these properties is provided by Bernstein and Lange in the "SafeCurves" web page [BL14b].

6.2.1 Standard security criteria

There are several standard criteria on which all methods cited on [BL14b] agree. The elliptic curve E must be defined over a prime field $\mathbb{F}_p$ or a binary field $\mathbb{F}_{2^p}$, for p a prime; its group order must be divisible by a large prime r; this prime must not match the field characteristic; and the embedding degree must be large. Over a prime field $\mathbb{F}_p$ the embedding degree is defined as the smallest positive integer k so that r divides $p^k - 1$. Brainpool requires $k \geq (r-1)/100$, and P1363 imposes a weaker requirement.

For efficiency and security reasons we focus on prime fields, a recommendation supported by Brainpool and the more recent NIST/NSA documents [NSA05].

6.2.2 Additional security criteria

SafeCurves imposes several further requirements to avoid "conflicts between simplicity, efficiency, and security". Specifically, it requires curves to support "simple, fast, complete, constant-time" algorithms for single-coordinate single-scalar multiplication and for multi-scalar multiplication. Montgomery curves [Mon87] meet the single-coordinate single-scalar requirement; Edwards curves [Edw07], when chosen to be complete [BL07], meet all of the requirements. Compared to Weierstrass curves, these curves make it easier to implement the curve arithmetic correctly: scalar multiplication is a very regular operation without exceptional cases that require special handling and that could reveal information about the scalar. The NIST curves do not meet these requirements.

SafeCurves also requires curves to be twist-secure. Twist-security means that the order of the twist, namely $2p+2-|E(\mathbb{F}_p)|$, is nearly prime. This criterion eliminates security problems caused by single-coordinate single-scalar multiplication algorithms that do not take extra effort to validate their inputs: for example, when a curve is given in Montgomery form and only the x-coordinate is transmitted and used, twist-security eliminates the need to check that the incoming x-coordinate is on the curve.

The NIST curve constants are not explained: in the SafeCurves terminology, the NIST curve choice is not "rigid". This has led to speculation about how the NIST curves were designed and about whether the NSA has implemented a back door in the choice of the curves. Our curve is "fully rigid": the prime and all curve constants are fully explained here.

6.2.3 Choice of prime field

Our target in designing the new curve was to generate an elliptic curve at a security level larger than 2^{192} that meets the SafeCurves requirements and that supports efficient implementations. To this aim we start with finding a prime for which field elements can be efficiently represented and modulo which reductions are efficient. Prime numbers of the form $2^u - c$ for $12 \cdot 32 < u < 13 \cdot 32$ and $0 < c < 32$ are rare: the only possibilities are $2^{389}-21$, $2^{401}-31$, $2^{413}-21$, and $2^{414}-17$. We selected $p = 2^{414}-17$ because 17 is the smallest c in this list; it also has the lowest Hamming weight. Section 6.4 explains how we perform arithmetic in $\mathbb{F}_p$; this prime also leaves enough space in the limbs when we represent field elements as 16 words of 32 bits that carries between the limbs and reductions modulo p can be delayed for long enough to be useful in the curve arithmetic. The next larger candidate prime would be $2^{444}-17$ which does not have this feature; our p is already very large for our security needs.

6.2.4 Choice of curve shape

For efficient and secure arithmetic in Diffie–Hellman key exchange and digital signature applications we insist on a curve in Edwards form. Note that each curve in Edwards form is birationally equivalent to one in Montgomery form, so there is no need to choose one over the other. The coefficient d in the Edwards curve $x^2 + y^2 = 1 + dx^2y^2$ appears as a factor in the addition formulas, so choosing d to be small in absolute value is good for efficiency. For security we choose a complete Edwards curve (d is not a square in $\mathbb{F}_p$) and insist on the same level of twist-security as Curve25519—the cofactors of the curve and its twist are in $\{4, 8\}$.

6.2.5 A safe curve

Curve41417 (named after the prime field) is defined as

$$x^2 + y^2 = 1 + 3617x^2y^2 \text{ over } \mathbb{F}_p, \qquad p = 2^{414} - 17.$$

Its order is $8r$, where

$$r = 2^{411} - 33364140863755142520810177694098385178984727200411208589594 75.$$

The order of the twist is also 8 times a prime. The value $d = 3617$ is the smallest integer in absolute value meeting the above security requirements.

6.3 ECC arithmetic

Our featured application is static Diffie–Hellman in which a user Alice computes her private key a and her public key $P_A = aP$ once and then publishes P_A. If Alice wants to communicate with user Bob she looks up Bob's public key P_B and computes aP_B. This means that the computations use variable base points. The computations involve the long-term secret key a and need to be protected against side-channel attacks by attackers sitting on the same device or having a connection to it. This means in particular that the scalar multiplication should run in constant time, independent of the scalar a, and that there should be no data-dependent branches or table lookups involving a.

We use a windowing method with fixed window width for constant-time single-scalar multiplication on Curve41417 in Edwards form. Our analysis also allows good estimates of, e.g., the cost of signature verification using Curve41417. Another option for single-scalar multiplication is the Montgomery ladder for the Montgomery form of Curve41417; this is not quite as fast as the Edwards form but has the advantage of fitting the computation into less SRAM.

6.3.1 Coordinate systems

The fastest doubling formulas in the EFD [BLb] for curves in Edwards form are in projective coordinates X, Y, Z with $x = X/Z, y = Y/Z$ for $Z \neq 0$. These take $3\mathbf{M} + 4\mathbf{S}$ per doubling where $\mathbf{M}$ and $\mathbf{S}$ denote field multiplication and field squaring respectively. (See the following subsection for the formulas used in our implementation.)

The fastest addition formulas are in extended coordinates X, Y, Z, T with $x = X/Z, y = Y/Z$, and $xy = T/Z$ for $Z \neq 0$. These take $9\mathbf{M} + 1\mathbf{m}_d$. Here $\mathbf{m}_d$ is a multiplication by curve constant d; for us $d = 3617$, which is significantly smaller than p, so this multiplication $\mathbf{m}_d$ is cheaper than general multiplications $\mathbf{M}$. (The curve $-x^2 + y^2 = 1 - dx^2y^2$ allows faster additions, saving $1\mathbf{M}$ in each addition. If -1 were a square in $\mathbb{F}_p$ then we could apply an isomorphism to that curve. However, -1 is not a square in $\mathbb{F}_p$, so that curve is not complete.)

Achieving the best performance requires combining these two coordinate systems: computing the extra T coordinate for a doubling output that will be used for addition, and skipping the extra T coordinate for an addition output that will be used only for doubling. This suggestion was made in [HWCD08], the paper introducing extended coordinates.

6.3.2 Point arithmetic formulas

This subsection presents the formulas that we use for doubling and addition of curve points. Most of these formulas are taken from the EFD [BLb]. To simplify the cost statements we count only field multiplications and squarings, not additions and subtractions.

Formulas for doubling

We use three different formulas for point doubling. The slowest formulas are the following:

Input:	X_1, Y_1, Z_1
Output:	X_3, Y_3, Z_3, T_3
Cost:	$4\mathbf{M} + 4\mathbf{S}$

$$\begin{array}{llll} A = X_1^2; & B = Y_1^2; & C = 2Z_1^2; & E = (X_1+Y_1)^2 - A - B; \\ G = A + B; & H = A - B; & F = G - C; & \\ X_3 = E \cdot F; & Y_3 = G \cdot H; & Z_3 = F \cdot G; & T_3 = E \cdot H. \end{array}$$

We use these formulas once in each five-doubling window, specifically for the last doubling before point addition. Each of the other four doublings costs just $3\mathbf{M} + 4\mathbf{S}$: we save $1\mathbf{M}$ by skipping the computation of T_3.

For the first doubling in the precomputation we use the following faster formulas.

Input: X_1, Y_1, T_1 where $Z_1 = 1$
Output: X_3, Y_3, Z_3, T_3
Cost: $3\mathbf{M} + 3\mathbf{S}$

$$\begin{array}{llll} A = X_1^2; & B = Y_1^2; & E = 2T_1; & \\ G = A + B; & H = A - B; & F = G - 2; & \\ X_3 = E \cdot F; & Y_3 = G \cdot H; & Z_3 = G^2 - 2G & T_3 = E \cdot H. \end{array}$$

Formulas for addition

All additions in the precomputation use the following formulas. These formulas save $1\mathbf{M}$ using $Z_2 = 1$.

Input: $X_1, Y_1, Z_1, T_1, X_2, Y_2, dT_2$ where $Z_2 = 1$
Output: X_3, Y_3, Z_3, T_3
Cost: 8**M**

$$A = X_1 \cdot X_2; \quad B = Y_1 \cdot Y_2; \quad C = dT_1 \cdot T_2; \quad E = (X_1 + Y_1) \cdot (X_2 + Y_2) - A - B;$$
$$F = Z_1 - C; \quad G = Z_1 + C; \quad H = B - A;$$
$$X_3 = E \cdot F; \quad Y_3 = G \cdot H; \quad Z_3 = F \cdot G; \quad T_3 = E \cdot H.$$

All additions in the main computation use the following formulas. These formulas save 1**M** by skipping the computation of T_3; the next operation is doubling, which does not use T.

Input: $X_1, Y_1, Z_1, T_1, X_2, Y_2, Z_2, dT_2$
Output: X_3, Y_3, Z_3
Cost: 8**M**

$$A = X_1 \cdot X_2; \quad B = Y_1 \cdot Y_2; \quad C = dT_1 \cdot T_2; \quad E = (X_1 + Y_1) \cdot (X_2 + Y_2) - A - B;$$
$$D = Z_1 \cdot Z_2; \quad F = D - C; \quad G = D + C; \quad H = B - A;$$
$$X_3 = E \cdot F; \quad Y_3 = G \cdot H; \quad Z_3 = F \cdot G.$$

6.3.3 Scalar multiplication

Constant-time sliding windows are difficult so we use fixed windows. We analyzed operation counts for signed fixed windows for window widths $w = 4$, $w = 5$, and $w = 6$, and concluded that $w = 5$ is optimal. We therefore precompute $0P_B = (0,1), P_B, 2P_B, \ldots, 16P_B$ and store the results in a table. We do table lookups in constant time using the same technique as in, e.g., [BDL+11]: we load the entire table into registers and perform the selection via arithmetic.

Precomputation is done as follows. We double P_B to obtain $2P_B$; add P_B to obtain $3P_B$; double $2P_B$ to obtain $4P_B$; add P_B to obtain $5P_B$; double $3P_B$ to obtain $6P_B$; add P_B to obtain $7P_B$; and so on through $16P_B$. We also multiply each resulting T coordinate by $d = 3617$, eliminating the multiplications by d in the main computation.

In total 8 doublings, 7 additions, and 16 multiplications by d are required. Note that these doublings are followed by additions and thus need one extra **M** for the T coordinate in the transition to extended coordinates. Note also that we have to compute T for P_B which costs 1**M**. For the first doubling, (X, Y, Z, T) is $(x, y, 1, xy)$. We save 1**S** by not having to compute Z^2 since $Z = 1$; we save another 1**S** by not having to compute $(x + y)^2$ but using the equality $(x + y)^2 - x^2 - y^2 = 2xy = 2T$; and we use $Z = 1$ again for an **S**−**M** tradeoff. The overall cost for the first doubling is 3**M**+3**S** while for the rest it is 4**M**+4**S**. Note that all additions in the precomputation are adding P_B which has $Z = 1$. We thus use mixed addition which saves 1**M**. This results in the total cost for precomputation of $1\mathbf{M} + (3\mathbf{M} + 3\mathbf{S}) + 7(4\mathbf{M} + 4\mathbf{S}) + 7(8\mathbf{M}) + 16\mathbf{m}_d = 88\mathbf{M} + 31\mathbf{S} + 16\mathbf{m}_d$.

The main computation uses a fixed pattern of five doublings followed by one addition. Four regular doublings in a block of five take 3**M** + 4**S** each. The fifth doubling in a block requires 1 more **M** to calculate T for the following addition. On the other hand, addition does not need to compute T since the following doubling is in projective coordinates. Furthermore, dT was precomputed for each T in the table. Therefore the addition takes only 8**M**. In total the five doublings and one addition take only $4(3\mathbf{M}+4\mathbf{S})+(4\mathbf{M}+4\mathbf{S})+(8\mathbf{M})=24\mathbf{M}+20\mathbf{S}$.

Note that, since the Edwards addition law is complete, no special handling is required for the neutral element $0P_B$. An addition when the coefficient of the scalar happens to be 0 is handled the same way as any other addition.

A scalar between 0 and $2^{414}-1$ uses 82 signed windows of width 5, after an initial selection from $0P_B, 1P_B, \dots, 16P_B$. The total cost for scalar multiplication including precomputation is $(88\mathbf{M}+31\mathbf{S}+16\mathbf{m}_d)+82(24\mathbf{M}+20\mathbf{S})=2056\mathbf{M}+1671\mathbf{S}+16\mathbf{m}_d$, plus 1 inversion and 2**M** to convert to $X/Z, Y/Z$ for output.

6.4 Karatsuba multiplication

Karatsuba, Toom, and the FFT are polynomial-multiplication methods that are asymptotically faster than schoolbook multiplication. However, for small input sizes the speedups are outweighed by the expense of more additions and subtractions, which in turn require more carries. These effects are particularly noticeable for polynomials of low degree — or equivalently for integers occupying just a few words. In software implementations of cryptography we rarely find integers large enough to justify use of FFT or Toom, and even Karatsuba's method is commonly only used in implementations of RSA and not ECC.

In this section we explain how to reduce the cost of carries by working with multiple levels of redundancy in the representation and thereby delaying carries. We also introduce "reduced refined Karatsuba", a new variant of the "refined Karatsuba" method; this variant eliminates some additions by merging Karatsuba multiplication with a subsequent modular reduction.

6.4.1 Redundant number representation

We decompose an integer f modulo $2^{414}-17$ into 16 integer pieces in radix $2^{414/16}=2^{25.875}$, i.e., we write f as $f_0+2^{26}f_1+2^{52}f_2+2^{78}f_3+2^{104}f_4+2^{130}f_5+2^{156}f_6+2^{182}f_7+2^{207}f_8+2^{233}f_9+2^{259}f_{10}+2^{285}f_{11}+2^{311}f_{12}+2^{337}f_{13}+2^{363}f_{14}+2^{389}f_{15}$. With this decomposition, each limb $f_0, f_1, \dots, f_{14}, f_{15}$ is small enough to fit into a 32-bit integer and to still have space to delay carries occurring when adding these pieces. The results of the 32-bit-by-32-bit multiplications fit into 64-bit words, and we can add thousands of them together before causing an overflow.

Note that f_7 is multiplied by 2^{207}, not 2^{208}. Having f_7 and f_{15} contain 25 bits makes $f_0, \dots, f_7$ symmetric to $f_8, \dots, f_{15}$, aiding vectorization. We considered using fewer limbs, but the advantage of saving multiplications is outweighed by the disadvantages of (1) extra carries and (2) extra vectorization overhead.

6.4.2 Two-level Karatsuba: decomposition strategy

As mentioned in Section 6.1, we use 2 Karatsuba levels. This fits nicely into the 128-bit Cortex-A8 vector units, and uses less arithmetic than 3 or 1 (or 0) Karatsuba levels.

We start with what Bernstein in [Ber09a] calls the "refined Karatsuba identity"

$$(F_0 + t^n F_1)(G_0 + t^n G_1) = (1 - t^n)(F_0 G_0 - t^n F_1 G_1) + t^n (F_0 + F_1)(G_0 + G_1).$$

This uses fewer additions than the original Karatsuba identity from [KO63].

For the *first* level of Karatsuba, we split one 16-limb integer f into two 8-limb integers F_0 and F_1 with $f = F_0 + 2^{207} F_1$ as:

$$\begin{aligned} F_0 &= f_0 + 2^{26} f_1 + 2^{52} f_2 + 2^{78} f_3 + 2^{104} f_4 + 2^{130} f_5 + 2^{156} f_6 + 2^{182} f_7 ; \\ F_1 &= f_8 + 2^{26} f_9 + 2^{52} f_{10} + 2^{78} f_{11} + 2^{104} f_{12} + 2^{130} f_{13} + 2^{156} f_{14} + 2^{182} f_{15}. \end{aligned}$$

We also decompose another integer g similarly to f. Then, we have

$$f g = (1 - 2^{207})(F_0 G_0 - 2^{207} F_1 G_1) + 2^{207} (F_0 + F_1)(G_0 + G_1).$$

For the *second* level of Karatsuba, we further split the 8 limbs of F_0 (and those of F_1) into two 4-limb integers F_{00}, F_{01} (and F_{10}, F_{11}) with $F_0 = F_{00} + 2^{104} F_{01}$ (and $F_1 = F_{10} + 2^{104} F_{11}$) as:

$$\begin{aligned} F_{00} &= f_0 + 2^{26} f_1 + 2^{52} f_2 + 2^{78} f_3 ; & F_{01} &= f_4 + 2^{26} f_5 + 2^{52} f_6 + 2^{78} f_7; \\ F_{10} &= f_8 + 2^{26} f_9 + 2^{52} f_{10} + 2^{78} f_{11}; & F_{11} &= f_{12} + 2^{26} f_{13} + 2^{52} f_{14} + 2^{78} f_{15}. \end{aligned}$$

We similarly split G_0 and G_1 to obtain G_{00}, G_{01}, G_{10}, and G_{11}. Then

$$\begin{aligned} F_0 G_0 &= (1 - 2^{104})(F_{00} G_{00} - 2^{104} F_{01} G_{01}) + 2^{104} (F_{00} + F_{01})(G_{00} + G_{01}); \\ F_1 G_1 &= (1 - 2^{104})(F_{10} G_{10} - 2^{104} F_{11} G_{11}) + 2^{104} (F_{10} + F_{11})(G_{10} + G_{11}). \end{aligned}$$

To compute $(F_0 + F_1)(G_0 + G_1)$ we first compute $F_0 + F_1$ and $G_0 + G_1$ without carries and then apply the same type of decomposition. For example, we split $F_0 + F_1$ into two 4-limb integers, namely $F_{00} + F_{10}$ and $F_{01} + F_{11}$.

6.4.3 Lowest-level multiplication

On the lowest level we need to multiply two 4-limb integers; we do this by schoolbook multiplication. For $F_{00} G_{00}$ this works as follows:

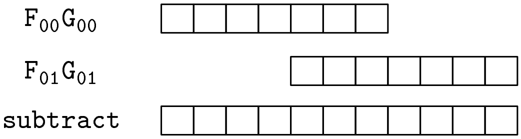
$$\begin{aligned} h_0 &= f_0 g_0, & & \\ h_1 &= f_0 g_1 + f_1 g_0, & h_4 &= f_1 g_3 + f_2 g_2 + f_3 g_1, \\ h_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0, & h_5 &= f_2 g_3 + f_3 g_2, \\ h_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0, & h_6 &= f_3 g_3. \end{aligned}$$

We store each input limb f_i and g_i in a word of 32 bits and use the processor's multiplication and addition units to compute each h_i. This takes 16 32-bit-by-32-bit multiplications and 9 64-bit additions. Each of the initial limbs has at most 26 bits and each of the h_i fits into 64 bits. The values h_4, h_5, and h_6 belong to the powers $2^{104}, 2^{130}$, and 2^{156}, i.e., they are implicitly multiplied by 2^{104}.

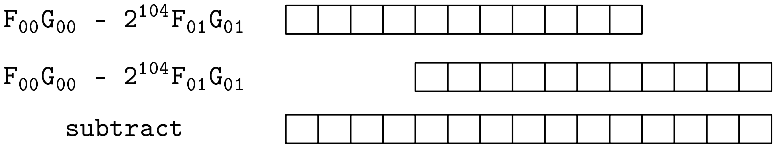
6.4.4 Middle-level recombination

After computing the three lowest-level products $F_{00}G_{00}, F_{01}G_{01}$ and $(F_{00}+F_{01})(G_{00}+G_{01})$, we obtain F_0G_0 as follows.

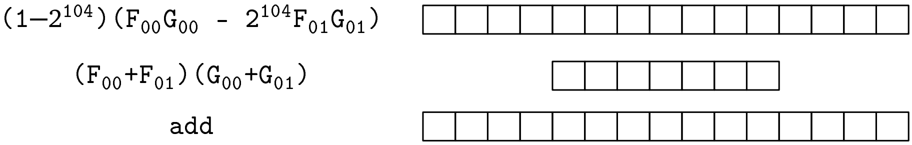
Step 1.1: Compute $F_{00}G_{00} - 2^{104}F_{01}G_{01}$. We merge $F_{01}G_{01}$ to $F_{00}G_{00}$ at the 2^{104} boundary using 3 subtractions of 64-bit words. In other words, we align the 5th limb of $F_{00}G_{00}$ with the 1st limb of $F_{01}G_{01}$ as shown in the following diagram. The result is thus 11 limbs long. The top limbs are not actually subtracted from 0; they are tracked as being implicitly negated.



Step 1.2: Compute $(1-2^{104})(F_{00}G_{00} - 2^{104}F_{01}G_{01})$. This is equivalent to merging $F_{00}G_{00} - 2^{104}F_{01}G_{01}$ to itself at the 2^{104} boundary. We conduct this merge similarly to Step 1.1: we align the 5th limb of $F_{00}G_{00} - 2^{104}F_{01}G_{01}$ with the 1st limb and subtract. The following diagram depicts this step. This merge requires 7 subtractions of 64-bit words, and the result is 15 limbs long.



Step 1.3: Compute F_0G_0. We finish this level of computation by adding $2^{104}(F_{00}+F_{01})(G_{00}+G_{01})$ to $(1-2^{104})(F_{00}G_{00} - 2^{104}F_{01}G_{01})$. This is done by merging the former to the latter at the 2^{104} boundary, i.e., the 5th limb of $(1-2^{104})(F_{00}G_{00} - 2^{104}F_{01}G_{01})$ is aligned with the 1st limb of $(F_{00}+F_{01})(G_{00}+G_{01})$ as shown in the following diagram. Note that this merge requires 7 additions of 64-bit words, and the result remains 15 limbs long.



When combining the results we need to pay attention to the 9th through 15th limbs. Those limbs are implicitly multiplied by 2^{207}. However, during the above computation they appear naturally as multiples of 2^{208} instead of 2^{207}. We therefore shift those seven limbs by one bit.

To summarize, the computation of the product F_0G_0 consists of

- 2×4 32-bit additions for $F_{00} + F_{01}$ and $G_{00} + G_{01}$;
- 3×16 32-bit-by-32-bit-producing-64-bit multiplications for $F_{00}G_{00}$, $F_{01}G_{01}$, and $(F_{00} + F_{01})(G_{00} + G_{01})$;
- 3×9 64-bit additions for computing the h_i;
- 1×3 64-bit subtractions for computing Step 1.1;
- 1×7 64-bit subtractions for computing Step 1.2;
- 1×7 64-bit additions for computing Step 1.3;
- 1×7 64-bit shifts for handling 2^{207} and 2^{208}.

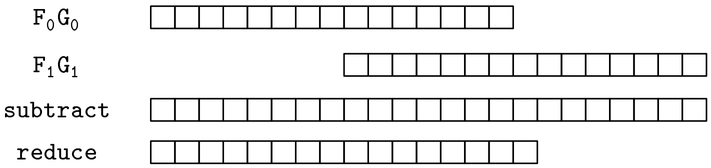
The total is 8 32-bit additions (counting subtractions as additions), 48 32-bit-by-32-bit multiplications, 44 64-bit additions, and 7 64-bit shifts.

We compute products F_1G_1 and $(F_0 + F_1)(G_0 + G_1)$ in the same way as F_0G_0. The total cost for these three products and the computation of $F_0 + F_1$ and $G_0 + G_1$ is 40 32-bit additions, 144 32-bit-by-32-bit multiplications, 132 64-bit additions, and 21 64-bit shifts.

6.4.5 Top-level recombination and reduction

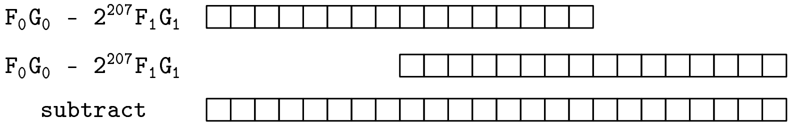
After computing F_0G_0 etc., we compute $fg = (1-2^{207})(F_0G_0 - 2^{207}F_1G_1) + 2^{207}(F_0 + F_1)(G_0 + G_1)$ as follows. This top-level recombination is immediately followed by a reduction, and we save some additions by interleaving the reduction into the refined-Karatsuba computation, a technique that we call "reduced refined Karatsuba". What is important here is that we reduce $F_0G_0 - 2^{207}F_1G_1$ before multiplying by $1 - 2^{207}$.

Step 2.1: Compute $F_0G_0 - 2^{207}F_1G_1$. This is similar to Step 1.1 but includes an extra reduction. The merge of F_1G_1 to F_0G_0 is at the 2^{207} boundary and uses 7 subtractions of 64-bit words; the 9th limb of F_0G_0 is aligned with the 1st limb of F_1G_1. The intermediate result is 23 limbs long. Then we reduce modulo $2^{414} - 17$: we multiply the 17th through 23rd limbs by 17 (using shifts and additions) and add to the 1st through 7th limbs. This requires another 7 shifts and 14 additions of 64-bit words. The result is thus only 16 limbs long, as indicated in the following diagram.

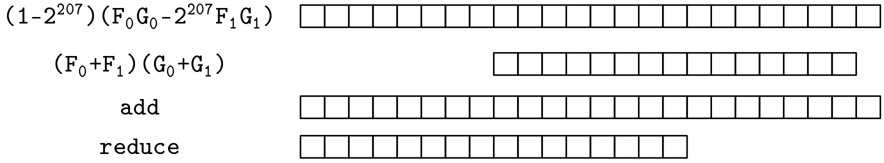


Step 2.2: Compute $(1 - 2^{207})(F_0G_0 - 2^{207}F_1G_1)$. This is similar to Step 1.2. The earlier reduction in Step 2.1 means that Step 2.2 uses only 8 subtractions of 64-bit

words. The result is 24 limbs long as shown in the following diagram. We do *not* perform an extra reduction here: by keeping this long result of 24 limbs, we save 8 shifts and 16 additions.



Step 2.3: Compute fg. We finish by adding 15-limb $2^{207}(F_0+F_1)(G_0+G_1)$ to 24-limb $(1-2^{207})(F_0G_0-2^{207}F_1G_1)$. This is done by merging the former to the latter at the 2^{207} boundary: i.e., the 9th limb of $(1-2^{207})(F_0G_0-2^{207}F_1G_1)$ is aligned with the 1st limb of $(F_0+F_1)(G_0+G_1)$. This merge requires 15 additions of 64-bit words and results in 24 limbs. We do another reduction similar to Step 2.1 to bring the result back to 16 limbs; this requires another 8 shifts and 16 additions of 64-bit words. The following diagram illustrates this step.



To summarize, computing fg from F_0G_0, F_1G_1 and $(F_0+F_1)(G_0+G_1)$ uses

- 7 64-bit subtractions for computing Step 2.1;
- 7 64-bit shift instructions for reduction in Step 2.1;
- 14 64-bit additions for reduction in Step 2.1;
- 8 64-bit subtractions for computing Step 2.2;
- 15 64-bit additions for computing Step 2.3;
- 8 64-bit shift instructions for reduction in Step 2.3;
- 16 64-bit additions for reduction in Step 2.3.

This sums up to 60 64-bit additions and 15 64-bit shift instructions. Therefore, the total cost for computing fg is 40 32-bit additions, 144 32-bit-by-32-bit multiplications, $132+60=192$ 64-bit additions, and $21+15=36$ 64-bit shifts.

6.4.6 Principles behind reduced refined Karatsuba

Our elimination of some additions can be viewed as following the general strategy of reducing *inputs* to a multiplication rather than *outputs* of a multiplication. Specifically, we reduce $F_0G_0-2^{207}F_1G_1$ before multiplying it by $1-2^{207}$; we do not reduce the product until after adding it to $(F_0+F_1)(G_0+G_1)$; if fg were being added to other products then we would similarly delay the reduction until after the addition. What is new here is seeing the multiplication by $1-t^n$ inside refined Karatsuba as a useful target of the general strategy, despite the sparsity of $1-t^n$.

6.5 Vectorization

The “NEON” vector unit in each Cortex-A8 core can compute a vector of two 64-bit products ac and bd in just 2 cycles given 32-bit inputs a, b, c, d. It can compute a vector of two 64-bit sums or four 32-bit sums in just 1 cycle. The latencies of these operations are actually higher, up to 7 cycles, but throughput is improved by pipelining. Taking advantage of this computational power requires that at every moment there are 2 or 4 identical computations to perform, and on top of this enough independent computations to hide latencies.

6.5.1 Karatsuba vectorization

Most of the computations in Section 6.4 are suitable for vectorization. For example, $F_{01}G_{01}$ takes $f_4, f_5, f_6, f_7, g_4, g_5, g_6, g_7$ as input; $F_{10}G_{10}$ takes $f_8, f_9, f_{10}, f_{11}, g_8, g_9, g_{10}$, g_{11}. There are no dependencies between these two identical sets of multiplications. Similar comments apply to $F_{00}G_{00}$ and $F_{11}G_{11}$; $(F_{00} + F_{10})(G_{00} + G_{10})$ and $(F_{01} + F_{11})(G_{01} + G_{11})$; and $(F_{00} + F_{01})(G_{00} + G_{01})$ and $(F_{10} + F_{11})(G_{10} + G_{11})$. The remaining multiplication consists of 16 32-bit products, which we partition into 8 vectorized products at the cost of some shuffling. Similarly, we vectorize between combining F_0G_0 and combining F_1G_1, and at the cost of some shuffling we vectorize within the computation of $(F_0 + F_1)(G_0 + G_1)$. NEON also supports a multiply-accumulate instruction, allowing us to eliminate many addition instructions.

6.5.2 Carry vectorization

At the end of the Karatsuba computation, reduction modulo p produces a product of the form $\sum_{i=0}^{7} m_i 2^{26i} + 2^{207} \sum_{i=0}^{7} m_{i+8} 2^{26i}$. We then use a sequence of carries to bring each limb down to 26 (or in some cases 25) bits. We vectorize between a carry $m_0 \to m_1$ and a carry $m_8 \to m_9$, between a carry $m_1 \to m_2$ and a carry $m_9 \to m_{10}$, etc.

Each carry has very high latency, so we perform four carry chains in parallel. Specifically, we vectorize between a carry $m_0 \to m_1$ and a carry $m_8 \to m_9$, and in parallel vectorize between a carry $m_4 \to m_5$ and a carry $m_{12} \to m_{13}$; we then vectorize between a carry $m_1 \to m_2$ and a carry $m_9 \to m_{10}$, and in parallel vectorize between a carry $m_5 \to m_6$ and a carry $m_{13} \to m_{14}$; and so on. This hides almost all latency.

6.5.3 Performance

See Section 6.1.4 for our Cortex-A8 performance results.

7

Kummer Surface Diffie–Hellman

The Eurocrypt 2013 paper “Fast cryptography in genus 2” by Bos, Costello, Hisil, and Lauter [BCHL13a] reported 117000 cycles on Intel’s Ivy Bridge microarchitecture for high-security constant-time scalar multiplication on a genus-2 Kummer surface. The eBACS site for publicly verifiable benchmarks [BLa] confirms 119032 “cycles to compute a shared secret” (quartiles: 118904 and 119232) for the `kumfp127g` software from [BCHL13a] measured on a single core of `h9ivy`, a 2012 Intel Core i5-3210M running at 2.5GHz. The software is not much slower on Intel’s previous microarchitecture, Sandy Bridge: eBACS reports 122716 cycles (quartiles: 122576 and 122836) for `kumfp127g` on `h6sandy`, a 2011 Intel Core i3-2310M running at 2.1GHz. (The quartiles demonstrate that rounding to a multiple of 1000 cycles, as in [BCHL13a], loses statistically significant information; this chapter follows eBACS in reporting medians of exact cycle counts.)

The paper reported that this was a “new software speed record” (“breaking the 120k cycle barrier”) compared to “all previous genus 1 and genus 2 implementations” of high-security constant-time scalar multiplication. Obviously the genus-2 cycle counts shown above are better than the (unverified) claim of 137000 Sandy Bridge cycles by Longa and Sica in [LS12b] (Asiacrypt 2012) for constant-time elliptic-curve scalar multiplication; the (unverified) claim of 153000 Sandy Bridge cycles by Hamburg in [Ham12] for constant-time elliptic-curve scalar multiplication; the 182708 cycles reported by eBACS on `h9ivy` for `curve25519`, a constant-time implementation by Bernstein, Duif, Lange, Schwabe, and Yang [BDL^{+}11] (CHES 2011) of Bernstein’s Curve25519 elliptic curve [Ber06a]; and the 194036 cycles reported by eBACS on `h6sandy` for `curve25519`.

One might conclude from these figures that genus-2 hyperelliptic-curve cryptography (HECC) solidly outperforms elliptic-curve cryptography (ECC). However, two newer papers claim better speeds for ECC, and a closer look reveals a strong argument that HECC should have trouble competing with ECC.

The first paper, [OLARH13] by Oliveira, López, Aranha, and Rodríguez-Henríquez (CHES 2013 best-paper award), is the new speed leader in eBACS for *non-constant-time* scalar multiplication; the paper reports a new Sandy Bridge speed record of 69500 cycles. Much more interesting is that the paper claims 114800 Sandy Bridge cycles for *constant-time* scalar multiplication, beating [BCHL13a]. eBACS reports 119904 cycles, but this is still faster than [BCHL13a].

The second paper, [FHLS14] by Faz-Hernández, Longa, and Sánchez, initially claimed 92000 Ivy Bridge cycles or 96000 Sandy Bridge cycles for constant-time scalar multiplication; a July 2014 update of the paper claims 89000 Ivy Bridge cycles or 92000 Sandy Bridge cycles. These claims are not publicly verifiable, but if they are even close to correct then they are faster than [BCHL13a].

Both of these new papers, like [LS12b], rely heavily on curve endomorphisms to eliminate many doublings, as proposed by Gallant, Lambert, and Vanstone [GLV01] (Crypto 2001), patented by the same authors in [GLV06] and [GLV11], and expanded by Galbraith, Lin, and Scott [GLS09] (Eurocrypt 2009). Specifically, [OLARH13] uses a GLS curve over a binary field to eliminate 50% of the doublings, while also taking advantage of Intel's new `pclmulqdq` instruction to multiply binary polynomials; [FHLS14] uses a GLV+GLS curve over a prime field to eliminate 75% of the doublings.

One can also use the GLV and GLS ideas in genus 2, as explored by Bos, Costello, Hisil, and Lauter starting in [BCHL13a] and continuing in [BCHL13b] (CHES 2013). However, the best GLV/GLS speed reported in [BCHL13b], 92000 Ivy Bridge cycles, provides only 2^{105} security and is not constant time. This is less impressive than the 119032 cycles from [BCHL13a] for constant-time DH at a 2^{125} security level, and less impressive than the reports in [OLARH13] and [FHLS14].

The underlying problem for HECC is easy to explain. All known HECC addition formulas are considerably slower than the state-of-the-art ECC addition formulas at the same security level. Almost all of the HECC options explored in [BCHL13a] are bottlenecked by additions, so the resulting time was larger.

The one exception is that HECC provides an extremely fast *ladder* (see Section 7.2), built from extremely fast *differential* additions and doublings, considerably faster than the Montgomery ladder frequently used for ECC. This is why [BCHL13a] was able to set DH speed records.

Unfortunately, differential additions do not allow arbitrary addition chains. Differential additions are incompatible with standard techniques for removing most or all doublings from fixed-base-point single-scalar multiplication, and with standard techniques for removing many doublings from multi-scalar multiplication. As a consequence, differential additions are incompatible with the GLV+GLS approach mentioned above for removing many doublings from single-scalar multiplication. This is why the DH speeds from [BCHL13a] were quickly superseded by DH speeds using GLV+GLS. A recent paper [CHS14] (Eurocrypt 2014) by Costello, Hisil, and Smith shows feasibility of combining differential additions and use of endomorphisms but reports 145000 Ivy Bridge cycles for constant-time software, much slower than the papers mentioned above.

This chapter introduces high-security constant-time variable-base-point DH software for Cortex-A8, Sandy Bridge, Ivy Bridge and Haswell CPUs. This software computes scalar multiplication on genus-2 hyperelliptic curves represented using their Kummer surfaces. The new speed records are achieved by exploiting vectorization without using GLS/GLV techniques.

Credits. The content of this chapter is based on the full version of the paper *"Kummer strikes back: new DH speed records"* [BCLS14] which is a joint work with Daniel J. Bernstein, Tanja Lange and Peter Schwabe. Note that "we" in this chapter is used to refer to the aforementioned authors.

Organization of this chapter. Section 7.1 summarizes contributions of this chapter and compares performance results of high-security DH speeds. Section 7.2 reviews scalar multiplication on Kummer surfaces. Section 7.3 shows how to decompose and optimize field multiplications on Sandy Bridge. Section 7.4 explains the vectorization of permutations in the Hadamard transform. Section 7.5 presents optimization techniques for Cortex-A8. Section 7.6 describes implementations on Haswell. Section 7.7 mentions possibile speedups by using lattice techniques. Section 7.8 discusses fixed-base scalar multiplication.

7.1 Overview of contributions

We show that HECC has an important compensating advantage, and we exploit this advantage to achieve new DH speed records. The advantage is that we are able to heavily *vectorize* the HECC ladder.

CPUs are evolving towards larger and larger vector units. A low-cost low-power ARM Cortex-A8 CPU core contains a 128-bit vector unit that every two cycles can compute two vector additions, each producing four sums of 32-bit integers, or one vector multiply-add, producing two results of the form $ab + c$ where a, b are 32-bit integers and c is a 64-bit integer. Every cycle a Sandy Bridge CPU core can compute a 256-bit vector floating-point addition, producing four double-precision sums, and at the same time a 256-bit vector floating-point multiplication, producing four double-precision products. A new Intel Haswell CPU core can carry out two 256-bit vector multiply-add instructions every cycle. Intel has announced future support for 512-bit vectors ("AVX-512").

Vectorization has an obvious attraction for a chip manufacturer: the costs of decoding an instruction are amortized across many arithmetic operations. The challenge for the algorithm designer is to efficiently vectorize higher-level computations so that the available circuitry is performing useful work during these computations rather than sitting idle. What we show here is how to fit HECC with surprisingly small overhead into commonly available vector units. This poses several algorithmic challenges, notably to minimize the permutations required for the Hadamard transform (see Section 7.4). We claim broad applicability of our techniques to modern CPUs, and to illustrate this we analyze all three of the microarchitectures mentioned in the previous paragraph.

Beware that different microarchitectures often have quite different performance. A paper that advertises a “better” algorithmic idea by reporting new record cycle counts on a new microarchitecture, not considered in the previous literature, might actually be reporting an idea that *loses* performance on *all* microarchitectures. We instead emphasize HECC performance on the widely deployed Sandy Bridge microarchitecture, since Sandy Bridge was shared as a target by the recent ECC speed-record papers listed above. We have now set a new Sandy Bridge DH speed record, demonstrating the value of vectorized HECC. We have also set DH speed records for Ivy Bridge, Haswell, and Cortex-A8.

7.1.1 Constant time: importance and difficulty

Before stating our performance results we emphasize that our software is truly constant time: the time that we use to compute nP is the same for every 251-bit scalar n and every point P. We strictly follow the rules stated by Bernstein in [Ber06a] (PKC 2006): we avoid “all input-dependent branches, all input-dependent array indices, and other instructions with input-dependent timings”. The importance of these data-flow requirements should be clear from, e.g., the Tromer–Osvik–Shamir attack [TOS10] (J. Cryptology 2010) recovering disk-encryption keys from the Linux kernel via cache timings, the Brumley–Tuveri attack [BT11] (ESORICS 2011) recovering ECDSA keys from OpenSSL via branch timings, and the recent “Lucky Thirteen” AlFardan–Paterson attack [AP13] (S&P 2013) recovering HTTPS plaintext via decryption timings.

Unfortunately, many of the speed reports in the literature are for cryptographic software that does not meet the same requirements. Sometimes the software is clearly labeled as taking variable time (for example, the ECC speed records from [OLARH13] state this quite explicitly), so it is arguably the user’s fault for deploying the software in applications that handle secret data; but in other cases non-constant-time software is incorrectly advertised as “constant time”.

Consider, for example, the scalar-multiplication algorithm stated in [BCHL13a, Algorithm 7], which includes a conditional branch for each bit of the scalar n. The “Side-channel resistance” section of the paper states “The main branch, i.e. checking if the bit is set (or not), can be converted into straight-line code by masking (pointers to) the in- and output. Since no lookup tables are used, and all modern cache sizes are large enough to hold the intermediate values … the algorithm (and runtime) becomes independent of input almost for free.”

Unfortunately, the argument in [BCHL13a] regarding cache sizes is erroneous, and this pointer-swapping strategy does not actually produce constant-time software. An operating-system interrupt can occur at any moment (for example, triggered by a network packet), knocking some or all data out of the cache (presumably at addresses predictable to, or controllable by, an attacker—it is helpful for the attacker that, for cost reasons, cache associativity is limited); see also the “flush+reload” attack from [YF14] and other local-spy attacks cited in [YF14]. If P_0 is knocked out of cache and the algorithm accesses P_0 then it suffers a cache miss; if both P_0 and P_1 are subsequently knocked out of cache and the algorithm accesses P_1, P_0 then it suffers two more cache misses. If, on the other hand, P_0 is knocked out of cache and the algorithm accesses P_1 then it does not suffer a cache miss; if both P_0 and P_1 are subsequently knocked out of cache and the algorithm accesses P_0, P_1 then it suffers two more cache misses. The total number of cache misses distinguishes these two examples, revealing whether the algorithm accessed P_0, P_1, P_0 or P_1, P_0, P_1.

We checked the `kumfp127g` software from [BCHL13a], and found that it contained exactly the branch indicated in [BCHL13a, Algorithm 7]. This exposes the software not just to data-cache-timing attacks but also to instruction-cache-timing attacks, branch-timing attacks, etc.; for background see, e.g., [ABG10] (CHES 2010). Evidently "can be converted" was a statement regarding possibilities, not a statement regarding what was actually done in the benchmarked software.

This is not an isolated example. We checked the fastest "constant-time" software from [CHS14] and found that it contained key-dependent branches. Specifically, the secret key `sk` is passed to a function `mon_fp_smul_2e127m1e2_djb`, which calls `decompose` and then `getchain` to convert the secret key into a `scanned` array and then calls `ec_fp_mdbladdadd_2e127m1e2_asm` repeatedly using various secret bits from that array; `ec_fp_mdbladdadd_2e127m1e2_asm` uses "`jmpq *(%rsi)`" to branch to different instructions depending on those bits. This exposes the software to instruction-cache-timing attacks.

The correct response to timing attacks is to use constant-time arithmetic instructions to simulate data-dependent branches, data-dependent table indices, etc.; see, e.g., Section 7.4.4. It is essential for "constant-time" cryptographic software to go to this effort. The time required for this simulation is often highly algorithm-dependent, and must be included in speed reports so that users are not misled regarding the costs of security.

Of course, the security assessment above was aided by the availability of the source code from [BCHL13a] and [CHS14]. For comparison, the public has no easy way to check the "constant time" claims for the software in [FHLS14], so for users the only safe assumption is that the claims are not correct. If that software is deployed somewhere then an attacker can be expected to do the necessary reverse-engineering work to discover and exploit the timing variability.

Our comparisons below are limited to software that has been advertised in the literature to be constant-time. Some of this software is not actually constant-time, as illustrated by the analysis above, and would become slower if it were fixed.

The authors of [BCHL13a] and [CHS14] have now updated their software, with credit to us. Their new (slower) software is not yet integrated into eBACS; our comparison table below shows the older software.

7.1.2 Performance results

eBACS shows that on a single core of `h6sandy` our DH software ("`kummer`") uses just 88916 Sandy Bridge cycles (quartiles: 88868 and 89184). On a single core of `h9ivy` our software uses 88448 cycles (quartiles: 88424 and 88476).

