



Universidade Estadual de Campinas
Instituto de Computação



Armando Faz Hernandez

High-Performance Elliptic Curve Cryptography: A SIMD Approach to Modern Curves

Criptografia de Curvas Elípticas de Alto Desempenho:
Uma Abordagem SIMD para Curvas Modernas

CAMPINAS
2022

Armando Faz Hernandez

**High-Performance Elliptic Curve Cryptography: A SIMD
Approach to Modern Curves**

**Criptografia de Curvas Elípticas de Alto Desempenho: Uma
Abordagem SIMD para Curvas Modernas**

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Julio Cesar Lopez Hernandez

Este exemplar corresponde à versão final da Tese defendida por Armando Faz Hernandez e orientada pelo Prof. Dr. Julio Cesar Lopez Hernandez.

CAMPINAS
2022

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

F297h Faz Hernández, Armando, 1987-
High-performance elliptic curve cryptography : a SIMD approach to modern curves / Armando Faz Hernández. – Campinas, SP : [s.n.], 2022.

Orientador: Julio César López Hernández.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia. 2. Criptografia de curvas elípticas. 3. Corpos finitos (Álgebra). 4. Aritmética de computador. 5. Algoritmos paralelos. I. López Hernández, Julio César, 1961-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações Complementares

Título em outro idioma: Criptografia de curvas elípticas de alto desempenho : uma abordagem SIMD para curvas modernas

Palavras-chave em inglês:

Cryptography

Elliptic curves cryptography

Finite fields (Algebra)

Computer arithmetic

Parallel algorithms

Área de concentração: Ciência da Computação

Titulação: Doutor em Ciência da Computação

Banca examinadora:

Julio Cesar Lopez Hernandez

Routo Terada

Eduardo Moraes de Moraes

Sara Díaz Cardell

Marco Aurélio Amaral Henriques

Data de defesa: 23-09-2022

Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)

- ORCID do autor: <https://orcid.org/0000-0001-5502-8666>

- Currículo Lattes do autor: <http://lattes.cnpq.br/9952901649594455>



Universidade Estadual de Campinas
Instituto de Computação



Armando Faz Hernandez

**High-Performance Elliptic Curve Cryptography: A SIMD
Approach to Modern Curves**

**Criptografia de Curvas Elípticas de Alto Desempenho: Uma
Abordagem SIMD para Curvas Modernas**

Comissão Examinadora:

- Prof. Dr. Julio Cesar Lopez Hernandez
Instituto de Computação
Universidade Estadual de Campinas
- Prof. Dr. Marco Aurélio Amaral Henriques
Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas
- Prof. Dr. Routo Terada
Instituto de Matemática e Estatística
Universidade de São Paulo
- Dr. Eduardo Moraes de Moraes
Protocol Labs
- Profa. Dra. Sara Díaz Cardell
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 23 de setembro de 2022

*Con todo mi cariño
para Ma, Pu y Ri.*

*Enseñarás a volar,
pero no volarán tu vuelo.
Enseñarás a soñar,
pero no soñarán tu sueño.
Enseñarás a vivir,
pero no vivirán tu vida.*

*Sin embargo...
en cada vuelo,
en cada vida,
en cada sueño,
perdurará siempre la huella
del camino enseñado.*



Acknowledgments

This research project was partially supported by the following fellowships and grants.

2015-2018 Secure Execution of Cryptographic Algorithms.

Grant #2014/50704-7, São Paulo Research Foundation (FAPESP).

2013-2016 Software Implementation of Cryptographic Algorithms.

Energy-Efficient Security for SoC devices. Intel Strategic Research Alliance.
Intel University Research Office.

2014 Security and Reliability of Information: Theory and Practice.

Grant #2013/25977-7, São Paulo Research Foundation (FAPESP).

2011 The São Paulo Advanced School of Cryptography, SP-ASCrypto 2011.

Grant #2011/50273-8, São Paulo Research Foundation (FAPESP).

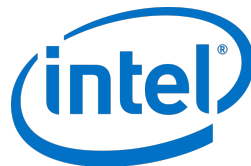
2018 ACM SIGSAC. Travel grant. Incheon, South Korea. June, 2018.