On a single core of `titan0`, an Intel Xeon E3-1275 V3 (Haswell), our software uses 60556 cycles (quartiles: 60444 and 68628). See Section 7.1.3 for previous

Haswell results.

For Cortex-A8 there is a difference in L1-cache performance between Cortex-A8-"fast" CPUs and Cortex-A8-"slow" CPUs. On `h7beagle`, a TI Sitara AM3359 (Cortex-A8-slow), our software uses 305395 cycles (quartiles: 305380 and 305413). On `h4mx515e`, a Freescale i.MX515 (Cortex-A8-fast), our software uses 273349 cycles (quartiles: 273337 and 273387). See Section 7.1.3 for previous Cortex-A8 results.

These cycle counts are the time for constant-time variable-scalar variable-base-point single-scalar multiplication using SUPERCOP's `crypto_dh` API. Our inputs and outputs are canonical representations of points as 48-byte strings and scalars as 32-byte strings. Our timings include more than just scalar multiplication on an internal representation of field elements; they also include the costs of parsing strings, all other necessary setup, the costs of conversion to inverted-affine form $(x/y, x/z, x/t)$ in the notation of Section 7.2, the costs of converting lazily reduced field elements to unique representatives, and the costs of converting to strings.

7.1.3 Cycle-count comparison

Table 7.1 summarizes reported high-security DH speeds for Cortex-A8, Sandy Bridge, Ivy Bridge, and Haswell.

This table is limited to software that *claims* to be constant time, and that claims a security level close to 2^{128}. This is the reason that the table does not include, e.g., the 767000 Cortex-A8 cycles and 108000 Ivy Bridge cycles claimed in [BCHL13b] for constant-time scalar multiplication on a Kummer surface; the authors claim only 103 bits of security for that surface. This is also the reason that the table does not include, e.g., the 69500 Sandy Bridge cycles claimed in [OLARH13] for non-constant-time scalar multiplication.

The table does not attempt to report whether the listed cycle counts are from software that actually meets the above security requirements. In some cases inspection of the software has shown that the security requirements are violated; see Section 7.1.1. "Open" means that the software is reported to be open source, allowing third-party inspection.

Our speeds, on the same platform targeted in [BCHL13a], solidly beat the HECC speeds from [BCHL13a]. Our speeds also solidly beat the Cortex-A8, Sandy Bridge, and Ivy Bridge speeds from all available ECC software, including [BDL+11], [BS12], [CHS14], and [OLARH13]; solidly beat the speeds claimed in [Ham12] and [LS12b]; and are even faster than the July 2014 Sandy Bridge/Ivy Bridge DH record claimed in [FHLS14], namely 92000/89000 cycles using unpublished software for GLV+GLS ECC. For Haswell, despite Haswell's exceptionally fast binary-field multiplier, our speeds beat the 61712 cycles from [OLARH13] for a GLS curve over a binary field. We set our new speed records using an HECC ladder that is conceptually much simpler than GLV and GLS, avoiding all the complications of scalar-dependent precomputations, lattice size issues, multi-scalar addition chains, endomorphism-rho security analysis, Weil-descent security analysis, and patents.

Table 7.1: Reported high-security DH speeds for Cortex-A8, Sandy Bridge, Ivy Bridge, and Haswell.

Arch.	Cycles	Ladder	Open	g	Field	Source
A8-slow	497389	yes	yes	1	$2^{255}-19$	[BS12]
A8-slow	305395	yes	yes	2	$2^{127}-1$	this chapter
A8-fast	460200	yes	yes	1	$2^{255}-19$	[BS12]
A8-fast	273349	yes	yes	2	$2^{127}-1$	this chapter
Sandy	194036	yes	yes	1	$2^{255}-19$	[BDL+11]
Sandy	153000?	yes	no	1	$2^{252}-2^{232}-1$	[Ham12]
Sandy	137000?	no	no	1	$(2^{127}-5997)^2$	[LS12b]
Sandy	122716	yes	yes	2	$2^{127}-1$	[BCHL13a]
Sandy	119904	no	yes	1	2^{254}	[OLARH13]
Sandy	96000?	no	no	1	$(2^{127}-5997)^2$	[FHLS14]
Sandy	92000?	no	no	1	$(2^{127}-5997)^2$	[FHLS14]
Sandy	88916	yes	yes	2	$2^{127}-1$	this chapter
Ivy	182708	yes	yes	1	$2^{255}-19$	[BDL+11]
Ivy	145000?	yes	yes	1	$(2^{127}-1)^2$	[CHS14]
Ivy	119032	yes	yes	2	$2^{127}-1$	[BCHL13a]
Ivy	114036	no	yes	1	2^{254}	[OLARH13]
Ivy	92000?	no	no	1	$(2^{127}-5997)^2$	[FHLS14]
Ivy	89000?	no	no	1	$(2^{127}-5997)^2$	[FHLS14]
Ivy	88448	yes	yes	2	$2^{127}-1$	this chapter
Haswell	161648	yes	yes	1	$2^{255}-19$	[BDL+11]
Haswell	110740	yes	yes	2	$2^{127}-1$	[BCHL13a]
Haswell	61712	no	yes	1	2^{254}	[OLARH13]
Haswell	60556	yes	yes	2	$2^{127}-1$	this chapter

Note: Cycle counts from eBACS are for `curve25519`, `kumfp127g`, `gls254prot`, and our `kummer` on `h7beagle` (Cortex-A8-slow), `h4mx515e` (Cortex-A8-fast), `h6sandy` (Sandy Bridge), `h9ivy` (Ivy Bridge), and `titan0` (Haswell). Cycle counts not from SUPERCOP are marked "?". ECC has $g = 1$; genus-2 HECC has $g = 2$. See text for security requirements.

7.2 Fast scalar multiplication on the Kummer surface

This section reviews the smallest number of field operations known for genus-2 scalar multiplication. Sections 7.3 and 7.4 optimize the performance of those field operations using 4-way vector instructions.

Vectorization changes the interface between this section and subsequent sections. What we actually optimize is not individual field operations, but rather pairs of operations, pairs of pairs, etc., depending on the amount of vectorization available from the CPU. Our optimization also takes advantage of sequences of operations such as the output of a squaring being multiplied by a small constant. What matters in this section is therefore not merely the *number* of field multiplications, squarings, etc., but also the *pattern* of those operations.

7.2.1 Only 25 multiplications

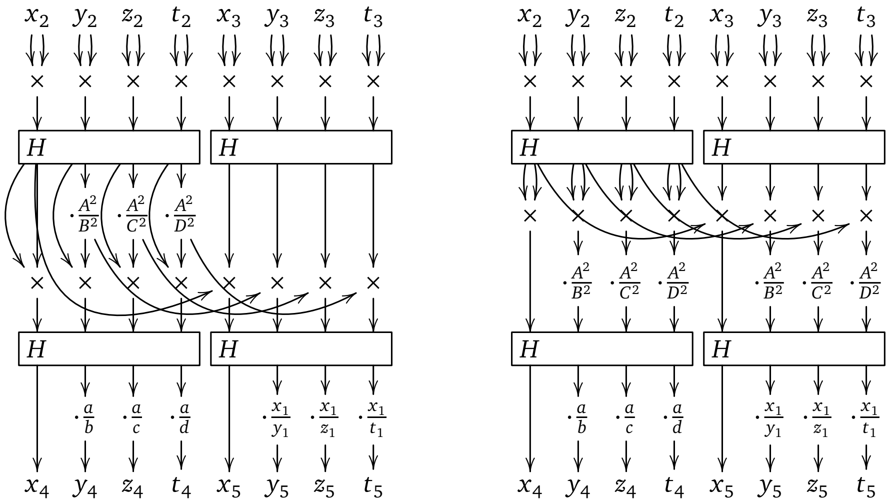
Around thirty years ago Chudnovsky and Chudnovsky wrote a classic paper [CC86] optimizing scalar multiplication inside the elliptic-curve method of integer factorization. At the end of the paper they also considered the performance of scalar multiplication on Jacobian varieties of genus-2 hyperelliptic curves. After mentioning various options they gave some details of one option, namely scalar multiplication on a Kummer surface.

A Kummer surface is related to the Jacobian of a genus-2 hyperelliptic curve in the same way that x-coordinates are related to a Weierstrass elliptic curve. There is a standard rational map X from the Jacobian to the Kummer surface; this map satisfies $X(P) = X(-P)$ for points P on the Jacobian and is almost everywhere exactly 2-to-1. Addition on the Jacobian does not induce an operation on the Kummer surface (unless the number of points on the surface is extremely small), but scalar multiplication $P \mapsto nP$ on the Jacobian induces scalar multiplication $X(P) \mapsto X(nP)$ on the Kummer surface. Not every genus-2 hyperelliptic curve can have its Jacobian mapped to the standard type of Kummer surface over the base field, but a noticeable fraction of curves can; see [Gau07].

Chudnovsky and Chudnovsky reported $14\mathbf{M}$ for doubling a Kummer-surface point, where $\mathbf{M}$ is the cost of field multiplication; and $23\mathbf{M}$ for "general addition", presumably differential addition, computing $X(Q+P)$ given $X(P), X(Q), X(Q-P)$. They presented their formulas for doubling, commenting on a "pretty symmetry" in the formulas and on the number of multiplications that were actually squarings. They did not present their formulas for differential addition.

Two decades later, in [Gau06], Gaudry reduced the total cost of differential addition and doubling, computing $X(2P), X(Q+P)$ given $X(P), X(Q), X(Q-P)$, to $25\mathbf{M}$, more precisely $16\mathbf{M} + 9\mathbf{S}$, more precisely $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$, where $\mathbf{S}$ is the cost of field squaring and $\mathbf{m}$ is the cost of multiplication by a curve constant. An ℓ-bit scalar-multiplication ladder therefore costs just $10\ell\mathbf{M} + 9\ell\mathbf{S} + 6\ell\mathbf{m}$.

Gaudry's formulas are shown in Figure 7.1(a). Each point on the Kummer surface is expressed projectively as four field elements $(x : y : z : t)$; one is free to replace $(x : y : z : t)$ with $(rx : ry : rz : rt)$ for any nonzero r. The "H" boxes are Hadamard



(a) $10\mathbf{M}+9\mathbf{S}+6\mathbf{m}$ ladder formulas.

(b) $7\mathbf{M}+12\mathbf{S}+9\mathbf{m}$ ladder formulas.

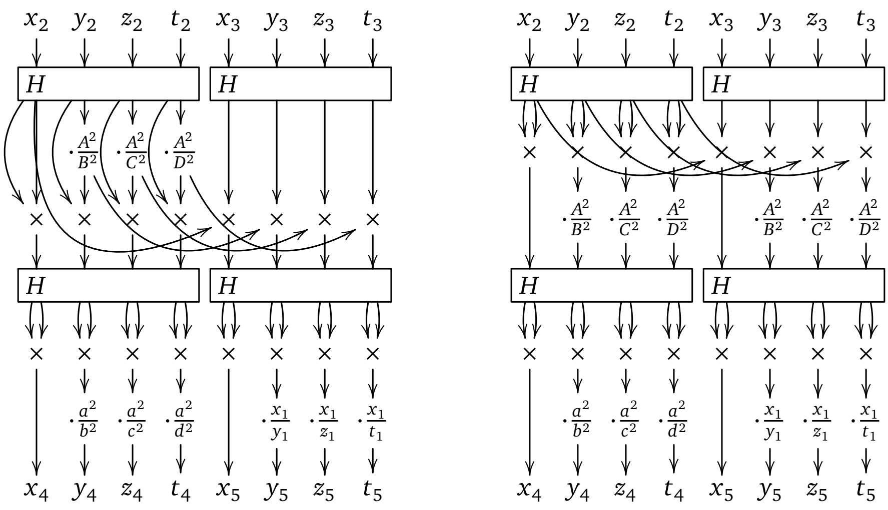
Figure 7.1: Ladder formulas for the Kummer surface. Inputs are $X(Q-P)=(x_1:y_1:z_1:t_1)$, $X(P)=(x_2:y_2:z_2:t_2)$, and $X(Q)=(x_3:y_3:z_3:t_3)$; outputs are $X(2P)=(x_4:y_4:z_4:t_4)$ and $X(P+Q)=(x_5:y_5:z_5:t_5)$. Formulas in (a) are from Gaudry [Gau06]; diagrams are copied from Bernstein [Ber06b].

transforms, each using 4 additions and 4 subtractions; see Section 7.4. The Kummer surface is parametrized by various constants $(a:b:c:d)$ and related constants $(A^2:B^2:C^2:D^2)=H(a^2:b^2:c^2:d^2)$. The doubling part of the diagram, from $(x_2:y_2:z_2:t_2)$ down to $(x_4:y_4:z_4:t_4)$, uses $3\mathbf{M}+5\mathbf{S}+6\mathbf{m}$, matching the $14\mathbf{M}$ reported by Chudnovsky and Chudnovsky; but the rest of the picture uses just $7\mathbf{M}+4\mathbf{S}$ extra, making remarkable reuse of the intermediate results of doubling. Figure 7.1(b) replaces $10\mathbf{M}+9\mathbf{S}+6\mathbf{m}$ with $7\mathbf{M}+12\mathbf{S}+9\mathbf{m}$, as suggested by Bernstein in [Ber06b]; this saves time if $\mathbf{m}$ is smaller than the difference $\mathbf{M}-\mathbf{S}$.

7.2.2 The original vs. the squared Kummer surface

Chudnovsky and Chudnovsky had used slightly different formulas for a slightly different surface, which we call the “squared Kummer surface”. Each point $(x:y:z:t)$ on the original Kummer surface corresponds to a point $(x^2:y^2:z^2:t^2)$ on the squared Kummer surface. Figure 7.2 presents the equivalent of Gaudry’s formulas for the squared Kummer surface, relabeling $(x^2:y^2:z^2:t^2)$ as $(x:y:z:t)$; the squarings at the top of Figure 7.1 have moved close to the bottom of Figure 7.2.

The number of field operations is the same either way, as stated in [Ber06b] with credit to André Augustyniak. However, the squared Kummer surface has a computational advantage over the original Kummer surface, as pointed out by Bernstein in [Ber06b]: constructing surfaces in which all of $a^2,b^2,c^2,d^2,A^2,B^2,C^2,D^2$ are small,



(a) $10\mathbf{M}+9\mathbf{S}+6\mathbf{m}$ ladder formulas. (b) $7\mathbf{M}+12\mathbf{S}+9\mathbf{m}$ ladder formulas.

Figure 7.2: Ladder formulas for the squared Kummer surface. Compare to Figure 7.1.

producing fast multiplications by constants in Figure 7.2, is easier than constructing surfaces in which all of $a, b, c, d, A^2, B^2, C^2, D^2$ are small, producing fast multiplications by constants in Figure 7.1.

7.2.3 Preliminary comparison to ECC

A Montgomery ladder step for ECC costs $5\mathbf{M}+4\mathbf{S}+1\mathbf{m}$, while a ladder step on the Kummer surface costs $10\mathbf{M}+9\mathbf{S}+6\mathbf{m}$ or $7\mathbf{M}+12\mathbf{S}+9\mathbf{m}$. Evidently ECC uses only about half as many operations. However, for security ECC needs primes around 256 bits (such as the convenient prime $2^{255}-19$), while the Kummer surface can use primes around 128 bits (such as the even more convenient prime $2^{127}-1$), and presumably this saves more than a factor of 2.

Several years ago, in [Ber06b], Bernstein introduced 32-bit Intel Pentium M software for generic Kummer surfaces (i.e., $\mathbf{m}=\mathbf{M}$) taking about 10% fewer cycles than his Curve25519 software, which at the time was the speed leader for ECC. Gaudry, Houtmann, and Thomé, as reported in [GL09, comparison table], introduced 64-bit software for Curve25519 and for a Kummer surface; the second option was slightly faster on AMD Opteron K8 but the first option was slightly faster on Intel Core 2. It is not at all clear that one can reasonably extrapolate to today's CPUs.

Bernstein's cost analysis concluded that HECC could be as much as 1.5× faster than ECC on a Pentium M (cost 1355 vs. cost 1998 in [Ber06b, page 31]), depending on the exact size of the constants $a^2, b^2, c^2, d^2, A^2, B^2, C^2, D^2$. This motivated a systematic search through small constants to find a Kummer surface providing high security and high twist security. But this was more easily said than done: genus-2 point counting was much more expensive than elliptic-curve point counting.

7.2.4 The Gaudry–Schost Kummer surface

Years later, after a 1000000-CPU-hour computation relying on various algorithmic improvements to genus-2 point counting, Gaudry and Schost announced in [GS12a] that they had found a secure Kummer surface $(a^2 : b^2 : c^2 : d^2) = (11 : -22 : -19 : -3)$ over $\mathbb{F}_p$ with $p = 2^{127} - 1$, with Jacobian order and twisted Jacobian order equal to

$$16 \cdot 1809251394333065553571917326471206521441306174399683558571672623546356726339,$$

$$16 \cdot 1809251394333065553414675955050290598923508843635941313077767297801179626051$$

respectively. This is exactly the surface that was used for the HECC speed records in [BCHL13a]. We obtain even better speeds for the same surface.

Note that, as mentioned by Bos, Costello, Hisil, and Lauter in [BCHL13a], the constants $(1 : a^2/b^2 : a^2/c^2 : a^2/d^2) = (1 : -1/2 : -11/19 : -11/3)$ in Figure 7.2 are projectively the same as $(-114 : 57 : 66 : 418)$. The common factor 11 between $a^2 = 11$ and $b^2 = -22$ helps keep these integers small. The constants $(1 : A^2/B^2 : A^2/C^2 : A^2/D^2) = (1 : -3 : -33/17 : -33/49)$ are projectively the same as $(-833 : 2499 : 1617 : 561)$.

7.3 Decomposing field multiplication

The only operations in Figures 7.1 and 7.2 are the H boxes, which we analyze in Section 7.4, and field multiplications, which we analyze in this section. Our goal here is to obtain the smallest possible number of CPU cycles for **M**, **S**, etc. modulo $p = 2^{127} - 1$.

This prime has been considered before, for example in [Ber04] and [Ber06b]. What is new here is fitting arithmetic modulo this prime, for the pattern of operations shown in Figure 7.2, into the vector abilities of modern CPUs. There are four obvious dimensions of vectorizability:

- Vectorizing across the "limbs" that represent a field element such as x_2. The most obvious problem with this approach is that, when f is multiplied by g, each limb of f needs to communicate with each limb of g and each limb of output. A less obvious problem is that the optimal number of limbs is CPU-dependent and is usually nonzero modulo the vector length. Each of these problems poses a challenge in organizing and reshuffling data inside multiplications.

- Vectorizing across the four field elements that represent a point. All of the multiplications in Figure 7.2 are visually organized into 4-way vectors, except that in some cases the vectors have been scaled to create a multiplication by 1. Even without vectorization, most of this scaling is undesirable for any surface with small a^2, b^2, c^2, d^2: e.g., for the Gaudry–Schost surface we replace

$(1 : a^2/b^2 : a^2/c^2 : a^2/d^2)$ with $(-114 : 57 : 66 : 418)$. The only remaining exception is the multiplication by 1 in $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$ where $X(Q-P) = (x_1 : y_1 : z_1 : t_1)$. Vectorizing across the four field elements means that this multiplication costs $1\mathbf{M}$, increasing the cost of a ladder step from $7\mathbf{M} + 12\mathbf{S} + 12\mathbf{m}$ to $8\mathbf{M} + 12\mathbf{S} + 12\mathbf{m}$.

- Vectorizing between doubling and differential addition. For example, in Figure 7.2(b), squarings are imperfectly paired with multiplications on the third line; multiplications by constants are perfectly paired with multiplications by the same constants on the fourth line; squarings are perfectly paired with squarings on the sixth line; and multiplications by constants are imperfectly paired with multiplications by inputs on the seventh line. There is some loss of efficiency in, e.g., pairing the squaring with the multiplication, since this prohibits using faster squaring methods.

- Vectorizing across a batch of independent scalar-multiplication inputs, in applications where a suitably sized batch is available. This is relatively straightforward but increases cache traffic, often to problematic levels. In this chapter we focus on the traditional case of a single input.

The second dimension of vectorizability is, as far as we know, a unique feature of HECC, and one that we heavily exploit for high performance.

For comparison, one can try to vectorize the well-known Montgomery ladder for ECC [Mon87] across the field elements that represent a point, but (1) this provides only two-way vectorization (x and z), not four-way vectorization; and (2) many of the resulting pairings are imperfect. The Montgomery ladder for Curve25519 was vectorized by Costigan and Schwabe in [CS09] for the Cell, and then by Bernstein and Schwabe in [BS12] for the Cortex-A8, but both of those vectorizations had substantially higher overhead than our new vectorization of the HECC ladder.

7.3.1 Sandy Bridge floating-point units

The only fast multiplier available on Intel's 32-bit platforms for many years, from the original Pentium twenty years ago through the Pentium M, was the floating-point multiplier. This was exploited by Bernstein for cryptographic computations in [Ber04], [Ber06a], etc.

The conventional wisdom is that this use of floating-point arithmetic was rendered obsolete by the advent of 64-bit platforms: in particular, Intel now provides a reasonably fast 64-bit integer multiplier. However, floating-point units have also become more powerful; evidently Intel sees many applications that rely critically upon fast floating-point arithmetic. We therefore revisit Bernstein's approach, with the added challenge of vectorization.

We next describe the relevant features of the Sandy Bridge; see [Fog14] for more information. Our optimization of HECC for the Sandy Bridge occupies the rest of Sections 7.3 and 7.4. The Ivy Bridge has the same features and should be expected to produce essentially identical performance for this type of code. The Haswell has important differences and is analyzed in Section 7.6; the Cortex-A8 is analyzed in Section 7.5.

Each Sandy Bridge core has several 256-bit vector units operating in parallel on vectors of 4 double-precision floating-point numbers:

- Port 0 handles one vector multiplication each cycle, with latency 5.
- Port 1 handles one vector addition each cycle, with latency 3.
- Port 5 handles one permutation instruction each cycle. The selection of permutation instructions is limited and is analyzed in detail in Section 7.4.
- Ports 2, 3, and 4 handle vector loads and stores, with latency 4 from L1 cache and latency 3 to L1 cache. Load/store throughput is limited in various ways, never exceeding one 256-bit load per cycle.

Recall that a double-precision floating-point number occupies 64 bits, including a sign bit, a power of 2, and a "mantissa". Every integer between -2^{53} and 2^{53} can be represented exactly as a double-precision floating-point number. More generally, every real number of the form $2^e i$, where e is a small integer and i is an integer between -2^{53} and 2^{53}, can be represented exactly as a double-precision floating-point number. The computations discussed here do not approach the lower or upper limits on e, so we do not review the details of the limits.

Our final software uses fewer multiplications than additions, and fewer permutations than multiplications. This does not mean that we were free to use extra multiplications and permutations: if multiplications and permutations are not finished quickly enough then the addition unit will sit idle waiting for input. In many cases, noted below, we have the flexibility to convert multiplications to additions, reducing latency; we found that in some cases this saved time despite the obvious addition bottleneck.

7.3.2 Optimizing field multiplication

We decompose an integer f modulo $2^{127}-1$ into six floating-point limbs in (non-integer) radix $2^{127/6}$. This means that we write f as $f_0+f_1+f_2+f_3+f_4+f_5$ where f_0 is a small multiple of 2^0, f_1 is a small multiple of 2^{22}, f_2 is a small multiple of 2^{43}, f_3 is a small multiple of 2^{64}, f_4 is a small multiple of 2^{85}, and f_5 is a small multiple of 2^{106}. (The exact meaning of "small" is defined by a rather tedious, but verifiable, collection of bounds on the floating-point numbers appearing in each step of the program. It should be obvious that a simpler definition of "small" would compromise efficiency; for example, H cannot be efficient unless the bounds on H intermediate results and outputs are allowed to be larger than the bounds on H inputs.)

If g is another integer similarly decomposed as $g_0+g_1+g_2+g_3+g_4+g_5$ then f_0g_0 is a multiple of 2^0, $f_0g_1+f_1g_0$ is a multiple of 2^{22}, $f_0g_2+f_1g_1+f_2g_0$ is a multiple of 2^{43}, etc. Each of these sums is small enough to fit exactly in a double-precision floating-point number, and the total of these sums is exactly fg. What we actually compute are the sums

$$\begin{aligned}
h_0 &= f_0g_0 + 2^{-127}f_1g_5 + 2^{-127}f_2g_4 + 2^{-127}f_3g_3 + 2^{-127}f_4g_2 + 2^{-127}f_5g_1,\\
h_1 &= f_0g_1 + f_1g_0 + 2^{-127}f_2g_5 + 2^{-127}f_3g_4 + 2^{-127}f_4g_3 + 2^{-127}f_5g_2,\\
h_2 &= f_0g_2 + f_1g_1 + f_2g_0 + 2^{-127}f_3g_5 + 2^{-127}f_4g_4 + 2^{-127}f_5g_3,\\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + 2^{-127}f_4g_5 + 2^{-127}f_5g_4,\\
h_4 &= f_0g_4 + f_1g_3 + f_2g_2 + f_3g_1 + f_4g_0 + 2^{-127}f_5g_5,\\
h_5 &= f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + f_5g_0,
\end{aligned}$$

whose total h is congruent to fg modulo $2^{127}-1$.

There are 36 multiplications f_ig_j here, and 30 additions. (This operation count does not include carries; we analyze carries below.) One can collect the multiplications by 2^{-127} into 5 multiplications such as $2^{-127}(f_4g_5 + f_5g_4)$. We use another approach, precomputing $2^{-127}f_1, 2^{-127}f_2, 2^{-127}f_3, 2^{-127}f_4, 2^{-127}f_5$, for two reasons: first, this reduces the latency of each h_i computation, giving us more flexibility in scheduling; second, this gives us an opportunity to share precomputations when the input f is reused for another multiplication.

7.3.3 Optimizing field squaring and constant field multiplication

For field squaring **S**, i.e., for $f = g$, we have

$$\begin{aligned}
h_0 &= f_0f_0 + 2^{-127}2f_1f_5 + 2^{-127}2f_2f_4 + 2^{-127}f_3f_3,\\
h_1 &= 2f_0f_1 + 2^{-127}2f_2f_5 + 2^{-127}2f_3f_4,\\
h_2 &= 2f_0f_2 + f_1f_1 + 2^{-127}2f_3f_5 + 2^{-127}f_4f_4,\\
h_3 &= 2f_0f_3 + 2f_1f_2 + 2^{-127}2f_4f_5,\\
h_4 &= 2f_0f_4 + 2f_1f_3 + f_2f_2 + 2^{-127}f_5f_5,\\
h_5 &= 2f_0f_5 + 2f_1f_4 + 2f_2f_3.
\end{aligned}$$

We precompute $2f_1, 2f_2, 2f_3, 2f_4, 2f_5$ and $2^{-127}f_3, 2^{-127}f_4, 2^{-127}f_5$; this costs 8 multiplications, where 5 of the multiplications can be freely replaced by additions. The rest of **S**, after this precomputation, takes 21 multiplications and 15 additions, plus the cost of carries.

For constant field multiplication **m** we have simply $h_0 = cf_0$, $h_1 = cf_1$, etc., costing 6 multiplications plus the cost of carries. This does not work for arbitrary field constants, but it does work for the small constants stated in Section 7.2.4.

7.3.4 Carries

The output limbs h_i from **M** are too large to be used in a subsequent multiplication. We carry $h_0 \to h_1$ by rounding $2^{-22}h_0$ to an integer c_0, adding $2^{22}c_0$ to h_1, and subtracting

$2^{22}c_0$ from h_0. This takes 3 additions (the CPU has a rounding instruction, `vroundpd`, that costs just 1 addition) and 2 multiplications. The resulting h_0 is guaranteed to be between -2^{21} and 2^{21}.

We could similarly carry $h_1 \to h_2 \to h_3 \to h_4 \to h_5$, and carry $h_5 \to h_0$ as follows: round $2^{-127}h_5$ to an integer c_5, add c_5 to h_0, and subtract $2^{127}c_5$ from h_5. One final carry $h_0 \to h_1$, for a total of 7 carries (21 additions and 14 multiplications), would then guarantee that all of $h_0, h_1, h_2, h_3, h_4, h_5$ are small enough to be input to a subsequent multiplication.

The problem with this carry chain is that it has extremely high latency: 5 cycles for $2^{-22}h_0$, 3 more cycles for c_0, 5 more cycles for $2^{22}c_0$, and 3 more cycles to add to h_1, all repeated 7 times, for a total of 112 cycles, plus the latency of obtaining h_0 in the first place. The ladder step in Figure 7.2 has a serial chain of $H \to \mathbf{M} \to \mathbf{m} \to H \to \mathbf{S} \to \mathbf{M}$, for a total latency above 500 cycles, i.e., above 125500 cycles for a 251-bit ladder.

We do better in six ways. First, we use only 6 carries in $\mathbf{M}$ rather than 7, if the output will be used only for $\mathbf{m}$. Even if the output h_0 is several bits larger than 2^{22}, it will not overflow the small-constant multiplication, since our constants are all bounded by 2^{12}.

Second, pushing the same idea further, we do these 6 carries in parallel. First we round in parallel to obtain $c_0, c_1, c_2, c_3, c_4, c_5$, then we subtract in parallel, then we add in parallel, allowing all of $h_0, h_1, h_2, h_3, h_4, h_5$ to end up several bits larger than they would have been with full carries.

Third, we also use 6 parallel carries for a multiplication that *is* an $\mathbf{m}$. There is no need for a chain, since the initial $h_0, h_1, h_2, h_3, h_4, h_5$ cannot be very large.

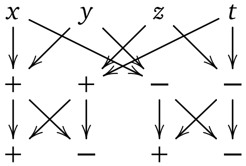
Fourth, we also use 6 parallel carries for each $\mathbf{S}$. This allows the $\mathbf{S}$ output to be somewhat larger than the input, but this still does not create overflows in the subsequent $\mathbf{M}$. At this point the only remaining block of 7 carries is in the $\mathbf{M}^4$ by $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$, where $\mathbf{M}^4$ means a vector of four field multiplications.

Fifth, for that $\mathbf{M}^4$, we run two carry chains in parallel, carrying $h_0 \to h_1$ and $h_3 \to h_4$, then $h_1 \to h_2$ and $h_4 \to h_5$, then $h_2 \to h_3$ and $h_5 \to h_0$, then $h_3 \to h_4$ and $h_0 \to h_1$. This costs 8 carries rather than 7 but chops latency in half.

Finally, for that $\mathbf{M}^4$, we use the carry approach from [Ber04]: add the constant $\alpha_{22} = 2^{22}(2^{52} + 2^{51})$ to h_0, and subtract α_{22} from the result, obtaining the closest multiple of 2^{22} to h_0; add this multiple to h_1 and subtract it from h_0. This costs 4 additions rather than 3, but reduces carry latency from 16 to 9, and also saves two multiplications.

7.4 Permutations in the Hadamard transform

The Hadamard transform H in Section 7.2 is defined as follows: $H(x, y, z, t) = (x + y + z + t, x + y - z - t, x - y + z - t, x - y - z + t)$. Evaluating this as written would use 12 field additions (counting subtraction as addition), but a standard "fast Hadamard transform" reduces the 12 to 8:



We copied this diagram from Bernstein [Ber06b].

Our representation of field elements for the Sandy Bridge (see Section 7.3) requires 6 limb additions for each field addition. There is no need to carry before the subsequent multiplications; this is the main reason that we use 6 limbs rather than 5.

In a ladder step there are 4 copies of H, each requiring 8 field additions, each requiring 6 limb additions, for a total of 192 limb additions. This operation count suggests that 48 vector instructions suffice. Sandy Bridge has a helpful `vaddsubpd` instruction that computes $(a-e, b+f, c-g, d+h)$ given (a,b,c,d) and (e,f,g,h), obviously useful inside H.

However, we cannot simply vectorize across x, y, z, t. In Section 7.3 we were multiplying one x by another, at the same time multiplying one y by another, etc., with no permutations required; in this section we need to add x to y, and this requires permutations.

The Sandy Bridge has a vector permutation unit acting in parallel with the adder and the multiplier, as noted in Section 7.3. But this does not mean that the cost of permutations can be ignored. A long sequence of permutations inside H will force the adder and the multiplier to remain idle, since only a small fraction of the work inside **M** can begin before H is complete.

Our original software used 48 vector additions and 144 vector permutations for the 4 copies of H. We then tackled the challenge of minimizing the number of permutations. We ended up reducing this number from 144 to just 36. This section presents the details; analyzes conditional swaps, which end up consuming further time in the permutation unit; and concludes by analyzing the total number of operations used in our Sandy Bridge software.

7.4.1 Limitations of the Sandy Bridge permutations

There is a latency-1 permutation instruction `vpermilpd` that computes (y, x, t, z) given (x, y, z, t). `vaddsubpd` then produces $(x-y, y+x, z-t, t+z)$, which for the moment we abbreviate as (e, f, g, h). At this point we seem to be halfway done: the desired output is simply $(f+h, f-h, e+g, e-g)$.

If we had (f, h, e, g) at this point, rather than (e, f, g, h), then we could apply `vpermilpd` and `vaddsubpd` again, obtaining $(f-h, h+f, e-g, g+e)$. One final `vpermilpd` would then produce the desired $(f+h, f-h, e+g, e-g)$. The remaining problem is the middle permutation of (e, f, g, h) into (f, h, e, g).

Unfortunately, Sandy Bridge has very few options for moving data between the left half of a vector, in this case (e, f), and the right half of a vector, in this case (g, h).

There is a `vperm2f128` instruction (1-cycle throughput but latency 2) that produces (g,h,e,f), but it cannot even produce (h,g,f,e), never mind a combination such as (f,h,e,g). (Haswell has more permutation instructions, but Ivy Bridge does not. This is not a surprising restriction: n-bit vector units are often designed as $n/2$-bit vector units operating on the left half of a vector in one cycle and the right half in the next cycle, but this means that any communication between left and right requires careful attention in the circuitry. A similar left-right separation is even more obvious for the Cortex-A8.) We could shift some permutation work to the load/store unit, but this would have very little benefit, since simulating a typical permutation requires quite a few loads and stores.

The `vpermilpd` instruction $(x,y,z,t) \mapsto (y,x,t,z)$ mentioned above is one of a family of 16 `vpermilpd` instructions that produce (x or y, x or y, z or t, z or t). There is an even more general family of 16 `vshufpd` instructions that produce (a or b, x or y, c or d, z or t) given (a,b,c,d) and (x,y,z,t). In the first versions of our software we applied `vshufpd` to (e,f,g,h) and (g,h,e,f), obtaining (f,h,g,e), and then applied `vpermilpd` to obtain (f,h,e,g).

Overall a single H handled in this way uses, for each limb, 2 `vaddsubpd` instructions and 6 permutation instructions, half of which are handling the permutation of (e,f,g,h) into (f,h,e,g). The total for all limbs is 12 additions and 36 permutations, and the large “bubble” of permutations ends up forcing many idle cycles for the addition unit. This occurs four times in each ladder step.

7.4.2 Changing the input/output format

There are two obvious sources of inefficiency in the computation described above. First, we need a final permutation to convert $(f-h,f+h,e-g,e+g)$ into $(f+h,f-h,e+g,e-g)$. Second, the middle permutation of (e,f,g,h) into (f,h,e,g) costs three permutation instructions, whereas (g,h,e,f) would cost only one.

The first problem arises from a tension between Intel’s `vaddsubpd`, which always subtracts in the first position, and the definition of H, which always adds in the first position. A simple way to resolve this tension is to store (t,z,y,x) instead of (x,y,z,t) for the input, and (t',z',y',x') instead of (x',y',z',t') for the output; the final permutation then naturally disappears. It is easy to adjust the other permutations accordingly, along with constants such as $(1,a^2/b^2,a^2/c^2,a^2/d^2)$.

However, this does nothing to address the second problem. Different permutations of (x,y,z,t) as input and output end up requiring different middle permutations, but these middle permutations are never exactly the left-right swap provided by `vperm2f128`.

We do better by generalizing the input/output format to allow negations. For example, if we start with $(x,-y,z,t)$, permute into $(-y,x,t,z)$, and apply `vaddsubpd`, we obtain $(x+y,x-y,z-t,t+z)$. Observe that this is not the same as the $(x-y,x+y,z-t,t+z)$ that we obtained earlier: the first two entries have been exchanged.

It turns out to be best to negate z, i.e., to start from $(x,y,-z,t)$. Then `vpermilpd` gives $(y,x,t,-z)$, and `vaddsubpd` gives $(x-y,x+y,-z-t,t-z)$, which we now abbreviate as (e,f,g,h). Next `vperm2f128` gives (g,h,e,f), and independently

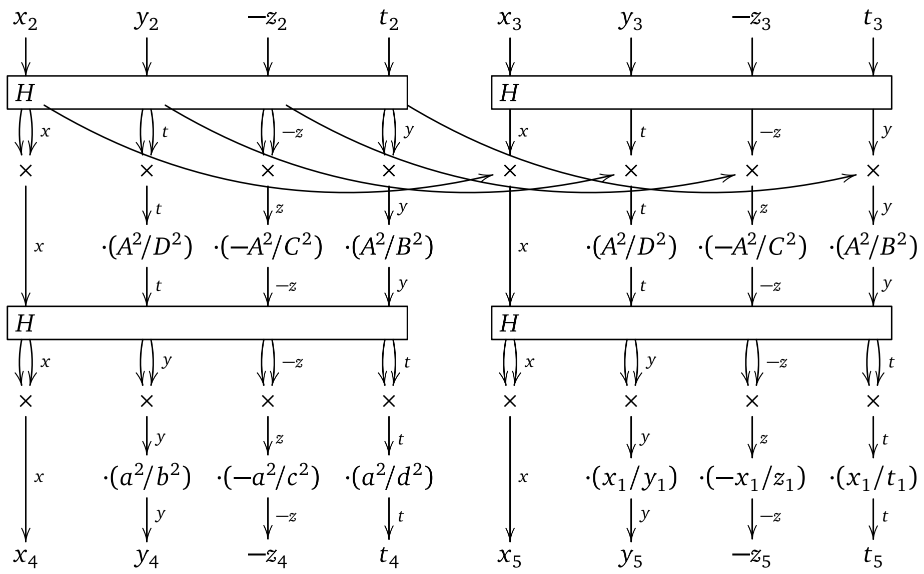


Figure 7.3: Output format that we use for each operation in the right side of Figure 7.2 on Sandy Bridge, including permutations and negations to accelerate H.

`vpermilpd` gives (f, e, h, g). Finally, `vaddsubpd` gives $(f - g, h + e, h - e, f + g)$. This is exactly $(x', t', -z', y')$ where $(x', y', z', t') = H(x, y, z, t)$.

The output format here is not the same as the input format: the positions of t and y have been exchanged. Fortunately, Figure 7.2 is partitioned by the H rows into two separate universes, and there is no need for the universes to use the same format. We use the $(x, y, -z, t)$ format at the top and bottom, and the $(x, t, -z, y)$ format between the two H rows. It is easy to see that exactly the same sequence of instructions works for all the copies of H, either producing $(x, y, -z, t)$ format from $(x, t, -z, y)$ format or vice versa.

$\mathbf{S}^4$ and $\mathbf{M}^4$ do not preserve negations: in effect, they switch from $(x, t, -z, y)$ format to (x, t, z, y) format. This is not a big problem, since we can reinsert the negation at any moment using a single multiplication or low-latency logic instruction (floating-point numbers use a sign bit rather than two's-complement, so negation is simply xor with a 1 in the sign bit). Even better, in Figure 7.2(b), the problem disappears entirely: each $\mathbf{S}^4$ and $\mathbf{M}^4$ is followed immediately by a constant multiplication, and so we simply negate the appropriate constants. The resulting sequence of formats is summarized in Figure 7.3.

Each H now costs 12 additions and just 18 permutations. The number of non-addition cycles that need to be overlapped with operations before and after H has dropped from the original 24 to just 6.

7.4.3 Exploiting double precision

We gain a further factor of 2 by temporarily converting from radix $2^{127/6}$ to radix $2^{127/3}$ during the computation of H. This means that, just before starting H, we replace the six limbs $(h_0, h_1, h_2, h_3, h_4, h_5)$ representing $h_0 + h_1 + h_2 + h_3 + h_4 + h_5$ by three limbs $(h_0 + h_1, h_2 + h_3, h_4 + h_5)$. These three sums, and the intermediate H results, still fit into double-precision floating-point numbers.

It is essential to switch each output integer back to radix $2^{127/6}$ so that each output limb is small enough for the subsequent multiplication. Converting three limbs into six is slightly less expensive than three carries; in fact, converting from six to three and back to six uses exactly the same operations as three carries, although in a different order.

We further reduce the conversion cost by the following observation. Except for the $\mathbf{M}^4$ by $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$, each of our multiplication results uses six carries, as explained in Section 7.3.4. However, if we are about to add h_0 to h_1 for input to H, then there is no reason to carry $h_0 \to h_1$, so we simply skip that carry; we similarly skip $h_2 \to h_3$ and $h_4 \to h_5$. These skipped carries exactly cancel the conversion cost.

For the $\mathbf{M}^4$ by $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$ the analysis is different: h_0 is large enough to affect h_2, and if we skipped carrying $h_0 \to h_1 \to h_2$ then the output of H would no longer be safe as input to a subsequent multiplication. We thus carry $h_0 \to h_1$, $h_2 \to h_3$, and $h_4 \to h_5$ in parallel; and then $h_1 \to h_2$, $h_3 \to h_4$, and $h_5 \to h_0$ in parallel. In effect this $\mathbf{M}^4$ uses 9 carries, counting the cost of conversion, whereas in Section 7.3.4 it used only 8.

To summarize, all of these conversions for all four H cost just one extra carry, while reducing 48 additions and 72 permutations to 24 additions and 36 permutations.

7.4.4 Conditional swaps

A ladder step starts from an input $(X(nP), X((n+1)P))$, which we abbreviate as $L(n)$, and produces $L(2n)$ as output. Swapping the two halves of the input, applying the same ladder step, and swapping the two halves of the output produces $L(2n+1)$ instead; one way to see this is to observe that $L(-n-1)$ is exactly the swap of $L(n)$.

Consequently one can reach $L(2n+\epsilon)$ for $\epsilon \in \{0,1\}$ by starting from $L(n)$, conditionally swapping, applying the ladder step, and conditionally swapping again, where the condition bit is exactly ϵ. A standard ladder reaches $L(n)$ by applying this idea recursively. A standard *constant-time* ladder reaches $L(n)$ by applying this idea for exactly ℓ steps, starting from $L(0)$, where n is known in advance to be between 0 and $2^\ell - 1$. An alternate approach is to first add to n an appropriate multiple of the order of P, producing an integer known to be between (e.g.) $2^{\ell+1}$ and $2^{\ell+2} - 1$, and then start from $L(1)$. We use a standard optimization, merging the conditional swap after a ladder step into the conditional swap before the next ladder step, so that there are just $\ell + 1$ conditional swaps rather than 2ℓ.

One way to conditionally swap field elements x and x' using floating-point arithmetic is to replace (x, x') with $(x + b(x' - x), x' - b(x' - x))$ where b is the condition bit, either 0 or 1. This takes three additions and one multiplication (times 6 limbs, times 4 field elements to swap). It is better to use logic instructions: replace each addition with `xor`, replace each multiplication with `and`, and replace b with an all-1 or all-0 mask computed from b. On the Sandy Bridge, logic instructions have low latency and are handled by the permutation unit, which is much less of a bottleneck for us than the addition unit.

We further improve the performance of the conditional swap as follows. The $\mathbf{M}^4$ on the right side of Figure 7.3 is multiplying H of the left input by H of the right input. This is commutative: it does not depend on whether the inputs are swapped. We therefore put the conditional swap *after* the first row of H computations, and multiply the H outputs directly, rather than multiplying the swap outputs. This trick has several minor effects and one important effect.

A minor advantage is that this trick removes all use of the right half of the swap output; i.e., it replaces the conditional swap with a conditional move. This reduces the original 24 logic instructions to just 18.

Another minor advantage is that the Sandy Bridge has a vectorized conditional-select instruction `vblendvpd`. This instruction occupies the permutation unit for 2 cycles, so it is no better than the 4 traditional logic instructions for a conditional swap: a conditional swap requires two conditional selects. However, this instruction *is* better than the 3 traditional logic instructions for a conditional move: a conditional move requires only one conditional select. This replaces the original logic instructions with 6 conditional-select instructions, consuming just 12 cycles.

A minor disadvantage is that the first $\mathbf{M}^4$ and $\mathbf{S}^4$ are no longer able to share precomputations of multiplications by 2^{-127}. This costs us 3 multiplication instructions.

The important effect is that this trick reduces latency, allowing the $\mathbf{M}^4$ to start much sooner. Adding this trick immediately produced a 5% reduction in our cycle counts.

7.4.5 Total operations

We treat Figure 7.2(b) as $2\mathbf{M}^4 + 3\mathbf{S}^4 + 3\mathbf{m}^4 + 4H$. The main computations of h_i, not counting precomputations and carries, cost 30 additions and 36 multiplications for each $\mathbf{M}^4$, 15 additions and 21 multiplications for each $\mathbf{S}^4$, and 0 additions and 6 multiplications for each $\mathbf{m}^4$. The total here is 105 additions and 153 multiplications.

The $\mathbf{M}^4$ by $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$ allows precomputations outside the loop. The other $\mathbf{M}^4$ consumes 5 multiplications for precomputations, and each $\mathbf{S}^4$ consumes 8 multiplications for precomputations; the total here is 29 multiplications. We had originally saved a few multiplications by sharing precomputations between the first $\mathbf{S}^4$ and the first $\mathbf{M}^4$, but this is incompatible with the more important trick described in Section 7.4.4.

There are a total of 24 additions in the four H, as explained in Section 7.4.3. There are also 51 carries (counting the conversions described in Section 7.4.3 as carries), each consuming 3 additions and 2 multiplications, for a total of 153 additions and 102 multiplications.

The grand total is 282 additions and 284 multiplications, evidently requiring at least 284 cycles for each iteration of the main loop. Recall that there are various options to trade multiplications for additions: each $\mathbf{S}^4$ has 5 precomputed doublings that can each be converted from 1 multiplication to 1 addition, and each carry can be converted from 3 additions and 2 multiplications to 4 additions and 0 multiplications (or 4 additions and 1 multiplication for $h_5 \to h_0$). We could use either of these options to eliminate one multiplication, reducing the 284-cycle lower bound to 283 cycles, but to reduce latency we ended up instead using the first option to eliminate 10 multiplications and the second option to eliminate 35 multiplications, obtaining a final total of 310 additions and 239 multiplications. These totals have been computer-verified.

We wrote functions in assembly for $\mathbf{M}^4$, $\mathbf{S}^4$, etc., but were still over 500 cycles. Given the Sandy Bridge floating-point latencies, and the requirement to keep *two* floating-point units constantly busy, we were already expecting instruction scheduling to be much more of an issue for this software than for typical integer-arithmetic software. We used various standard optimization techniques that were already used in several previous DH speed records: we merged the functions into a single loop, reorganized many computations to save registers, and eliminated many loads and stores. After building a new Sandy Bridge simulator and experimenting with different instruction schedules we ended up with our current loop, just 338 cycles, and a total of 88916 Sandy Bridge cycles for scalar multiplication. The main loop explains 84838 of these cycles; the remaining cycles are spent outside the ladder, mostly on converting $(x : y : z : t)$ to $(x/y : x/z : x/t)$ for output.

7.5 Cortex-A8

The low-power ARM Cortex-A8 core is the CPU core in the iPad 1, iPhone 4, Samsung Galaxy S, Motorola Droid X, Amazon Kindle 4, etc. Today a Cortex-A8 CPU, the Allwinner A10, costs just $5 in bulk and is widely used in low-cost tablets, set-top boxes, etc. Like Sandy Bridge, Cortex-A8 is not the most recent microarchitecture, but its very wide deployment and use make it a sensible choice of platform for optimization and performance comparisons.

Bernstein and Schwabe in [BS12] (CHES 2012) analyzed the vector capabilities of the Cortex-A8 for various cryptographic primitives, and in particular set a new speed record for high-security DH, namely 460200 Cortex-A8 cycles. We do much better, just 274593 Cortex-A8 cycles, measured on a Freescale i.MX515. Our basic vectorization approach is the same for Cortex-A8 as for Sandy Bridge, and many techniques are reused, but there are also many differences. The rest of this section explains the details.

7.5.1 Cortex-A8 vector units

Each Cortex-A8 core has two 128-bit vector units operating in parallel on vectors of four 32-bit integers or two 64-bit integers:

- The arithmetic port takes one cycle for vector addition, with latency 2; or two cycles for vector multiplication (two 64-bit products ac, bd given 32-bit inputs

a, b and c, d), with latency 7. Logic operations also use the arithmetic port.

- The load/store port handles loads, stores, and permutations. ARM's Cortex-A8 documentation [ARM10] indicates that the load/store port can carry out one 128-bit load every cycle. Beware, however, that there are throughput limits on the L1 cache. We have found experimentally that the common TI Sitara Cortex-A8 CPU (used, e.g., in the Beaglebone Black development board) needs three cycles from one load until the next (this is what we call "Cortex-A8-slow"), while other Cortex-A8 CPUs ("Cortex-A8-fast") can handle seven consecutive cycles of loads without penalty.

There are three obvious reasons for Cortex-A8 cycle counts to be much larger than Sandy Bridge cycle counts: registers are only 128 bits, not 256 bits; there are only 2 ports, not 6; and multiplication throughput is 1 every 2 cycles, not 1 every cycle. However, there are also speedups on Cortex-A8. There is (as in Haswell's floating-point units—see Section 7.6) a vector multiply-accumulate instruction with the same throughput as vector multiplication. A sequence of m consecutive multiply-accumulate instructions that all accumulate into the same register executes in $2m$ cycles (unlike Haswell), effectively reducing multiplication latency from 7 to 1. Furthermore, Cortex-A8 multiplication produces 64-bit integer products, while Sandy Bridge gives only 53-bit-mantissa products.

7.5.2 Representation

We decompose an integer f modulo $2^{127}-1$ into *five* integer pieces in radix $2^{127/5}$: i.e., we write f as $f_0+2^{26}f_1+2^{51}f_2+2^{77}f_3+2^{102}f_4$. Compared to Sandy Bridge, having 20% more room in 64-bit integers than in 53-bit floating-point mantissas allows us to reduce the number of limbs from 6 to 5. We require the small integers f_0, f_1, f_2, f_3, f_4 to be *unsigned* because this reduces carry cost from 4 integer instructions to 3.

We arrange four integers x, y, z, t modulo $2^{127}-1$ in five 128-bit vectors, namely, (x_0, y_0, x_1, y_1); (x_2, y_2, x_3, y_3); (x_4, y_4, z_4, t_4); (z_0, t_0, z_1, t_1); (z_2, t_2, z_3, t_3). This representation is designed to minimize permutations in $\mathbf{M}$, $\mathbf{S}$, and H. For example, computing $(x_0+z_0, y_0+t_0, x_1+z_1, y_1+t_1)$ takes just one addition without any permutations. The Cortex-A8 multiplications take two pairs of inputs at a time, rather than four as on Sandy Bridge, so there is little motivation to put (x_0, y_0, z_0, t_0) into a vector.

7.5.3 Optimizing field multiplication

Given an integer f as above and an integer $g = g_0 + 2^{26}g_1 + 2^{51}g_2 + 2^{77}g_3 + 2^{102}g_4$, the product fg modulo $2^{127}-1$ is $h = h_0 + 2^{26}h_1 + 2^{51}h_2 + 2^{77}h_3 + 2^{102}h_4$, with

$$\begin{aligned}
h_0 &= f_0g_0 + 2f_1g_4 + 2f_2g_3 + 2f_3g_2 + 2f_4g_1, \\
h_1 &= f_0g_1 + f_1g_0 + f_2g_4 + 2f_3g_3 + f_4g_2, \\
h_2 &= f_0g_2 + 2f_1g_1 + f_2g_0 + 2f_3g_4 + 2f_4g_3, \\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + f_4g_4, \\
h_4 &= f_0g_4 + 2f_1g_3 + f_2g_2 + 2f_3g_1 + f_4g_0.
\end{aligned}$$

There are 25 multiplications $f_i g_j$; additions are free as part of multiply-accumulate instructions. We precompute $2f_1, 2f_2, 2f_3, 2f_4$ so that these values can be reused for another multiplication. These precomputations can be done by using either 4 shift or 4 addition instructions. Both shift and addition use 1 cycle per instruction, but addition has a lower latency. See Section 7.5.6 for the cost of carries.

7.5.4 Optimizing field squaring

The idea of optimizing field squaring **S** in Cortex-A8 is quite similar to Sandy Bridge; for details see Section 7.3.3. We state here only the operation count. Besides precomputation and carry, we use 15 multiplication instructions; some of those are actually multiply-accumulate instructions. From now on we describe both multiplication instructions and multiply-accumulate instructions as “multiplications” without further comment.

7.5.5 Optimizing constant field multiplication

For constant field multiplication **m**, we compute only $h_0 = cf_0$, $h_1 = cf_1$, $h_2 = cf_2$, $h_3 = cf_3$, and $h_4 = cf_4$, again exploiting the small constants stated in Section 7.2.4.

Recall that we use *unsigned* representation. We always multiply absolute values, then negate results as necessary by subtracting from $2^{129} - 4$: $n_0 = 2^{28} - 4 - h_0$, $n_1 = 2^{27} - 4 - h_1$, $n_2 = 2^{28} - 4 - h_2$, $n_3 = 2^{27} - 4 - h_3$, $n_4 = 2^{27} - 4 - h_4$.

Negating any subsequence of x, y, z, t costs at most 5 vector subtractions. Negating only x or y, or both x and y, costs only 3 subtractions, because our representation keeps x, y within 3 vectors. The same comment applies to z and t. The specific **m** in Section 7.2.4 end up requiring a total of 13 subtractions with the same cost as 13 additions.

7.5.6 Carries

Each multiplication uses at worst 6 serial carries $h_1 \to h_2 \to h_3 \to h_4 \to h_0 \to h_1$, each costing 3 additions. Various carries are eliminated by the ideas of Section 7.3.4.

7.5.7 Hadamard transform

Each of the 8 field additions in H requires 5 additions of limbs. One ladder step has four H, for a total of 160 limb additions, i.e., at least 40 vector additions.

Four of the field additions in H are actually subtractions. We handle subtractions by the same strategy as Section 7.5.6. The extra constants cost another 5 vector additions per H.

The detailed sequence of operations that we use on the Cortex-A8 is as follows. The Hadamard transform receives as input

$$\begin{aligned}
r_0 &= (x_0, y_0, x_1, y_1);\\
r_1 &= (x_2, y_2, x_3, y_3);\\
r_2 &= (x_4, y_4, z_4, t_4);\\
r_3 &= (z_0, t_0, z_1, t_1);\\
r_4 &= (z_2, t_2, z_3, t_3).
\end{aligned}$$

The output will be 5 registers $s_0, \dots, s_4$ with

$$\begin{aligned}
s_0&=((x_0+y_0)+(z_0+t_0),(x_0+y_0)-(z_0+t_0),(x_1+y_1)+(z_1+t_1),(x_1+y_1)-(z_1+t_1)),\\
s_1&=((x_0-y_0)+(z_0-t_0),(x_0-y_0)-(z_0-t_0),(x_1-y_1)+(z_1-t_1),(x_1-y_1)-(z_1-t_1)),\\
s_2&=((x_4+y_4)+(z_4+t_4),(x_4+y_4)-(z_4+t_4),(x_4-y_4)+(z_4-t_4),(x_4-y_4)-(z_4-t_4)),\\
s_3&=((x_2+y_2)+(z_2+t_2),(x_2+y_2)-(z_2+t_2),(x_3+y_3)+(z_3+t_3),(x_3+y_3)-(x_3+y_3)),\\
s_4&=((x_2-y_2)+(z_2-t_2),(x_2-y_2)-(z_2-t_2),(x_3-y_3)+(z_3-t_3),(x_3-y_3)-(x_3-y_3)).
\end{aligned}$$

We begin with vector addition and subtraction to produce

$$\begin{aligned}
t_0 &= (x_0 + z_0, y_0 + t_0, x_1 + z_1, y_1 + t_1),\\
t_1 &= (x_0 - z_0, y_0 - t_0, x_1 - z_1, y_1 - t_1),\\
t_2 &= (x_2 + z_2, y_2 + t_2, x_3 + z_3, y_3 + t_3),\\
t_3 &= (x_2 - z_2, y_2 - t_2, x_3 - z_3, y_3 - t_3).
\end{aligned}$$

We would next like to add/subtract $x_0 + z_0$ with $y_0 + t_0$, also $x_1 + z_1$ with $y_1 + t_1$, and so on. Unfortunately, there are no instructions to add/subtract among or between left/right halves of vectors. There is a Cortex-A8 instruction `vtrn` which allows permuting two vectors (a, b, c, d) (e, f, g, h) to produce (a, e, c, g) (b, f, d, h), and can also permute two vectors (a, b, c, d) (e, f, g, h) to produce (a, b, c, g) (e, f, d, h). Another helpful instruction is `vswp` which swaps left and right halves of two vectors in various ways such as (a, b, c, d) $(e, f, g, h) \to (a, b, e, f)$ (c, d, g, h), and (a, b, c, d) $(e, f, g, h) \to (a, b, g, h)$ (e, f, c, d).

We apply `vtrn` to t_0 and t_1 to produce

$$\begin{aligned}
t_4 &= (x_0 + z_0, x_0 - z_0, x_1 + z_1, x_1 - z_1),\\
t_5 &= (y_0 + t_0, y_0 - t_0, y_1 + t_1, y_1 - t_1).
\end{aligned}$$

We then add and subtract to produce

$$\begin{aligned}
t_6 &= (x_0 + z_0 + y_0 + t_0, x_0 - z_0 + y_0 - t_0, x_1 + z_1 + y_1 + t_1, x_1 - z_1 + y_1 - t_1),\\
t_7 &= (x_0 + z_0 - y_0 - t_0, x_0 - z_0 - y_0 + t_0, x_1 + z_1 - y_1 - t_1, x_1 - z_1 - y_1 + t_1).
\end{aligned}$$

These are two of the desired output vectors from the Hadamard transform.

We could repeat similar steps for t_2 and t_3, but then there would be considerable overhead in handling the one remaining vector. To avoid arithmetic overhead we permute three vectors together while performing arithmetic on two at a time. Specifically, we apply `vtrn` to t_2 and t_3 to produce

$$\begin{aligned} t_8 &= (x_2+z_2, x_2-z_2, x_3+z_3, x_3-z_3), \\ t_9 &= (y_2+t_2, y_2-t_2, y_3+t_3, y_3-t_3); \end{aligned}$$

next use `vswp` $(a,b,c,d)\ (e,f,g,h) \to (a,b,e,f)\ (c,d,g,h)$ to t_8, r_2 to produce

$$\begin{aligned} t_{10} &= (x_2+z_2, x_2-z_2, x_4, y_4), \\ t_{11} &= (x_3+z_3, x_3-z_3, z_4, t_4); \end{aligned}$$

and then use `vswp` $(a,b,c,d)\ (e,f,g,h) \to (a,b,g,h)\ (e,f,c,d)$ to t_9, t_{11} to produce

$$\begin{aligned} t_{12} &= (y_2+t_2, y_2-t_2, z_4, t_4), \\ t_{13} &= (x_3+z_3, x_3-z_3, y_3+t_3, y_3-t_3). \end{aligned}$$

We then add and subtract t_{10} and t_{12} to produce

$$\begin{aligned} t_{14} &= (x_2+z_2+y_2+t_2, x_2-z_2+y_2-t_2, x_4+z_4, y_4+t_4), \\ t_{15} &= (x_2+z_2-y_2-t_2, x_2-z_2-y_2+t_2, x_4-z_4, y_4-t_4). \end{aligned}$$

Next, we perform another sequence of permutations as follows: starting with using `vswp` $(a,b,c,d)\ (e,f,g,h) \to (e,f,c,d)\ (a,d,g,h)$ to t_{14} and t_{13} to produce

$$\begin{aligned} t_{16} &= (x_3+z_3, x_3-z_3, x_4+z_4, y_4+t_4), \\ t_{17} &= (x_2+z_2+y_2+t_2, x_2-z_2+y_2-t_2, y_3+t_3, y_3-t_3); \end{aligned}$$

then using `vswp` $(a,b,c,d)\ (e,f,g,h) \to (a,b,e,f)\ (c,d,g,h)$ to t_{17} and t_{15} to produce

$$\begin{aligned} t_{18} &= (x_2+z_2+y_2+t_2, x_2-z_2+y_2-t_2, x_2+z_2-y_2-t_2, x_2-z_2-y_2+t_2), \\ t_{19} &= (y_3+t_3, y_3-t_3, x_4-z_4, y_4-t_4); \end{aligned}$$

and then using `vtrn` $(a,b,c,d)\ (e,f,g,h) \to (a,b,c,g)\ (e,f,d,h)$ to t_{16} and t_{19} to produce

$$\begin{aligned} t_{20} &= (x_3+z_3, x_3-z_3, x_4+z_4, x_4-z_4), \\ t_{21} &= (y_3+t_3, y_3-t_3, y_4+t_4, y_4-t_4). \end{aligned}$$

Now we are ready to add and subtract t_{20} with t_{21} to produce

$$\begin{aligned} t_{22} &= (x_3+z_3+y_3+t_3, x_3-z_3+y_3-t_3, x_4+z_4+y_4+t_4, x_4-z_4+y_4-t_4), \\ t_{23} &= (x_3+z_3-y_3-t_3, x_3-z_3-y_3+t_3, x_4+z_4-y_4-t_4, x_4-z_4-y_4+t_4). \end{aligned}$$

Finally, we use `vswp` $(a,b,c,d)\ (e,f,g,h) \to (a,b,e,f)\ (c,d,g,h)$ to t_{22} and t_{23} to produce

$$\begin{aligned} t_{24} &= (x_3+z_3+y_3+t_3, x_3-z_3+y_3-t_3, x_3+z_3-y_3-t_3, x_3-z_3-y_3+t_3), \\ t_{25} &= (x_4+z_4+y_4+t_4, x_4-z_4+y_4-t_4, x_4+z_4-y_4-t_4, x_4-z_4-y_4+t_4); \end{aligned}$$

and `vswp` $(a,b,c,d)\ (e,f,g,h) \to (a,b,e,f)\ (c,d,g,h)$ to t_{18} and t_{24} to produce

$$\begin{aligned} t_{26} &= (x_2+z_2+y_2+t_2, x_2-z_2+y_2-t_2, x_3+z_3+y_3+t_3, x_3-z_3+y_3-t_3),\\ t_{27} &= (x_2+z_2-y_2-t_2, x_2-z_2-y_2+t_2, x_3+z_3-y_3-t_3, x_3-z_3-y_3+t_3). \end{aligned}$$

The vectors $t_6, t_7, t_{25}, t_{26}, t_{27}$ are the final results of the Hadamard transform.

7.5.8 Total arithmetic

We view Figure 7.2(b) as $4\mathbf{M}^2 + 6\mathbf{S}^2 + 6\mathbf{m}^2 + 4H$. Here we combine x multiplications and y multiplications into a vectorized $\mathbf{M}^2$, and similarly combine z multiplications and t multiplications; this fits well with the Cortex-A8 vector multiplication instruction, which outputs two products.

The main computations of h_i, not counting precomputations and carries, cost 0 additions and 25 multiplications for each $\mathbf{M}$, 0 additions and 15 multiplications for each $\mathbf{S}$, 0 additions and 5 multiplications for each $\mathbf{m}$, and 15 additions for each H block. The total here is 60 additions and 220 multiplications.

Each $\mathbf{M}$ costs 4 additions for precomputations, and each $\mathbf{S}$ also costs 4 additions for precomputations. Some precomputations can be reused. The cost of precomputations is 20 additions.

There are 10 carry blocks using 6 carries each, and 6 carry blocks using 5 carries each. Each carry consists of 1 shift, 1 addition, and 1 logical `and`. This cost is equivalent to 3 additions. There are another 13 additions needed to handle negation. Overall the carries cost 283 additions. Two conditional swaps, each costing 9 additions, sum up to 18 additions.

In total we have 381 additions and 220 multiplications in our inner loop. This means that the inner loop takes at least 821 cycles.

We scheduled instructions carefully but ended up with some overhead beyond arithmetic: even though the arithmetic and the load/store unit can operate in parallel, latencies and the limited number of registers leave the arithmetic unit idle for some cycles. Sobole's simulator at [Sob12], which we found very helpful, reports 966 cycles. Actual measurements report 986 cycles; the 251 ladder steps thus account for 247486 of our 273349 cycles.

7.6 Haswell

Compared to Sandy Bridge (and Ivy Bridge), Haswell devotes considerably more transistors to multiplication. This section analyzes the impact of Haswell's new multipliers, and in particular explains how we achieved 60556 Haswell cycles for HECC.

7.6.1 Binary-polynomial multipliers

Haswell has an unusual level of hardware support for fast binary-polynomial multiplication. One should expect this to make binary-field ECC much faster on Haswell than on other CPUs, as illustrated by the impressive 61712 Haswell cycles in [OLARH13] for constant-time binary-field GLV+GLS ECC. (Note that recent advances in index-calculus discrete-logarithm algorithms for *multiplicative* groups of binary fields do not threaten the binary-field *curves* used in [OLARH13].)

Our HECC speeds are nevertheless faster. Perhaps there are further improvements in the approach of [OLARH13], but it is clear that HECC is an excellent cross-platform option and will continue to benefit from increased CPU support for vectorization, while it is not at all clear whether Intel will continue to expand the circuit area devoted to binary-field arithmetic, or whether other CPU manufacturers will follow. Note also that there are formulas in [GL09] for Kummer surfaces of binary hyperelliptic curves, opening up the possibility of a unification of these techniques.

7.6.2 Vectorized floating-point multipliers

Each Haswell core has *two* vectorized floating-point multiplication units: port 0 and port 1 each handle one 256-bit vector multiplication each cycle. Even better, these multipliers include integrated adders, so in one cycle a Haswell core can compute $ab+c$ and $de+f$, while in one cycle a Sandy Bridge core can compute at best ab and $d+f$. One should not think that this trivially reduces our cycle counts by a factor of 2:

- Multiplication latency is still 5 cycles, so to keep both multipliers busy one needs 10 independent multiplications. For comparison, keeping the Sandy Bridge multiplier busy needs only 5 independent multiplications.

- The integrated adders are useful only for additions (or subtractions) of products. There is no improvement to the performance of other addition instructions: each 256-bit vector addition consumes port 1 for one cycle. Obviously one can also take advantage of port 0 by rewriting $b+c$ as $1b+c$, but this still costs 1 arithmetic instruction rather than 0.5 arithmetic instructions, and it also increases latency from 3 to 5.

A further detail of importance for us is that the `vroundpd` instruction on the Haswell costs as much as two additions, whereas on Sandy Bridge it costs just one. We therefore switched to the carry approach from [Ber04] described in Section 7.3.4. This reduced our starting 284 multiplications on Sandy Bridge to 190 multiplications: it eliminated 2 multiplications in each of 51 carries, except that the 8 carries $h_5 \to h_0$ (see Section 7.3.4) still require 1 multiplication each. Meanwhile this increased the original 282 additions to 333 additions.

Most of the additions in Sections 7.3.4 and 7.4 cannot be integrated into multiplications but the additions in Sections 7.3.2 and 7.3.3 naturally have the form “add ab into c”. After integrating additions into multiplications we were left with 220 additions and 190 multiplications. These numbers have been computer-verified.

This arithmetic consumes at least 205 arithmetic cycles, i.e., at least 51455 cycles for 251 iterations of the main loop. The actual performance of our main loop at this point was much worse, 333 cycles. After initial rescheduling we thought that our total cost was 72276 cycles, including 68272 cycles from the main loop, although as noted above these figures were corrupted by Turbo Boost; the correct figures would have been even higher. We then switched to exploring a different approach described below.

7.6.3 Vectorized integer multipliers

Each Haswell core has one 4-way vectorized $32 \times 32 \to 64$ integer multiplier (port 0) and two 4-way vectorized 64-bit integer adders (port 5, competing with permutations, and port 1). Compared to floating-point vectors, Haswell's integer vectors have the obvious disadvantage of producing only 4 products per cycle instead of 8; on the other hand, integer vectors provide a noticeably better spread of operations across ports, the ability to efficiently use 5 limbs rather than 6, and a much lower addition latency.

We split each field element x into 5 unsigned limbs x_0, x_1, x_2, x_3, x_4; we use unsigned limbs here because Haswell's vectorized 64-bit right-shift instruction handles only unsigned integers. Each limb fits into 32 bits. We arrange four field elements (x, y, z, t) as five 256-bit vectors $(x_i, 0, y_i, 0, z_i, 0, t_i, 0)$, matching the input format used by the vectorized integer-multiplication instruction.

M involves 25 32-bit multiplications, 20 64-bit additions, and 6 carries, after a precomputation involving 4 32-bit doublings. Each carry consists of 1 shift, 1 addition, and 1 mask. For the precomputation we (1) use additions rather than shifts, because shift instructions compete with the multiplier on port 0, and (2) use 64-bit additions rather than 32-bit additions, because the extra efficiency of 32-bit additions is outweighed by the cost of extra permutations.

$\mathbf{M}^4$ thus uses 25 (vectorized) multiplication instructions, 26 or 30 addition instructions depending on whether precomputation is included, 6 shift instructions, and 6 mask instructions. Similarly, $\mathbf{S}^4$ uses 15 multiplication instructions, 20 addition instructions, 6 shift instructions, and 6 mask instructions; $\mathbf{m}^4$ uses 5 multiplication instructions, 5 addition instructions, 5 shift instructions, and 5 mask instructions. We use absolute values of all constants in $\mathbf{m}^4$; this means that we are implicitly negating x as output of $\mathbf{m}^4$, but we compensate for this in the H steps, as discussed in a moment.

We apply H to two inputs (x, y, z, t) and (x', y', z', t') at once. These two inputs actually consist of five vectors $(-x_i, 0, y_i, 0, z_i, 0, t_i, 0)$ and five vectors $(-x'_i, 0, y'_i, 0, z'_i, 0, t'_i, 0)$. For each i we build $(-x_i, -x'_i, y_i, y'_i, z_i, z'_i, t_i, t'_i)$ at the expense of a shift and a xor. (We could alternatively apply the same format conversions to four out of the five vectors in one H input, reducing the data flow between the left and right halves of Figure 7.2(b).) Undoing this format conversion at the end of H^2 might seem to involve a shift and a mask, but we actually skip the mask: each H output is used solely for multiplication, and the multiplication instructions implicitly mask their inputs.

The input to the core of H^2 is thus $(-x_i, -x'_i, y_i, y'_i, z_i, z'_i, t_i, t'_i)$, which for this paragraph we abbreviate as $(-x, y, z, t)$. For the first Hadamard level we obtain

- $(x - 1, y, z, -t - 1)$ by a xor of $(-x, y, z, t)$ with $(-1, 0, 0, -1)$, exploiting the two's-complement representation of integers;

- $(y, -x, t, z)$ by a permutation; and
- $(x + y - 1, y - x, z + t, z - t - 1)$ by an addition.

We then obtain $(x + y - 1, z + t, y - x, z - t - 1)$ by a (latency-3) permutation. For the second Hadamard level we obtain

- $(x + y - 1, -z - t - 1, x - y - 1, z - t - 1)$ by a xor with $(0, -1, -1, 0)$;
- $(z + t, x + y - 1, z - t - 1, y - x)$ by a permutation; and
- $(x + y + z + t - 1, x + y - z - t - 2, x - y + z - t - 2, -(x - y - z + t) - 1)$ by an addition.

A final addition of $(1, 2, 2, 1)$ would produce $(x + y + z + t, x + y - z - t, x - y + z - t, -(x - y - z + t))$. We actually add something larger (equal to $(1, 2, 2, 1)$ modulo $2^{255} - 19$) to produce unsigned results, as in Section 7.5. The negation of $x - y - z + t$ disappears as in Figure 7.3.

Overall H^2 uses 11 vector instructions—3 additions, 2 shifts, 3 permutations, and 3 logic instructions—for each of the 5 limbs. There are also 5 conditional-select instructions after the first H^2, as in Section 7.4.

We treat Figure 7.2(b) as $2\mathbf{M}^4 + 3\mathbf{S}^4 + 3\mathbf{m}^4 + 2H^2$, for a total of 110 multiplications, 161 additions, 65 shifts, 75 logic instructions, and 30 permutations, plus the 5 conditional-select instructions, each of which is as expensive as 2 permutations. These totals have been computer-verified.

The 175 multiplications and shifts use port 0; the 161 additions use port 1; the $30 + 5 \cdot 2$ permutations use port 5; and the 75 logic instructions can use any of these ports. The most obvious bottleneck is 175 cycles for port 0. The most obvious scheduling challenge is to avoid having port 0 distracted by the logic operations. Our current software uses 219 cycles for the main loop.

7.7 Appendix: Lattice techniques

The maximum possible speedup from the following idea is small and as far as we can tell is outweighed by overhead, so we do not use it in our software, but we briefly describe the idea because it might be useful in other contexts.

One could scale $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$ so that each limb is smaller, hopefully small enough to eliminate the need for carry chains in the relevant **M**. There are 24 limbs (on Sandy Bridge) and approximately 2^{127} possible scalings, so one would expect a scaling to exist that makes all the limbs 5 bits smaller. However, finding this scaling appears to be a medium-dimensional lattice problem that would cost more to solve than it could possibly save. Scaling to four integers below 2^{96} would be a much easier lattice computation and would save the multiplications by top coefficients, but still does not appear to be worthwhile.

For comparison, scaling $(1/x_1 : 1/y_1 : 1/z_1 : 1/t_1)$ to $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$ is a one-dimensional lattice problem. The potential advantage of the higher-dimensional lattices in the previous paragraph is that they are compatible with our vectorization across the four coefficients.

7.8 Appendix: Fixed-base scalar multiplication

> *There is no doubt that the 'grail' of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We* should *forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*
>
> —Donald E. Knuth, 1974 [Knu74]

The simplest way to generate a DH key is to apply our variable-base-point scalar-multiplication software to the fixed base point

$$(x/y : x/z : x/t) = (6 : 142514137003289508520683013813888233576 : 1)$$

of prime order (the prime ending 339). The software takes constant time, and in particular takes the same time for DH key generation as it does for DH shared-secret computation.

7.8.1 Optimizing fixed-base scalar multiplication

The simplest approach is not the fastest approach. It is well known that fixed-base scalar multiplication, and in particular DH key generation, can be made much faster than variable-base scalar multiplication.

The standard way to speed up fixed-base scalar multiplication in genus 2 is to precompute various multiples of the base point on the Jacobian. For Jacobian addition formulas see, e.g., [BCHL13a] and [HC14]. An alternate "hyper-and-elliptic" approach proposed recently in [BL14a] is to carry out fixed-base scalar multiplication using precomputed points on an auxiliary elliptic curve, taking advantage of the speed of elliptic-curve additions, and then map from the elliptic curve to the Jacobian; this does not work with the Kummer surface that we used, but it does work with other small-coefficient Kummer surfaces constructed in [BL14a]. Mapping from Jacobian to Kummer takes a few dozen additional multiplications.

For memory-limited platforms there are still significant speedups from a small number of precomputed points. For example, precomputing just 4 points reduces the number of doublings by a factor of 4. When the number of precomputed points is sufficiently small we could also carry out lattice precomputations as in Section 7.7 to noticeably reduce the size of the projective representations of those points; alternatively, we could generate new base points meeting various size criteria.

7.8.2 Reusing DH keys

There are two major types of DH keys: *long-term* (or "static") DH keys used as traditional identifiers, and *ephemeral* DH keys that are erased to provide forward secrecy. For example, HTTPS supports certified long-term "ECDH" SSL keys assigned to web servers, and it also supports ephemeral "ECDHE" SSL keys that provide forward secrecy.

A long-term DH key involves just one fixed-base scalar multiplication (for key generation) and is then bottlenecked by variable-base scalar multiplications using public keys of other parties. In this context it is obvious that there is negligible benefit in speeding up fixed-base scalar multiplication.

The same optimization applies to ephemeral DH: an implementor who finds that key-generation time is a bottleneck can simply reuse ephemeral DH keys. If a DH key is reused just 1000 times, and key generation is implemented in the simplest way as variable-base scalar multiplication, then key generation is below 0.1% of the total DH cost. Of course, standard fixed-base speedups (see above) make DH key generation another few times faster, but this has negligible benefit. The critical speedup comes from key reuse.

As a concrete example, Microsoft's SSL library (SChannel) reuses ephemeral DH keys for 2 hours, according to [CFN+14, page 8]. Even if the reuse intervals were drastically shortened, and keys were discarded after just 2 seconds, the ephemeral key-generation time (a small fraction of a millisecond using ECC or HECC) would be completely unnoticeable.

These are not new observations: they are the reason that the cost of variable-base scalar multiplication is the primary metric used in the DH literature. For example, [Ber06a] ("new Diffie–Hellman speed records") mentions that fixed-base speedups exist and then says that these speedups are "negligible in the Diffie–Hellman context . . . since each key is used many times". As another example, when [BCHL13a] ("Fast cryptography") advertises a "new software speed record" for "constant-time scalar multiplication", it is referring to variable-base scalar multiplication; [BCHL13a] says far less about fixed-base performance. As yet another example, when [CHS14] ("Faster compact Diffie–Hellman") advertises "fast elliptic curve scalar multiplication, optimized for Diffie–Hellman Key Exchange", it is referring to variable-base scalar multiplication.

Of course, in contexts where it is important for every key to be erased as quickly as possible, or where any key reuse would give away valuable metadata, one should switch to the simplest form of ephemeral DH. This means carrying out one fixed-base scalar multiplication to generate a single-use key, and one variable-base scalar multiplication to generate a shared secret. Even in this (arguably important) corner of the DH universe, fixed-base scalar multiplication (with the standard optimizations) consumes relatively little time, so making variable-base scalar multiplication N% faster is more important than making fixed-base scalar multiplication N% faster.

Fixed-base scalar multiplication is far more important outside the DH context: for example, it is the primary bottleneck in signature generation. See, e.g., [BDL+11]. We do not discuss this further: this chapter focuses on DH.

8

PandA: Pairings and Arithmetic

Since the late 1990s and early 2000s, when Ohgishi, Sakai, Kasahara [OSK99, SOK00, SOK01] and Joux [Jou00, Jou04] presented the first constructive uses of cryptographic pairings, many pairing-based cryptographic protocols have been proposed. Early work such as the identity-based encryption scheme by Boneh and Franklin [BF01] and the short signature scheme by Boneh, Lynn and Shacham [BLS04], were followed by a flood of papers presenting more and more pairing-based schemes with exciting, new cryptographic functionalities. Examples include schemes for hierarchical identity-based encryption [HL02, GS02], attribute-based encryption [SW05], systems for non-interactive zero-knowledge proofs [GS12b, Gro10], and randomizable proofs and anonymous credentials [BCC⁺09].

In a highly related—but often somewhat independent—line of research, the performance of pairing computation was drastically improved. Milestones in this line of research were the construction of various families of pairing-friendly curves (for an overview, see [FST10]), many optimizations for the pairing algorithm including denominator elimination in the Miller loop [BKLS02], faster algorithms to compute the final exponentiation [SBC⁺09], and the introduction of loop-shortening techniques [HSV06], that led to the notion of *optimal pairings* [Ver10]. Recently, several papers presented high-speed software that computes 128-bit secure pairings for various Intel and AMD processors [NNS10, BGM⁺10, AKL⁺11, Mit13], and for ARM processors with NEON support [SRH13]. These efforts reduced the time required to compute a pairing at the 128-bit security level on current processors to below 0.5 ms.

Unfortunately, these advances in pairing performance do not immediately speed up pairing-based protocols. The reason is that protocols need much more than just fast pairings. They need fast arithmetic in all involved groups, fast hashing into elliptic-curve groups, fast multi-scalar multiplication (and multi-exponentiation), or specific optimizations for computing products of pairings. This means that, even if authors of speed-record papers for pairing computation make their software available, this software is typically not “complete” from a protocol designer’s point of view, and does not necessarily include these other operations; and it is often not easy to use when it comes to prototyping a new pairing-based protocol to evaluate its practical performance. Also, once a protocol implementation has settled for one pairing library, it typically requires a significant effort to switch to another software or library.

Furthermore, as Scott points out in [Sco11], which optimizations to the pairing computation or other arithmetic operations are most useful, strongly depends on the pairing-based protocol that is being implemented. Pairings are used in such protocols in different flavors, where in some scenarios pairing computation is the dominant cost in the overall protocol and in others the large number of non-pairing operations may be the bottleneck (see, for example, [PGHR13]). If the protocol contains many more group exponentiations than it has pairing computations, in some cases it might even make sense to choose different pairing-friendly curves to allow faster group operations at the cost of a slightly more expensive pairing (see the ratios of group exponentiation and pairing costs in [BCN13]). In an implementation that has been tailored for high-speed pairings only, it is often difficult to account for such tradeoffs.

This chapter introduces PandA, a software framework that intends to address the above concerns by making improvements in pairing (and more generally group-arithmetic) performance easily usable for protocol designers. The project is inspired by the eBACS benchmarking project [BLa] that defines APIs for various typical cryptographic primitives and protocols (such as hash functions, stream ciphers, public-key encryption, and cryptographic signatures). PandA can be seen as a generalization of eBACS to lower-level functions in the elliptic-curve and pairing setting.

Credits. The content of this chapter is based on the paper *“PandA: Pairings and Arithmetic”* [CNRS13] which is a joint work with Michael Naehrig, Pance Ribarski, and Peter Schwabe. Note that “we” in this chapter is used to refer to the aforementioned authors.

Organization of the chapter. Section 8.1 describes an overview of the PandA framework. Section 8.2 explains the PandA API. Section 8.3 gives details of our reference implementation of this API and reports benchmark results of all arithmetic operations. Section 8.4 considers BLS signatures as an example that shows how easy it is to implement pairing-based protocols that achieve state-of-the-art performance using the PandA API.

8.1 PandA framework

This chapter introduces PandA, a software framework for *Pairings and Arithmetic*. It is designed to bring together advances in the efficient computation of cryptographic pairings and the development and implementation of pairing-based protocols. The intention behind the PandA framework is to give protocol designers and implementors easy access to a toolbox of all functions needed for implementing pairing-based cryptographic protocols, while making it possible to use state-of-the-art algorithms for pairing computation and group arithmetic. PandA offers an API in the C programming language and all arithmetic operations run in constant time to protect against timing attacks. The framework also makes it easy to consistently test and benchmark the lower-level functions used in pairing-based protocols.

8.1.1 Type-1, Type-2 and Type-3 pairings

Currently our reference implementation of the PandA API only implements a particular set of parameters for Type-3 pairings, but the API is designed to support arbitrary pairing-friendly curves. However, Section 8.2 explains how the API supports also Type-1 pairings. Until recently the standard approach to implementing high-security (e.g., 128-bit secure) Type-1 pairings was using supersingular curves over binary or ternary fields. However, advances on solving discrete-logarithm problems in multiplicative groups of small-characteristic fields by Joux in [Jou13], by Göloğlu, Granger, McGuire, and Zumbrägel in [GGMZ13], by Barbulescu, Gaudry, Joux, and Thomé in [BGJT13], and by Adj, Menezes, Oliveira, and Rodríguez-Henríquez in [AMORH13] have raised serious concerns about the security of such constructions. Granger commented that he does not “think the coffin has been firmly nailed shut just yet!” [1], and it is indeed not clear that all small-characteristic pairings are broken, but there is a strong consensus that pairings on curves over small-characteristic fields are not recommended anymore. We are therefore planning to include a reference implementation of Type-1 pairings that uses an approach similar to the ones described in [TSK$^+$14] and [ZW14].

We follow Chatterjee and Menezes stating in [CM11] that “Type 2 pairings are merely inefficient implementations of Type 3 pairings, and appear to offer no benefit for protocols based on asymmetric pairings from the point of view of functionality, security, and performance”. Thus, we do not explicitly support Type-2 pairings, but it would be straightforward to include Type-2 pairings in PandA (the only difference from an API perspective is missing hashing into the second group of pairing arguments).

The Type-3 pairing setting in this chapter is as follows. The pairing is a non-degenerate, bilinear function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_1$ and $\mathbb{G}_2$ are groups of prime order r consisting of rational points on an ordinary, pairing-friendly elliptic curve E defined over a finite field $\mathbb{F}_p$ of prime characteristic p. The elliptic curve E has a small embedding degree k, which means that the group $\mathbb{G}_T$ is the group of r-th roots of unity in the multiplicative group $\mathbb{F}_{p^k}^*$, i.e., all three groups have prime order r.

8.1.2 Arithmetic in non-pairing groups

PandA also has an API for arithmetic in groups that do not support efficient computation of pairings (like non-pairing-friendly elliptic curves). If protocols do not need

---

[1] see `http://ellipticnews.wordpress.com/2013/05/22/joux-kills-pairings-in-characteristic-2/`

efficient pairing computation they can choose from a much larger pool of groups in which the DLP is hard. When choosing from this larger pool one can typically pick groups with more efficient arithmetic. The group API supports all functions that are also supported for each of the three groups in the pairing setting. Our reference implementation of this API uses the group of the twisted Edwards curve that is also used for Ed25519 signatures [BDL+11, BDL+12]. However, this chapter focuses on the description of the pairing setting in PandA.

8.1.3 The importance of constant-time algorithms

Aside from attacks against the hard problems that the security of modern cryptography is based on, major threats to cryptographic software are side-channel attacks. In particular timing attacks (that can even be carried out remotely in many cases) prove to be a very powerful attack tool. See [OST06, TOS10], [BT11], [AP13], [YF14] for some examples of timing attacks against cryptographic software.

One could argue that a framework which is designed to evaluate the performance of cryptographic protocols should not pay attention to these issues, but rather keep the API simple, and add suitable timing-attack protection only for "real-world" software. We disagree for two reasons. First, once some pieces of unprotected cryptographic software have been written and publicized, it is almost impossible to ensure that it does not end up in some real-world software. Second, and more importantly, protecting software against timing-attacks does not add a constant overhead; the cost highly depends on protocol design, and algorithm and parameter choices made on a high level. For example, the completeness of the group law on Edwards curves [BL07, BBJ+08] makes it easy to protect group addition against timing attacks. It is possible to protect Weierstrass-curve point addition against timing attacks (see Section 8.3) but it involves a significant overhead.

Optimizing performance of unprotected implementations of cryptographic protocols may thus lead to wrong decisions that are very hard to correct later. PandA acknowledges this fact by offering timing-attack protected (constant-time) versions of all arithmetic operations. For operations that do not involve any secret data (such as signature verification) there are possibly faster non-constant-time versions of all group-arithmetic operations. These unprotected versions of functions have to be chosen explicitly; the default is the constant-time versions.

8.1.4 Related work

There exist various cryptographic *libraries* that expose low-level functionality such as group arithmetic and pairings through their API. However, the API that gives access to this low-level functionality is typically tailored to suit the specific needs of the higher-level primitives of the library. It is usually not designed for efficient implementation of arbitrary new protocols. Some libraries that use group arithmetic even decide to not expose the low-level functionality through the API, because this functionality was never written to be used outside the specific needs of the higher-level protocols. See, for example, the high-level API of NaCl [BLS12]. Two notable examples of cryptographic libraries with a convenient API for pairings and group arithmetic are RELIC [AG] and Miracl [Cer].

A library which has been explicitly designed for the use in arbitrary pairing-based protocols is the PBC library [Lyn]. This careful design is the reason that it is still the preferred library for the implementation of various protocols; despite the fact that it does not offer state-of-the-art performance and (by default) no high-security curves. The PandA API is designed with the same usage profile as PBC in mind. However, the reference implementation of the PandA API presented in this chapter offers state-of-the art performance with a curve choice that offers 128 bits of security. Furthermore, PandA is designed as a framework that supports submissions by various designers to keep reflecting the state-of-the-art in group-arithmetic and pairing performance.

Another framework for easy implementation of cryptographic protocols is Charm [AGM+13]. Charm offers a high-level Python API and uses multiple cryptographic libraries to achieve good performance. For pairing-based cryptography it uses the PBC library. Charm is a higher-level framework than PandA; we see PandA not in competition to Charm but rather hope that Charm will eventually include some of PandA's high-performance pairing and group-arithmetic implementations to speed up protocols implemented in its high-level API.

8.2 PandA API and functionality

The API of PandA is inspired by the API of eBACS, which means in particular that the API is also for the C programming language. There are various advantages of using C. It is the language most commonly used for speed-record-setting cryptographic software (often combined with assembly), so a C API makes it easy to integrate fast software into PandA. Furthermore, protocols that need group arithmetic, pairings, and, for example, a hash function or a stream cipher, can easily combine software from PandA with software that is tested and benchmarked in eBACS.

In the eBACS API all functions are within the `crypto` namespace, i.e., all function names begin with `crypto_`. Similarly, all functions and data types related to arithmetic in groups that support efficient bilinear-pairing computation are in the `bgroup` namespace (for "bilinear group"); the API for group arithmetic without pairings uses the `group` namespace.

8.2.1 PandA data types

The functionality that is tested and benchmarked in PandA is on a lower level in the design of cryptographic protocols. In the eBACS project, complete cryptographic primitives and protocols are benchmarked, while PandA benchmarks arithmetic operations that are meant to be used to implement cryptographic protocols. This has consequences for the data type of inputs and outputs. In eBACS, all functions receive inputs as byte arrays (C data type `unsigned char`), the length of these arrays is specified in arguments of type `unsigned long long`. Outputs are again written to byte arrays. A typical implementation of a cryptographic protocol in eBACS first converts the input byte arrays to an internal representation for fast computation that depends on the architecture, then performs all computations in this representation, and then transforms the output to a unique representation as a byte array. These transformations typically contribute only little overhead to the cost of a cryptographic protocol if they are done only at the beginning and the end of the *protocol*. Protocols implemented using the PandA API typically need a sequence of functions from the PandA API and we clearly want to avoid transformations at the beginning and the end of each *function*. Implementations of the PandA API therefore define 4 data types—for elements of the three groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ and for scalars (modulo the group order)—in a file called `api.h`. These data types (`struct` in C) are called `bgroup_g1e`, `bgroup_g2e`, `bgroup_g3e`, and `bgroup_scalar`. The API provides two functions, one is used to convert an element of $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, or a scalar to a unique byte array of fixed length (`pack`), the other one converts such a byte array back to a group element or scalar (`unpack`). Implementations of the PandA API furthermore specify the size of packed elements in `api.h`:

```
#define BGROUP_G1E_PACKEDBYTES 32
#define BGROUP_G2E_PACKEDBYTES 64
#define BGROUP_G3E_PACKEDBYTES 384
#define BGROUP_SCALAR_PACKEDBYTES 32
```

indicating that packed elements of $\mathbb{G}_1$ need 32 bytes, packed elements of $\mathbb{G}_2$ need 64 bytes, etc. According to this file, PandA automatically generates the header file `panda_bgroup.h` that defines all functions of the API. Implementations of Type-1 pairings omit the implementation of $\mathbb{G}_2$ and instead include a line

```
#define BGROUP_TYPE1
```

in the file `api.h`. For the group $\mathbb{G}_1$, the `unpack` and `pack` functions are

```
int bgroup_g1e_unpack(
          bgroup_g1e *r,
          const unsigned char b[BGROUP_G1E_PACKEDBYTES]);

void bgroup_g1e_pack(
          unsigned char r[BGROUP_G1E_PACKEDBYTES],
          const bgroup_g1e *b);
```

Following eBACS convention, the `unpack` function returns an integer value, which is zero whenever a valid byte array is received that can be unpacked to a group element. On input of an invalid byte array that does not correspond to a packed group element, the function returns a nonzero integer. In the following, we mostly describe the API for arithmetic in $\mathbb{G}_1$ as an example. Equivalent functions exist for $\mathbb{G}_2$ and $\mathbb{G}_T$.

8.2.2 PandA constants

For each of the three groups, a PandA implementation has to define two constants, namely, a generator and the neutral element. For the group $\mathbb{G}_1$ these are called `bgroup_g1e_base` and `bgroup_g1e_neutral`. Each implementation needs to ensure that the pairing evaluated at `bgroup_g1e_base` and `bgroup_g2e_base` gives `bgroup_g3e_base` as result. Furthermore, each PandA implementation has to define two constants of type `bgroup_scalar` for the element zero and the element one in the ring of integers modulo the order r of the groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$. These constants are called `bgroup_scalar_zero` and `bgroup_scalar_one`.

8.2.3 Comparing group elements

One way to compare two group elements for equality is to use the `bgroup_g1e_pack` function on both of them and compare the resulting byte arrays for equality. This is typically not the most efficient way to compare equality (except if packing of elements is required anyway). For example, consider two elliptic-curve points in projective coordinates. Conversion to a *unique* byte array requires transformation to affine coordinates, i.e., two inversions and several multiplications. Comparison for equality only needs a few multiplications. The API therefore has a comparison function

```
int bgroup_g1e_equals(const bgroup_g1e *a, const bgroup_g1e *b);
```

which returns 1 if the two elements are equal and 0 if they are not.

As explained in the introduction, this function must be guaranteed to not leak timing information about the two arguments. For cases where none of the two inputs is secret, there is a function

```
int bgroup_g1e_equals_publicinputs(const bgroup_g1e *a, const bgroup_g1e *b);
```

which behaves the same way but is not guaranteed to not leak timing information and may be faster than the constant-time version.

8.2.4 Addition and doubling

In concrete implementations of pairings, the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are typically additive groups, while the group $\mathbb{G}_T$ is a multiplicative group. Hence, the core operations for group arithmetic are additions and doublings in $\mathbb{G}_1$ and $\mathbb{G}_2$ and multiplications and squarings in $\mathbb{G}_T$. It makes sense to treat all three groups as abstract abelian groups and therefore use a common notation for the group operation in all of them. Many papers that treat a pairing as a black box use multiplicative notation for $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$. Instead, the PandA API uses additive notation following the `crypto_scalarmult` API of the SUPERCOP benchmarking framework used in eBACS.

Addition of two elements, doubling, and negation (computing the inverse of an element) are done through the functions:

```
void bgroup_g1e_add(bgroup_g1e *r, const bgroup_g1e *a, const bgroup_g1e *b);
void bgroup_g1e_double(bgroup_g1e *r, const bgroup_g1e *a);
void bgroup_g1e_negate(bgroup_g1e *r, const bgroup_g1e *a);
```

Note that the return value is always written to the first argument pointer (as in the eBACS API and also, for example, in the GMP API [Gra]). Note also that the implementation needs to ensure that the addition and doubling functions work for all elements of the group as inputs and that no timing information leaks about these inputs or the output. As before, there are also potentially faster non-constant-time versions of these functions:

```
void bgroup_g1e_add_publicinputs(bgroup_g1e *r,const bgroup_g1e *a,const bgroup_g1e *b);
void bgroup_g1e_double_publicinputs(bgroup_g1e *r,const bgroup_g1e *a);
void bgroup_g1e_negate_publicinputs(bgroup_g1e *r,const bgroup_g1e *a);
```

8.2.5 Scalar multiplication

The default function for performing a scalar multiplication is simply

```
void bgroup_g1e_scalarmult(bgroup_g1e *r,const bgroup_g1e *a,const bgroup_scalar *s);
```

This function can be made much faster when multiplying a fixed base point that is known at compile time. This potentially faster version is supported for the generator `bgroup_g1e_base` through

```
void bgroup_g1e_scalarmult_base(bgroup_g1e *r, const bgroup_scalar *s);
```

Another improvement can be implemented for multi-scalar multiplication, i.e., whenever a sum $\sum_{i=0}^{m-1} s_i P_i$ of several scalar multiples needs to be computed for m scalars $s_0, \ldots, s_{m-1}$ and m group elements $P_0, \ldots, P_{m-1}$. Such computations are supported through the function below, in which the last (`unsigned long long`) argument specifies the number m of scalar multiplications to be performed in the sum.

```
void bgroup_g1e_multiscalarmult(
                bgroup_g1e *r, const bgroup_g1e *a,
                const bgroup_scalar *s, unsigned long long alen);
```

Again, all group elements have to be supported as inputs, constant-time behavior has to be ensured by implementations, and the API also supports non-constant-time (`publicinputs`) versions of the functions. The input `alen` is considered public also for the constant-time version.

8.2.6 Hashing to $\mathbb{G}_1$ and $\mathbb{G}_2$

Many protocols require hashing of arbitrary bit strings to group elements in $\mathbb{G}_1$ and $\mathbb{G}_2$, which is also supported by the PandA API. The corresponding function for hashing into $\mathbb{G}_1$ is:

```
void bgroup_g1e_hashfromstr(
                bgroup_g1e *r, const unsigned char *a, unsigned long long alen);
```

As for the previous functions, there is also a non-constant-time (`publicinputs`) version of this function. Due to the different ways in which the constant-time and non-constant-time functions are computed, it can be the case that the hashed values obtained by evaluating each version on the same input bit string are different. It is not necessary to insist that both versions compute the same result, because we expect that throughout a protocol, the same input string to a hash function is always either public or private. Therefore, one can consistently select the right version of the function and thus take advantage of faster non-constant-time algorithms.

8.2.7 Arithmetic on scalars

Various functions are supported for arithmetic on scalars modulo the group order, which are required in various protocols (for example, ECDSA signatures). Specifically these functions are the following:

```
void bgroup_scalar_setrandom(bgroup_scalar *r);
void bgroup_scalar_add(bgroup_scalar *r,const bgroup_scalar *s,const bgroup_scalar *t);
void bgroup_scalar_sub(bgroup_scalar *r,const bgroup_scalar *s,const bgroup_scalar *t);
void bgroup_scalar_negate(bgroup_scalar *r,const bgroup_scalar *s);
void bgroup_scalar_mul(bgroup_scalar *r,const bgroup_scalar *s,const bgroup_scalar *t);
void bgroup_scalar_square(bgroup_scalar *r,const bgroup_scalar *s);
void bgroup_scalar_invert(bgroup_scalar *r,const bgroup_scalar *s);
int  bgroup_scalar_equals(const bgroup_scalar *s,const bgroup_scalar *t);
```

Arithmetic on scalars is typically not the performance bottleneck in pairing-based protocols; furthermore we do not expect significant speedups for non-constant-time versions of scalar arithmetic. Therefore, the API does not include `publicinputs` versions of functions for arithmetic on scalars.

8.2.8 Pairings and products of pairings

Finally, the API function for computing a pairing is

```
void bgroup_pairing(bgroup_g3e *r, const bgroup_g1e *a, const bgroup_g2e *b);
```

Some protocols need—or can make use of—the product of several pairings (for an example see BLS signatures in Section 8.4). Computing the product of two pairings can be significantly faster than computing two independent pairings and then multiplying the results. One reason is that the final exponentiation has to be done only once; another reason is that squarings inside the Miller loop can be shared between the two pairings. To support these important speedups, the PandA API includes a function

```
void bgroup_pairing_product(
            bgroup_g3e *r, const bgroup_g1e *a, const bgroup_g2e *b,
            unsigned long long alen);
```

providing the product of `alen` pairings.

8.3 PandA reference implementation

This section describes our reference implementation of the API functions from Section 8.2. The implementation provides a 128-bit secure, Type-3 pairing framework.

8.3.1 Choice of parameters

At the 128-bit security level, the most suitable choice of pairing-friendly curve is a Barreto-Naehrig curve [BN06] over a prime field of size roughly 256 bits. We use the 254-bit curve $E = E_{2,254}$ that has been proposed in [PJNB11] and has also been used in [AKL$^+$11]. The curve parameter $u = -(2^{62} + 2^{55} + 1)$ yields 254-bit primes $p = p(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1$ and $r = r(u) = 36u^4 + 36u^3 + 18u^2 + 6u + 1$, and $E : y^2 = x^3 + 2$ over $\mathbb{F}_p$. Since the embedding degree is $k = 12$, the implementation needs to provide the field extension $\mathbb{F}_{p^{12}}$. This extension is implemented in the standard way as a tower $\mathbb{F}_p \subset \mathbb{F}_{p^2} \subset \mathbb{F}_{p^6} \subset \mathbb{F}_{p^{12}}$.

As usual, the elliptic-curve groups are $\mathbb{G}_1 = E(\mathbb{F}_p)$ and $\mathbb{G}_2$ is the p-eigenspace of the p-power Frobenius in the r-torsion group $E(\mathbb{F}_{p^{12}})[r]$, which is represented by an isomorphic group $\mathbb{G}_2' = E'(\mathbb{F}_{p^2})[r]$, where E' is defined over $\mathbb{F}_{p^2}$ and is a sextic twist of E, i.e., they are isomorphic over $\mathbb{F}_{p^{12}}$. Whenever we work with elements in $\mathbb{G}_2$, we make use of their representation as elements in $\mathbb{G}_2'$, i.e., they are curve points with coefficients in $\mathbb{F}_{p^2}$ and arithmetic is actually arithmetic on E' over $\mathbb{F}_{p^2}$.

8.3.2 Algorithms

Packing and unpacking. To pack elements of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$, we use the usual way of point compression on elliptic curves. For elliptic-curve arithmetic, points are in Jacobian coordinates. To pack a point, it is first transformed to affine coordinates. The packed representation is the 32-byte array containing the point's 254-bit affine x-coordinate together with the least significant bit of its y-coordinate in one of the remaining two free bits. The other free bit is used to represent the point at infinity.

Given such a byte array, the unpacking algorithm recovers the x-coordinate and solves the curve equation for the y-coordinate, choosing the right square root according to the least significant bit given in the array. The core of this operation is a square root computation, for which we use different algorithms in $\mathbb{G}_1$ and $\mathbb{G}_2$. Since $p \equiv 3 \bmod 4$, in $\mathbb{G}_1$, we use $a^{(p+1)/4}$ to compute the square root of $a \in \mathbb{F}_p$. The unpack algorithm in $\mathbb{G}_2$ uses [AR14, Algorithm 9] to compute the square root. After obtaining a point on the curve, it needs to be checked whether it has order r, i.e., whether it is in the correct subgroup.

The elements of $\mathbb{G}_T$ are kept as elements of $\mathbb{F}_{p^{12}}^*$. The packing algorithm constructs a unique byte array composed of the twelve $\mathbb{F}_p$-coefficients of the unique $\mathbb{F}_{p^{12}}$-element in $\mathbb{G}_T$. The unpack algorithm simply converts the byte array back to an $\mathbb{F}_{p^{12}}$-element and checks that the order of the element is r. Pairing values can be compressed to one third the length of an $\mathbb{F}_{p^{12}}$-element by using the techniques described in [SB04, GPS04, NBS08, AKL$^+$11].

Comparison. To compare elements of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$, we need to compare points that are represented in (projective) Jacobian coordinates. The standard way of comparing these redundant representations is to multiply through by the respective powers of the Z-coordinate. This does not need inversions, in contrast to a conversion to affine coordinates. Comparison in the group $\mathbb{G}_T$ can directly compare $\mathbb{F}_{p^{12}}$-elements or the respective compressed representations.

Hashing to $\mathbb{G}_1$ and $\mathbb{G}_2$. The standard non-constant-time algorithm to hash an arbitrary string to a point on an elliptic curve is the "try-and-increment" method introduced in [BLS04]. The message is concatenated with a counter and hashed by a cryptographic hash function to an element of the underlying finite field. If this element is a valid x-coordinate, compute one of the corresponding y coordinates; otherwise increase the counter and repeat the procedure. We use this method for non-constant-time hashing to $\mathbb{G}_1$ and $\mathbb{G}_2$.

For constant-time hashing to $\mathbb{G}_1$ and $\mathbb{G}_2$ we use the algorithm described in [FT12] which is based on the algorithm by Shallue and van de Woestijne [SvdW06]. The conditional branches in the algorithm (in particular choosing between one out of three possible solutions) are implemented through constant-time conditional-copy operations.

Group addition. We represent elements of $\mathbb{G}_1$ and $\mathbb{G}_2$ in Jacobian coordinates. For non-constant-time addition we use the addition formulas by Bernstein and Lange that take 11 multiplications and 5 squarings [2]. If the inputs happen to be one of the special cases that are not handled by the formulas we use conditional branches to switch to doubling or to returning the point at infinity. Doubling uses the formulas by Lange that take 5 squarings and 2 multiplications [3].

Constant-time complete addition on a Weierstrass curve is not easy to do efficiently. There exist no complete formulas [BHWL95, Theorem 1]. The unified formulas proposed in [Hış10, 5.5.2] can handle doublings but they achieve this by moving the special cases to other points (specifically, addition of points of the form (x_1, y_1) and $(x_2, -y_1)$ with $x_1 \neq x_2$). Here, we evaluate two sets of formulas and use constant-time conditional copies to choose between the two outputs. We do that with the addition and doubling formulas described above. Note that protocols are typically not bottlenecked by additions but rather by scalar multiplications. Constant-time scalar-multiplication can use much faster dedicated addition as long as we can be sure that scalars are smaller than the group order. This is also compatible with the GLV/GLS decomposition described in the next paragraph.

Scalar multiplication. For the scalar multiplication algorithms that we implemented in PandA for each of the three groups, we distinguish between constant-time algorithms and their more efficient counterparts public inputs. For each case, there are three algorithms: general scalar multiplication, scalar multiplication of a fixed base point, and multi-scalar multiplication.

Scalar multiplication of a fixed base point that is known at compile time is done by precomputing 512 multiples of that point in a table and then using these to compute the scalar multiple. The method we use is described in detail by Bernstein et al. [BDL⁺11, BDL⁺12, Section 4]. Since we do not expect a significant speedup by

---

[2] `http://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#addition-add-2007-bl`

[3] `http://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#doubling-dbl-2009-l`

moving from the constant-time to a variable-time version, we also use the constant-time algorithm in the function on public inputs.

The standard case of scalar multiplication uses efficient endomorphisms on the BN curve by splitting the scalars via 2-dimensional GLV in $\mathbb{G}_1$ and 4-dimensional GLS decomposition in $\mathbb{G}_2$ and $\mathbb{G}_T$. See the work by Bos, Costello, and Naehrig [BCN13] for details. In $\mathbb{G}_1$, we slightly differ from the method in [BCN13]. After the scalar decomposition in the constant-time function, we save a few additions by using a signed fixed window of width 5 and two additions per lookup, instead of the table with window width 2 and one addition. The function on public inputs uses a signed sliding window of width 5. The constant-time algorithms in $\mathbb{G}_2$ and $\mathbb{G}_T$ are as described in [BCN13], the variable-time algorithms use signed sliding windows of width 4.

The variable-time algorithm for multi-scalar multiplication begins by applying the GLV/GLS scalar decomposition. For small batch sizes, we then use joint-signed-sliding-window scalar multiplication; for larger batch sizes (> 16 for $\mathbb{G}_1$ and > 8 for $\mathbb{G}_2$ and $\mathbb{G}_T$) we use Bos-Coster scalar multiplication (described in [dR95, Section 4]). For the constant-time version, due to the slow complete addition routine, the function currently simply carries out each scalar multiplication separately and adds them together at the end. For the group $\mathbb{G}_T$, it seems worthwhile to implement exponentiations of compressed values using the methods of Stam and Lenstra [SL02]. We are planning to consider this optimization.

Pairing computation. The pairing algorithm computes the optimal ate pairing on the same BN curve as [AKL⁺11]. Unlike [AKL+11] we do not use standard projective coordinates but Jacobian coordinates as in [NNS10]. We use lazy reduction for arithmetic in the extension fields as described in [AKL+11]. The final exponentiation implements the same approach as [BGM+10], we use the cyclotomic squarings from [GS10, Section 3.1], but we do not use the compressed squarings described in [AKL+11, Section 5.2].

Low-level arithmetic. The low-level arithmetic in $\mathbb{F}_p$ and arithmetic on scalars are implemented in `AMD64` assembly. We use Montgomery representation for elements in $\mathbb{F}_p$; scalars are represented in "standard" form because in scalar multiplication we need access to the binary representation. Modular reduction of scalars uses Barrett reduction [Bar86].

We have not yet implemented inlined arithmetic in $\mathbb{F}_{p^2}$ in assembly. We are planning to include this optimization and expect significant performance improvements for pairing computation and for arithmetic in $\mathbb{G}_2$ and $\mathbb{G}_T$.

We also have a compatible implementation of the field arithmetic entirely written in C to support other platforms. We will continue to optimize the software with assembly implementations for other platforms, in particular ARM processors with NEON support.

8.3.3 Performance

We benchmarked our software (with $\mathbb{F}_p$ arithmetic implemented in assembly) on one core of an Intel Core i5-3210M processor with Turbo Boost and Hyperthreading disabled. For each function we carried out 100 computations on random inputs. The median and quartiles of the cycle counts measured in these experiments are reported in Tables 8.1, 8.2, 8.3, and 8.4.

Table 8.1: Cycle counts for arithmetic operations in G_1 on Intel Core i5-3210M.

API function	**25% quartile**	**median**	**75% quartile**
`bgroup_g1e_unpack`	39140	39184	39212
`bgroup_g1e_pack`	39512	39548	39568
`bgroup_g1e_hashfromstr (59 bytes)`	198780	198908	198964
`bgroup_g1e_add`	6052	6080	6100
`bgroup_g1e_double`	1204	1216	1224
`bgroup_g1e_negate`	36	36	40
`bgroup_g1e_scalarmult`	346852	347024	347180
`bgroup_g1e_scalarmult_base`	128468	128596	128696
`bgroup_g1e_multiscalarmult`			
($n = 2$)	705564	705820	706056
($n = 3$)	1058308	1058644	1059128
($n = 4$)	1411188	1411644	1411944
($n = 8$)	2822252	2823148	2826864
($n = 32$)	11294736	11296364	11298420
($n = 128$)	45181816	45186732	45193356
`bgroup_g1e_equals`	1124	1132	1140
`bgroup_g1e_hashfromstr_publicinputs (59 bytes)`	41752	83168	83696
`bgroup_g1e_add_publicinputs`	2456	2468	2476
`bgroup_g1e_double_publicinputs`	1180	1192	1200
`bgroup_g1e_negate_publicinputs`	36	36	40
`bgroup_g1e_scalarmult_publicinputs`	284228	288240	290788
`bgroup_g1e_scalarmult_base_publicinputs`	102184	104024	105772
`bgroup_g1e_multiscalarmult_publicinputs`			
($n = 2$)	415076	419860	423440
($n = 3$)	551124	556792	560712
($n = 4$)	710416	715396	722000
($n = 8$)	1229100	1238660	1246568
($n = 32$)	4727808	4741472	4752772
($n = 128$)	14590168	14605364	14635184
`bgroup_g1e_equals_publicinputs`	576	580	588

8.4 Implementing protocols with PandA

In this section we consider BLS signatures [BLS04] as a small example of a pairing-based protocol implemented in PandA. We choose this example because it illustrates the use of most API functions of PandA and because cryptographic signatures (unlike more complex cryptographic protocols) are supported by the eBACS benchmarking project [BLa]. The software presented in this section implements the eBACS API for cryptographic signatures.

8.4.1 The BLS signature scheme

We briefly describe the three algorithms — key generation, signing, and verification — of the BLS scheme for an asymmetric, Type-3 pairing. Let $Q \in \mathbb{G}_2$ be a system-wide fixed base point for $\mathbb{G}_2$.

Table 8.2: Cycle counts for arithmetic operations in G_2 on Intel Core i5-3210M.

API function	**25% quartile**	**median**	**75% quartile**
`bgroup_g2e_unpack`	1864580	1864884	1865396
`bgroup_g2e_pack`	42080	42124	42160
`bgroup_g2e_hashfromstr (59 bytes)`	2435116	2435564	2439536
`bgroup_g2e_add`	16048	16072	16096
`bgroup_g2e_double`	2924	2940	2948
`bgroup_g2e_negate`	60	60	64
`bgroup_g2e_scalarmult`	764628	764808	765088
`bgroup_g2e_scalarmult_base`	336788	336916	337060
`bgroup_g2e_multiscalarmult`			
($n = 2$)	1563312	1563668	1564040
($n = 3$)	2344964	2345496	2346704
($n = 4$)	3126720	3127116	3131192
($n = 8$)	6253984	6257528	6258700
($n = 32$)	25024136	25027200	25031036
($n = 128$)	100103176	100117420	100157284
`bgroup_g2e_equals`	3100	3112	3124
`bgroup_g2e_hashfromstr_publicinputs (59 bytes)`	298524	299884	894696
`bgroup_g2e_add_publicinputs`	6572	6596	6608
`bgroup_g2e_double_publicinputs`	2960	2972	2992
`bgroup_g2e_negate_publicinputs`	60	60	64
`bgroup_g2e_scalarmult_publicinputs`	612012	625636	635656
`bgroup_g2e_scalarmult_base_publicinputs`	273468	278372	283056
`bgroup_g2e_multiscalarmult_publicinputs`			
($n = 2$)	1031736	1043332	1060796
($n = 3$)	1477392	1492796	1510148
($n = 4$)	1889684	1912744	1928124
($n = 8$)	3443640	3467764	3489032
($n = 32$)	10293932	10329088	10366420
($n = 128$)	32941972	32991824	33061804
`bgroup_g2e_equals_publicinputs`	3104	3116	3120

Key generation. Pick a random scalar $s \in \mathbb{Z}_r^*$. Compute the scalar multiple $R \leftarrow sQ$. Return R as the public key and s as the private key.

Signing. Hash the message m to an element M in $\mathbb{G}_1$. Use the private key s to compute $S = sM$. Return the x-coordinate of the result S as the signature σ.

Verification. Upon receiving a signature σ, a message m, and the public key R, find an element $S \in \mathbb{G}_1$ such that its x-coordinate corresponds to σ and it has order r. If no such point exists, reject the signature. Next calculate $t_1 \leftarrow e(S,Q)$. After that, compute the hash $M \in \mathbb{G}_1$ of the message m, and compute $t_2 \leftarrow e(M,R)$. The signature is accepted if $t_1 = t_2$ or $t_1 = -t_2$ and rejected otherwise. Note that we use additive notation in $\mathbb{G}_T$.

This scheme requires one scalar multiplication for key generation, one scalar multiplication for signature generation, and the computation and comparison of two pairing values for signature verification.

8.4.2 Implementation with PandA

Our example implementation follows the eBATS API which consists of three functions, namely, `crypto_sign_keypair`, `crypto_sign`, and `crypto_sign_open`. The

Table 8.3: Cycle counts for arithmetic operations in G_3 on Intel Core i5-3210M.

API function	**25% quartile**	**median**	**75% quartile**
`bgroup_g3e_unpack`	1832068	1832404	1833044
`bgroup_g3e_pack`	424	424	428
`bgroup_g3e_add`	8020	8032	8048
`bgroup_g3e_double`	5548	5560	5572
`bgroup_g3e_negate`	172	176	180
`bgroup_g3e_scalarmult`	1120300	1120552	1120936
`bgroup_g3e_scalarmult_base`	608964	609148	609320
`bgroup_g3e_multiscalarmult`			
($n = 2$)	2255028	2255624	2258896
($n = 3$)	3382628	3383284	3387392
($n = 4$)	4510336	4511420	4515516
($n = 8$)	9024924	9025736	9026820
($n = 32$)	36100180	36103240	36109596
($n = 128$)	144408660	144446076	144467856
`bgroup_g3e_equals`	8304	8324	8336
`bgroup_g3e_add_publicinputs`	8024	8044	8056
`bgroup_g3e_double_publicinputs`	5548	5556	5568
`bgroup_g3e_negate_publicinputs`	176	176	180
`bgroup_g3e_scalarmult_publicinputs`	852272	864136	877804
`bgroup_g3e_scalarmult_base_publicinputs`	609004	609188	609352
`bgroup_g3e_multiscalarmult_publicinputs`			
($n = 2$)	2255104	2255424	2258836
($n = 3$)	3382688	3383368	3387800
($n = 4$)	4510680	4512684	4515652
($n = 8$)	4272080	4297668	4330036
($n = 32$)	12768868	12803832	12843124
($n = 128$)	40764052	40825956	40876608
`bgroup_g3e_equals_publicinputs`	8304	8320	8332

Table 8.4: Cycle counts for pairing computation on Intel Core i5-3210M.

API function	**25% quartile**	**median**	**75% quartile**
`bgroup_pairing`	2566580	2567116	2572096
`bgroup_pairing_product`			
($n = 2$)	3831724	3832644	3837688
($n = 3$)	5089192	5093724	5094728
($n = 4$)	6347328	6351260	6352588
($n = 8$)	11380604	11381384	11383420
($n = 32$)	41565448	41569424	41588976
($n = 128$)	162321836	162364916	162387468

details of each function are as follows.

The function `crypto_sign_keypair` generates the public and private key pair. It requires one fixed-base scalar multiplication in $\mathbb{G}_2$. The complete code for keypair generation is given in Figure 8.1. The macro `CRYPTO_BYTES` is required by the eBACS API and is set to `BGROUP_G1E_PACKEDBYTES` in a file called `api.h`.

The function `crypto_sign` computes the signature upon receiving the message. This function also requires hashing to $\mathbb{G}_1$ (we assume that the message is public and use the `publicinputs` version) and one scalar multiplication in $\mathbb{G}_1$. The complete

```
int crypto_sign_keypair(
  unsigned char *pk,
  unsigned char *sk
)
{
  // private key //
  bgroup_scalar x;
  bgroup_scalar_setrandom(&x);
  bgroup_scalar_pack(sk, &x);

  // public key //
  bgroup_g2e r;
  bgroup_g2_scalarmult_base(&r, &x);
  bgroup_g2_pack(pk, &r);

  return 0;
}
```

Figure 8.1: Public and private key generation.

```
int crypto_sign(
  unsigned char *sm,
  unsigned long long *smlen,
  const unsigned char *m,
  unsigned long long mlen,
  const unsigned char *sk)
{

  bgroup_g1e p, p1;
  bgroup_scalar x;
  int i,r;

  bgroup_g1e_hashfromstr_publicinputs(&p, m, mlen);
  r = bgroup_scalar_unpack(&x, sk);
  bgroup_g1e_scalarmult(&p1, &p, &x);
  bgroup_g1e_pack(sm, &p1);

  for (i = 0; i < mlen; i++)
  sm[i + CRYPTO_BYTES] = m[i];
  *smlen = mlen + CRYPTO_BYTES;

  return -r;
}
```

Figure 8.2: Signature generation.

code for signing is given in Figure 8.2.

The function `crypto_sign_open` verifies whether the signature belongs to the message. A naive method to compare whether two pairing values are equal is to first compute those two pairings, then compare the results. It is obvious that one can avoid the computation of two pairings. Instead, one computes a product of two pairings and checks whether it is equal to one. In our case, the signature is the packed value of the elliptic-curve point, which includes the information on the sign of the correct y-coordinate. We therefore compute the unique point S corresponding to the signature σ. To verify, we only need to check whether $e(-S,Q) \cdot e(M,R) = 1$. This way, verification needs hashing to $\mathbb{G}_1$ and one pairing-product computation. The code

```
int crypto_sign_open(
  unsigned char *m,
  unsigned long long *mlen,
  const unsigned char *sm,
  unsigned long long smlen,
  const unsigned char *pk)
{

  bgroup_g1e p[2];
  bgroup_g2e q[2];
  bgroup_g3e r;
  unsigned long long i;
  int ok;

  ok  = !bgroup_g1e_unpack(p, sm);
  bgroup_g1e_negate_publicinputs(p, p);
  q[0] = bgroup_g2e_base;
  bgroup_g1e_hashfromstr_publicinputs(p+1, sm + CRYPTO_BYTES, smlen - CRYPTO_BYTES);
  ok &= !bgroup_g2e_unpack(q+1, pk);
  bgroup_pairing_product(&r, p, q, 2);

  ok &= bgroup_g3e_equals(&r, &bgroup_g3e_neutral);

  if (ok)
  {
    for (i = 0; i < smlen - CRYPTO_BYTES; i++)
      m[i] = sm[i + CRYPTO_BYTES];
    *mlen = smlen - CRYPTO_BYTES;
    return 0;
  }
  else
  {
    for (i = 0; i < smlen - CRYPTO_BYTES; i++)
      m[i] = 0;
    *mlen = (unsigned long long) (-1);
    return -1;
  }
}
```

Figure 8.3: Signature verification.

for signature verification is given in Figure 8.3.

8.4.3 Performance

We benchmarked the BLS implementation on the same Core i5-3210M running at 2.5 GHz that we used for the detailed benchmarks of our reference implementation of the API. Key generation takes 378848 cycles. Signing (of a 59-byte message) takes 434640 cycles (this is a median of 10000 measurements, the quartiles are 428616 and 511764). Verification of a signature on a 59-byte message takes 5832584 cycles (again, this is a median, the quartiles are 5797640 and 5874292). To our knowledge, these are the fastest reported speeds of a BLS signature implementation at the 128-bit security level. We would like to compare performance with the BLS implementation by Scott included in SUPERCOP. However, it seems that the software fails to build on 64-bit platforms; consequently eBACS does not contain benchmark results of the "bls" software on such platforms.

We ran the benchmark included in the RELIC framework (version 0.3.5) on the same machine that we used for benchmarking. The times reported by this RELIC benchmark are 609966 nanoseconds for BLS key generation, 510775 nanoseconds for signing and 6910615 nanoseconds for verification. At a clock speed of 2.5 GHz, this corresponds to 1524915 cycles for key generation, 1276937 cycles for signing, and 17276537 cyles for verification; about three times slower than our implementation.

9

NTRU Prime

This chapter presents an efficient implementation of a high-security **prime-degree large-Galois-group inert-modulus** ideal-lattice-based cryptosystem. “Prime degree” etc. are defenses against potential attacks; see Section 9.1. The reader can, if desired, skip Section 9.1 in favor of the following short summary:

- Rings of the form $(\mathbb{Z}/q)[x]/(x^p - 1)$, where p is a prime and q is a power of 2, are used in the classic NTRU cryptosystem [HPS98], and have none of the recommended defenses presented in this chapter.
- Rings of the form $(\mathbb{Z}/q)[x]/(x^p + 1)$, where p is a power of 2 and $q \in 1 + 2p\mathbb{Z}$ is a prime, are used in typical “Ring-LWE-based” cryptosystems such as [ADPS16], and have none of the recommended defenses presented in this chapter.
- Fields of the form $(\mathbb{Z}/q)[x]/(x^p - x - 1)$, where p is prime, are used in “NTRU Prime”, introduced in this chapter, and have all of the recommended defenses.

Specifically, the presented implementation takes only about 50000 cycles on one core of an Intel Haswell CPU for **constant-time** multiplication in the field $(\mathbb{Z}/9829)[x]/(x^{739} - x - 1)$.

This chapter defines a public-key cryptosystem called “Streamlined NTRU Prime 9829^{739}” using this field, aiming for the standard design goal of IND-CCA2 security at the standard 2^{128} **post-quantum** security level. This cryptosystem provides several implementation advantages and security-auditing advantages beyond the choice of ring: it has shorter ciphertexts than ring elements, for example, and it eliminates the annoying possibility of “decryption failures” that appear in most lattice-based cryptosystems. The security analysis indicates that Streamlined NTRU Prime 9829^{739} actually provides a large security margin beyond the targeted security level, compen-

Table 9.1: Comparison of multiplication results.

Dfn.	Con.	Cycles	Ring	Technique	Source
yes	yes	50000	$(\mathbb{Z}/9829)[x]/(x^{739}-x-1)$	Toom etc.	this chapter
no	yes	33000	$(\mathbb{Z}/12289)[x]/(x^{1024}+1)$	NTT	"new hope" [ADPS16],[LN16]
no	no	100000	$(\mathbb{Z}/2048)[x]/(x^{743}-1)$	sparse input	`ntruees743ep1` [Kum14]

Note: "Dfn." means that the ring has this chapter's defenses against potential attacks. "Con." means that the software runs in constant time. "Cycles" is approximate multiplication time on an Intel Haswell; see text for calculations. All rings are used in public-key cryptosystems aiming for $\geq 2^{128}$ post-quantum security.

sating for a severe lack of clarity in the literature regarding the actual cost of lattice attacks.

Multiplication is the main bottleneck in both encryption and decryption, so the performance of the presented software easily outperforms, e.g., pre-quantum Curve25519 as a public-key cryptosystem. The implementation takes advantage of Haswell's vectorized multiplier, but modern Curve25519 implementations such as [Cho15b] and [FL15] also do this, so it is also expected to outperform Curve25519 on other platforms.

The presented public keys are field elements, easily squeezed into 1232 bytes. This chapter explains how to further squeeze ciphertexts (transporting 256-bit session keys) into just 1141 bytes. Obviously these sizes are not competitive with 256-bit ECC key sizes, but they are small enough for many applications: for example, the presented ciphertexts fit into the 1500-byte Ethernet MTU for plaintexts up to a few hundred bytes, avoiding the implementation hassle of packet fragmentation.

The previous state of the art in implementations of lattice-based cryptography was last year's paper "Post-quantum key exchange: a new hope" [ADPS16] by Alkim, Ducas, Pöppelmann, and Schwabe. Most of the implementations before [ADPS16] are, in many views, obviously unsuitable for deployment because they access the CPU cache at secret addresses, taking variable time and allowing side-channel attacks. A reimplementation [LN16] by Longa and Naehrig of the system in [ADPS16] achieves better speeds on the same platform using an improved NTT. Gueron and Schlieker [GS16] improved the algorithm for generating a uniformly random polynomial obtaining further speedups on a newer Intel archtecture called Skylake. It does not appear to have anything faster than "new hope" at a high security level. For example, [BLa] reports 98904 Haswell cycles for `ntruees743ep1` encryption; almost all of this time is for multiplication in $(\mathbb{Z}/2048)[x]/(x^{743}-1)$ using variable-time sparse-polynomial-multiplication algorithms.

Like this chapter, [ADPS16] and [LN16] target the Haswell CPU, require constant-time implementations, and aim for more than 2^{128} post-quantum security. Unlike this chapter, [ADPS16] follows the classic NTRU/Ring-LWE tradition of using cyclotomic rings. More precisely, [ADPS16] uses the same type of ring $(\mathbb{Z}/q)[x]/(x^p+1)$ as previous Ring-LWE papers, specifically with $p = 1024$ and $q = 12289 = 12 \cdot 1024 + 1$.

An obvious disadvantage of requiring the lattice dimension p to be a power of 2, as in [ADPS16], is that security levels are quite widely separated. There is a "claim of 94 bits of post quantum security" in [ADPS16] for one dimension-512 system; it does not seem to have any dimension-512 system that is claimed today to reach the standard 2^{128} post-quantum security target. Jumping to the next power of 2, namely $p = 1024$, means at least doubling key sizes, ciphertext sizes, encryption time, etc. This severe discontinuity in the security-performance graph means that [ADPS16] is unable to offer any options truly comparable to the better-tuned $p = 743$ in `ntruees743ep1` or $p = 739$ in this chapter. Of course, it is possible to present this deficiency as a security feature, stating that $p = 1024$ offers a "large margin"; but dimension is only one contributing factor to security, and size does matter.

The conventional wisdom is that, despite this disadvantage, rings of the type used in [ADPS16] are particularly efficient. These rings allow multiplication at the cost of three "number-theoretic transforms" (NTTs), i.e., fast Fourier transforms over finite fields, with only a small overhead for "pointwise multiplication". This multiplication strategy relies critically on choosing an NTT-friendly polynomial such as $x^{1024} + 1$ and choosing an NTT-friendly prime such as 12289.

Tweaking the polynomial and prime, as required by the conservative security recommendation made in this chapter, would make the NTTs several times more expensive. The best NTT-based method known to multiply in, e.g., $(\mathbb{Z}/8819)[x]/(x^{1021} - x - 1)$ requires replacing $x^{1021} - x - 1$ with $x^{2048} + 1$, and also replacing 8819 with two or three NTT-friendly primes. The conventional wisdom therefore implies that there is a very large penalty for requiring a large Galois group (NTT-friendly polynomials always have small Galois groups) and an inert modulus (NTT-friendly primes are never inert).

This chapter does much better by scrapping the NTTs and multiplying in a completely different way, using an optimized combination of several layers of Karatsuba's method and Toom's method. This approach does not need NTT-friendly polynomials, and it does not need NTT-friendly primes. The 50000-cycle speed is still slower than multiplications in [ADPS16], but it is quite close: one multiplication in [ADPS16] takes about 40000 cycles. (Each forward NTT in [ADPS16] takes 10968 cycles; each reverse NTT takes 12128 cycles; [LN16, Table 1] reports 9100 cycles and 9300 cycles respectively; the time for pointwise multiplication is not stated in [ADPS16] or [LN16] but can be extrapolated from [GOPS13] to take about 5000 cycles.) To summarize, the recommended defenses do *not* create a large speed penalty.

Credits. The content of this chapter is based on the paper *"NTRU Prime"* [BCLvV16] on ePrint which is a joint work with Daniel J. Bernstein, Tanja Lange and Christine van Vredendaal. Note that "we" in this chapter is used to refer to the aforementioned authors.

Organization of this chapter. Section 9.1 explores potential attacks to some rings used in lattice-based cryptography. Section 9.2 specifies Streamlined NTRU Prime, a public-key cryptosystem. Section 9.3 describes the design of Streamlined NTRU Prime as a lattice-based encryption system. Section 9.4 analyzes the security of Streamlined

NTRU Prime. Section 9.5 presents the parameter generation algorithm. Section 9.6 explains how multiplication is performed in Streamlined NTRU Prime. Section 9.7 shows vectorization techniques. Section 9.8 presents a complete non-constant-time reference implementation. Section 9.9 compares public-key encryption and unauthenicated key exchange. Section 9.10 discusses worst-case-to-average-case reductions. Section 9.11 reviews sieving algorithms.

9.1 Avoiding rings with worrisome structure

Practically all proposals for ideal-lattice-based cryptography use cyclotomic rings such as $\mathbb{Z}[x]/(x^{1024}+1)$. These rings have many automorphisms, such as $x \mapsto x^3$. Typical encryption proposals work specifically in cyclotomic quotient rings such as $(\mathbb{Z}/12289)[x]/(x^{1024}+1)$, allowing further nontrivial ring homomorphisms such as $x \mapsto 7$.

In February 2014, we publicly recommended[1] changing the choice of rings in ideal-lattice-based cryptography. Part of the recommendation is to switch from cyclotomic rings to new rings very far from having any non-identity automorphisms: specifically, rings of the form $\mathbb{Z}[x]/(x^p - x - 1)$. Another part of the recommendation is to use quotient *fields* such as $(\mathbb{Z}/9829)[x]/(x^{739} - x - 1)$, eliminating further nonzero homomorphisms. The complete recommendation has a precise mathematical definition explained below.

Attacks published *after* this recommendation have already built an excellent track record for the recommendation. For example, it is now generally agreed that the Smart–Vercauteren system [SV10] *using the old rings* is broken by a polynomial-time quantum attack and by a subexponential-time pre-quantum attack. Nobody has extended these attacks to our new rings. The attacks rely upon various interesting operations that are available in the old rings and not in the new rings; see below.

As another example, Bauch, Bernstein, de Valence, Lange, and van Vredendaal have very recently announced a *polynomial-time pre-quantum attack* against the Smart–Vercauteren system using multiquadratic rings. This attack relies even more heavily upon automorphisms and subrings.

9.1.1 Cryptographic risk management

An important part of cryptographer Alice's job is to extrapolate beyond known attacks (just like other scientists formulating theories beyond current knowledge), with the goal of making the safest decisions about an unclear future. For example:

- Dobbertin, Bosselaers, and Preneel wrote in 1996 [DBP96] that "it is anticipated that these techniques can be used to produce collisions for MD5" and recommended switching to other hash functions long before the MD5 attack by Wang and Yu [WY05].

---

[1] See the blog post [Ber14] by Daniel J. Bernstein; see also the 2013 note [Ber13, page 2].

- Many discrete-logarithm experts recommended prime fields (see, e.g., [BCC$^+$05, page 25]) long before recent attacks such as [Jou13] against small-characteristic finite-field discrete logarithms.
- Many discrete-logarithm experts specifically recommend prime-field ECDL over small-characteristic ECDL (see, e.g., [BCC$^+$05, page 62]), even though none of the efforts to break small-characteristic ECDL have been successful (see the recent survey [GG16]).
- Experts frequently recommend one unbroken system over another unbroken system of similar speed, saying that possible attack strategies against the first system are better understood.
- Our general recommendation included, as a special case, switching the Smart–Vercauteren system from the old rings to the new rings. This came before the polynomial-time quantum attacks against the system using the old rings.

History shows that this approach, despite its uncertainties, produces much better security than shortsighted cryptographer Bob focusing on the extreme question of what has already been broken.

Our recommendation to change the choice of rings was, and is, a broad recommendation for ideal-lattice-based cryptography: it applies to Smart–Vercauteren, to NTRU, to Ring-LWE-based cryptosystems, etc. We suggest the name “NTRU Prime” for the resulting cryptosystems; the “NTRU” part of the name acknowledges that we are tweaking the classic NTRU cryptosystem, and “Prime” reflects the fact that our modifications eliminate several different types of factorizations.

We emphasize that normal NTRU parameters are not affected by any of the known attacks discussed in this section. The same holds for Ring-LWE based on cyclotomics. However, we are skeptical of the notion that the most recent papers are the end of the attack story.

There is widespread agreement on the general goal of removing unnecessary algebraic structure from cryptography. As an example, the common recommendation of prime fields for DL takes various operations (in particular, automorphisms) away from attackers, and all available evidence is that this rescues *some* fraction of DL systems without enabling any new attacks. This provides ample justification for the recommendation, not merely in the cases where there are already known attacks (such as FFDL) but also in other cases (such as ECDL). Similarly, our recommendation of the new rings takes various operations (again, in particular, automorphisms) away from attackers, and all available evidence is that this rescues *some* fraction of lattice-based systems without enabling any new attacks. This provides ample justification for the recommendation, not merely in the cases where there are already known attacks (such as Smart–Vercauteren) but also in other cases (such as NTRU).

Of course, one could speculate that our recommendation does not help security. A recent paper by Albrecht, Bai, and Ducas [ABD16] broke some “overstretched NTRU assumptions” using the old rings, but Kirchner and Fouque [KF16] extended this attack to all rings, and one could speculate that *all* attacks against the old rings can be somehow adapted to the new rings. One could even speculate that our recommendation somehow *hurts* security.

But the big picture today is the opposite. Some systems are unbroken with rings following our recommendations and broken with rings violating our recommendations; there is nothing the other way around. More importantly, we successfully *predicted* this picture as a consequence of worrisome structure, specifically unnecessary ring homomorphisms eliminated by our recommendations.

Ideas for improving security usually come at a huge cost in performance; see, e.g., Section 9.10. If following our recommendation makes lattice-based systems much more expensive, then one has to ask whether using the increased costs in other ways, such as larger lattice dimensions, would produce a larger security benefit. But our performance results show that high-security Streamlined NTRU Prime is competitive with, and in some ways even better than, previous lattice-based public-key encryption systems.

We again emphasize that we are *not* saying that standard NTRU parameters are known to be broken in a way rescued by NTRU Prime. We are saying that ignoring the important possibility of a break would not be sensible risk management. NTRU Prime provides a more conservative, less structured alternative to NTRU, the same way that prime-field FFDL/ECDL have always provided more conservative, less structured alternatives to small-characteristic FFDL/ECDL.

9.1.2 Case study: the Campbell–Groves–Shepherd attack

The October 2014 Campbell–Groves–Shepherd preprint "Soliloquy: a cautionary tale" [CGS14] sent shock waves through lattice-based cryptography. The preprint describes a lattice-based cryptosystem named "Soliloquy" that the authors say they privately developed in 2007, and then claims a polynomial-time quantum key-recovery attack against this system. As mentioned briefly in [CGS14], the key-recovery problem for this system is identical to the key-recovery problem for the Smart–Vercauteren system.

In these systems, everyone shares a standard monic irreducible polynomial $P \in \mathbb{Z}[x]$ with small coefficients. Smart and Vercauteren [SV10, Section 7] take power-of-2 cyclotomic polynomials, such as the polynomial $x^{1024}+1$, but [SV10, Section 3] allows more general polynomials, and [CGS14] allows any cyclotomic polynomial. The receiver's public key consists of an integer α and a prime number q dividing $P(\alpha)$. Note that $q\mathcal{R}+(x-\alpha)\mathcal{R}$ is a prime ideal of the ring $\mathcal{R}=\mathbb{Z}[x]/P$; the receiver's secret key is a small generator $g \in \mathcal{R}$ of this ideal. The encryption and decryption procedures are not difficult but are not relevant here.

The first stage of the attack finds *some* generator of the ideal, expressed as a product of powers of small ring elements. Biasse and Song questioned the claimed performance of the algorithm for this stage (and these claims do not seem to have been defended by the authors of [CGS14]) but subsequently presented a different polynomial-time quantum algorithm for this stage; see [BS15] and [BS16]. Even without quantum computers, well-known techniques complete this stage in subexponential time.

The second stage of the attack reduces the generator to a small generator (either g or something else that is just as good for decryption, such as $-g$). This is a closest-vector problem in what is called the "log-unit lattice". One normally expects CVPs to take exponential time, but *for cyclotomic polynomials* P one can efficiently write down a very short basis for the log-unit lattice (or at worst a small-index sublattice). This basis consists of logarithms of various "cyclotomic units", as explained very briefly in [CGS14] and analyzed in detail in the followup paper [CDPR16] by Cramer, Ducas, Peikert, and Regev. For example, for $P = x^{1024} + 1$, the ring $\mathcal{R}$ contains $(1-x^3)/(1-x) = 1 + x + x^2$, and also contains the reciprocal $(1-x)/(1-x^3) = (1-x^{2049})/(1-x^3) = 1 + x^3 + \cdots + x^{2046}$; so $(1-x^3)/(1-x)$ is a unit in R, a typical example of a cyclotomic unit.

9.1.3 Mathematical specification of our recommendation

We recommend taking a standard monic irreducible polynomial P whose degree is a prime p, and whose "Galois group" is as large as possible, isomorphic to the permutation group S_p of size $p!$. Most polynomials of degree p have Galois group S_p, and we specifically suggest the small polynomial $P = x^p - x - 1$, which is irreducible and has Galois group S_p; see [Sel56] and [Osa87]. Furthermore, in contexts using moduli (such as NTRU and Ring-LWE), we recommend taking a prime modulus q that is "inert" in $\mathcal{R}$, i.e., where P is irreducible in $(\mathbb{Z}/q)[x]$, i.e., where $(\mathbb{Z}/q)[x]/P$ is a field. This happens with probability approximately $1/p$ for a "random" prime q; see Table 9.2 for many examples of acceptable pairs (p, q).

One way to define the Galois group is as the group of automorphisms of the smallest field that contains all the complex roots of P. Consider, for example, the field $\mathbb{Q}(\zeta)$ where $\zeta = \exp(2\pi i/2048)$. The notation $\mathbb{Q}(\zeta)$ means the smallest field containing both $\mathbb{Q}$ and ζ; explicitly, $\mathbb{Q}(\zeta)$ is the set of complex numbers $q_0 + q_1\zeta + \cdots + q_{1023}\zeta^{1023}$ with $q_0, q_1, \ldots, q_{1023} \in \mathbb{Q}$. The complex roots of $P = x^{1024} + 1$ are $\zeta, \zeta^3, \zeta^5, \ldots, \zeta^{2047}$, all of which are in $\mathbb{Q}(\zeta)$, so $\mathbb{Q}(\zeta)$ is the smallest field that contains all the complex roots of P. There are exactly 1024 automorphisms of this field (invertible maps from the field to itself preserving $0, 1, +, -, \cdot$). These automorphisms are naturally labeled $1, 3, 5, \ldots, 2047$; the automorphism with label i maps ζ to ζ^i, so it maps ζ^j to ζ^{ij}. In other words, automorphism i permutes the complex roots of P the same way that ith powering does; the Galois group is thus isomorphic to the multiplicative group $(\mathbb{Z}/2048)^*$.

NTRU traditionally takes $P = x^p - 1$ with p prime and q a power of 2, typically 2048. These choices violate our recommendation in several ways. First of all, $x^p - 1$ is not irreducible. One can tweak NTRU to work modulo the cyclotomic polynomial $\Phi_p = (x^p - 1)/(x - 1)$, but this polynomial does not have prime degree. Furthermore, the Galois group of Φ_p has size only $p - 1$, vastly smaller than $(p-1)!$. Also, the modulus q is not prime.

Ring-LWE-based systems typically take $P = x^p + 1$ where p is a power of 2 and q is a prime in $1 + 2p\mathbb{Z}$. These choices also violate our recommendation in several ways. The polynomial P is irreducible, but it does not have prime degree. Furthermore, its

Galois group has size only p, vastly smaller than $p!$. The modulus q is prime, but P is very far from irreducible modulo q: in fact, it splits into linear factors modulo q.

9.1.4 How the recommendation stops attacks

The multiquadratic attack mentioned above exploits a large collection of subfields of the field $\mathbb{Q}[x]/P$. The degree of the field is a multiple of the degree of every subfield, so by taking a prime degree we obviously rule out any such attack: the only subfields of $\mathbb{Q}[x]/P$ are $\mathbb{Q}$ and the entire field $\mathbb{Q}[x]/P$.

To understand why we also require very large Galois groups, consider the suggestion from [ABD16] to use the field $\mathbb{Q}(\zeta + \zeta^{-1})$ with $\zeta = \exp(2\pi i/2p)$, where both p and $(p-1)/2$ are prime. This field has prime degree $(p-1)/2$ and thus stops subfield attacks. It does not, however, stop the attack of [CGS14]: one can easily write down a very short basis consisting of logs of cyclotomic units in this field, such as $(\zeta^3 - \zeta^{-3})/(\zeta - \zeta^{-1})$.

More generally, if a number field of prime degree p has a Galois group of size p then the field is a subfield of a cyclotomic field. Even more generally, the Kronecker–Weber theorem states that any “abelian” number field is a subfield of a cyclotomic field. This might not enable an attack along the lines of [CGS14] if the cyclotomic field has degree much larger than p, but we do not think that it is wise to rely on this.

Of course, prohibiting minimum size p is not the same as requiring maximum size $p!$; there is a large gap between p and $p!$. But having a Galois group of size, say, $2p$ means that one can write down a degree-$2p$ extension field with $2p$ automorphisms, and one can then try to apply these automorphisms to build many units, generalizing the construction of cyclotomic units. From the perspective of algebraic number theory, the fact that, e.g., $(\zeta^3 - \zeta^{-3})/(\zeta - \zeta^{-1})$ is a unit is not a numerical accident: it is the same as saying that the ideal I generated by $\zeta - \zeta^{-1}$ is also generated by $\zeta^3 - \zeta^{-3}$, i.e., that I is preserved by the $\zeta \mapsto \zeta^3$ automorphism. This in turn can be seen from the factorization of I into prime ideals, together with the structure of Galois groups acting on prime ideals—a rigid structural feature that is not specific to the cyclotomic case.

Having a much larger Galois group means that P will have at most a small number of roots in any field of reasonable degree. This eliminates all known methods of efficiently performing computations with more than a small number of automorphisms.

It is of course still possible to compute a minimum-length basis for the log-unit lattice, but all known methods are very slow. Cohen’s classic book “A course in computational algebraic number theory” [Coh93, page 217] describes the task of computing “a system of fundamental units” (i.e., a basis for the log-unit lattice) as one of the five “main computational tasks of algebraic number theory”. One can compute *some* basis in subexponential time by techniques similar to the number-field sieve for integer factorization, but for almost all P the resulting basis elements will not be very short and will not be close to orthogonal, and finding a very short basis takes exponential time by all known methods.

The theory of units generalizes to what are called “S-units”; see, e.g., [Coh00, Chapter 7]. For any polynomial P it is trivial to write down *a few* independent S-units for various S; we have heard speculation that this would somehow allow attacks. We point out that this is a reinvention of a special case of "free relations", a small speedup to NFS. Writing down a few independent S-units is much less than writing down an entire basis, and does not seem helpful for attacking log-unit CVP. The big picture is that, despite intensive research, log-unit CVP is a classic problem that seems to be hard for almost all number fields; the only known attacks exploit extremely special algebraic features of number fields with small Galois groups.

Finally, we choose q as an inert prime so that there are no ring homomorphisms from $(\mathbb{Z}/q)[x]/P$ to any smaller nonzero ring. The attack strategies of [EHL14], [ELOS15], and [CLS16] start by applying such homomorphisms; the attacks are restricted in other ways, but we see no reason to provide homomorphisms to the attacker in the first place. The examples from [ELOS15] were shown in [CIV16b] to be breakable in a simpler way without homomorphisms (some noise components turn out to always be 0); but, as pointed out in [CLS16, Section 2.2] (see also [CIV16a, Section 5]), many other examples have been broken by homomorphisms without being broken by the algorithm from [CIV16b]. It is sometimes claimed that "modulus switching" makes the choice of q irrelevant (for example, [LS12a] says "we prove that the arithmetic form of the modulus q is irrelevant to the computational hardness of LWE and RLWE"), but an attacker switching from q to another modulus will noticeably increase noise, interfering with typical attack algorithms.

9.2 Streamlined NTRU Prime

This section specifies "Streamlined NTRU Prime", a public-key cryptosystem. The next section compares Streamlined NTRU Prime to alternatives.

We emphasize that Streamlined NTRU Prime is designed for the standard goal of IND-CCA2 security, i.e., security against adaptive chosen-ciphertext attacks. A server can reuse a public key any number of times, amortizing the costs of key generation and key distribution. The cost of setting up a new session key, *including* post-quantum server authentication, is then just one encryption for the client and one decryption for the server. This gives Streamlined NTRU Prime important performance advantages over unauthenticated key-exchange mechanisms such as [ADPS16]; see Section 9.9 for a precise comparison.

We are essentially done with a complete implementation, and we will submit it to eBACS [BLa] for benchmarking. However, we caution potential users that many details of Streamlined NTRU Prime are new and require careful security review. We have not limited ourselves to the minimum changes that would be required to switch to NTRU Prime from an existing version of the NTRU public-key cryptosystem; we have taken the opportunity to rethink and reoptimize all of the details of NTRU from an implementation and security perspective. We recommend NTRU Prime, but it is too early to recommend Streamlined NTRU Prime.

9.2.1 Parameters

Streamlined NTRU Prime is actually a family of cryptosystems parametrized by positive integers (p, q, t) subject to the following restrictions: p is a prime number; q is a prime number; $t \geq 1$; $p \geq 3t$; $q \geq 32t + 1$; $x^p - x - 1$ is irreducible in the polynomial ring $(\mathbb{Z}/q)[x]$.

We abbreviate the ring $\mathbb{Z}[x]/(x^p - x - 1)$, the ring $(\mathbb{Z}/3)[x]/(x^p - x - 1)$, and the field $(\mathbb{Z}/q)[x]/(x^p - x - 1)$ as $\mathscr{R}$, $\mathscr{R}/3$, and $\mathscr{R}/q$ respectively. We refer to an element of $\mathscr{R}$ as **small** if all of its coefficients are in $\{-1, 0, 1\}$. We refer to a small element as **t-small** if exactly $2t$ of its coefficients are nonzero.

Our case study in this chapter is Streamlined NTRU Prime 9829^{739}. This specific cryptosystem has parameters $p = 739$, $q = 9829$, and $t = 246$. The following subsections specify the algorithms for general parameters but the reader may wish to focus on these particular parameters. Section 9.8 shows complete algorithms for key generation, encapsulation, and decapsulation in Streamlined NTRU Prime 9829^{739}, using the Sage [S+15] computer-algebra system.

9.2.2 Key generation

The receiver generates a public key as follows:

- Generate a uniform random small element $g \in \mathscr{R}$. Repeat this step until g is invertible in $\mathscr{R}/3$.
- Generate a uniform random t-small element $f \in \mathscr{R}$. (Note that f is nonzero and hence invertible in $\mathscr{R}/q$, since $t \geq 1$.)
- Compute $h = g/(3f)$ in $\mathscr{R}/q$. (By assumption q is a prime larger than 3, so 3 is invertible in $\mathscr{R}/q$, so $3f$ is invertible in $\mathscr{R}/q$.)
- Encode h as a string $\underline{h}$. The public key is $\underline{h}$.
- Save the following secrets: f in $\mathscr{R}$; and $1/g$ in $\mathscr{R}/3$.

See `keygen` in Section 9.8.

The encoding of public keys as strings is another parameter for Streamlined NTRU Prime. Each element of $\mathbb{Z}/q$ is traditionally encoded as $\lceil \log_2 q \rceil$ bits, so the public key is traditionally encoded as $p\lceil \log_2 q \rceil$ bits. If q is noticeably smaller than a power of 2 then one can easily compress a public key by merging adjacent elements of $\mathbb{Z}/q$, with a lower limit of $p \log_2 q$ bits. For example, an element of $\mathbb{Z}/q$ for $q = 9829$ is traditionally encoded as 14 bits, but three such elements are easily encoded together as 40 bits, saving 5% of the space; 739 elements of $\mathbb{Z}/q$ would traditionally take 10346 bits, but 246 triples and a final element take just 9856 bits. See Section 9.8 for further encoding details.

9.2.3 Encapsulation

Streamlined NTRU Prime is actually a “key encapsulation mechanism” (KEM). This means that the sender takes a public key as input and produces a ciphertext and session key as output. See Section 9.3.4 for comparison to older notions of public-key encryption.

Specifically, the sender generates a ciphertext as follows:

- Decode the public key $\underline{h}$, obtaining $h \in \mathscr{R}/q$.
- Generate a uniform random t-small element $r \in \mathscr{R}$.
- Compute $hr \in \mathscr{R}/q$.
- Round each coefficient of hr, viewed as an integer between $-(q-1)/2$ and $(q-1)/2$, to the nearest multiple of 3, producing $c \in \mathscr{R}$. (If $q \in 1+3\mathbb{Z}$, as in our case study $q = 9829$, then each coefficient of c is in $\{-(q-1)/2, \ldots, -6, -3, 0, 3, 6, \ldots, (q-1)/2\}$. If $q \in 2+3\mathbb{Z}$ then each coefficient of c is in $\{-(q+1)/2, \ldots, -6, -3, 0, 3, 6, \ldots, (q+1)/2\}$.)
- Encode c as a string $\overline{c}$.
- Hash r, obtaining a left half C (“key confirmation”) and a right half K.
- The ciphertext is the concatenation $C\,\overline{c}$. The session key is K.

See `encapsulate` in Section 9.8.

The hash function for r is another parameter for Streamlined NTRU Prime. We encode r as a byte string by adding 1 to each coefficient, obtaining an element of $\{0, 1, 2\}$ encoded as 2 bits in the usual way, and then packing 4 adjacent coefficients into a byte, consistently using little-endian form. See `encodeZx` in Section 9.8. We hash the resulting byte string with SHA-512, obtaining a 256-bit key confirmation C and a 256-bit session key K.

The encoding of ciphertexts c as strings $\overline{c}$ is another parameter for Streamlined NTRU Prime. This encoding can be more compact than the encoding of public keys because each coefficient of c is in a limited subset of $\mathbb{Z}/q$. Concretely, for $q = 9829$ and $p = 739$, we use 12 bits for each coefficient of c and thus 8872 bits (padded to a byte boundary) for $\overline{c}$, saving 10% compared to the size of a public key and 15% compared to separately encoding each element of $\mathbb{Z}/q$. See `encoderoundedRq` in Section 9.8. Key confirmation adds 256 bits to ciphertexts.

9.2.4 Decapsulation

The receiver decapsulates a ciphertext $C\,\overline{c}$ as follows:

- Decode $\overline{c}$, obtaining $c \in \mathscr{R}$.
- Multiply by $3f$ in $\mathscr{R}/q$.

- View each coefficient of $3fc$ in $\mathscr{R}/q$ as an integer between $-(q-1)/2$ and $(q-1)/2$, and then reduce modulo 3, obtaining a polynomial e in $\mathscr{R}/3$.
- Multiply by $1/g$ in $\mathscr{R}/3$.
- Lift e/g in $\mathscr{R}/3$ to a small polynomial $r' \in \mathscr{R}$.
- Compute c', C', K' from r' as in encapsulation.
- If r' is t-small, $c' = c$, and $C' = C$, then output K'. Otherwise output False.

See `decapsulate` in Section 9.8.

If $C\,\bar{c}$ is a legitimate ciphertext then c is obtained by rounding the coefficients of hr to the nearest multiples of 3; i.e., $c = m + hr$ in $\mathscr{R}/q$, where m is small. All coefficients of the polynomial $3fm + gr$ in $\mathscr{R}$ are in $[-16t, 16t]$ by Theorem 9.1 below, and thus in $[-(q-1)/2, (q-1)/2]$ since $q \geq 32t + 1$. Viewing each coefficient of $3fc = 3fm + gr$ as an integer in $[-(q-1)/2, (q-1)/2]$ thus produces exactly $3fm + gr \in \mathscr{R}$, and reducing modulo 3 produces $gr \in \mathscr{R}/3$; i.e., $e = gr$ in $\mathscr{R}/3$, so $e/g = r$ in $\mathscr{R}/3$. Lifting now produces exactly r since r is small; i.e., $r' = r$. Hence $(c', C', K') = (c, C, K)$. Finally, $r' = r$ is t-small, $c' = c$, and $C' = C$, so decapsulation outputs $K' = K$, the same session key produced by encapsulation.

Theorem 9.1. *Fix integers $p \geq 3$ and $t \geq 1$. Let $m, r, f, g \in \mathbb{Z}[x]$ be polynomials of degree at most $p-1$ with all coefficients in $\{-1, 0, 1\}$. Assume that f and r each have at most $2t$ nonzero coefficients. Then $3fm + gr \bmod x^p - x - 1$ has each coefficient in the interval $[-16t, 16t]$.*

Proof. We first show that fm has coefficients in $[-4t, 4t]$. Let $f = \sum_{i=0}^{p-1} f_i x^i$, a ternary polynomial of exactly $2t$ terms. Let $m = \sum_{i=0}^{p-1} m_i x^i$ a ternary polynomial. Now we rewrite fm as $\sum_{i=0}^{p-1} f_i(x^i m)$. Since this sum adds at most $2t$ terms of each degree, it just remains to be shown that the coefficients of $x^i m$ are all in the range $[-2, 2]$. To show this we observe that $xm \equiv m_{p-1} + (m_0 + m_{p-1})x + m_1 x^2 + \cdots + m_{p-2}x^{p-1} \bmod x^p - x - 1$, which has coefficients in the range $[-1, 1]$, except $|m_0 + m_{p-1}|$ may be 2. Generalizing this, for $2 \leq i < p$, we get $x^i m \equiv m_{p-i} + (m_{p-i} + m_{p-i-1})x + \cdots + (m_{p-2} + m_{p-1})x^{i-1} + (m_{p-1} + m_0)x^i + m_1 x^{i+1} + \cdots + m_{p-i-1}x^{p-1} \bmod x^p - x - 1$ with coefficients in $[-2, 2]$. Similar reasoning for gr implies that each coefficient of $gr \bmod x^p - x - 1$ is in $[-4t, 4t]$. Hence each coefficient of $3fm + gr \bmod x^p - x - 1$ is in $[-16t, 16t]$. □

9.3 The design space of lattice-based encryption

There are many different ideal-lattice-based public-key encryption schemes in the literature, including many versions of NTRU, many Ring-LWE-based cryptosystems, and now Streamlined NTRU Prime. These are actually many different points in a high-dimensional space of possible cryptosystems. We give a unified description of the advantages and disadvantages of what we see as the most important options in each dimension, in particular explaining the choices that we made in Streamlined NTRU Prime.

Beware that there are many interactions between options. For example, using Gaussian errors is incompatible with eliminating decryption failures, because there is always a small probability of large samples combining with large values. Using *truncated* Gaussian errors is compatible with eliminating decryption failures, but requires a much larger modulus q. Neither of these options is compatible with the simple tight KEM that we use.

9.3.1 The ring

The choice of cryptosystem includes a choice of a monic degree-p polynomial $P \in \mathbb{Z}[x]$ and a choice of a positive integer q. As in Section 9.2, we abbreviate the ring $\mathbb{Z}[x]/P$ as $\mathcal{R}$, and the ring $(\mathbb{Z}/q)[x]/P$ as $\mathcal{R}/q$.

The choices of P mentioned in the introduction of this chapter include $x^p - 1$ for prime p (classic NTRU); $x^p + 1$ where p is a power of 2; and $x^p - x - 1$ for prime p (NTRU Prime). Choices of q include powers of 2 (classic NTRU); split primes q; and inert primes q (NTRU Prime).

Of course, Streamlined NTRU Prime makes the NTRU Prime choices here. Most of the optimizations in Streamlined NTRU Prime can also be applied to other choices of P and q, with a few exceptions noted below.

9.3.2 The public key

The receiver's public key, which we call h, is an element of $\mathcal{R}/q$, secretly computed by dividing two small polynomials. It is invertible in $\mathcal{R}/q$ but has no other publicly visible structure.

An alternative is to transmit the public key h as two elements $d, hd \in \mathcal{R}/q$, where d is chosen as a uniform random invertible element of $\mathcal{R}/q$. This is what would be called "randomized projective coordinates" in the ECC context, whereas simply sending h would be called "affine coordinates". The advantage of representing h as a fraction $(hd)/d$ is that the receiver can skip all divisions in the secret computation of the public key h: the receiver simply computes h as a fraction, and then multiplies the numerator and denominator by a uniform random invertible element of $\mathcal{R}/q$ to hide all information beyond what h would have revealed. The obvious disadvantage of sending d, hd is that public keys become twice as large; a further disadvantage is that arithmetic on h turns into arithmetic on both d and hd. Key size is important, and we expect key generation to be amortized across many uses of h, so we skip this alternative in Streamlined NTRU Prime. We also skip the idea of supporting both key formats as a run-time option: this would complicate implementations.

9.3.3 Inputs and ciphertexts

Classic NTRU ciphertexts are elements of the form $m + hr \in \mathcal{R}/q$. Here $h \in \mathcal{R}/q$ is the public key as above, and m, r are small elements of $\mathcal{R}$ chosen by the sender. The multiplication of h by r is the main bottleneck in encryption and the main target of our implementation work; see Sections 9.6 and 9.7.

The receiver can quickly recover the input (m, r) from the ciphertext $m + hr$; see Section 9.2.4 for Streamlined NTRU Prime and Section 9.3.5 for a broader view. We say "input" rather than "plaintext" because in any modern public-key cryptosystem the input is randomized and is separated from the sender's plaintext by some hashing; see Section 9.3.4.

In the original NTRU specification [HPS98], m was allowed to be any element of $\mathscr{R}$ having all coefficients in a standard range. The range was $\{-1, 0, 1\}$ for all of the suggested parameters, with q not a multiple of 3, and we focus on this case for simplicity (although we note that some other lattice-based cryptosystems have taken the smaller range $\{0, 1\}$, or sometimes larger ranges).

Current NTRU specifications such as [HPS⁺15] prohibit m that have an unusually small number of 0's or 1's or -1's. For random m, this prohibition applies with probability $<2^{-10}$, and in case of failure the sender can try encoding the plaintext as a new m, but this is problematic for applications with hard real-time requirements. The reason for this prohibition is that classic NTRU gives the attacker an "evaluate at 1" homomorphism from $\mathscr{R}/q$ to $\mathbb{Z}/q$, leaking $m(1)$. The attacker scans many ciphertexts to find an occasional ciphertext where the value $m(1)$ is particularly far from 0; this value constrains the search space for the corresponding m by enough bits to raise security concerns. In NTRU Prime, $\mathscr{R}/q$ is a field, so this type of leak cannot occur.

Streamlined NTRU Prime actually uses a different type of ciphertext, which we call a "rounded ciphertext". The sender chooses a small r and computes $hr \in \mathscr{R}/q$. The sender obtains the ciphertext by rounding each coefficient of hr, viewed as an integer between $-(q-1)/2$ and $(q-1)/2$, to the nearest multiple of 3. This ciphertext can be viewed as an example of the original ciphertext $m + hr$, but with m chosen so that each coefficient of $m + hr$ is in a restricted subset of $\mathbb{Z}/q$. The advantage of choosing m in this way is that $\approx q/3$ possibilities take less space than q possibilities. See Section 9.2.3.

With the original ciphertexts, each coefficient of $m+hr$ leaves 3 possibilities for the corresponding coefficients of hr and m. With rounded ciphertexts, each coefficient of $m + hr$ also leaves 3 possibilities for the corresponding coefficients of hr and m, except that the boundary cases $-(q - 1)/2$ and $(q - 1)/2$ (assuming $q \in 1 + 3\mathbb{Z}$) leave only 2 possibilities. In a pool of 2^{64} rounded ciphertexts, the attacker might find one ciphertext that has 15 of these boundary cases out of 739 coefficients; these occasional exceptions have very little impact on known attacks. It would be possible to randomize the choice of multiples of 3 near the boundaries, but we prefer the simplicity of having the ciphertext determined entirely by r. It would also be possible to prohibit ciphertexts at the boundaries, but as above we prefer to avoid restarting the encryption process.

The original NTRU paper [HPS98, Section 5.4] drew analogies between NTRU and the McEliece code-based cryptosystem. A 1986 paper [Nie86] by Niederreiter proposed ciphertexts shorter than McEliece ciphertexts, and this feature has become standard in code-based cryptography. Given this background, one would expect to find many papers proposing ideal-lattice-based cryptosystems with shorter ciphertexts than NTRU; but, as far as we know, Streamlined NTRU Prime is the first such system. We have borrowed the "rounding" name from a few recent papers on "learning with rounding" (see [BPR12], [AKPW13], and [BGM+16]), but none of these papers achieves ciphertext sizes smaller than keysizes.

9.3.4 Padding and KEMs

In Streamlined NTRU Prime we use the modern "KEM+DEM" approach introduced by Shoup; see [Sho01]. This approach is much nicer for implementors than previous approaches to public-key encryption. For readers unfamiliar with this approach, we briefly review the analogous options for RSA encryption.

RSA maps an input m to a ciphertext $m^e \bmod n$, where (n, e) is the receiver's public key. When RSA was first introduced, its input m was described as the sender's plaintext. This was broken in reasonable attack models, leading to the development of various schemes to build m as some combination of fixed padding, random padding, and a short plaintext; typically this short plaintext is used as a shared secret key. This turned out to be quite difficult to get right, both in theory (see, e.g., [Sho02]) and in practice (see, e.g., [MSW+14]), although it does seem possible to protect against arbitrary chosen-ciphertext attacks by building m in a sufficiently convoluted way.

The "KEM+DEM" approach, specifically Shoup's "RSA-KEM" described in [Sho01] (also called "Simple RSA"), is much easier:

- Choose a uniform random integer m modulo n. This step does not even look at the plaintext.
- To obtain a shared secret key, simply apply a cryptographic hash function to m.
- Encrypt and authenticate the sender's plaintext using this shared key.

Any attempt to modify m, or the plaintext, will be caught by the authenticator.

"KEM" means "key encapsulation mechanism": $m^e \bmod n$ is an "encapsulation" of the shared secret key $H(m)$. "DEM" means "data encapsulation mechanism", referring to the encryption and authentication using this shared secret key. Authenticated ciphers are normally designed to be secure for many messages, so $H(m)$ can be reused to protect further messages from the sender to the receiver, or from the receiver back to the sender. It is also easy to combine KEMs, for example combining a pre-quantum KEM with a post-quantum KEM, by simply hashing the shared secrets together.

When NTRU was introduced, its input (m, r) was described as a sender plaintext m combined with a random r. This is obviously not secure against chosen-ciphertext attacks. Subsequent NTRU papers introduced various mechanisms to build (m, r) as increasingly convoluted combinations of fixed padding, random padding, and a short plaintext.

It is easy to guess that KEMs simplify NTRU, the same way that KEMs simplify RSA; we are certainly not the first to suggest this. However, all the NTRU-based KEMs we have found in the literature (e.g., [Sta05] and [Sak07]) construct the NTRU input (m, r) by hashing a shorter input and verifying this hash during decapsulation;

typically r is produced as a hash of m. These KEMs implicitly assume that m and r can be chosen independently, whereas rounded ciphertexts (see Section 9.3.3) have r as the sole input. Furthermore, it is not clear that generic-hash chosen-ciphertext attacks against these KEMs are as difficult as inverting the NTRU map from input to ciphertext: the security theorems are quite loose.

We instead follow a simple generic KEM construction introduced in the earlier paper [Den03, Section 6] by Dent, backed by a tight security reduction [Den03, Theorem 8] from inversion to generic-hash chosen-ciphertext attacks:

- Like RSA-KEM, this construction hashes the input, in our case r, to obtain the session key.
- Decapsulation verifies that the ciphertext is the correct ciphertext for this input, preventing per-input ciphertext malleability.
- The KEM uses additional hash output for key confirmation, making clear that a ciphertext cannot be generated except by someone who knows the corresponding input.

Key confirmation might be overkill from a security perspective, since a random session key will also produce an authentication failure; but key confirmation allows the KEM to be audited without regard to the authentication mechanism, and adds only 3% to our ciphertext size.

Dent's security analysis assumes that decryption works for all inputs. This assumption is not valid for most lattice-based encryption schemes (see Section 9.3.7), but it is valid for Streamlined NTRU Prime. This difference appears to account for most of the complications in subsequent papers on NTRU-based KEMs.

As a spinoff of analyzing KEM options, we found a fast chosen-ciphertext attack against the code-based KEM proposed in [Per13]. The problem is that [Per13] switches to predictable KEM output if decoding fails. The attacker can easily modify the ciphertext to flip a small number of bits (e.g., one or two bits) in the unknown error vector and to generate an authenticator from the predictable KEM output; decryption will succeed if the flipped error vector is decodable, and will almost certainly fail otherwise. Repeating these modifications, quickly reveals the unknown error vector from the pattern of decryption failures, similarly to Berson's attack [Ber97] on the plain McEliece system. McBits [BCS13] avoids this problem, because it uses a separate output bit from the KEM to indicate a decoding failure.

9.3.5 Key generation and decryption

Classic NTRU computes the public key as $3g/f$ in $\mathscr{R}/q$, where f and g are secret. Decryption computes $fc = fm + 3gr$, reduces modulo 3 to obtain fm, and multiplies by $1/f$ to obtain m.

The NTRU literature, except for the earliest papers, takes f of the form $1 + 3F$, where F is small. This eliminates the multiplication by the inverse of f modulo 3. In Streamlined NTRU Prime we have chosen to skip this speedup for two reasons. First, in the long run we expect cryptography to be implemented in hardware, where a multiplication in $\mathscr{R}/3$ is far less expensive than a multiplication in $\mathscr{R}/q$. Second, this speedup requires noticeably larger keys and ciphertexts for the same security level, and this is important for many applications, while very few applications will notice the CPU time for Streamlined NTRU Prime.

Streamlined NTRU Prime changes the position of the 3, taking h as $g/(3f)$ rather than $3g/f$. Decryption computes $3fc = 3fm + gr$, reduces modulo 3 to obtain gr, and multiplies by $1/g$ to obtain r. This change lets us compute (m, r) by first computing r and then multiplying by h, whereas otherwise we would first compute m and then multiply by $1/h$. One advantage is that we skip computing $1/h$; another advantage is that we need less space for storing a key pair. This $1/h$ issue does not arise for NTRU variants that compute r as a hash of m, but those variants are incompatible with rounded ciphertexts, as discussed in Section 9.3.4.

It is important for security to compute inverses such as $1/f$ in constant time. For Streamlined NTRU Prime, $\mathbb{Z}/q$ and $\mathscr{R}/q$ are fields of size q and q^p respectively, so inversion is the same as computing $(q-2)$nd powers and (q^p-2)nd powers respectively by Fermat's little theorem, and one can easily build constant-time exponentiations from our constant-time multiplications. We actually use a more complicated but much faster approach. We first convert the extended Euclidean algorithm into a one-coefficient-at-a-time Berlekamp–Massey/"almost-inverse" algorithm (see, e.g., [Sil99]). All of the conditional branches amount to simple input selections, which we convert into constant-time arithmetic. We compute the maximum number of iterations for the algorithm, and always perform this number of iterations, again using constant-time arithmetic so that dummy iterations produce the correct result.

9.3.6 The shape of small polynomials

As noted in Section 9.3.3, the coefficients of m are chosen from the limited range $\{-1, 0, 1\}$. The NTRU literature [HPS98, HSW05, HHHW09, HPS+15] generally puts the same limit on the coefficients of r, g, and f, except that if f is chosen with the shape $1 + 3F$ (see Section 9.3.5) then the literature puts this limit on the coefficients of F. Sometimes these "ternary polynomials" are further restricted to "binary polynomials", excluding coefficient -1.

The NTRU literature further restricts the Hamming weight of r, g, and f. Specifically, a cryptosystem parameter is introduced to specify the number of 1's and -1's. For example, there is a parameter t (typically called "d" in NTRU papers) so that r has exactly t coefficients equal to 1, exactly t coefficients equal to -1, and the remaining $p - 2t$ coefficients equal to 0. These restrictions allow decryption for smaller values of q (see Section 9.3.7), saving space and time. Beware, however, that if t is *too* small then there are attacks; see our security analysis in Section 9.4.

We keep the requirement that r have Hamming weight $2t$, and keep the requirement that these $2t$ nonzero coefficients are all in $\{-1, 1\}$, but we drop the requirement of an equal split between -1 and 1. This allows somewhat more choices of r. The same comments apply to f. Similarly, we require g to have all coefficients in $\{-1, 0, 1\}$ but we do not further limit the distribution of coefficients.

These changes would affect the conventional NTRU decryption procedure: they expand the *typical* size of coefficients of fm and gr, forcing larger choices of q to avoid *noticeable* decryption failures. But we instead choose q to avoid *all* decryption failures (see Section 9.3.7), and these changes do not expand our *bound* on the size of the coefficients of fm and gr.

The obvious way to generate t-small polynomials is choose a random position for a nonzero coefficient and repeat until the weight has reached $2t$. This takes variable time, raising security questions. Here is a constant-time alternative:

- Generate a target list $(\pm 1, \ldots, \pm 1, 0, \ldots, 0)$ starting with $2t$ nonzero entries, each chosen randomly as either 1 or -1.
- Use a constant-time algorithm to sort a list of p random numbers, and at the same time apply the same permutation to the target list.

This is theoretically perfect when the numbers do not collide. We generate 32-bit random numbers, replace the bottom 2 bits with the target list, sort the numbers, and extract the bottom 2 bits. (This produces 30-bit collisions once every few thousand ciphertexts, making smaller coefficients marginally more likely to appear near the beginning of the list. We could check for these collisions and restart if they occur, but the information leak is negligible.) Modern stream ciphers take only a few thousand cycles to generate 739 random 32-bit numbers, and a constant-time size-1024 sorting network (Batcher's "odd-even sorting network" [Bat68]) takes just 24064 constant-time compare-exchange steps, which we note are easily vectorizable.

NTRU papers starting with [HS03] have used "product-form polynomials", i.e., polynomials of the form $AB+C$. The weight of $AB+C$ is generally higher than the total weight of A, B, C (since the terms of A and B cross-multiply), and a rather small total weight of A, B, C maintains security against all known attacks. To multiply by $AB+C$ one can multiply by A, then multiply by B, then multiply the original input by C. This saves time for non-constant-time sparse-polynomial-multiplication algorithms, but it loses time for constant-time algorithms, so we ignore this idea. (Even *with* this idea, the best speeds for NTRU using sparse polynomial multiplication are not competitive with our speeds.)

Elsewhere in the literature on lattice-based cryptography one can find larger coefficients: consider, e.g., the quinary polynomials in [DDLL13], and the even wider range in [ADPS16]. In [SS11], the coefficients of f and g are sampled from a very wide discrete Gaussian distribution, allowing a proof regarding the distribution of g/f. However, this appears to produce *worse* security for any given key size. Specifically, there are no known attack strategies blocked by a Gaussian distribution, while the very wide distribution forces q to be very large to enable decryption (see Section 9.3.7), producing a much larger key size (and ciphertext size) for the same security level.

9.3.7 Choosing q

In Streamlined NTRU Prime we require $q \geq 32t+1$. Recall that decryption sees $3fm+gr$ in $\mathcal{R}/q$ and tries to deduce $3fm+gr$ in $\mathcal{R}$; the condition $q \geq 32t+1$

guarantees that this works, since each coefficient of $3fm+gr$ in $\mathscr{R}$ is between $-(q-1)/2$ and $(q-1)/2$ by Theorem 9.1. Taking different shapes of m, r, f, g, or changing the polynomial $P = x^p - x - 1$, would change the bound $32t+1$; for example, replacing g by $1+3G$ would change $32t+1$ into $48t+3$.

In lattice-based cryptography it is standard to take somewhat smaller values of q. The idea is that coefficients in $3fm+gr$ are produced as sums of many $+1$ and -1 terms, and these terms usually cancel, rather than conspiring to produce the maximum conceivable coefficient. But this approach raises several questions:

- Will users randomly encounter decryption failures?
- Can attackers trigger decryption failures by generating many more ciphertexts?
- Can attackers tweak these ciphertexts to trigger decryption failures?
- What should implementors do if decryption *does* fail?

The literature on the first two questions is already quite complicated, and it is difficult to find literature on the third and fourth questions. The only safe assumption is that decryption failures compromise security, allowing attackers to learn f from the pattern of decryption failures; see [HNP+03], and see also the discussion of NTRU complications in Section 9.3.4. In the context of code-based cryptography using QC-MDPC codes, Guo, Johansson, and Stankovski [GJS16] recently showed a complete key recovery from decryption failures. We prefer to guarantee that decryption works, making the security analysis simpler and more robust.

9.4 Security of Streamlined NTRU Prime

In this section we adapt existing *pre-quantum* NTRU attack strategies to the context of Streamlined NTRU Prime and quantify their effectiveness. In particular, we account for the impact of changing x^p-1 to x^p-x-1, and using small f rather than $f = 1+3F$ with small F.

Underestimating attack cost can *damage* security, for reasons explained in [BL13a, full version, Appendix B.1.2], so we prefer to use accurate cost estimates. However, accurately evaluating the cost of lattice attacks is generally quite difficult. The literature very often explicitly resorts to underestimates. Comprehensively fixing this problem is beyond the scope of this chapter, but we have started work in this direction, as illustrated by Section 9.11. At the same time it is clear that the best attack algorithms known today are much better than the best attack algorithms known a few years ago, so it is unreasonable to expect that the algorithms have stabilized. We plan to periodically issue updated security estimates to reflect ongoing work.

9.4.1 Meet-in-the-middle attack

Odlyzko's meet-in-the-middle attack [HGSW03, How07] on NTRU works by splitting the space of possible keys $\mathscr{F}$ into two parts such that $\mathscr{F} = \mathscr{F}_1 \oplus \mathscr{F}_2$. Then in each loop of the algorithm partial keys are drawn from $\mathscr{F}_1$ and $\mathscr{F}_2$ until a collision function (defined in terms of the public key h) indicates that $f_1 \in \mathscr{F}_1$ and $f_2 \in \mathscr{F}_2$ have been found such that $f = f_1 + f_2$ is the private key.

The number of choices for f is $\binom{p}{t}\binom{p-t}{t}$ in classic NTRU and $\binom{p}{2t}2^{2t}$ in Streamlined NTRU Prime. A first estimate is that the number of loops in the algorithm is the square root of the number of choices of f. However, this estimate does not account for equivalent keys. In classic NTRU, a key (f, g) is equivalent to all of the rotated keys $(x^i f, x^i g)$ and to the negations $(-x^i f, -x^i g)$, and the algorithm succeeds if it finds any of these rotated keys. The $2p$ rotations and negations are almost always distinct, producing a speedup factor very close to $\sqrt{2p}$.

The structure of the NTRU Prime ring is less friendly to this attack. Say f has degree $p - c$; typically c is around $p/2t$, since there are $2t$ terms in f. Multiplying f by $x, x^2, \ldots, x^{c-1}$ produces elements of $\mathscr{F}$, but multiplying f by x^c replaces x^{p-c} with $x^p \bmod x^p - x - 1 = x + 1$, changing its weight and thus leaving $\mathscr{F}$. It is possible but rare for subsequent multiplications by x to reenter $\mathscr{F}$. Similarly, one expects only about $p/2t$ divisions by x to stay within $\mathscr{F}$, for a total of only about p/t equivalent keys, or $2p/t$ when negations are taken into account. We have confirmed these estimates with experiments.

One could modify the attack to use a larger set $\mathscr{F}$, but this seems to lose more than it gains. Furthermore, similar wraparounds for g compromise the effectiveness of the collision function. To summarize, the extra term in $x^p - x - 1$ seems to increase the attack cost by a factor around $\sqrt{t}$, compared to classic NTRU; i.e., the rotation speedup is only around $\sqrt{2p/t}$ rather than $\sqrt{2p}$.

On the other hand, some keys f allow considerably more rotations. We have decided to conservatively assume a speedup factor of $\sqrt{2(p-t)}$, even though we do not know how to achieve this speedup for random keys f. This means that the number of loops before this attack is expected to find f is

$$L = \sqrt{\binom{p}{2t}2^{2t}} \Big/ \sqrt{2(p-t)}. \tag{9.1}$$

In each loop, t vectors of size p are added and their coefficients are reduced modulo q. We thus estimate the attack cost as Lpt. The storage requirement of the attack is approximately $L \log_2 L$. We can reduce this storage by applying collision search to the meet-in-the-middle attack (see [OW99, vV16]). In this case we can reduce the storage capacity by a factor s at the expense of increasing the running time by a factor $\sqrt{s}$.

9.4.2 Streamlined NTRU Prime lattice

As with NTRU we can embed the problem of recovering the private keys f, g into a lattice problem. Saying $3h = g/f$ in $\mathscr{R}/q$ is the same as saying $3hf + qk = g$ in $\mathscr{R}$ for some polynomial k; in other words, there is a vector (k, f) of length $2p$ such that

$$\begin{pmatrix} k & f \end{pmatrix}\begin{pmatrix} qI & 0 \\ H & I \end{pmatrix} = \begin{pmatrix} k & f \end{pmatrix} B = \begin{pmatrix} g & f \end{pmatrix},$$

where H is a matrix with the i'th vector corresponding to $x^i \cdot 3h \bmod x^p - x - 1$ and I is the $p \times p$ identity matrix. We will call B the *Streamlined NTRU Prime public lattice basis*. This lattice has determinant q^p. The vector (g, f) has norm at most $\sqrt{2p}$. The Gaussian heuristic states that the length of the shortest vector in a random lattice is approximately $\det(B)^{1/(2p)}\sqrt{\pi e p} = \sqrt{\pi e p q}$, which is much larger than $\sqrt{2p}$, so we expect (g, f) to be the shortest nonzero vector in the lattice.

Finding the secret keys is thus equivalent to solving the Shortest Vector Problem (SVP) for the Streamlined NTRU Prime public lattice basis. The fastest currently known method to solve SVP in the NTRU public lattice is the hybrid attack, which we discuss below.

A similar lattice can be constructed to instead try to find the input pair (m, r). However, there is no reason to expect the attack against (m, r) to be easier than the attack against (g, f): r has the same range as f, and m has essentially the same range as g. Recall that Streamlined NTRU Prime does not have classic NTRU's problem of leaking $m(1)$. There are occasional boundary constraints on m (see Section 9.3.3), and there is also an $\mathscr{R}/3$ invertibility constraint on g, but these effects are minor.

9.4.3 Hybrid security

The best known attack against the NTRU lattice is the hybrid lattice-basis reduction and meet-in-the-middle attack described in [How07]. The attack works in two phases: the lattice basis reduction phase and the meet-in-the-middle phase.

In the lattice reduction step it is observed that applying lattice reduction techniques will mostly reduce the middle vectors of the basis [Sch03]. Therefore the strategy is to apply lattice-basis reduction, for example BKZ 2.0 [CN11a], to a submatrix B' of the public basis B. We then get a reduced basis $T = UBY$:

$$\left(\begin{array}{c|c|c} I_w & 0 & 0 \\ \hline 0 & U' & 0 \\ \hline 0 & 0 & I_{w'} \end{array}\right) \cdot \left(\begin{array}{c|c|c} qI_w & 0 & 0 \\ \hline * & B' & 0 \\ \hline * & * & I_{w'} \end{array}\right) \cdot \left(\begin{array}{c|c|c} I_w & 0 & 0 \\ \hline 0 & Y' & 0 \\ \hline 0 & 0 & I_{w'} \end{array}\right) = \left(\begin{array}{c|c|c} qI_w & 0 & 0 \\ \hline * & T' & 0 \\ \hline * & * & I_{w'} \end{array}\right)$$

Here Y is orthonormal and T' is again in lower triangular form.

In the meet-in-the-middle phase we can use a meet-in-the-middle algorithm to guess options for the last w' coordinates of the key by guessing halves of the key and looking for collisions. If the lattice basis was reduced sufficiently in the first phase, a collision resulting in the private key will be found by applying a rounding algorithm to the half-key guesses. More details on how to do this can be found in [How07].

To estimate the security against this attack we adapt the analysis of [HPS+15] to the set of keys that we use in Streamlined NTRU Prime. Let w be the dimension of I_w and w' be the dimension of $I_{w'}$. For a sufficiently reduced basis the meet-in-the-middle phase will require on average

$$-\frac{1}{2}\left(\log_2(2(p-t)) + \sum_{\substack{0 \le a \le 2t \\ 0 \le b \le 2t-a}} \binom{w'}{a}\binom{w'-a}{b} v(a,b)\log_2(v(a,b))\right) \tag{9.2}$$

work, where the $\log_2(2(p-t))$ term accounts for equivalent keys and

$$v(a,b) = \frac{\sum_{i=a}^{2t-b} \binom{p-w'}{i-a}\binom{p-w'-i+a}{2t-b-i}}{2^{2t}\binom{p}{2t}} = \frac{2^{-a-b}\binom{p-w'}{2t-a-b}}{\binom{p}{2t}}.$$

The quality of a basis after lattice reduction can be measured by the Hermite factor $\delta = ||\mathbf{b_1}||/\det(B)^{1/p}$. Here $||\mathbf{b_1}||$ is the length of the shortest vector among the rows of B. To be able to recover the key in the meet-in-the-middle phase, the $(2p-w-w') \times (2p-w-w')$ matrix T' has to be sufficiently reduced. For given w and w' this is the case if the lattice reduction reaches the required value of δ. This Hermite factor has to satisfy

$$\log_2(\delta) \le \frac{(p-w)\log_2(q)}{(2p-(w+w'))^2} - \frac{1}{2p-(w'+w)}. \tag{9.3}$$

We use the BKZ 2.0 simulator of [CN11a] to determine the best BKZ 2.0 parameters, specifically the "block size" β and the number of "rounds" n, needed to reach a root Hermite factor δ. To get a concrete security estimate of the work required to perform BKZ-2.0 with parameters β and n we use the conservative formula determined by [HPS+15] from the experiments of [CN11b]:

$$\text{Estimate}(\beta, p, n) = 0.000784314\beta^2 + 0.366078\beta - 6.125 + \log_2(p \cdot n) + 7. \tag{9.4}$$

This estimate and the underlying experiments rely on "enumeration"; see Section 9.11 for a comparison to "sieving". This analysis also assumes that the probabily of two halves of the key colliding is 1. We will also conservatively assume this, but a more realistic estimate can be found in [Wun16]. Using these estimates we can determine the optimal w and w' to attack a parameter set and thereby estimate its security.

9.4.4 Algebraic attacks

The attack strategy of Ding [Din10], Arora–Ge [AG11], and Albrecht–Cid–Faugère–Fitzpatrick–Perret [ACF+14] takes subexponential time to break dimension-n LWE with noise width $o(\sqrt{n})$, and polynomial time to break LWE with constant noise width. However, these attacks require many LWE samples, whereas typical cryptosystems such as NTRU and NTRU Prime provide far less data to the attacker. When these attacks are adapted to cryptosystems that provide only (say) $2n$ samples, they end up taking more than $2^{0.5n}$ time, even when the noise is limited to $\{0,1\}$. See generally [ACF+14, Theorem 7] and [Lyu16, Case Study 1].

9.5 Parameters

Algorithm 9.1 searches for (p,q,t,λ), where λ is Section 9.4's estimate of the *pre-quantum* security level for parameters (p,q,t). For example, we used Algorithm 9.1 to find our recommended parameters $(p,q,t) = (739, 9829, 246)$ with estimated pre-quantum security 2^{232}. We expect *post-quantum* security levels to be somewhat lower

Algorithm 9.1 Determine parameter sets for security level above ℓ.

```
Input: Upper bound $q_b$ for $q$, range $[p_1, p_2]$ for $p$, lower bound $\ell$ for security level
Output: Viable parameters $p$, $q$ and $t$ with security level $\lambda$.
$p \leftarrow p_1 - 1$ (the prime we are currently investigating)
while $p \leq p_2$ do
    $p \leftarrow \mathrm{nextprime}(p)$
    $Q \leftarrow \mathrm{viableqs}(p, q_b)$
    for $q \in Q$ do
        $t \leftarrow \min(\lfloor (q-1)/32 \rfloor, \lfloor p/3 \rfloor)$
        $\lambda_1 \leftarrow \mathrm{mitmcosts}(p, t)$
        if $\lambda_1 \geq \ell$ then
            Find $w$, $w'$, $\beta$, $n$ such that BKZ-2.0 costs are approximately equal to
                meet-in-the-middle costs in the hybrid attack.
            $\lambda_2 \leftarrow \max(\mathrm{hybridbkzcost}, \mathrm{hybridmitmcost})$
            return $p, q, t, \min(\lambda_1, \lambda_2)$
```

(e.g., [LMvdP15] saves a factor 1.1 in the best known asymptotic SVP exponents), and lattice security remains a tricky research topic, but there is a comfortable security margin above our target 2^{128}.

In the parameter generation algorithm the subroutine `nextprime`(i) returns the first prime number $>i$. The subroutine `viableqs`(p, q_b) returns all primes q larger than p and smaller than q_b for which it holds that $x^p - x - 1$ is irreducible in $(\mathbb{Z}/q)[x]$. The subroutine `mitmcosts`(p, t) uses the estimates from Equation (9.1) to determine the bitsecurity level of the parameters against a straightforward meet-in-the-middle attack. To find w, w', β, n we set w to the `hybridbkzcost` of the previous iteration (initially 0) and do a binary search for w' such that the two phases of the hybrid attack are of equal cost. For each w' we determine the Hermite factor required with Equation (9.3), use the BKZ-2.0 simulator to determine the optimal β and n to reach the required Hermite factor and use Equations (9.4) and (9.2) to determine the `hybridbkzcost` and `hybridmitmcost`.

Note that this algorithm outputs the largest value of t such that there are no decryption failures according to Theorem 9.1 and that no more than 2/3 of the coefficients of f are set. Experiments show that decreasing t to t_1 linearly decreases the security level by approximately $t - t_1$.

The results of the algorithm for $q_b = 20000$, $[p_1, p_2] = [500, 950]$ and $\ell = 128$ are displayed in Table 9.2.

9.6 Polynomial multiplication

The main bottleneck operation in both encryption and decryption is polynomial multiplication. It is well known that schoolbook multiplication is asymptotically superseded by Karatsuba's method, Toom's method, and FFTs. For large input sizes, it is

Table 9.2: Streamlined NTRU Prime parameter sets with $465 < p < 970$ and $q < 20000$.

Parameters				
p	q	t	λ	key size
467	3911	122	130	5573
479	5689	159	132	5976
	6089	159	130	6022
491	6287	163	135	6196
	8627	163	130	6420
	9277	163	129	6472
499	8243	166	134	4692
	9029	166	132	6558
503	2879	89	137	5781
	8663	167	134	6580
509	10939	169	133	6830
	11087	169	133	6840
521	15271	173	132	7242
	15359	173	132	7246
	16001	173	132	7277
523	3331	104	147	6121
	7151	174	146	6697
	7159	174	146	6698
	11003	174	138	7022
	19853	174	129	7467
541	2297	71	140	6041
	2437	76	150	6087
	17789	180	137	7639
557	4759	148	164	6805
	9323	185	155	7345
	13339	185	148	7633
	13963	185	148	7669
	14827	185	147	7718
	18719	185	143	7906
563	10627	187	155	7531
569	1579	49	129	6046
	3929	122	166	6794
	11489	189	156	7675
	12781	189	154	7763
	16193	189	150	7957
571	4201	131	168	6873
	7177	190	166	7315
577	1861	58	141	6268
	19081	192	150	8205
587	5233	163	176	7252
	8263	195	170	7639
593	1381	43	131	6186
	14423	197	161	8193
	19697	197	156	8460
599	7001	199	178	7652
	9551	199	171	7920
601	17257	200	161	8460
	18701	200	159	8529
607	6317	197	184	7664
	17747	202	163	8568
	18749	202	162	8617
613	1459	45	137	6444
	3319	103	174	7170
	4363	136	183	7412
	9157	204	178	8068
	10529	204	175	8191
	11867	204	173	8297
	12109	204	173	8315
	13799	204	170	8431
617	19469	204	164	8735
	1511	47	140	6517
	12829	205	173	8421
	16007	205	169	8618
	16073	205	169	8621
619	2297	71	160	6912
	6907	206	187	7895
	9397	206	180	8170
	9679	206	180	8196
	12203	206	175	8403
	13933	206	172	8522
	14173	206	172	8537

Parameters				
p	q	t	λ	key size
631	2081	65	158	6956
	2693	84	171	7191
	11287	210	182	8495
	16481	210	174	8840
641	13691	213	182	8808
643	6247	195	199	8108
	14737	214	181	8904
	15797	214	180	8969
	17189	214	178	9047
631	3559	111	188	7633
	16573	215	180	9069
	16883	215	180	9086
	18461	215	178	9170
653	2311	72	169	7297
	4621	144	199	7950
	8419	217	197	8515
	17477	217	182	9203
	17627	217	182	9211
	19163	217	180	9290
	19507	217	179	9307
659	2137	66	166	7290
	6781	211	205	8388
	7481	219	203	8481
	19571	219	182	9395
661	13327	220	191	9058
	15907	220	187	9226
673	1493	46	149	7097
	9413	224	204	8884
	17123	224	190	9465
677	3251	101	192	7899
683	5623	175	214	8509
	13313	227	200	9358
691	1499	46	147	7290
	5471	170	217	8581
	6449	201	219	8745
	12281	230	205	9387
701	17921	233	201	9905
709	10337	236	217	9455
	11923	236	214	9601
719	2087	65	179	7929
	2351	73	185	8053
	5153	161	226	8867
	9133	239	225	9460
	10531	239	221	9608
	10739	239	221	9628
	14831	239	213	9963
	19079	239	208	10224
727	5827	182	232	9094
	12241	242	221	9873
	17317	242	213	10236
739	9829	246	232	9802
	10859	246	229	9908
	12713	246	225	10076
	17183	246	218	10397
	19429	246	216	10528
743	7541	235	240	9571
	11251	247	231	10000
	16451	247	221	10407
	17959	247	219	10501
751	3067	95	211	8699
	3823	119	223	8938
757	1193	37	138	7737
	3727	116	224	8981
	6869	214	247	9649
	7879	246	246	9799
	10979	252	237	10161
	12973	252	233	10344
	13789	252	231	10410
	14737	252	230	10483
761	1619	50	170	8113
	4091	127	229	9131

Parameters				
p	q	t	λ	key size
761	4591	143	248	9258
	7883	246	248	9851
	13829	253	233	10468
	14107	253	232	10490
	19001	253	225	10817
769	1433	44	162	8063
	6599	206	251	9758
	17729	256	231	10854
773	877	27	131	7558
	2099	65	190	8531
	8317	257	252	10066
	9811	257	247	10251
	13757	257	239	10628
787	4243	132	240	9485
797	1259	39	158	8208
809	1801	56	187	8749
	6113	191	265	10176
	14107	269	254	11152
811	8543	266	269	10593
	10457	270	263	10829
	11831	270	260	10974
	14083	270	255	11177
823	4513	141	255	9992
	8069	252	275	10681
	11197	274	267	11070
827	7219	225	276	10601
	9767	275	273	10961
	13159	275	264	11317
	19081	275	254	11760
829	1657	51	183	8866
	12227	276	267	11256
	13037	276	265	11333
	14107	276	263	11427
853	9721	284	285	11300
	12377	284	278	11597
	15511	284	272	11875
857	5167	161	274	10572
	13367	285	278	11747
	14797	285	275	11872
859	12487	286	280	11690
	17203	286	272	12087
863	1523	47	182	9125
	4111	128	263	10361
	8779	274	293	11306
881	3217	100	248	10265
	7673	239	300	11370
	15733	293	284	12283
883	8089	252	302	11463
	14639	294	287	12219
887	13007	295	292	12123
907	7727	241	311	11715
	8807	275	313	11886
	12109	302	304	12303
	15467	302	297	12622
	17389	302	293	12776
919	11827	306	311	12434
	13933	306	306	12652
	14771	306	304	12729
929	12953	309	313	12692
	14011	309	310	12797
	18229	309	302	13149
937	1823	56	206	10150
	12401	312	318	12742
941	2521	78	239	10634
	10781	313	324	12606
947	3917	122	282	11303
953	6343	198	322	12038
	8237	257	333	12397
	16693	317	316	13367
967	8243	257	338	12580

Note: The estimated pre-quantum security level is 2^{λ}. Parameter sets with $\lambda < 128$ are omitted. Keysize is computed as $\lceil p \log_2(q) \rceil$ (see Section9.2.2).

clear that the FFT is the best. However, for small to medium input sizes, it is unclear which methods or combinations of methods are best.

We analyzed many different combinations of schoolbook multiplication, refined Karatsuba, the arbitrary-degree variant of Karatsuba for degrees 3, 4, 5, or 6, and Toom's method for splitting into 3, 4, 5, or 6 pieces. We considered sizes up to $1024n \times 1024n$ (where n reflects the number of bits, limbs or terms). We analyzed the resulting ranges of double-precision floating-point numbers (53-bit mantissa) for various input sizes, making sure to avoid overflows.

After comparing the results of this analysis to the parameter possibilities in Table 9.2, we decided to focus on $768n \times 768n$. This section explains how we decompose $768n$ into $128n$ using Toom6, then decompose $128n$ into $4n$ using five levels of refined Karatsuba, then use schoolbook multiplication for $4n \times 4n$. All 6 pieces of the Toom6 computation are of the same size, each half of refined Karatsuba at each level is of the same size, and everything is 4-way vectorizable; we exploit this in Section 9.7.

9.6.1 Top level

At the top level we use Toom6 to decompose $768n$ into 6 pieces of size $128n$. For instance, let one of the $768n$ polynomials be $a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{767}x^{767}$. It is then decomposed into

$$a(x,y) = A_0(x) + A_1(x)y + A_2(x)y^2 + A_3(x)y^3 + A_4(x)y^4 + A_5(x)y^5,$$

where $y = x^{128}$ and

$$\begin{aligned}
A_0(x) &= a_0 \quad + a_1 \quad x + a_2 \quad x^2 + \cdots + a_{127}x^{127};\\
A_1(x) &= a_{128} + a_{129}x + a_{130}x^2 + \cdots + a_{255}x^{127};\\
A_2(x) &= a_{256} + a_{257}x + a_{258}x^2 + \cdots + a_{383}x^{127};\\
A_3(x) &= a_{384} + a_{385}x + a_{386}x^2 + \cdots + a_{511}x^{127};\\
A_4(x) &= a_{512} + a_{513}x + a_{514}x^2 + \cdots + a_{639}x^{127};\\
A_5(x) &= a_{640} + a_{641}x + a_{642}x^2 + \cdots + a_{767}x^{127}.
\end{aligned}$$

Let another polynomial be $b(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{767}x^{767}$. We can decompose this in a similar way as a, such that the multiplication of a and b becomes

$$\begin{aligned}
ab = C_0 + C_1x^{128} + C_2x^{256} + C_3x^{384} + C_4x^{512} + C_5x^{640}&\\
+ C_6x^{768} + C_7x^{896} + C_8x^{1024} + C_9x^{1152} + C_{10}x^{1280},&
\end{aligned}$$

where

$$
\begin{aligned}
C_0 &= A_0B_0;\\
C_1 &= A_0B_1 + A_1B_0;\\
C_2 &= A_0B_2 + A_1B_1 + A_2B_0;\\
C_3 &= A_0B_3 + A_1B_2 + A_2B_1 + A_3B_0;\\
C_4 &= A_0B_4 + A_1B_3 + A_2B_2 + A_3B_1 + A_4B_0;\\
C_5 &= A_0B_5 + A_1B_4 + A_2B_3 + A_3B_2 + A_4B_1 + A_5B_0;\\
C_6 &= A_1B_5 + A_2B_4 + A_3B_3 + A_4B_2 + A_5B_1;\\
C_7 &= A_2B_5 + A_3B_4 + A_4B_3 + A_5B_2;\\
C_8 &= A_3B_5 + A_4B_4 + A_5B_3;\\
C_9 &= A_4B_5 + A_5B_4;\\
C_{10} &= A_5B_5.
\end{aligned}
$$

Note that we leave out x but A_i and B_i are polynomials in x.

There are 11 values C_i, therefore Toom6 requires 11 multiplications (see below) in order to interpolate those C_i. We chose to evaluate y at 0, ±1, ±2, ±3, ±4, 5 and ∞. One of the advantages of using + and − is that some intermediate results can be reused to save some computations. For example, to compute $s_1 = a(x,1) \cdot b(x,1)$ and $s_{-1} = a(x,-1) \cdot b(x,-1)$, we first compute $T_0 = A_0(x) + A_2(x) + A_4(x)$ and $T_1 = A_1(x) + A_3(x) + A_5(x)$, then compute $T_0 + T_1$ and $T_0 - T_1$ to obtain s_1 and s_{-1} respectively. The 11 multiplications of $128n \times 128n$ that we need to compute are

$$
\begin{aligned}
y &= 0: A_0 \cdot B_0;\\
y &= 1: (A_0 + A_1 + A_2 + A_3 + A_4 + A_5) \cdot (B_0 + B_1 + B_2 + B_3 + B_4 + B_5);\\
y &= -1: (A_0 - A_1 + A_2 - A_3 + A_4 - A_5) \cdot (B_0 - B_1 + B_2 - B_3 + B_4 - B_5);\\
y &= 2: (A_0 + 2A_1 + 2^2A_2 + 2^3A_3 + 2^4A_4 + 2^5A_5) \cdot (B_0 + 2B_1 + 2^2B_2 + 2^3B_3 + 2^4B_4 + 2^5B_5);\\
y &= -2: (A_0 - 2A_1 + 2^2A_2 - 2^3A_3 + 2^4A_4 - 2^5A_5) \cdot (B_0 - 2B_1 + 2^2B_2 - 2^3B_3 + 2^4B_4 - 2^5B_5);\\
y &= 3: (A_0 + 3A_1 + 3^2A_2 + 3^3A_3 + 3^4A_4 + 3^5A_5) \cdot (B_0 + 3B_1 + 3^2B_2 + 3^3B_3 + 3^4B_4 + 3^5B_5);\\
y &= -3: (A_0 - 3A_1 + 3^2A_2 - 3^3A_3 + 3^4A_4 - 3^5A_5) \cdot (B_0 - 3B_1 + 3^2B_2 - 3^3B_3 + 3^4B_4 - 3^5B_5);\\
y &= 4: (A_0 + 4A_1 + 4^2A_2 + 4^3A_3 + 4^4A_4 + 4^5A_5) \cdot (B_0 + 4B_1 + 4^2B_2 + 4^3B_3 + 4^4B_4 + 4^5B_5);\\
y &= -4: (A_0 - 4A_1 + 4^2A_2 - 4^3A_3 + 4^4A_4 - 4^5A_5) \cdot (B_0 - 4B_1 + 4^2B_2 - 4^3B_3 + 4^4B_4 - 4^5B_5);\\
y &= 5: (A_0 + 5A_1 + 5^2A_2 + 5^3A_3 + 5^4A_4 + 5^5A_5) \cdot (B_0 + 5B_1 + 5^2B_2 + 5^3B_3 + 5^4B_4 + 5^5B_5);\\
y &= \infty: A_5 \cdot B_5.
\end{aligned}
$$

9.6.2 Middle level

The middle-level 11 multiplications of $128n \times 128n$ inside Toom6 are computed using 5-level refined Karatsuba. Recall the “refined Karatsuba identity” from [Ber09a, Section 2]:

$$(F_0+T^nF_1)(G_0+T^nG_1)=(1-T^n)(F_0G_0-T^nF_1G_1)+T^n(F_0+F_1)(G_0+G_1).$$

Level 1. For the first level of Karatsuba, we split one $128n$ of f (and one $128n$ of g) into two $64n$'s, namely, F_0 and F_1, with $F=F_0+x^{64}F_1$ (and into two $64n$'s, namely, G_0 and G_1, with $G=G_0+x^{64}G_1$) as

$$\begin{aligned} F_0&=f_0+f_1x+f_2x^2+\cdots+f_{63}x^{63}; & F_1&=f_{64}+f_{65}x+f_{66}x^2+\cdots+f_{127}x^{63};\\ G_0&=g_0+g_1x+g_2x^2+\cdots+g_{63}x^{63}; & G_1&=g_{64}+g_{65}x+g_{66}x^2+\cdots+g_{127}x^{63}. \end{aligned}$$

Then, we have

$$fg=(1-x^{64})(F_0G_0-x^{64}F_1G_1)+x^{64}(F_0+F_1)(G_0+G_1).$$

Level 2. For the second level, we further split one $64n$ of F_0 (and those of F_1) into two $32n$'s, namely, F_{00} and F_{01}, with $F_0=F_{00}+x^{32}F_{01}$ (and into F_{10} and F_{11} with $F_1=F_{10}+x^{32}F_{11}$) as

$$\begin{aligned} F_{00}&=f_0+f_1x+f_2x^2+\cdots+f_{31}x^{31}; & F_{01}&=f_{32}+f_{33}x+f_{34}x^2+\cdots+f_{63}x^{31};\\ F_{10}&=f_{64}+f_{65}x+f_{66}x^2+\cdots+f_{95}x^{31}; & F_{11}&=f_{96}+f_{97}x+f_{98}x^2+\cdots+f_{127}x^{31}. \end{aligned}$$

Let $F_2=F_0+F_1$, then F_2 is further split into F_{20} and F_{21} with $F_2=F_{20}+x^{32}f_{21}$ as

$$\begin{aligned} F_{20}&=(f_0+f_{64})+(f_1+f_{65})x+(f_2+f_{66})x^2+\cdots+(f_{31}+f_{95})x^{31};\\ F_{21}&=(f_{32}+f_{96})+(f_{33}+f_{97})x+(f_{34}+f_{98})x^2+\cdots+(f_{63}+f_{127})x^{31}. \end{aligned}$$

Similarly, we split G_0, G_1 and $G_2=G_0+G_1$ to obtain G_{00}, G_{01}, G_{10}, G_{11}, G_{20} and G_{21}. Then, we have

$$\begin{aligned} F_0G_0&=(1-x^{32})(F_{00}G_{00}-x^{32}F_{01}G_{01})+x^{32}(F_{00}+F_{01})(G_{00}+G_{01});\\ F_1G_1&=(1-x^{32})(F_{10}G_{10}-x^{32}F_{11}G_{11})+x^{32}(F_{10}+F_{11})(G_{10}+G_{11});\\ F_2G_2&=(1-x^{32})(F_{20}G_{20}-x^{32}F_{21}G_{21})+x^{32}(F_{20}+F_{21})(G_{20}+G_{21}). \end{aligned}$$

Level 3. For the third level, we further split one $32n$ into two $16n$'s, for instance, F_{00} is split into F_{000} and F_{001} with $F_{00}=F_{000}+x^{16}F_{001}$ as

$$F_{000}=f_0+f_1x+f_2x^2+\cdots+f_{15}x^{15};\qquad F_{001}=f_{16}+f_{17}x+f_{18}x^2+\cdots+f_{31}x^{15}.$$

Similar splits also apply to F_{01}, F_{10}, F_{11}, F_{20} and F_{21}.

Let $F_{02}=F_{00}+F_{01}$, then F_{02} is further split into F_{020} and F_{021} with $F_{02}=F_{020}+x^{16}F_{021}$ as

$$\begin{aligned} F_{020}&=(f_0+f_{32})+(f_1+f_{33})x+(f_2+f_{34})x^2+\cdots+(f_{15}+f_{47})x^{15};\\ F_{021}&=(f_{16}+f_{48})+(f_{17}+f_{49})x+(f_{18}+f_{50})x^2+\cdots+(f_{31}+f_{63})x^{15}. \end{aligned}$$

Similar splits also apply to $F_{12} = F_{10} + F_{11}$ and $F_{22} = F_{20} + F_{21}$.

We do the same for G_{ij} where $0 \leq i, j \leq 2$. Then, we have

$$F_{ij}G_{ij} = (1 - x^{16})(F_{ij0}G_{ij0} - x^{16}F_{ij1}G_{ij1}) + x^{16}(F_{ij0} + F_{ij1})(G_{ij0} + G_{ij1});$$

where $0 \leq i, j \leq 2$.

Level 4. For the fourth level, we further split one $16n$ into two $8n$'s, for instance, F_{000} is split into F_{0000} and F_{0001} with $F_{000} = F_{0000} + x^8 F_{0001}$ as

$$F_{0000} = f_0 + f_1x + f_2x^2 + \cdots + f_7x^7; \qquad F_{0001} = f_8 + f_9x + f_{10}x^2 + \cdots + f_{15}x^7.$$

Similar splits also apply to F_{ijk} and G_{ijk} where $0 \leq i, j, k \leq 2$. Then we have

$$F_{ijk}G_{ijk} = (1 - x^8)(F_{ijk0}G_{ijk0} - x^8 F_{ijk1}G_{ijk1}) + x^8(F_{ijk0} + F_{ijk1})(G_{ijk0} + G_{ijk1});$$

where $0 \leq i, j, k \leq 2$.

Level 5. Finally, for the fifth level, we further split one $8n$ into two $4n$'s, for instance, F_{0000} is split into F_{00000} and F_{00001} with $F_{0000} = F_{00000} + x^4 F_{00001}$ as

$$F_{00000} = f_0 + f_1x + f_2x^2 + f_3x^3; \qquad F_{00001} = f_4 + f_5x + f_6x^2 + f_7x^3.$$

Similar splits also apply to F_{ijkl} and G_{ijkl} where $0 \leq i, j, k, l \leq 2$. Then we have

$$\begin{aligned} F_{ijkl}G_{ijkl} = (1 - x^8)(F_{ijkl0}G_{ijkl0} &- x^8 F_{ijkl1}G_{ijkl1}) \\ &+ x^8(F_{ijkl0} + F_{ijkl1})(G_{ijkl0} + G_{ijkl1}); \end{aligned}$$

where $0 \leq i, j, k, l \leq 2$.

9.6.3 Lowest level

The lowest-level multiplication of $4n \times 4n$ is computed using schoolbook multiplication. For instance, $F_{00000}G_{00000}$ is computed as follows

$$\begin{aligned} h_0 &= f_0g_0; & h_4 &= f_1g_3 + f_2g_2 + f_3g_1; \\ h_1 &= f_0g_1 + f_1g_0; & h_5 &= f_2g_3 + f_3g_2; \\ h_2 &= f_0g_2 + f_1g_1 + f_2g_0; & h_6 &= f_3g_3. \\ h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0; \end{aligned}$$

Since we use 5-level Karatsuba, we need to perform 3^5 $4n \times 4n$ multiplications to do one 128×128 computation.

9.7 Vectorization

Each Haswell core has two 256-bit floating-point vector multiplication units. These compute 4-way vectorized multiplications which are also integrated with additions.

This means that in one cycle, the Haswell CPU can compute 8 independent multiply-accumulate instructions $ab + c$ for 64-bit double-precision inputs a, b, c.

Most of the computations in Section 9.6 are obviously suitable for vectorization. We vectorize the computations of Toom6 for decomposing a and b in the top level (before the 11 multiplications) by taking four consecutive polynomial coefficients for each vector. For example, to evaluate at $y = 1$, we have to compute $A_0 + A_1$ as part of $A_0 + A_1 + A_2 + A_3 + A_4 + A_5$. We take a_0, a_1, a_2, a_3 as one vector and $a_{128}, a_{129}, a_{130}, a_{131}$ as another vector, then add them together. We then move to the next four coefficients of A_0 and A_1.

We also vectorize computations of refined Karatsuba for splitting inputs in the middle level (before the schoolbook multiplication), similarly to the Toom6 decomposition. For example, to compute $F_{20} = F_0 + F_1 = (f_0 + f_{64}) + (f_1 + f_{65})x + (f_2 + f_{66})x^2 + \cdots + (f_{31} + f_{95})x^{31}$, we take f_0, f_1, f_2, f_3 as inputs for one vector and $f_{64}, f_{65}, f_{66}, f_{67}$ for another vector, then add them together.

The $4n \times 4n$ schoolbook multiplications in the lowest level violate this vector structure. Each schoolbook multiplication uses 16 multiplications, including 9 multiply-accumulate instructions, with extensive communication across input lanes. Instead of trying to vectorize *inside* a schoolbook multiplication, we transpose inputs and vectorize *across* independent schoolbook multiplications. Inside a $128n \times 128n$ multiplication there are many (specifically, $3^5 = 243$) of these independent multiplications. For example, we vectorize 16 multiplications of $F_{00000}G_{00000}$, $F_{00001}G_{00001}$, $F_{00010}G_{00010}$ and $F_{00011}G_{00011}$ together.

We transpose the results of schoolbook multiplication back to the original format. We vectorize the merging of results in refined Karatsuba and the interpolation of C_i in Toom6 in the same way as splitting refined Karatsuba and decomposing Toom6.

We benchmarked our software on an Intel Haswell CPU, being careful to disable Turbo Boost. Each multiplication in $(\mathbb{Z}/9829)[x]/(x^{739} - x - 1)$ takes just 51488 cycles.

9.8 Reference implementation

This section presents a complete non-constant-time reference implementation of the proposed Streamlined NTRU Prime 9829^{739} using the Sage computer-algebra system.

```
p = 739; q = 9829; t = 246
Zx.<x> = ZZ[]; R.<xp> = Zx.quotient(x^p-x-1)
Fq = GF(q); Fqx.<xq> = Fq[]; Rq.<xqp> = Fqx.quotient(x^p-x-1)
F3 = GF(3); F3x.<x3> = F3[]; R3.<x3p> = F3x.quotient(x^p-x-1)

import itertools
def concat(lists): return list(itertools.chain.from_iterable(lists))

def nicelift(u):
  return lift(u + q//2) - q//2

def nicemod3(u): # r in 0,1,-1 with u-r in ...,-3,0,3,...
  return u - 3*round(u/3)

def int2str(u,bytes):
  return "".join([chr((u//256^i)%256) for i in range(bytes)])
```

```
def str2int(s):
  return sum([ord(s[i])*256^i for i in range(len(s))])

def encodeZx(m): # assumes coefficients in range -1,0,1,2
  m = [m[i]+1 for i in range(p)] + [0]*(-p % 4)
  return "".join([int2str(m[i]+m[i+1]*4+m[i+2]*16+m[i+3]*64,1) for i in range(0,len(m),4)])

def decodeZx(mstr):
  m = [str2int(mstr[i:i+1]) for i in range(len(mstr))]
  m = concat([[m[i]%4,(m[i]//4)%4,(m[i]//16)%4,m[i]//64] for i in range(len(m))])
  return Zx([m[i]-1 for i in range(p)])

def encodeRq(h):
  h = [lift(h[i]) for i in range(p)] + [0]*(-p % 3)
  h = "".join([int2str(h[i]+h[i+1]*10240+h[i+2]*10240^2,5) for i in range(0,len(h),3)])
  return h[0:1232]

def decodeRq(hstr):
  h = [str2int(hstr[i:i+5]) for i in range(0,len(hstr),5)]
  h = concat([[h[i]%10240,(h[i]//10240)%10240,h[i]//10240^2] for i in range(len(h))])
  if max(h) >= q: raise Exception("pk out of range")
  return Rq(h)

def encoderoundedRq(c):
  c = [1638 + nicelift(c[i]/3) for i in range(p)] + [0]*(-p % 2)
  c = "".join([int2str(c[i]+c[i+1]*4096,3) for i in range(0,len(c),2)])
  return c[0:1109]

def decoderoundedRq(cstr):
  c = [str2int(cstr[i:i+3]) for i in range(0,len(cstr),3)]
  c = concat([[c[i]%4096,c[i]//4096] for i in range(len(c))])
  if max(c) > 3276: raise Exception("c out of range")
  return 3*Rq([c[i]-1638 for i in range(p)])

def randomR(): # R element with 2t coeffs +-1
  L = [2*randrange(2^31) for i in range(2*t)]
  L += [4*randrange(2^30)+1 for i in range(p-2*t)]
  L.sort()
  L = [(L[i]%4)-1 for i in range(p)]
  return Zx(L)

def keygen():
  while True:
    g = Zx([randrange(3)-1 for i in range(p)])
    if R3(g).is_unit(): break
  f = randomR()
  h = Rq(g)/(3*Rq(f))
  pk = encodeRq(h)
  return pk,encodeZx(f) + encodeZx(R(lift(1/R3(g)))) + pk

import hashlib
def hash(s): h = hashlib.sha512(); h.update(s); return h.digest()

def encapsulate(pk):
  h = decodeRq(pk)
  r = randomR()
  hr = h * Rq(r)
  m = Zx([-nicemod3(nicelift(hr[i])) for i in range(p)])
  c = Rq(m) + hr
  fullkey = hash(encodeZx(r))
  return fullkey[:32] + encoderoundedRq(c),fullkey[32:]

def decapsulate(cstr,sk):
  f,ginv,h = decodeZx(sk[:185]),decodeZx(sk[185:370]),decodeRq(sk[370:])
  confirm,c = cstr[:32],decoderoundedRq(cstr[32:])
  f3mgr = Rq(3*f) * c
```

```
  f3mgr = [nicelift(f3mgr[i]) for i in range(p)]
  r = R3(ginv) * R3(f3mgr)
  r = Zx([nicemod3(lift(r[i])) for i in range(p)])
  hr = h * Rq(r)
  m = Zx([-nicemod3(nicelift(hr[i])) for i in range(p)])
  checkc = Rq(m) + hr
  fullkey = hash(encodeZx(r))
  if sum([r[i]==0 for i in range(p)]) != p-2*t: return False
  if checkc != c: return False
  if fullkey[:32] != confirm: return False
  return fullkey[32:]

for keys in range(5):
  pk,sk = keygen()
  for ciphertexts in range(5):
    c,k = encapsulate(pk)
    assert decapsulate(c,sk) == k

print len(pk),"bytes in public key"
print len(sk),"bytes in secret key"
print len(c),"bytes in ciphertext"
print len(k),"bytes in shared secret"
```

9.9 Appendix: Public-key encryption vs. unauthenticated key exchange

We mentioned earlier that multiplication in [ADPS16] costs only about 40000 cycles and about 33000 cycles in [LN16]. However, the complete cryptographic operations reported in [LN16] cost 105k cycles for the client and 92.1k+15.7k=107.8k cycles for the server. Gueron and Schlieker [GS16] have very recently reported 80087 Skylake cycles for the client and 84119 Skylake cycles for the server; Skylake is newer and often faster than Haswell, making a direct comparison difficult, but it is clear that the client is spending at least as much time as two multiplications, whereas one would expect encryption to be bottlenecked by a single multiplication. A comparison of network traffic is even more striking: in the latest version of [ADPS16], the server sends 1824 bytes and the client sends 2048 bytes; for us a ciphertext is just 1141 bytes.

The main reason for the scale of costs in [ADPS16] is that [ADPS16] is targeting unauthenticated key exchange, a larger operation than traditional public-key encryption. The bigger picture is that [ADPS16] and this chapter are actually taking two completely different approaches to securing communication. Both approaches support the most urgent goal of post-quantum cryptography, namely encrypting today's data in a way that will not be decrypted by future quantum computers. Both approaches also support server authentication, so that the client will not be fooled into encrypting data to a "man in the middle" rather than the server. However, the details and costs of these two approaches are fundamentally quite different.

In the first approach, the server's long-term identifier is a public key for a *signature system*. To start a secure session, the client and server perform an unauthenticated post-quantum key exchange (as in [ADPS16]), obtaining a shared secret key used to authenticate and encrypt subsequent messages by standard symmetric techniques.

The server signs a hash of the key exchange, so the client knows that it is talking to the server; this is what stops the "man in the middle". At the end of the session, the client and server erase the shared secret key.

In the second approach, the server's long-term identifier is a public key for an *encryption system*. To start a secure session, the client sends a ciphertext to the server. Decryption provides both the client and the server with a shared secret key used to authenticate and encrypt subsequent messages; see our discussion of KEMs in Section 9.3.4. The client knows that it is talking to the server since nobody else has the shared secret key.

The second approach is less expensive for several reasons:

- In the first approach, the client cost is signature verification *plus* the client side of unauthenticated key exchange. In the second approach, the client cost is merely one public-key encryption.
- In the first approach, the server cost is signature generation *plus* the server side of unauthenticated key exchange. In the second approach, the server cost is merely one decryption.
- In the first approach, the network traffic is a signature *plus* unauthenticated key exchange. In the second approach, the network traffic is merely one ciphertext.

As a concrete example of total costs for the first approach, the very recent paper [dPLP16, Table 2] combines [ADPS16] with a lattice-based signature system, reporting a total of more than 4700 bytes for post-quantum authenticated-server key exchange after significant compression effort. CPU time is not reported in [dPLP16] but presumably the signatures add significantly to the cost of [ADPS16].

As another example, [ADPS16] suggests combining post-quantum unauthenticated key exchange with the current world of pre-quantum signatures. The total traffic is then 3936 bytes, assuming a typical 64-byte ECC signature. The total CPU time is the time reported in [ADPS16] *plus* the cost of generating and verifying an ECC signature.

This type of combination provides *transitional* security: if the signature is verified before the attacker has a quantum computer then both integrity and confidentiality are protected, even against future quantum computers. However, it does not provide post-quantum security: if the signature is verified *after* the attacker has a quantum computer then neither integrity nor confidentiality is protected. Users will thus need *another* upgrade, switching all deployments to post-quantum signatures before attackers have quantum computers.

For comparison, we use just 1141 bytes of network traffic to set up a session key with true post-quantum server authentication. There is no need for a subsequent upgrade.

9.9.1 Key erasure ("forward secrecy")

An attacker who steals physical server hardware has a copy of the server's long-term secret key. The attacker can pose as the server for as long as this key is valid: often

Table 9.3: Bandwidth used by different techniques of authenticated-server key exchange

Total bytes	Client bytes	Server bytes	Key erasure	Source
1141	1141	0	no	this chapter
3514	1141+1141	1232	yes	this chapter
3514	1141+1232	1141	yes	this chapter
3872+sig.	2048	1824+sig.	yes	[ADPS16]+sig.

Note: The above bandwidth calculation assumes client already knows server's long-term key. In first line, long-term key is encryption key; client sends ciphertext; session key is hash of plaintext. In second line, server also sends short-term encryption key; client sends another ciphertext to that key; session key is hash of two plaintexts. In third line, short-term encryption key is generated by client rather than server. In fourth line, long-term key is signature key, and server signs hash of unauthenticated key exchange. "sig." means signature.

90 days, often much longer. Furthermore, if the key is an encryption key, then the attacker can decrypt any previously recorded ciphertexts for this key. The low-cost encryption approach described above does not provide fast key erasure: there are many ciphertexts encrypted to the server's long-term key, and the plaintexts expose useful information.

For comparison, in a TLS "ECDHE" session, the client and server exchange short-term ECC keys and use the corresponding Diffie–Hellman shared secret as a session key. The client and server subsequently switch to new short-term ECC keys and erase their old secret keys, so they have no way to recompute the shared secret. Servers nevertheless retain keys for "many months" in some cases, as explained in [Lan13], but this period is no longer tied to the lifetime of the signature key that identifies the server; it is possible for a good implementation to erase keys much more quickly. Generating a new short-term key every minute has negligible cost.

The least expensive way to add post-quantum security to ECDHE is to add a layer of post-quantum public-key encryption using a long-term server key. Attackers stealing keys are stopped by ECDHE if they do not have quantum computers, and attackers with quantum computers are stopped by the post-quantum encryption if they are not stealing keys. However, it might still be possible for an attacker to use key theft to break the post-quantum system *and* a quantum computer to break ECDHE, so there is a comprehensible security benefit to deploying post-quantum key erasure.

Post-quantum key erasure is compatible with the second approach described at the beginning of Section 9.9, although it does increase costs. The simplest protocol allowing key erasure is as follows. The server maintains a long-term post-quantum public key and also a short-term post-quantum public key. The client sends a ciphertext to each of these keys, and the two plaintexts are hashed to produce the shared secret key. An attacker who later steals the server's secrets does not know the short-term secret key (the key has been erased), cannot decrypt the ciphertext sent to that key, and cannot compute the shared secret.

With this protocol, the client performs two public-key encryption operations, and the server performs two decryption operations, assuming key-generation costs are amortized. This is an *authenticated-server* key-exchange protocol using the same amount of computation as a naive *unauthenticated* key-exchange protocol in which the client sends a ciphertext to a short-term server key and the server sends a ciphertext to a short-term client key. One would expect an optimized unauthenticated key-exchange protocol to be somewhat faster than this, but it is not at all clear that this speedup can outweigh the costs of generating and verifying signatures.

As for bandwidth, this protocol requires the server to send its short-term post-quantum public key to the client. This could be smaller or larger than the signature in the first approach. See Table 9.3 for a comparison of the bandwidth of various techniques for authenticated-server key exchange. Similar comments apply to authenticated-client authenticated-server key exchange, but for simplicity we skip the details.

To summarize, a post-quantum public-key cryptosystem is a simple, highly flexible tool that provides all desired long-term security features. Combining signatures with unauthenticated key exchange is more complicated, does not add any security features, and does not seem to provide significant performance benefits.

9.9.2 Questioning the value of unauthenticated key exchange

With the above analysis in mind, we review the arguments given in [ADPS16, Section 2.3] for studying unauthenticated key exchange.

The first argument is that the "protection of stored transcripts against future decryption using quantum computers" is "much more urgent" than post-quantum authentication. This is indisputably the most urgent issue for many users. However, unauthenticated key exchange is not the only way to achieve this protection: a pure post-quantum public-key cryptosystem provides the same protection, and *also* straightforwardly provides authentication.

The second argument is that the unauthenticated key-exchange protocol given in [ADPS16] is simpler than earlier authenticated key-exchange protocols. However, this ignores the possibility of building authenticated key exchange in an even simpler way from post-quantum public-key encryption.

The third argument is that combining unauthenticated key exchange with signatures allows the two components to be designed and optimized separately. However, it is not clear how this is better than building the same security features from a single well-optimized component, namely a public-key cryptosystem.

9.10 Appendix: Worst-case-to-average-case reductions

We now consider an argument against our recommended defenses: specifically, an argument that using cyclotomic fields with split modulus (i.e., with P splitting into linear factors in $(\mathbb{Z}/q)[x]$) is *desirable* for security, whereas we recommend *against* this choice.

The argument begins with statements from Lyubashevsky, Peikert, and Regev [LPR13] that the Ring-LWE problem—with cyclotomic P, split q, and a wide error—has "very strong hardness guarantees" and in turn provides a "truly practical lattice-based public-key cryptosystem with an efficient security reduction". These statements allude to a conversion

- from any attack algorithm against the cryptosystem
- into an algorithm to solve the worst case of a "hard" SVP-like ideal-lattice problem.

This conversion is, internally, the composition of three theorems, first producing an algorithm to attack Decision-Ring-LWE, then producing an algorithm to attack Search-Ring-LWE, and finally producing an algorithm to attack the "hard" problem.

These statements and theorems have created a common belief that some "truly practical" lattice-based cryptosystems are guaranteed to be secure. The particular cryptosystems that are subjected to this belief have cyclotomic P (occasionally generalized to Galois P, but never beyond) and split q. The belief is not directly contradicted by, e.g., attacks against the Smart–Vercauteren system: no theorems of this type have been proven for that system.

A more extreme argument against NTRU Prime—*and* against NTRU *and* Ring-LWE-based systems—is the argument that one should actually use the original LWE problem. This argument begins with similar theorems, but this time the conversion (introduced by Regev [Reg05]) applies to different "LWE-based" cryptosystems, and the conversion ends with the worst case of a "hard" SVP-like lattice problem. If all relevant parameters are equal then this problem is clearly at least as difficult to break as the worst case of an SVP-like ideal-lattice problem, since ideal lattices are a special case of lattices.

We have four counterarguments to both of these arguments. First, asymptotic attacks against SVP have improved dramatically in the last few years, reducing the asymptotic security level of d-dimensional lattices from approximately $0.41d$ bits to approximately $0.29d$ bits. This does not mean that there is any loss of security in, e.g., NTRU (see Section 9.11 below), but it calls into question the notion that SVP has been thoroughly studied.

Second, even in the extreme context of LWE, the allegedly "hard" SVP-like lattice problems are *not* the classic SVP problem. The same issue is even more obvious in the context of Ring-LWE: the "hard" SVP-like ideal-lattice problems are considerably more complicated, and less attractive to cryptanalysts, than truly well-known problems such as SVP. It is not easy to justify the notion that these allegedly "hard" problems have been studied more thoroughly than, e.g., the problem of breaking the NTRU cryptosystem. Simply labeling problems as "hard" does not make them so.

Third, it is not true that the cryptosystems have been proven to be as difficult to break as the "hard" problems. The underlying issue is that the conversion is very far from tight. Even if one assumes that there are no better attacks against the "hard" problems than the attacks known today, the conversion does not guarantee a reasonable cryptographic security level for any reasonably efficient cryptosystem. We have not found any paper proposing a specific lattice-based cryptosystem for which the conversion is meaningful.

Our work analyzing the tightness of this conversion is less detailed than an independent analysis by Chatterjee, Koblitz, Menezes, and Sarkar [CKMS16, Section 6]. The analysis in [CKMS16] features an astonishing 2^{504} tightness gap for reasonable LWE parameters. The analysis concludes that there is a "flimsy scientific foundation" for "the claim that, because of worst-case/average-case reductions, the more recent lattice-based encryption schemes have better security than classical NTRU".

Fourth, even if the conversion were tight, switching to these "provable" cryptosystems would impose considerable costs, as illustrated by the analysis of [CWB14]. This raises an important question mentioned in Section 9.1: would it be better for security to use the increased costs in other ways? In particular, taking larger parameters in the original cryptosystem would straightforwardly increase security against all known attacks. Compared to this option, switching to "provable" cryptosystems at the same cost means *reducing* security against known attacks. The argument to impose this security loss is, fundamentally, speculation that attacks against the original cryptosystem will improve much more than attacks against an SVP-like ideal-lattice problem that is claimed to be "hard" but that has little evidence of serious study.

Replacing Ring-LWE with the more extreme case of LWE somewhat simplifies the statement of the "hard" problem but imposes even more serious costs: "size be damned", in the words of a commentator who does not wish to be identified. Switching to LWE is likely to make network traffic a bottleneck. If this is affordable then much larger parameters in the original cryptosystem should also be affordable, producing an even larger increase in security against all known attacks. The argument for LWE thus rests on an even more extreme speculation regarding the progress of attacks.

For comparison, our recommendation to switch from NTRU to Streamlined NTRU Prime generally *reduces* costs and thus does not produce a security loss against known attacks. This recommendation is thus quite different from recommendations to switch to cryptosystems based on Ring-LWE, or more extreme cryptosystems based on LWE.

Notice that none of these counterarguments are questioning the *correctness* of the proofs of the aforementioned theorems. The core problem is that, even if the theorems are correct exactly as stated, there are severe restrictions in what the theorems actually say.

9.11 Appendix: Sieving algorithms

The security estimates in Section 9.4 rely on enumeration algorithms [Poh81, FP85, Kan83, HPS+15]. For very large dimensions, the performance of enumeration algorithms is slightly super-exponential and is known to be suboptimal. The provable sieving algorithms of Pujol and Stehlé [PS09] solve dimension-β SVP in time $2^{2.465\ldots\beta+o(\beta)}$ and space $2^{1.233\ldots\beta+o(\beta)}$, and more recent SVP algorithms [ADRS14] take time $2^{\beta+o(\beta)}$. More importantly, under heuristic assumptions, sieving is much faster. The most recent work on lattice sieving (see [BDGL16, Laa15]) has pushed the heuristic complexity down to $2^{0.292\ldots\beta+o(\beta)}$.

Simply comparing 0.292β to enumeration exponents suggests that sieving could be faster than enumeration for sizes of β of relevance to cryptography. However, this comparison ignores two critical caveats regarding the performance of sieving. First, a closer look at polynomial factors indicates that the $o(\beta)$ here is positive. Consider, e.g., [BDGL16, Figure 3], which reports a best fit of $2^{0.387\beta-15}$ for its fastest sieving experiments. The comparison in [MW15] takes this caveat into account and concludes that the sieving cutoff is “far out of reach”.

Second, sieving needs much more storage as β grows: at least $2^{0.208\ldots\beta+o(\beta)}$ bits of storage, again with positive $o(\beta)$. Furthermore, sieving is bottlenecked by random access to storage, and this random access also becomes slower as the amount of storage increases. The slowdown is approximately the square root of the storage in realistic cost models; see, e.g., [BK81].

Enumeration fits into very little memory even for large β. Kuo, Schneider, Dagdelen, Reichelt, Buchmann, Cheng, and Yang [KSD+11] showed that enumeration parallelizes effectively within and across GPUs. An attacker who can afford enough hardware for sieving for large β can instead use the same amount of hardware for enumeration, obtaining an almost linear parallelization speedup.

We do not mean to suggest that the operation-count ratio should be multiplied by the sieving storage (accounting for this enumeration speedup) *and* further by the square root of the storage (accounting for the cost of random access inside sieving): this would ignore the possibility of a speedup from parallelizing sieving. “Mesh” sorting algorithms such as the Schnorr–Shamir algorithm [SS86] sort n small items in time just $O(\sqrt{n})$, which is optimal in realistic models of parallel computation; these algorithms can be used as subroutines inside sieving, reducing the asymptotic cost penalty to just $2^{0.104\ldots\beta+o(\beta)}$. However, this is still much less effective parallelization than [KSD+11].

This cost penalty for sieving is ignored in measurements such as [MBL15] and [BDGL16, Figure 3], and in the resulting comparisons such as [MW15]. These measurements are limited to sieving sizes that fit into DRAM on a single computer, and do not account for the important increase in memory cost as β increases. Another way to see the same issue would be to scale sieving *down* to a small enough size to fit into GPU multiprocessors; this would demonstrate a sieving speedup for smaller β, for fundamentally the same reason that there will be a sieving slowdown for larger β.

In the absence of any realistic analyses of sieving cost for large β, we have decided to omit sieving from our security estimates. There is very little reason to believe that sieving can beat enumeration inside any attack that fits within our 2^{128} security target.

Bibliography

[AB10] Michel Abdalla and Paulo S. L. M. Barreto, editors. *Progress in Cryptology – LATINCRYPT 2010, First International Conference on Cryptology and Information Security in Latin America, Puebla, Mexico, August 8–11, 2010, Proceedings*, volume 6212 of *Lecture Notes in Computer Science*. Springer, 2010.

[ABD16] Martin R. Albrecht, Shi Bai, and Léo Ducas. A Subfield Lattice Attack on Overstretched NTRU Assumptions - Cryptanalysis of Some FHE and Graded Encoding Schemes. In *CRYPTO 2016 [RK16]*, pages 153–178, 2016. `https://eprint.iacr.org/2016/127`.

[Abe10] Masayuki Abe, editor. *Advances in Cryptology – ASIACRYPT 2010, 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5–9, 2010, Proceedings*, volume 6477 of *Lecture Notes in Computer Science*. Springer, 2010.

[ABG10] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES 2010 [MS10]*, pages 110–124, 2010. `http://www.iacr.org/archive/ches2010/62250105/62250105.pdf`.

[ACD$^+$05] Roberto M. Avanzi, Henri Cohen, Christophe Doche, Gerhard Frey, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2005.

[ACF$^+$14] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. Algebraic Algorithms for LWE. *IACR Cryptology ePrint Archive*, 2014. `http://eprint.iacr.org/2014/1018`.

[AD11] Vijay Atluri and Claudia Díaz, editors. *Computer Security – ESORICS 2011, 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12–14, 2011, Proceedings*, volume 6879 of *Lecture Notes in Computer Science*. Springer, 2011.

[ADI11] Jithra Adikari, Vassil S. Dimitrov, and Laurent Imbert. Hybrid binary-ternary number system for elliptic curve cryptosystems. *IEEE Transactions on Computers*, 60:254–265, 2011.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *USENIX 2016 [HS16]*, pages 327–343, 2016. https://eprint.iacr.org/2015/1092.

[ADRS14] Divesh Aggarwal, Daniel Dadush, Oded Regev, and Noah Stephens-Davidowitz. Solving the Shortest Vector Problem in 2^n Time via Discrete Gaussian Sampling. *CoRR*, abs/1412.7994, 2014. http://arxiv.org/abs/1412.7994.

[AFI68] *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April – 2 May 1968*, volume 32 of *AFIPS Conference Proceedings*. Thomson Book Company, Washington D.C., 1968.

[AG] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. http://code.google.com/p/relic-toolkit/.

[AG11] Sanjeev Arora and Rong Ge. New Algorithms for Learning in Presence of Errors. In *ICALP 2011 [AHS11]*, pages 403–415, 2011.

[AGM⁺13] Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. Charm: A Framework for Rapidly Prototyping Cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, 2013. http://eprint.iacr.org/2011/617/.

[AH03] Helmut Alt and Michel Habib, editors. *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science, Berlin, Germany, February 27 – March 1, 2003, Proceedings*, volume 2607 of *Lecture Notes in Computer Science*. Springer, 2003.

[AHS11] Luca Aceto, Monika Henzinger, and Jirí Sgall, editors. *Automata, Languages and Programming, 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4–8, 2011, Proceedings, Part I*, volume 6755 of *Lecture Notes in Computer Science*. Springer, 2011.

[AKL⁺11] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio López. Faster Explicit Formulas for Computing Pairings over Ordinary Curves. In *EUROCRYPT 2011 [Pat11]*, pages 48–68, 2011. http://eprint.iacr.org/2010/526/.

[AKPW13] Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with Rounding, Revisited – New Reduction, Properties and Applications. In *CRYPTO 2013 [CG13]*, pages 57–74, 2013.

[ALNR11] Christophe Arène, Tanja Lange, Michael Naehrig, and Christophe Ritzenthaler. Faster Computation of the Tate Pairing. *Journal of Number Theory*, 131:842–857, 2011. http://eprint.iacr.org/2009/155.

[AMORH13] Gora Adj, Alfred Menezes, Thomaz Oliveira, and Francisco Rodríguez-Henríquez. Weakness of $\mathbb{F}_{3^{6\cdot 509}}$ for Discrete Logarithm Cryptography. *IACR Cryptology ePrint Archive*, 2013. `http://eprint.iacr.org/2013/446/`.

[ANS11] Agence nationale de la sécurité des systèmes d'information. Publication d'un paramétrage de courbe elliptique visant des applications de passeport électronique et de l'administration électronique française, 2011. `https://tinyurl.com/nhog26h`.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: breaking the TLS and DTLS record protocols. In *S&P 2013 [IEE13]*, pages 526–540, 2013. `http://www.isg.rhul.ac.uk/tls/TLStiming.pdf`.

[APFV09] Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors. *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2–5, 2009, Proceedings*, volume 5536 of *Lecture Notes in Computer Science*. Springer, 2009.

[AR14] Gora Adj and Francisco Rodríguez-Henríquez. Square Root Computation over Even Extension Fields. *IEEE Transactions on Computers*, 63(11):2829–2841, 2014. `http://eprint.iacr.org/2012/685`.

[ARM10] ARM Limited. Cortex-A8 Technical Reference Manual, Revision r3p2, 2010. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf`.

[Aro50] Siegfried Heinrich Aronhold. Zur Theorie der homogenen Functionen dritten Grades von drei Variabeln. *Journal für die reine und angewandte Mathematik*, 1850 (39):140–159, 1850. `http://www.degruyter.com/view/j/crll.1850.issue-39/crll.1850.39.140/crll.1850.39.140.xml`.

[Bar86] Paul Barrett. Implementing the Rivest, Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *CRYPTO'86 [Odl86]*, pages 311–323, 1986.

[Bat68] Kenneth E. Batcher. Sorting Networks and Their Applications. In *AFIPS 1968 [AFI68]*, pages 307–314, 1968.

[BBC+14] Benjamin Black, Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Elliptic Curve Cryptography (ECC) Nothing Up My Sleeve (NUMS) Curves and Curve Generation, 2014. `https://tools.ietf.org/html/draft-black-numscurves-00`.

[BBC+15] Benjamin Black, Joppe W. Bos, Craig Costello, Adam Langley, Patrick Longa, and Michael Naehrig. Rigid Parameter Generation for Elliptic Curve Cryptography, 2015. `https://tools.ietf.org/html/draft-black-rpgecc-01`.

[BBJ+08] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards Curves. In *AFRICACRYPT 2008 [Vau08]*, pages 389–405, 2008. `http://cr.yp.to/papers.html#twisted`.

[BBLP07] Daniel J. Bernstein, Peter Birkner, Tanja Lange, and Christiane Peters. Optimizing double-base elliptic-curve single-scalar multiplication. In *INDOCRYPT 2007 [SRY07]*, pages 167–182, 2007. `https://eprint.iacr.org/2007/414`.

[BC13] Guido Bertoni and Jean-Sébastien Coron, editors. *Cryptographic Hardware and Embedded Systems – CHES 2013, 15th International Workshop, Santa Barbara, CA, USA, August 20–23, 2013, Proceedings*, volume 8086 of *Lecture Notes in Computer Science*. Springer, 2013.

[BCC+05] Steve Babbage, Dario Catalano, Carlos Cid, Christian Gehrmann, Louis Granboulan, Tanja Lange, Arjen Lenstra, Mats Näslund, Phong Nguyen, Christof Paar, Jan Pelzl, Thomas Pornin, Bart Preneel, Matt Robshaw, Andy Rupp, Nigel Smart, and Michael Ward. ECRYPT Yearly Report on Algorithms and Keysizes, 2005. `https://www.cosic.esat.kuleuven.be/ecrypt/ecrypt1/documents/D.SPA.16-1.0.pdf`.

[BCC+09] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. Randomizable Proofs and Delegatable Anonymous Credentials. In *CRYPTO 2009 [Hal09]*, pages 108–125, 2009. `http://research.microsoft.com/pubs/122759/anoncred.pdf`.

[BCC+15] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. How to manipulate curve standards: a white paper for the black hat. In *SSR 2015 [CM15]*, pages 109–139, 2015. `http://bada55.cr.yp.to/bada55-20150927.pdf`.

[BCHL13a] Joppe W. Bos, Craig Costello, Hüseyin Hışıl, and Kristin Lauter. Fast cryptography in genus 2. In *EUROCRYPT 2013 [JN13]*, pages 194–210, 2013. `http://eprint.iacr.org/2012/670`.

[BCHL13b] Joppe W. Bos, Craig Costello, Hüseyin Hışıl, and Kristin Lauter. High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition. In *CHES 2013 [BC13]*, pages 331–348, 2013. `http://eprint.iacr.org/2013/146`.

[BCKL15] Daniel J. Bernstein, Chitchanok Chuengsatiansup, David Kohel, and Tanja Lange. Twisted Hessian curves. In *LATINCRYPT 2015 [LR15]*, pages 269–294, 2015. `http://eprint.iacr.org/2015/781`.

[BCL14] Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. In *CHES 2014 [BR14]*, pages 316–334, 2014. `http://cr.yp.to/ecdh/curve41417-20140706.pdf`.

[BCL17] Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Double-base scalar multiplication revisited. *IACR Cryptology ePrint Archive*, 2017. `http://eprint.iacr.org/2017/037.pdf`.

[BCLN15] Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Selecting Elliptic Curves for Cryptography: An Efficiency and Security Analysis. *Journal of Cryptographic Engineering*, pages 1–28, 2015. `https://eprint.iacr.org/2014/130/`.

[BCLS14] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Peter Schwabe. Kummer Strikes Back: New DH Speed Records. In *ASIACRYPT 2014 [SI14]*, pages 317–337, 2014. `http://cr.yp.to/hecdh/kummer-20141028.pdf`.

[BCLvV16] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime. *IACR Cryptology ePrint Archive*, 2016. `http://eprint.iacr.org/2016/461.pdf`.

[BCN13] Joppe W. Bos, Craig Costello, and Michael Naehrig. Exponentiating in Pairing Groups. In *SAC 2013 [LLL14]*, 2013. `http://cryptosith.org/papers/#exppair`.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast Constant-Time Code-Based Cryptography. In *CHES 2013 [BC13]*, pages 250–272, 2013.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA 2016 [Kra16]*, pages 10–24, 2016. `http://dx.doi.org/10.1137/1.9781611974331.ch2`.

[BDL+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *CHES 2011 [PT11]*, pages 124–142, 2011. `http://eprint.iacr.org/2011/368`.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. `https://eprint.iacr.org/2011/368`.

[Ben14] Josh Benaloh, editor. *Topics in Cryptology – CT-RSA 2014, The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25–28, 2014, Proceedings*, volume 8366 of *Lecture Notes in Computer Science*. Springer, 2014.

[Ber97] Thomas A. Berson. Failure of the McEliece Public-Key Cryptosystem Under Message-Resend and Related-Message Attack. In *CRYPTO'97 [Jr.97]*, pages 213–220, 1997.

[Ber04] Daniel J. Bernstein. Floating-point arithmetic and message authentication, 2004. `http://cr.yp.to/antiforgery/hash127-20040918.pdf`.

[Ber06a] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *PKC 2006 [YDKM06]*, pages 207–228, 2006. `http://cr.yp.to/ecdh/curve25519-20060209.pdf`.

[Ber06b] Daniel J. Bernstein. Elliptic vs. hyperelliptic, part 1, 2006. `http://cr.yp.to/talks/2006.09.20/slides-djb-20060920-a4.pdf`.

[Ber09a] Daniel J. Bernstein. Batch binary Edwards. In *CRYPTO 2009 [Hal09]*, pages 317–336, 2009. `http://binary.cr.yp.to/bbe-20090604.pdf`.

[Ber09b] Daniel J. Bernstein. Complete addition laws for all elliptic curves over finite fields (talk slides), 2009. `http://cr.yp.to/talks/2009.07.17/slides.pdf`.

[Ber13] Daniel J. Bernstein. Complexity news: discrete logarithms in multiplicative groups of small-characteristic finite fields—the algorithm of Barbulescu, Gaudry, Joux, Thomé, 2013. `https://cr.yp.to/talks/2013.07.18/slides-djb-20130718-a4.pdf`.

[Ber14] Daniel J. Bernstein. A subfield-logarithm attack against ideal lattices, 2014. `https://blog.cr.yp.to/20140213-ideal.html`.

[BF01] Dan Boneh and Matthew K. Franklin. Identity-Based Encryption from the Weil Pairing. In *CRYPTO 2001 [Kil01]*, pages 213–229, 2001. `http://www.iacr.org/archive/crypto2001/21390212.pdf`.

[BGJT13] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. *CoRR*, abs/1306.4244, 2013. `http://eprint.iacr.org/2013/400/`.

[BGM$^+$10] Jean-Luc Beuchat, Jorge E. González Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco, Rodríguez-Henríquez, and Tadanori Teruya. High-Speed Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves. In *Pairing 2010 [JMO10]*, pages 21–39, 2010. `http://eprint.iacr.org/2010/354/`.

[BGM$^+$16] Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the Hardness of Learning with Rounding over Small Modulus. In *TCC 2016 [KM16]*, pages 209–224, 2016.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *ACM CCS'13 [SGY13]*, pages 967–980, 2013. `http://elligator.cr.yp.to/`.

[BHWL95] Wieb Bosma and Jr. Hendrik W. Lenstra. Complete Systems of Two Addition Laws for Elliptic Curves. *Journal of Number Theory*, 53:229–240, 1995.

[BI04] Valerie Berthe and Laurent Imbert. On Converting Numbers to the Double-Base Number System. In *SPIE'04 [Luk04]*, pages 70–78, 2004.

[BJ02] Eric Brier and Marc Joye. Weierstraß Elliptic Curves and Side-Channel Attacks. In *PKC 2002 [NP02]*, pages 335–345, 2002. `http://joye.site88.net/papers/BJ02espa.pdf`.

[BJ03] Olivier Billet and Marc Joye. The Jacobi Model of an Elliptic Curve and Side-Channel Analysis. In *AAECC 2003 [FHP03]*, pages 34–42, 2003. `eprint.iacr.org/2002/125`.

[BK81] Richard P. Brent and H. T. Kung. The Area-Time Complexity of Binary Multiplication. *J. ACM*, 28(3):521–534, 1981.

[BK12] Joppe W. Bos and Thorsten Kleinjung. ECM at Work. In *ASIACRYPT 2012 [WS12]*, pages 467–484, 2012. `http://eprint.iacr.org/2012/089`.

[BKLS02] Paulo S.L.M. Barreto, H. Y. Kim, B. Lynn, and M. Scott. Efficient algorithms for pairing-based cryptosystems. In *CRYPTO 2002 [Yun02]*, pages 354–368, 2002. `http://eprint.iacr.org/2002/008`.

[BLa] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`.

[BLb] Daniel J. Bernstein and Tanja Lange. Explicit-Formulas Database. `http://hyperelliptic.org/EFD`, accessed 13 June 2014.

[BL06] Rana Barua and Tanja Lange, editors. *Progress in Cryptology – INDOCRYPT 2006, 7th International Conference on Cryptology in India, Kolkata, India, December 11–13, 2006, Proceedings*, volume 4329 of *Lecture Notes in Computer Science*. Springer, 2006.

[BL07] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In *ASIACRYPT 2007 [Kur07]*, pages 29–50, 2007. `http://cr.yp.to/newelliptic/newelliptic-20070906.pdf`.

[BL08] Daniel J. Bernstein and Tanja Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. *Contemporary Mathematics*, 461:1–19, 2008. https://eprint.iacr.org/2007/455.

[BL11] Daniel J. Bernstein and Tanja Lange. A complete set of addition laws for incomplete Edwards curves. *Journal of Number Theory*, 131:858–872, 2011. http://cr.yp.to/newelliptic/completed-20101006.pdf.

[BL13a] Daniel J. Bernstein and Tanja Lange. Non-uniform Cracks in the Concrete: The Power of Free Precomputation. In *ASIACRYPT 2013 [SS13]*, pages 321–340, 2013.

[BL13b] Daniel J. Bernstein and Tanja Lange. Security dangers of the NIST curves, 2013. http://cr.yp.to/talks/2013.09.16/slides-djb-20130916-a4.pdf.

[BL14a] Daniel J. Bernstein and Tanja Lange. Hyper-and-elliptic-curve cryptography. *LMS Journal of Computation and Mathematics*, 17:181–202, 2014. Special Issue A (Algorithmic Number Theory Symposium XI).

[BL14b] Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography, 2014. http://safecurves.cr.yp.to, accessed 13 June 2014.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004. http://crypto.stanford.edu/~dabo/pubs/papers/weilsigs.ps.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT 2012 [HN12]*, pages 159–176, 2012. https://cryptojedi.org/papers/coolnacl-20120725.pdf.

[BMSZ13] Joppe W. Bos, Peter L. Montgomery, Daniel Shumow, and Gregory M. Zaverucha. Montgomery multiplication using vector instructions. In *SAC 2013 [LLL14]*, pages 471–489, 2013. http://eprint.iacr.org/2013/519.

[BN06] Paulo S.L.M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *SAC 2005 [PT06]*, pages 319–331, 2006. http://cryptosith.org/papers/pfcpo.pdf.

[Bon03] Dan Boneh, editor. *Advances in Cryptology – CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17–21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*. Springer, 2003.

[Bos00] Wieb Bosma, editor. *Algorithmic Number Theory, 4th International Symposium, ANTS-IV, Leiden, The Netherlands, July 2–7, 2000, Proceedings*, volume 1838 of *Lecture Notes in Computer Science*. Springer, 2000.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom Functions and Lattices. In *EUROCRYPT 2012 [PJ12]*, pages 719–737, 2012.

[BR14] Lejla Batina and Matthew Robshaw, editors. *Cryptographic Hardware and Embedded Systems – CHES 2014, 16th International Workshop, Busan, South Korea, September 23–26, 2014, Proceedings*, volume 8731 of *Lecture Notes in Computer Science*. Springer, 2014.

[BR16] Billy Bob Brumley and Juha Röning, editors. *Secure IT Systems – 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2–4, 2016, Proceedings*, volume 10014 of *Lecture Notes in Computer Science*, 2016.

[Bra05] ECC Brainpool. ECC Brainpool standard curves and curve generation, 2005. `http://www.ecc-brainpool.org/download/Domain-parameters.pdf`.

[BS09] Ljiljana Brankovic and Willy Susilo, editors. *Seventh Australasian Information Security Conference, AISC 2009, Wellington, New Zealand, January 2009*, volume 98 of *CRPIT*. Australian Computer Society, 2009.

[BS12] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In *CHES 2012 [PS12]*, pages 320–339, 2012. `http://cr.yp.to/highspeed/neoncrypto-20120320.pdf`.

[BS15] Jean-François Biasse and Fang Song. On the quantum attacks against schemes relying on the hardness of finding a short generator of an ideal in $\mathbb{Q}(\zeta_{p^n})$, 2015. `http://cacr.uwaterloo.ca/techreports/2015/cacr2015-12.pdf`.

[BS16] Jean-François Biasse and Fang Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In *SODA 2016 [Kra16]*, pages 893–902, 2016.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *ESORICS 2011 [AD11]*, pages 355–371, 2011. `https://eprint.iacr.org/2011/232`.

[Buh98] Joe Buhler, editor. *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21–25, 1998, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*. Springer, 1998.

[Cay81] Arthur Cayley. On the 34 Concomitants of the Ternary Cubic. *American Journal of Mathematics*, 4:1–15, 1881.

[CC86] David V. Chudnovsky and Gregory V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, 7:385–434, 1986.

[CDPR16] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering Short Generators of Principal Ideals in Cyclotomic Rings. In *EUROCRYPT 2016 [FC16b]*, pages 559–585, 2016.

[Cer] Certivox. MIRACL Cryptographic SDK. http://www.certivox.com/miracl.

[Cer00a] Certicom Research. SEC 1: Elliptic Curve Cryptography, Version 1.0, 2000. http://www.secg.org/SEC1-Ver-1.0.pdf.

[Cer00b] Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0, 2000. http://www.secg.org/SEC2-Ver-1.0.pdf.

[Cer09] Certicom Research. SEC 1: Elliptic Curve Cryptography, Version 2.0, 2009. http://www.secg.org/sec1-v2.pdf.

[Cer10] Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0, 2010. http://www.secg.org/sec2-v2.pdf.

[CFN$^+$14] Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, and Hovav Shacham. On the practical exploitability of Dual EC in TLS implementations. In *USENIX 2014 [FJ14]*, pages 319–335, 2014. https://projectbullrun.org/dual-ec/index.html.

[CG13] Ran Canetti and Juan A. Garay, editors. *Advances in Cryptology – CRYPTO 2013, 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*. Springer, 2013.

[CGS14] Peter Campbell, Michael Groves, and Dan Shepherd. Soliloquy: a cautionary tale, 2014. http://docbox.etsi.org/Workshop/2014/201410_CRYPTO/S07_Systems_and_Attacks/S07_Groves_Annex.pdf.

[Che11] Liqun Chen, editor. *Cryptography and Coding, 13th IMA International Conference, IMACC 2011, Oxford, UK, December 2011, Proceedings*, volume 7089 of *Lecture Notes in Computer Science*. Springer, 2011.

[Cho15a] Tung Chou. Sandy2x: fastest Curve25519 implementation ever, 2015. http://csrc.nist.gov/groups/ST/ecc-workshop-2015/presentations/session6-chou-tung.pdf.

[Cho15b] Tung Chou. Sandy2x: New Curve25519 Speed Records. In *SAC 2015 [DK16]*, pages 145–160, 2015.

[CHS14] Craig Costello, Hüseyin Hışıl, and Benjamin Smith. Faster compact Diffie–Hellman: endomorphisms on the x-line. In *EUROCRYPT 2014 [NO14]*, pages 183–200, 2014. `https://eprint.iacr.org/2013/692`.

[CIV16a] Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. On Error Distributions in Ring-based LWE. *LMS Journal of Computation and Mathematics*, 19:130–145, 2016. Special Issue A (Algorithmic Number Theory Symposium XII).

[CIV16b] Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. Provably Weak Instances of Ring-LWE Revisited. In *EUROCRYPT 2016 [FC16a]*, pages 147–167, 2016.

[cKKNP01] Çetin Kaya Koç, David Naccache, and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems – CHES 2001, Third International Workshop, Paris, France, May 14–16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*. Springer, 2001.

[CKMS16] Sanjit Chatterjee, Neal Koblitz, Alfred Menezes, and Palash Sarkar. Another Look at Tightness II: Practical Issues in Cryptography. *IACR Cryptology ePrint Archive*, 2016. `https://eprint.iacr.org/2016/360`.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[CLS16] Hao Chen, Kristin Lauter, and Katherine E. Stange. Vulnerable Galois RLWE Families and Improved Attacks. *IACR Cryptology ePrint Archive*, 2016. `https://eprint.iacr.org/2016/193`.

[CM11] Sanjit Chatterjee and Alfred Menezes. On Cryptographic Protocols Employing Asymmetric Pairings – The Role of Ψ Revisited. *Discrete Applied Mathematics*, 159:1311–1322, 2011. `http://eprint.iacr.org/2009/480/`.

[CM15] Liqun Chen and Shin'ichiro Matsuo, editors. *Security Standardisation Research, Second International Conference, SSR 2015, Tokyo, Japan, December 15–16, 2015, Proceedings*, volume 9497 of *Lecture Notes in Computer Science*. Springer, 2015.

[CMO98] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *ASIACRYPT 1998 [OP98]*, pages 51–65, 1998. `http://www.math.u-bordeaux.fr/~cohen/asiacrypt98.dvi`.

[CN11a] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better Lattice Security Estimates. In *ASIACRYPT 2011 [LW11]*, pages 1–20, 2011. `http://dx.doi.org/10.1007/978-3-642-25385-0_1`.

[CN11b] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better Lattice Security Estimates (Full Version), 2011. `http://www.di.ens.fr/~ychen/research/Full_BKZ.pdf`.

[CNRS13] Chitchanok Chuengsatiansup, Michael Naehrig, Pance Ribarski, and Peter Schwabe. Panda: Pairings and arithmetic. In *Pairing 2013 [CZ14]*, pages 229–250, 2013. `https://cryptojedi.org/papers/panda-20131204.pdf`.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, 1993.

[Coh00] Henri Cohen. *Advanced Topics in Computational Number Theory*, volume 193 of *Graduate Texts in Mathematics*. Springer, 2000.

[Coo66] Stephen A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, 1966. `http://cr.yp.to/bib/1966/cook.html`.

[Cra05] Ronald Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22–26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*. Springer, 2005.

[CS09] Neil Costigan and Peter Schwabe. Fast elliptic-curve cryptography on the Cell Broadband Engine. In *AFRICACRYPT 2009 [Pre09]*, pages 368–385, 2009. `https://cryptojedi.org/papers/celldh-20090331.pdf`.

[CT15] Alex Capuñay and Nicolas Thériault. Computing Optimal 2-3 Chains for Pairings. In *LATINCRYPT 2015 [LR15]*, pages 225–244, 2015.

[CT16] Jung Hee Cheon and Tsuyoshi Takagi, editors. *Advances in Cryptology – ASIACRYPT 2016, 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4–8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*. Springer, 2016.

[CWB14] Daniel Cabarcas, Patrick Weiden, and Johannes A. Buchmann. On the Efficiency of Provably Secure NTRU. In *PQCrypto 2014 [Mos14]*, pages 22–39, 2014.

[CZ14] Zhenfu Cao and Fangguo Zhang, editors. *Pairing-Based Cryptography – Pairing 2013, 6th International Conference, Beijing, China, November 22–24, 2013, Revised Selected Papers*, volume 8365 of *Lecture Notes in Computer Science*. Springer, 2014.

[DBP96] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In *FSE 1996 [Gol96]*, pages 71–82, 1996.

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice Signatures and Bimodal Gaussians. In *CRYPTO 2013 [CG13]*, pages 40–56, 2013.

[Den03] Alexander W. Dent. A Designer's Guide to KEMs. In *IMA 2003 [Pat03]*, pages 133–151, 2003. `https://eprint.iacr.org/2002/174`.

[DH08] Christophe Doche and Laurent Habsieger. A tree-based approach for computing double-base chains. In *ACISP 2008 [MSS08]*, pages 433–446, 2008. `http://web.science.mq.edu.au/~doche/tree_DBNS.pdf`.

[DHH$^+$15] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 77:493–514, 2015.

[DI06] Christophe Doche and Laurent Imbert. Extended Double-Base Number System with Applications to Elliptic Curve Cryptography. In *INDOCRYPT 2006 [BL06]*, pages 335–348, 2006.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DIK06] Christophe Doche, Thomas Icart, and David R. Kohel. Efficient scalar multiplication by isogeny decompositions. In *PKC 2006 [YDKM06]*, pages 191–206, 2006. `http://eprint.iacr.org/2005/420`.

[DIM05] Vassil S. Dimitrov, Laurent Imbert, and Pradeep Kumar Mishra. Efficient and Secure Elliptic Curve Point Multiplication Using Double-Base Chains. In *ASIACRYPT 2005 [Roy05]*, pages 59–78, 2005. `https://www.iacr.org/archive/asiacrypt2005/059/059.pdf`.

[DIM08] Vassil S. Dimitrov, Laurent Imbert, and Pradeep Kumar Mishra. The Double-Base Number System and Its Application to Elliptic Curve Cryptography. *Mathematics of Computation*, 77(262):1075–1104, 2008. `http://www.ams.org/journals/mcom/2008-77-262/S0025-5718-07-02048-0/S0025-5718-07-02048-0.pdf`.

[Din10] Jintai Ding. Solving LWE problem with bounded errors in polynomial time. *IACR Cryptology ePrint Archive*, 2010. `http://eprint.iacr.org/2010/558`.

[DK16] Orr Dunkelman and Liam Keliher, editors. *Selected Areas in Cryptography – SAC 2015, 22nd International Conference, Sackville, NB, Canada, August 12–14, 2015, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*. Springer, 2016.

[DKS09] Christophe Doche, David R. Kohel, and Francesco Sica. Double-Base Number System for Multi-scalar Multiplications. In *EUROCRYPT 2009 [Jou09]*, 2009.

[DL05] Christophe Doche and Tanja Lange. *Arithmetic of Elliptic Curves*, pages 267–302. HEHCC [ACD$^+$05]. 2005.

[Doc14] Christophe Doche. On the Enumeration of Double-Base Chains with Applications to Elliptic Curve Cryptography. In *ASIACRYPT 2014 [SI14]*, pages 297–316, 2014. `http://eprint.iacr.org/2014/371`.

[dPLP16] Rafaël del Pino, Vadim Lyubashevsky, and David Pointcheval. The Whole is Less Than the Sum of Its Parts: Constructing More Efficient Lattice-Based AKEs. In *SCN 2016 [ZP16]*, pages 273–291, 2016.

[dR95] Peter de Rooij. Efficient exponentiation using precomputation and vector addition chains. In *EUROCRYPT'94 [San94]*, pages 389–399, 1995.

[Edw07] Harold M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44:393–422, 2007. `http://www.ams.org/bull/2007-44-03/S0273-0979-07-01153-6/home.html`.

[EHL14] Kirsten Eisenträger, Sean Hallgren, and Kristin E. Lauter. Weak Instances of PLWE. In *SAC 2014 [JY14]*, pages 183–194, 2014.

[ELOS15] Yara Elias, Kristin E. Lauter, Ekin Ozman, and Katherine E. Stange. Provably Weak Instances of Ring-LWE. In *CRYPTO 2015 [GR15]*, pages 63–92, 2015.

[FC16a] Marc Fischlin and Jean-Sébastien Coron, editors. *Advances in Cryptology – EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8–12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*. Springer, 2016.

[FC16b] Marc Fischlin and Jean-Sébastien Coron, editors. *Advances in Cryptology – EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8–12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*. Springer, 2016.

[FHLS14] Armando Faz-Hernández, Patrick Longa, and Ana H. Sánchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. In *CT-RSA 2014 [Ben14]*, pages 1–27, 2014. `http://eprint.iacr.org/2013/158`.

[FHP03] Marc P. C. Fossorier, Tom Høholdt, and Alain Poli, editors. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 15th International Symposium, AAECC-15, Toulouse, France, May 12–16, 2003, Proceedings*, volume 2643 of *Lecture Notes in Computer Science*. Springer, 2003.

[FJ10] Reza Rezaeian Farashahi and Marc Joye. Efficient arithmetic on Hessian curves. In *PKC 2010 [NP10]*, pages 243–260, 2010.

[FJ14] Kevin Fu and Jaeyeon Jung, editors. *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20–22, 2014*. USENIX Association, 2014.

[FL15] Armando Faz-Hernández and Julio López. Fast Implementation of Curve25519 Using AVX2. In *LATINCRYPT 2015 [LR15]*, pages 329–345, 2015.

[Fog14] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, 2014. `http://agner.org/optimize/`.

[FP85] Ulrich Fincke and Michael Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44(170):463–471, 1985. `http://www.jstor.org/stable/2007966`.

[FP16] Sara Foresti and Giuseppe Persiano, editors. *Cryptology and Network Security – 15th International Conference, CANS 2016, Milan, Italy, November 14–16, 2016, Proceedings*, volume 10052 of *Lecture Notes in Computer Science*, 2016.

[FPRE15] Jean-Pierre Flori, Jérôme Plût, Jean-René Reinhard, and Martin Ekerå. Diversity and transparency for ECC, 2015. `http://csrc.nist.gov/groups/ST/ecc-workshop-2015/papers/session4-flori-jean-pierre.pdf`.

[Fra04] Matthew K. Franklin, editor. *Advances in Cryptology – CRYPTO 2004, 24th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15–19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*. Springer, 2004.

[FST10] David Freeman, Michael Scott, and Edlyn Teske. A Taxonomy of Pairing-Friendly Elliptic Curves. *Journal of Cryptology*, 23(2):224–280, 2010. `http://eprint.iacr.org/2006/372/`.

[FT12] Pierre-Alain Fouque and Mehdi Tibouchi. Indifferentiable Hashing to Barreto-Naehrig Curves. In *LATINCRYPT 2012 [HN12]*, pages 1–17, 2012. `www.di.ens.fr/~fouque/pub/latincrypt12.pdf`.

[FWZ12] Reza Rezaeian Farashahi, Hongfeng Wu, and Chang-An Zhao. Efficient arithmetic on elliptic curves over fields of characteristic three. In *SAC 2012 [KW13]*, pages 135–148, 2012.

[Gab13] Philippe Gaborit, editor. *Post-Quantum Cryptography, 5th International Workshop, PQCrypto 2013, Limoges, France, June 4–7, 2013, Proceedings*, volume 7932 of *Lecture Notes in Computer Science*. Springer, 2013.

[Gau06] Pierrick Gaudry. Variants of the Montgomery form based on Theta functions, 2006. `http://www.loria.fr/~gaudry/publis/toronto.pdf`, [Gau07].

[Gau07] Pierrick Gaudry. Fast genus 2 arithmetic based on Theta functions. *Journal of Mathematical Cryptology*, 1:243–265, 2007. `http://webloria.loria.fr/~gaudry/publis/arithKsurf.pdf`, [Gau06].

[GF05] Harold N. Gabow and Ronald Fagin, editors. *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC'05. ACM, 2005.

[GG14] Juan A. Garay and Rosario Gennaro, editors. *Advances in Cryptology – CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17–21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*. Springer, 2014.

[GG16] Steven D. Galbraith and Pierrick Gaudry. Recent progress on the elliptic curve discrete logarithm problem. *Designs, Codes and Cryptography*, 78(1):51–72, 2016.

[GGMZ13] Faruk Göloğlu, Robert Granger, Gary McGuire, and Jens Zumbrägel. Solving a 6120-bit DLP on a Desktop Computer. In *SAC 2013 [LLL14]*, 2013. `http://eprint.iacr.org/2013/306`.

[GH15] Tim Güneysu and Helena Handschuh, editors. *Cryptographic Hardware and Embedded Systems – CHES 2015, 17th International Workshop, Saint-Malo, France, September 13–16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*. Springer, 2015.

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors. In *ASIACRYPT 2016 [CT16]*, pages 789–815, 2016.

[GKZ14] Robert Granger, Thorsten Kleinjung, and Jens Zumbrägel. Breaking “128-bit secure” supersingular binary curves (or how to solve discrete logarithms in $\mathbb{F}_{2^{4\cdot 1223}}$ and $\mathbb{F}_{2^{12\cdot 367}}$). In *CRYPTO 2014 [GG14]*, 2014. http://eprint.iacr.org/2014/119.

[GL09] Pierrick Gaudry and David Lubicz. The arithmetic of characteristic 2 Kummer surfaces and of elliptic Kummer lines. *Finite Fields and Their Applications*, 15:246–260, 2009. http://hal.inria.fr/inria-00266565/PDF/c2.pdf.

[GLS09] Steven Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In *EUROCRYPT 2009 [Jou09]*, pages 518–535, 2009. https://eprint.iacr.org/2008/194.

[GLV01] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *CRYPTO 2001 [Kil01]*, pages 190–200, 2001. http://www.iacr.org/archive/crypto2001/21390189.pdf.

[GLV06] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. U.S. patent 7110538: method for accelerating cryptographic operations on elliptic curves, 2006. http://www.freepatentsonline.com/7110538.html.

[GLV11] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. U.S. patent 7995752: method for accelerating cryptographic operations on elliptic curves, 2011. http://www.freepatentsonline.com/7995752.html.

[GM00] Steven D. Galbraith and James McKee. The probability that the number of points on an elliptic curve over a finite field is prime. *Journal of the London Mathematical Society*, 62:671–684, 2000. https://www.math.auckland.ac.nz/~sgal018/cm.pdf.

[Gol96] Dieter Gollmann, editor. *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21–23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*. Springer, 1996.

[GOPS13] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software Speed Records for Lattice-Based Signatures. In *PQCrypto 2013 [Gab13]*, pages 67–82, 2013.

[GPS04] Robert Granger, Dan Page, and Martijn Stam. On Small Characteristic Algebraic Tori in Pairing-Based Cryptography. *IACR Cryptology ePrint Archive*, 2004. http://eprint.iacr.org/2004/132.

[GPS08] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008. `http://www.sciencedirect.com/science/article/pii/S0166218X08000449`.

[GR15] Rosario Gennaro and Matthew Robshaw, editors. *Advances in Cryptology – CRYPTO 2015, 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16–20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*. Springer, 2015.

[Gra] Torbjörn Granlund. GMP 5.1.3: GNU multiple precision arithmetic library. `http://gmplib.org`.

[Gro10] Jens Groth. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *ASIACRYPT 2010 [Abe10]*, pages 321–340, 2010. `http://www.cs.ucl.ac.uk/staff/J.Groth/ShortNIZK.pdf`.

[GS02] Craig Gentry and Alice Silverberg. Hierarchical ID-Based Cryptography. In *ASIACRYPT 2002 [Zhe02]*, pages 548–566, 2002. `http://www.cs.ucdavis.edu/~franklin/ecs228/pubs/extra_pubs/hibe.pdf`.

[GS10] Robert Granger and Michael Scott. Faster Squaring in the Cyclotomic Subgroup of Sixth Degree Extensions. In *PKC 2010 [NP10]*, pages 209–223, 2010.

[GS12a] Pierrick Gaudry and Éric Schost. Genus 2 point counting over prime fields. *Journal of Symbolic Computation*, 47:368–400, 2012. `http://www.csd.uwo.ca/~eschost/publications/countg2.pdf`.

[GS12b] Jens Groth and Amit Sahai. Efficient Non-Interactive Proof Systems for Bilinear Groups. *SIAM Journal on Computing*, 41(5):1193–1232, 2012. `http://www0.cs.ucl.ac.uk/staff/J.Groth/WImoduleFull.pdf`.

[GS16] Shay Gueron and Fabian Schlieker. Speeding up R-LWE Post-quantum Key Exchange. In *NordSec 2016 [BR16]*, pages 187–198, 2016.

[GSCB14] Diana Goehringer, Marco Domenico Santambrogio, João M. P. Cardoso, and Koen Bertels, editors. *Reconfigurable Computing: Architectures, Tools, and Applications, 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14–16, 2014, Proceedings*, volume 8405 of *Lecture Notes in Computer Science*. Springer, 2014.

[GT07] Pierrick Gaudry and Emmanuel Thomé. The mpFq library and implementing curve-based key exchanges. In *SPEED: software performance enhancement for encryption and decryption*, pages 49–64, 2007. `http://www.loria.fr/~gaudry/papers.en.html`.

[Hal09] Shai Halevi, editor. *Advances in Cryptology – CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16–20, 2009, Proceedings*, volume 5677 of *Lecture Notes in Computer Science*. Springer, 2009.

[Ham12] Mike Hamburg. Fast and compact elliptic-curve cryptography. *IACR Cryptology ePrint Archive*, 2012. `http://eprint.iacr.org/2012/309`.

[Ham14] Mike Hamburg. New Ed448-Goldilocks release, 2014. `https://moderncrypto.org/mail-archive/curves/2014/000101.html`.

[Har86] Juris Hartmanis, editor. *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC'86. ACM, 1986.

[HC14] Hüseyin Hışıl and Craig Costello. Jacobian Coordinates on Genus 2 Curves. In *ASIACRYPT 2014 [SI14]*, pages 338–357, 2014. `https://eprint.iacr.org/2014/385`.

[HCD07] Hüseyin Hışıl, Gary Carter, and Ed Dawson. New formulae for efficient elliptic curve arithmetic. In *INDOCRYPT 2007 [SRY07]*, pages 138–151, 2007.

[Hes44] Otto Hesse. Über die Elimination der Variabeln aus drei algebraischen Gleichungen vom zweiten Grade mit zwei Variabeln. *Journal für die reine und angewandte Mathematik*, 28:68–96, 1844.

[HGSW03] Nick Howgrave-Graham, Joseph H Silverman, and William Whyte. A Meet-in-the-Middle Attack on an NTRU Private key. Technical report, Technical report, NTRU Cryptosystems, June 2003. Report, 2003. `https://www.securityinnovation.com/uploads/Crypto/NTRUTech004v2.pdf`.

[HHHW09] Philip S. Hirschhorn, Jeffrey Hoffstein, Nick Howgrave-Graham, and William Whyte. Choosing NTRUEncrypt Parameters in Light of Combined Lattice Reduction and MITM Approaches. In *ACNS 2009 [APFV09]*, pages 437–455, 2009.

[Hış10] Hüseyin Hışıl. *Elliptic curves, group law, and efficient computation*. PhD thesis, Queensland University of Technology, 2010.

[HL02] Jeremy Horwitz and Ben Lynn. Toward Hierarchical Identity-Based Encryption. In *EUROCRYPT 2002 [Knu02]*, pages 466–481, 2002. `http://theory.stanford.edu/~horwitz/pubs/hibe.pdf`.

[HN12] Alejandro Hevia and Gregory Neven, editors. *Progress in Cryptology – LATINCRYPT 2012, 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7–10,* *2012, Proceedings*, volume 7533 of *Lecture Notes in Computer Science*. Springer, 2012.

[HNP$^+$03] Nick Howgrave-Graham, Phong Q. Nguyen, David Pointcheval, John Proos, Joseph H. Silverman, Ari Singer, and William Whyte. The Impact of Decryption Failures on the Security of NTRU Encryption. In *CRYPTO 2003 [Bon03]*, pages 226–246, 2003.

[Hor19] W. G. Horner. A New Method of Solving Numerical Equations of All Orders, by Continuous Approximation. *Philosophical Transactions of the Royal society of London*, 109:308–335, 1819.

[How07] Nick Howgrave-Graham. A Hybrid Lattice-Reduction and Meet-in-the-Middle Attack Against NTRU. In *CRYPTO 2007 [Men07]*, pages 150–169, 2007.

[HPP06] Florian Hess, Sebastian Pauli, and Michael E. Pohst, editors. *Algorithmic Number Theory, 7th International Symposium, ANTS-VII, Berlin, Germany, July 23–28, 2006, Proceedings*, volume 4076 of *Lecture Notes in Computer Science*. Springer, 2006.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A Ring-Based Public Key Cryptosystem. In *ANTS-III [Buh98]*, pages 267–288, 1998.

[HPS$^+$15] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, William Whyte, and Zhenfei Zhang. Choosing Parameters for NTRUEncrypt. *IACR Cryptology ePrint Archive*, 2015. `https://eprint.iacr.org/2015/708`.

[HS03] Jeffrey Hoffstein and Joseph H. Silverman. Random small Hamming weight products with applications to cryptography. *Discrete Applied Mathematics*, 130(1):37–49, 2003.

[HS16] Thorsten Holz and Stefan Savage, editors. *Proceedings of the25th USENIX Security Symposium, Austin, TX, USA, August 10–12, 2016*. USENIX Association, 2016.

[HSSW15] Michael Hutter, Jürgen Schilling, Peter Schwabe, and Wolfgang Wieser. NaCl's `crypto_box` in Hardware. In *CHES 2015 [GH15]*, pages 81–101, 2015. `https://cryptojedi.org/papers/naclhw-20150616.pdf`.

[HSV06] Florian Hess, Nigel P. Smart, and Frederik Vercauteren. The Eta Pairing Revisited. *IEEE Transactions on Information Theory*, 52(10):4595–4602, 2006. `http://eprint.iacr.org/2006/110`.

[HSW05] Nick Howgrave-Graham, Joseph H. Silverman, and William Whyte. Choosing Parameter Sets for NTRUEncrypt with NAEP and SVES-3. *IACR Cryptology ePrint Archive*, 2005. `https://eprint.iacr.org/2005/045`.

[Hus03] Dale Husemöller. *Elliptic curves*, volume 111 of *Graduate Texts in Mathematics*. Springer, 2nd edition, 2003.

[HWCD08] Hüseyin Hışıl, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In *ASIACRYPT 2008 [Pie08]*, 2008.

[HWCD09] Hüseyin Hışıl, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Faster group operations on elliptic curves. In *AISC 2009 [BS09]*, pages 7–19, 2009. `https://eprint.iacr.org/2007/441`.

[ICP15] *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1–4, 2015*. IEEE Computer Society, 2015. `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7349544`.

[IEE00] Institute of Electrical and Electronics Engineers. IEEE 1363-2000: Standard specifications for public key cryptography, 2000. Preliminary draft at `http://grouper.ieee.org/groups/1363/P1363/draft.html`.

[IEE13] *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, California, USA, May 19–22, 2013*. IEEE Computer Society, 2013.

[Ind15] Piotr Indyk, editor. *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4–6, 2015*. SIAM, 2015.

[JcKKP02] Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems – CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13–15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*. Springer, 2002.

[JFF$^+$83] David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors. *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC'83. ACM, 1983.

[JLMS13] Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors. *Applied Cryptography and Network Security, 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25–28, 2013, Proceedings*, volume 7954 of *Lecture Notes in Computer Science*. Springer, 2013.

[JMO10] Marc Joye, Atsuko Miyaji, and Akira Otsuka, editors. *Pairing-Based Cryptography – Pairing 2010 , 4th International Conference, Yamanaka Hot Spring, Japan, December 13–15, 2010, Proceedings*, volume 6487 of *Lecture Notes in Computer Science*. Springer, 2010.

[JN13] Thomas Johansson and Phong Q. Nguyen, editors. *Advances in Cryptology – EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26–30, 2013, Proceedings*, volume 7881 of *Lecture Notes in Computer Science*. Springer, 2013.

[Jou00] Antoine Joux. A One Round Protocol for Tripartite Diffie-Hellman. In *ANTS-IV [Bos00]*, pages 385–393, 2000. `http://cgi.di.uoa.gr/~aggelos/crypto/page4/assets/joux-tripartite.pdf`.

[Jou04] Antoine Joux. A One Round Protocol for Tripartite Diffie–Hellman. *Journal of Cryptology*, 17(4):263–276, 2004.

[Jou09] Antoine Joux, editor. *Advances in Cryptology – EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26–30, 2009, Proceedings*, volume 5479 of *Lecture Notes in Computer Science*. Springer, 2009.

[Jou13] Antoine Joux. A New Index Calculus Algorithm with Complexity $L(1/4 + o(1))$ in Small Characteristic. In *SAC 2013 [LLL14]*, pages 355–379, 2013. `http://eprint.iacr.org/2013/095/`.

[JQ01] Marc Joye and Jean-Jacques Quisquater. Hessian elliptic curves and side-channel attacks. In *CHES 2001 [cKKNP01]*, pages 402–410, 2001. `http://joye.site88.net/`.

[Jr.97] Burton S. Kaliski Jr., editor. *Advances in Cryptology – CRYPTO'97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17–21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*. Springer, 1997.

[JSS15] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. Practical Invalid Curve Attacks on TLS-ECDH. In *ESORICS 2015 [PRW15]*, 2015. `https://www.nds.rub.de/research/publications/ESORICS15/`.

[JY14] Antoine Joux and Amr M. Youssef, editors. *Selected Areas in Cryptography – SAC 2014, 21st International Conference, Montreal, QC, Canada, August 14–15, 2014, Revised Selected Papers*, volume 8781 of *Lecture Notes in Computer Science*. Springer, 2014.

[Kan83] Ravi Kannan. Improved Algorithms for Integer Programming and Related Lattice Problems. In *STOC'83 [JFF$^+$83]*, pages 193–206, 1983. `http://doi.acm.org/10.1145/800061.808749`.

[Kel03] John Kelsey. Choosing a DRBG Algorithm, 2003. `https://github.com/matthewdgreen/nistfoia/blob/master/6.4.2014%20production/011%20-%209.12%20Choosing%20a%20DRBG%20Algorithm.pdf`.

[KF16] Paul Kirchner and Pierre-Alain Fouque. Comparison between Subfield and Straightforward Attacks on NTRU. *IACR Cryptology ePrint Archive*, 2016. `https://eprint.iacr.org/2016/717`.

[Kil01] Joe Kilian, editor. *Advances in Cryptology – CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*. Springer, 2001.

[KM16] Eyal Kushilevitz and Tal Malkin, editors. *Theory of Cryptography, 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10–13, 2016, Proceedings, Part I*, volume 9562 of *Lecture Notes in Computer Science*. Springer, 2016.

[Knu74] Donald Knuth. Structured Programming with go to Statements. *Computing Surveys*, 6:261–301, 1974.

[Knu02] Lars R. Knudsen, editor. *Advances in Cryptology – EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 – May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*. Springer, 2002.

[KO63] Anatoly A. Karatsuba and Yuri Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[Kob96] Neal Koblitz, editor. *Advances in Cryptology – CRYPTO'96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18–22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*. Springer, 1996.

[Kob98] Neal Koblitz. *Algebraic Aspects of Cryptography*, volume 3 of *Algorithms and Computation in Mathematics*. Springer, 1998.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO'96 [Kob96]*, pages 104–113, 1996. `http://www.cryptography.com/public/pdf/TimingAttacks.pdf`.

[Koh11] David Kohel. Addition law structure of elliptic curves. *Journal of Number Theory*, 131:894–919, 2011.

[Koh15] David Kohel. The geometry of efficient arithmetic on elliptic curves. *Contemporary Mathematics*, 637, 2015.

[Kra16] Robert Krauthgamer, editor. *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10–12, 2016*. SIAM, 2016.

[KSD+11] Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes A. Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. Extreme Enumeration on GPU and in Clouds: How Many Dollars You Need to Break SVP Challenges. In *CHES 2011 [PT11]*, pages 176–191, 2011.

[Kum14] Virendra Kumar. `ntruees743ep1` software, 2014. Included in [BLa].

[Kur07] Kaoru Kurosawa, editor. *Advances in Cryptology – ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2–6, 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*. Springer, 2007.

[KW13] Lars R. Knudsen and Huapeng Wu, editors. *Selected Areas in Cryptography, 19th International Conference, SAC 2012, Windsor, Canada, August 15–16, 2012, Revised Selected Papers*, volume 7707 of *Lecture Notes in Computer Science*. Springer, 2013.

[Laa15] Thijs Laarhoven. Sieving for Shortest Vectors in Lattices Using Angular Locality-Sensitive Hashing. In *CRYPTO 2015 [GR15]*, pages 3–22, 2015. `http://dx.doi.org/10.1007/978-3-662-47989-6_1`.

[Lan13] Adam Langley. How to botch TLS forward secrecy, 2013. `https://www.imperialviolet.org/2013/06/27/botchingpfs.html`.

[LC14] Brian LaMacchia and Craig Costello. Deterministic generation of elliptic curves (a.k.a. “NUMS” Curves), 2014. `https://www.ietf.org/proceedings/90/slides/slides-90-cfrg-5.pdf`.

[LLL14] Tanja Lange, Kristin Lauter, and Petr Lisonek, editors. *Selected Areas in Cryptography – SAC 2013, 20th International Conference, Burnaby, BC, Canada, August 14–16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*. Springer, 2014.

[LM10] Manfred Lochter and Johannes Merkle. RFC 5639: Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation, 2010. `https://tools.ietf.org/html/rfc5639`.

[LM13] Adam Langley and Andrew Moon. Implementations of a fast elliptic-curve Digital Signature Algorithm, 2013. `https://github.com/floodyberry/ed25519-donna`.

[LMS04] Florian Luca, David Jose Mireles, and Igor E. Shparlinski. MOV attack in various subgroups on elliptic curves. *Illinois Journal of Mathematics*, 48(3):1041–1052, July 2004. `https://projecteuclid.org/euclid.ijm/1258131069`.

[LMSS14] Manfred Lochter, Johannes Merkle, Jörn-Marc Schmidt, and Torsten Schütze. Requirements for Standard Elliptic Curves, 2014. Position Paper of the ECC Brainpool, `http://www.ecc-brainpool.org/20141001_ECCBrainpool_PositionPaper.pdf`.

[LMvdP15] Thijs Laarhoven, Michele Mosca, and Joop van de Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77:375–400, 2015.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *CANS 2016 [FP16]*, pages 124–139, 2016.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors over Rings. *J. ACM*, 60(6):43, 2013.

[LR15] Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors. *Progress in Cryptology – LATINCRYPT 2015, 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23–26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*. Springer, 2015.

[LS01] Pierre-Yvan Liardet and Nigel P. Smart. Preventing SPA/DPA in ECC systems using the Jacobi form. In *CHES 2001 [cKKNP01]*, pages 391–401, 2001.

[LS12a] Adeline Langlois and Damien Stehlé. Hardness of decision (R)LWE for any modulus. *IACR Cryptology ePrint Archive*, 2012. `https://eprint.iacr.org/2012/091`.

[LS12b] Patrick Longa and Francesco Sica. Four-dimensional Gallant–Lambert–Vanstone scalar multiplication. In *ASIACRYPT 2012 [WS12]*, pages 718–739, 2012. `http://eprint.iacr.org/2011/608`.

[Luk04] Franklin T. Luk, editor. *SPIE'04: Advanced Signal Processing Algorithms, Architectures, and Implementations XIV, Aug 2, 2004, Denver, Colorado (USA), Proceedings*, volume 5559, 2004.

[LW11] Dong Hoon Lee and Xiaoyun Wang, editors. *Advances in Cryptology – ASIACRYPT 2011, 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4–8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*. Springer, 2011.

[Lyn] Ben Lynn. PBC Library – The Pairing-Based Cryptography Library. `http://crypto.stanford.edu/pbc/`.

[Lyu16] Vadim Lyubashevsky. Future Directions in Lattice Cryptography (talk slides), 2016. `http://troll.iis.sinica.edu.tw/pkc16/slides/Invited_Talk_II--Directions_in_Practical_Lattice_Cryptography.pptx`.

[MBL15] Artur Mariano, Christian H. Bischof, and Thijs Laarhoven. Parallel (Probable) Lock-Free Hash Sieve: A Practical Sieving Algorithm for the SVP. In *ICPP 2015 [ICP15]*, pages 590–599, 2015.

[MC14] Eric M. Mahé and Jean-Marie Chauvet. Fast GPGPU-Based Elliptic Curve Scalar Multiplication. *IACR Cryptology ePrint Archive*, 2014. `https://eprint.iacr.org/2014/198.pdf`.

[Men07] Alfred Menezes, editor. *Advances in Cryptology – CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*. Springer, 2007.

[Men09] Alfred Menezes. An Introduction to Pairing-Based Cryptography. *Contemporary Mathematics*, 477, 2009. `http://www.math.uwaterloo.ca/~ajmeneze/publications/pairings.pdf`.

[Mer14] Johannes Merkle. Re: [Cfrg] ECC reboot (Was: When's the decision?), 2014. `https://www.ietf.org/mail-archive/web/cfrg/current/msg05353.html`.

[Mit13] Shigeo Mitsunari. A Fast Implementation of the Optimal Ate Pairing over BN curve on Intel Haswell Processor. *IACR Cryptology ePrint Archive*, 2013. `http://eprint.iacr.org/2013/362/`.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve method of factorization. *Mathematics of Computation*, 48:243–264, 1987. `https://cr.yp.to/bib/1987/montgomery.pdf`.

[Mos14] Michele Mosca, editor. *Post-Quantum Cryptography, 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1–3, 2014, Proceedings*, volume 8772 of *Lecture Notes in Computer Science*. Springer, 2014.

[MS10] Stefan Mangard and François-Xavier Standaert, editors. *Cryptographic Hardware and Embedded Systems – CHES 2010, 12th International Workshop, Santa Barbara, USA, August 17–20, 2010, Proceedings*, volume 6225 of *Lecture Notes in Computer Science*. Springer, 2010.

[MSS08] Yi Mu, Willy Susilo, and Jennifer Seberry, editors. *Information Security and Privacy, 13th Australasian Conference, ACISP 2008, Wollongong, Australia, July 7–9, 2008, Proceedings*, volume 5107 of *Lecture Notes in Computer Science*. Springer, 2008.

[MSW$^+$14] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *USENIX 2014 [FJ14]*, pages 733–748, 2014. `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer`.

[MW15] Daniele Micciancio and Michael Walter. Fast Lattice Point Enumeration with Minimal Overhead. In *SODA 2015 [Ind15]*, pages 276–294, 2015.

[NBS08] Michael Naehrig, Paulo S.L.M. Barreto, and Peter Schwabe. On compressible pairings and their computation. In *AFRICACRYPT 2008 [Vau08]*, pages 371–388, 2008. `http://eprint.iacr.org/2007/429/`.

[Nie86] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15:159–166, 1986.

[NIS00] National Institute for Standards and Technology. FIPS PUB 186-2: Digital signature standard (DSS), 2000. `http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf`.

[NIS13] National Institute for Standards and Technology. FIPS PUB 186-4: Digital signature standard (DSS), 2013. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`.

[NNS10] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In *LATINCRYPT 2010 [AB10]*, pages 109–123, 2010. updated version: `https://cryptojedi.org/papers/dclxvi-20100714.pdf`.

[NO14] Phong Q. Nguyen and Elisabeth Oswald, editors. *Advances in Cryptology – EUROCRYPT 2014, 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11–15, 2014, Proceedings*, volume 8441 of *Lecture Notes in Computer Science*. Springer, 2014.

[NP02] David Naccache and Pascal Paillier, editors. *Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002, Paris, France, February 12–14, 2002, Proceedings*, volume 2274 of *Lecture Notes in Computer Science*. Springer, 2002.

[NP10] Phong Q. Nguyen and David Pointcheval, editors. *Public Key Cryptography – PKC 2010, 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26–28, 2010, Proceedings*, volume 6056 of *Lecture Notes in Computer Science*. Springer, 2010.

[NSA05] National Security Agency. Suite B Cryptography / Cryptographic Interoperability, 2005. `https://web.archive.org/web/20150724150910/https://www.nsa.gov/ia/programs/suiteb_cryptography/`.

[Odl86] Andrew M. Odlyzko, editor. *Advances in Cryptology – CRYPTO'86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*. Springer, 1986.

[OLARH13] Thomaz Oliveira, Julio López, Diego F. Aranha, and Francisco Rodríguez-Henríquez. Lambda coordinates for binary elliptic curves. In *CHES 2013 [BC13]*, pages 311–330, 2013. `https://eprint.iacr.org/2013/131`.

[OP98] Kazuo Ohta and Dingyi Pei, editors. *Advances in Cryptology – ASIACRYPT'98, International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, October 18–22, 1998, Proceedings*, volume 1514 of *Lecture Notes in Computer Science*. Springer, 1998.

[OS02] Katsuyuki Okeya and Kouichi Sakurai. Fast Multi-scalar Multiplication Methods on Elliptic Curves with Precomputation Strategy Using Montgomery Trick. In *CHES 2002 [JcKKP02]*, pages 564–578, 2002.

[Osa87] Hiroyuki Osada. The Galois groups of the polynomials $X^n + aX^1 + b$. *Journal of Number Theory*, 25(2):230–238, 1987.

[OSC10a] State Commercial Cryptography Administration (OSCCA), China. Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves, December 2010. `http://www.oscca.gov.cn/UpFile/2010122214822692.pdf`.

[OSC10b] State Commercial Cryptography Administration (OSCCA), China. Recommanded Curve Parameters for Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves, December 2010. `http://www.oscca.gov.cn/UpFile/2010122214836668.pdf`.

[OSK99] Kiyoshi Ohgishi, Ryuichi Sakai, and Masao Kasahara. Notes on ID-based Key Sharing Systems over Elliptic Curve (in Japanese). Technical Report ISEC99-57, IEICE, 1999.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA 2006 [Poi06]*, pages 1–20, 2006. `http://eprint.iacr.org/2005/271/`.

[OW99] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[Pat03] Kenneth G. Paterson, editor. *Cryptography and Coding, 9th IMA International Conference, Cirencester, UK, December 16–18, 2003, Proceedings*, volume 2898 of *Lecture Notes in Computer Science*. Springer, 2003.

[Pat11] Kenneth G. Paterson, editor. *Advances in Cryptology – EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15–19, 2011, Proceedings*, volume 6632 of *Lecture Notes in Computer Science*. Springer, 2011.

[Per13] Edoardo Persichetti. Secure and Anonymous Hybrid Encryption from Coding Theory. In *PQCrypto 2013 [Gab13]*, pages 174–187, 2013. `http://dx.doi.org/10.1007/978-3-642-38616-9_12`.

[PGHR13] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *S&P 2013 [IEE13]*, pages 238–252, 2013. `http://eprint.iacr.org/2013/279`.

[Pie08] Josef Pieprzyk, editor. *Advances in Cryptology – ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7–11, 2008, Proceedings*, volume 5350 of *Lecture Notes in Computer Science*. Springer, 2008.

[PJ12] David Pointcheval and Thomas Johansson, editors. *Advances in Cryptology – EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15–19, 2012, Proceedings*, volume 7237 of *Lecture Notes in Computer Science*. Springer, 2012.

[PJNB11] Geovandro C.C.F. Pereira, Marcos A. Simplício Jr, Michael Naehrig, and Paulo S.L.M. Barreto. A Family of Implementation-Friendly BN Elliptic Curves. *Journal of Systems and Software*, 84(8):1319–1326, 2011. `http://cryptojedi.org/papers/#fast-bn`.

[Poh81] Michael Pohst. On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. *SIGSAM Bulletin*, 15(1):37–44, feb 1981.

[Poi06] David Pointcheval, editor. *Topics in Cryptology – CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13–17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*. Springer, 2006.

[Pre09] Bart Preneel, editor. *Progress in Cryptology – AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21–25, 2009, Proceedings*, volume 5580 of *Lecture Notes in Computer Science*. Springer, 2009.

[PRW15] Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors. *Computer Security – ESORICS 2015, 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21–25, 2015, Proceedings, Part I*, volume 9326 of *Lecture Notes in Computer Science*. Springer, 2015.

[PS09] Xavier Pujol and Damien Stehlé. Solving the Shortest Lattice Vector Problem in Time $2^{2.465n}$. *IACR Cryptology ePrint Archive*, 2009. https://eprint.iacr.org/2009/605.

[PS12] Emmanuel Prouff and Patrick Schaumont, editors. *Cryptographic Hardware and Embedded Systems – CHES 2012, 14th International Workshop, Leuven, Belgium, September 9–12, 2012, Proceedings*, volume 7428 of *Lecture Notes in Computer Science*. Springer, 2012.

[PT06] Bart Preneel and Stafford E. Tavares, editors. *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11–12, 2005, Revised Selected Papers*, volume 3897 of *Lecture Notes in Computer Science*. Springer, 2006.

[PT11] Bart Preneel and Tsuyoshi Takagi, editors. *Cryptographic Hardware and Embedded Systems – CHES 2011, 13th International Workshop, Nara, Japan, September 28–October 1, 2011, Proceedings*, volume 6917 of *Lecture Notes in Computer Science*. Springer, 2011.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC'05 [GF05]*, pages 84–93, 2005.

[RK16] Matthew Robshaw and Jonathan Katz, editors. *Advances in Cryptology – CRYPTO 2016, 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14–18, 2016, Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*. Springer, 2016.

[Roy05] Bimal K. Roy, editor. *Advances in Cryptology – ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4–8, 2005, Proceedings*, volume 3788 of *Lecture Notes in Computer Science*. Springer, 2005.

[RS62] J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962. https://projecteuclid.org/euclid.ijm/1255631807.

[S$^+$15] William A. Stein et al. *Sage Mathematics Software (version 6.8)*. The Sage Development Team, 2015. http://www.sagemath.org.

[Sak07] Halvor Sakshaug. Security analysis of the NTRUEncrypt public key encryption scheme, 2007. brage.bibsys.no/xmlui/bitstream/handle/11250/258846/426901_FULLTEXT01.pdf.

[San94] Alfredo De Santis, editor. *Advances in Cryptology – EUROCRYPT'94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9–12, 1994, Proceedings*, volume 950 of *Lecture Notes in Computer Science*. Springer, 1994.

[SB04] Michael Scott and Paulo S.L.M. Barreto. Compressed Pairings. In *CRYPTO 2004 [Fra04]*, pages 140–156, 2004.

[SBC$^+$09] Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J. Kachisa. On the Final Exponentiation for Calculating Pairings on Ordinary Elliptic Curves. In *Pairing 2009 [SW09]*, pages 78–88, 2009. `eprint.iacr.org/2008/490/`.

[Sch03] Claus-Peter Schnorr. Lattice Reduction by Random Sampling and Birthday Methods. In *STACS 2003 [AH03]*, pages 145–156, 2003.

[Sco99] Michael Scott. Re: NIST announces set of Elliptic Curves, 1999. `https://groups.google.com/forum/message/raw?msg=sci.crypt/mFMukSsORmI/FpbHDQ6hM_MJ`.

[Sco11] Michael Scott. On the Efficient Implementation of Pairing-Based Protocols. In *IMACC 2011 [Che11]*, pages 296–308, 2011. `http://eprint.iacr.org/2011/334/`.

[Sel56] Ernst S. Selmer. On the irreducibility of certain trinomials. *Mathematica Scandinavica*, 4:287–302, 1956.

[SG14] Pascal Sasdrich and Tim Güneysu. Efficient Elliptic-Curve Cryptography Using Curve25519 on Reconfigurable Devices. In *ARC 2014 [GSCB14]*, pages 25–36, 2014. `https://www.hgi.rub.de/media/sh/veroeffentlichungen/2014/03/25/paper_arc14_curve25519.pdf`.

[SGY13] Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors. *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4–8, 2013*. ACM, 2013.

[Sho01] Victor Shoup. A Proposal for an ISO Standard for Public Key Encryption. *IACR Cryptology ePrint Archive*, 2001. `https://eprint.iacr.org/2001/112`.

[Sho02] Victor Shoup. OAEP Reconsidered. *Journal of Cryptology*, 15(4):223–249, 2002.

[SI14] Palash Sarkar and Tetsu Iwata, editors. *Advances in Cryptology – ASIACRYPT 2014, 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014, Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*. Springer, 2014.

[Sil99] Joseph H. Silverman. Almost inverses and fast NTRU key creation, 1999. https://assets.securityinnovation.com/static/downloads/NTRU/resources/NTRUTech014.pdf.

[Sil09] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. 2009.

[Sim40] Thomas Simpson. *Essays on several curious and useful subjects in speculative and mix'd mathematics, illustrated by a variety of examples*. 1740. https://cr.yp.to/bib/1740/simpson.html.

[SL02] Martijn Stam and Arjen K. Lenstra. Efficient Subgroup Exponentiation in Quadratic and Sixth Degree Extensions. In *CHES 2002 [JcKKP02]*, pages 318–332, 2002.

[Sma01] Nigel P. Smart. The Hessian form of an elliptic curve. In *CHES 2001 [cKKNP01]*, pages 118–125, 2001.

[Sma05] Nigel P. Smart, editor. *Cryptography and Coding, 10th IMA International Conference, Cirencester, UK, December 19–21, 2005, Proceedings*, volume 3796 of *Lecture Notes in Computer Science*. Springer, 2005.

[Sob12] Étienne Sobole. Calculateur de cycle pour le Cortex A8, 2012. http://pulsar.webshaker.net/ccc/index.php.

[SOK00] Ryuichi Sakai, Kiyoshi Ohgishi, and Masao Kasahara. Cryptosystems based on pairing. In *The 2000 Symposium on Cryptography and Information Security*, pages 135–148, 2000.

[SOK01] Ryuichi Sakai, Kiyoshi Ohgishi, and Masao Kasahara. Cryptosystems based on Pairing over Elliptic Curve (in Japanese). In *The 2001 Symposium on Cryptography and Information Security*, pages 23–26, 2001.

[SRH13] Ana Helena Sánchez and Francisco Rodríguez-Henríquez. NEON Implementation of an Attribute-Based Encryption Scheme. In *ACNS 2013 [JLMS13]*, pages 322–338, 2013. http://cacr.uwaterloo.ca/techreports/2013/cacr2013-07.pdf.

[SRY07] Kannan Srinathan, C. Pandu Rangan, and Moti Yung, editors. *Progress in Cryptology – INDOCRYPT 2007, 8th International Conference on Cryptology in India, Chennai, India, December 9–13, 2007, Proceedings*, volume 4859 of *Lecture Notes in Computer Science*. Springer, 2007.

[SS86] Claus-Peter Schnorr and Adi Shamir. An Optimal Sorting Algorithm for Mesh Connected Computers. In *STOC'86 [Har86]*, pages 255–263, 1986.

[SS11] Damien Stehlé and Ron Steinfeld. Making NTRU as Secure as Worst-Case Problems over Ideal Lattices. In *EUROCRYPT 2011 [Pat11]*, pages 27–47, 2011.

[SS13] Kazue Sako and Palash Sarkar, editors. *Advances in Cryptology – ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1–5, 2013, Proceedings, Part II*, volume 8270 of *Lecture Notes in Computer Science*. Springer, 2013.

[Sta05] Martijn Stam. A Key Encapsulation Mechanism for NTRU. In *IMA 2005 [Sma05]*, pages 410–427, 2005.

[SV10] Nigel P. Smart and Frederik Vercauteren. Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In *PKC 2010 [NP10]*, pages 420–443, 2010.

[SvdW06] Andrew Shallue and Christiaan van de Woestijne. Construction of Rational Points on Elliptic Curves over Finite Fields. In *ANTS-VII [HPP06]*, pages 510–524, 2006.

[SW05] Amit Sahai and Brent Waters. Fuzzy Identity-Based Encryption. In *EUROCRYPT 2005 [Cra05]*, pages 457–473, 2005. `http://eprint.iacr.org/2004/086/`.

[SW09] Hovav Shacham and Brent Waters, editors. *Pairing-Based Cryptography – Pairing 2009, Third International Conference, Palo Alto, CA, USA, August 12–14, 2009, Proceedings*, volume 5671 of *Lecture Notes in Computer Science*. Springer, 2009.

[Too63] Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23:37–71, 2010. `http://people.csail.mit.edu/tromer/papers/cache-joc-official.pdf`.

[TSK$^+$14] Tadanori Teruya, Kazutaka Saito, Naoki Kanayama, Yuto Kawahara, Tetsutaro Kobayashi, and Eiji Okamoto. Constructing Symmetric Pairings over Supersingular Elliptic Curves with Embedding Degree Three. In *Pairing 2013 [CZ14]*, pages 97–112, 2014.

[Vau08] Serge Vaudenay, editor. *Progress in Cryptology – AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11–14, 2008, Proceedings*, volume 5023 of *Lecture Notes in Computer Science*. Springer, 2008.

[Ver10] Frederik Vercauteren. Optimal Pairings. *IEEE Transactions on Information Theory*, 56(1), 2010. `http://www.cosic.esat.kuleuven.be/publications/article-1039.pdf`.

[vV16] Christine van Vredendaal. Reduced Memory Meet-in-the-Middle Attack against the NTRU Private Key. *LMS Journal of Computation and Mathematics*, 19:43–57, 2016. Special Issue A (Algorithmic Number Theory Symposium XII).

[Wik15a] Wikipedia. Nothing up my sleeve number, 2015. https://en.wikipedia.org/wiki/Nothing_up_my_sleeve_number (accessed 27 September 2015).

[Wik15b] Wikipedia. NewDES, 2015. https://en.wikipedia.org/wiki/NewDES (accessed 27 September 2015).

[WS12] Xiaoyun Wang and Kazue Sako, editors. *Advances in Cryptology – ASIACRYPT 2012, 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2–6, 2012, Proceedings*, volume 7658 of *Lecture Notes in Computer Science*. Springer, 2012.

[Wun16] Thomas Wunderer. Revisiting the Hybrid Attack: Improved Analysis and Refined Security Estimates. *IACR Cryptology ePrint Archive*, 2016. https://eprint.iacr.org/2016/733.

[WY05] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In *EUROCRYPT 2005 [Cra05]*, pages 19–35, 2005.

[X999] Accredited Standards Committee X9. American National Standard X9.62-1999, Public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ECDSA), 1999. Preliminary draft at http://grouper.ieee.org/groups/1363/Research/Other.html.

[X901] Accredited Standards Committee X9. American National Standard X9.63-2001, Public key cryptography for the financial services industry: key agreement and key transport using elliptic curve cryptography, 2001. Preliminary draft at http://grouper.ieee.org/groups/1363/Research/Other.html.

[YDKM06] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors. *Public Key Cryptography – PKC 2006, 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24–26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*. Springer, 2006.

[YF14] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX 2014 [FJ14]*, pages 719–732, 2014. https://eprint.iacr.org/2013/448.

[Yun02] Moti Yung, editor. *Advances in Cryptology – CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18–22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*. Springer, 2002.

[Zhe02] Yuliang Zheng, editor. *Advances in Cryptology – ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1–5, 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*. Springer, 2002.

[ZP16] Vassilis Zikas and Roberto De Prisco, editors. *Security and Cryptography for Networks – 10th International Conference, SCN 2016, Amalfi, Italy, August 31 – September 2, 2016, Proceedings*, volume 9841 of *Lecture Notes in Computer Science*. Springer, 2016.

[ZW14] Xusheng Zhang and Kunpeng Wang. Fast Symmetric Pairing Revisited. In *Pairing 2013 [CZ14]*, pages 131–148, 2014.

Index

Summary

Optimizing Curve-Based Cryptography

This thesis focuses on analyzing and improving the performance of public-key cryptographic schemes. Contributions include fast and secure software implementation of elliptic-curve cryptography. The content of this thesis is categorized into the following 4 topics.

Curves and arithmetic: Improving the underlying arithmetic often leads to speeding up the overall performance. The main bottleneck in elliptic-curve-based protocols is scalar multiplication. One way to speed up this operation is to accelerate the underlying point arithmetic operations. Another way is to express scalars in different representations such that there are fewer point operations, for example, by using double-base-chain representations.

Following the former approach, this thesis introduces faster formulas for point operations on twisted Hessian and twisted Edwards curves. For the latter, this thesis proposes a better algorithm to construct double-base chains which leads to faster scalar multiplication.

There exist many standards concerning elliptic curves, for example, ANSI, ANSSI, ISO, Brainpool. Each standard tries to ensure security. However, there are some flexibilities which allow vulnerable curves to be standardized. This thesis explores those possibilities and shows that it is possible to manipulate those standards by quantifying how much work the attackers need to perform.

Fast finite fields: Implementation is a way to put cryptography into practice. It is desirable to have fast implementations, and it is essential to have secure implementations. It is recommended that when implementing cryptographic software, side-channel attacks should be taken into account. Therefore, implementations presented in this thesis are ensured to run in constant time and contain neither branches depending on secret input nor array indices depending on secrets. This way, these implementations are secure against software side-channel attacks.

In terms of optimization, various techniques are applied such as utilizing vector units to parallelize computation, interleaving instructions to hide latency, and representing numbers in a way to prevent overflow. The interplay among different layers of computation from low to high level is also taken into consideration. For example, this thesis uses details of instruction sets on each microarchitecture, suitable combinations of fast multiplication algorithms such as Karatsuba and Toom, and state-of-the-art point arithmetic formulas. This thesis presents implementations setting speed

records for genus-2 hyperelliptic curves using the Kummer surface for representing elements and a very-high-security elliptic curve called Curve41417.

Pairing-based cryptography: One advantage of pairing-based cryptography is that it has features that cannot be achieved by any other schemes. For example, without pairings, short signatures, identity-based encryption, and attribute-based encryption would not be realizable. This thesis introduces a software framework called PandA which stands for Pairing and Arithmetic. This framework aims to provide designers and implementors access to a state-of-the-art elliptic curve arithmetic toolbox containing functions necessary for implementing pairing-based protocols. The default for these functions is to run in constant time, so that they will not be vulnerable to side-channel attacks. However, there are options for non-constant time, which may be faster, in case the computation does not deal with secret data.

Lattice-based cryptography: Curve-based cryptography is threatened by the coming of quantum computers. With the abilities of quantum computers, adversaries would be able to break curve-based cryptography in polynomial time. To ensure that the security of cryptography still holds even in the presence of quantum computers, post-quantum cryptography schemes have been proposed. This thesis presents NTRU Prime, a post-quantum lattice-based scheme. Its implementation achieves very competitive performance compared to other implementations of lattice-based schemes.

Curriculum Vitae

Chitchanok Chuengsatiansup was born on 17 September 1987, in Bangkok, Thailand.

In 2010, she graduated from Chulalongkorn University receiving the Bachelor degree of Engineering program in Computer Engineering with first class honors. Her graduation project's title is *"How to Write Best Paper: An Approach to Mining Characteristics of Good Papers"*; it was supervised by Boonserm Kijsirikul.

Chitchanok furthered her education in Japan under the Japanese Government scholarship (文部科学省奨学金). From April 2010 until March 2011, she was a research student at Imai's Laboratory, Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo. After that she enrolled in the Master's Program in Computer Science at the same university, The University of Tokyo. She graduated in March 2013. The title of her thesis is *"Improved Elliptic Curve Arithmetic by Reordering Operation Sequences"* (演算順序変更による楕円曲線算術の改善); it was supervised by Prof. Dr. Hiroshi Imai and Vorapong Supakitpaisarn.

Starting May 2013, Chitchanok began her Ph.D. studies in the Cryptographic Implementations group at Eindhoven University of Technology under supervision of Prof. Dr. Daniel J. Bernstein and Prof. Dr. Tanja Lange. Her work during this period was supported by the Netherlands Organisation for Scientific Research (NWO) under grants 639.073.005. Her research interests include efficient cryptographic software implementations, elliptic and hyperelliptic curve cryptography, pairing-based cryptography, lattice-based cryptography and secure messaging.