2013-2017 Latincrypt and ASCrypto committees. Travel grants and stipends.

Latincrypt eds. 2014, 2015, and 2017. ASCrypto eds. 2013, 2015, and 2017.

2017-2018 Serviço de Apoio ao Estudante (SAE).

Bolsa Pesquisa-Empresa.



Resumo

A criptografia baseada em curvas elípticas fornece métodos eficientes para a criptografia de chave pública. Pesquisa recente tem mostrado a superioridade das curvas de Montgomery e de Edwards sobre as curvas de Weierstrass pois elas precisam de menos operações aritméticas. O uso destas curvas modernas, porém, traz consigo diversos desafios na construção de algoritmos criptográficos deixando em aberto novos alvos de otimizações.

Nosso objetivo principal é propor otimizações algorítmicas e técnicas de implementação para os algoritmos criptográficos baseados em curvas elípticas. Visando acelerar a execução destes algoritmos, nossa abordagem fundamenta-se na utilização extensões ao conjunto de instruções da arquitetura. Além daquelas específicas para a criptografia, nós usamos extensões que seguem o paradigma de cômputo paralelo SIMD (do inglês, *Single Instruction, Multiple Data*). Neste modelo, o processador executa a mesma operação sobre um conjunto de dados de forma paralela. Nós investigamos como aplicar o modelo SIMD na implementação de algoritmos de curvas elípticas.

Como parte de nossas contribuições, projetamos algoritmos paralelos para a aritmética de corpos primos e de curvas elípticas. Projetamos também um algoritmo de multiplicação escalar para calcular $P + kQ$ e uma fórmula otimizada para calcular $3P$ em curvas de Montgomery. Estes algoritmos encontraram aplicabilidade na criptografia baseada em isogenias. Usando extensões SIMD tais como SSE, AVX e AVX2, desenvolvemos implementações otimizadas dos seguintes algoritmos criptográficos: X25519, X448, SIDH, ECDH, ECDSA, EdDSA e qDSA. Testes de desempenho mostram que essas implementações são mais rápidas do que as implementações existentes no estado da arte.

Nosso estudo confirma que o uso de extensões ao conjunto de instruções da arquitetura é uma ferramenta efetiva para otimizar implementações de algoritmos criptográficos baseados em curvas elípticas. Seja isto um incentivo não somente para aqueles que procuram acelerar os programas em geral, mas também para que os fabricantes de computadores incluam mais extensões avançadas para apoiar a demanda crescente da criptografia.

Abstract

Cryptography based on elliptic curves is endowed with efficient methods for public-key cryptography. Recent research has shown the superiority of the Montgomery and Edwards curves over the Weierstrass curves as they require fewer arithmetic operations. Using these modern curves has, however, introduced several challenges to the cryptographic algorithm's design, opening up new opportunities for optimization.

Our main objective is to propose algorithmic optimizations and implementation techniques for cryptographic algorithms based on elliptic curves. In order to speed up the execution of these algorithms, our approach relies on the use of extensions to the instruction set architecture. In addition to those specific for cryptography, we use extensions that follow the Single Instruction, Multiple Data (SIMD) parallel computing paradigm. In this model, the processor executes the same operation over a set of data in parallel. We investigated how to apply SIMD to the implementation of elliptic curve algorithms.

As part of our contributions, we design parallel algorithms for prime field and elliptic curve arithmetic. We also design a new three-point ladder algorithm for the scalar multiplication $P + kQ$, and a faster formula for calculating $3P$ on Montgomery curves. These algorithms have found applicability in isogeny-based cryptography. Using SIMD extensions such as SSE, AVX, and AVX2, we develop optimized implementations of the following cryptographic algorithms: X25519, X448, SIDH, ECDH, ECDSA, EdDSA, and qDSA. Performance benchmarks show that these implementations are faster than existing implementations in the state of the art.

Our study confirms that using extensions to the instruction set architecture is an effective tool for optimizing implementations of cryptographic algorithms based on elliptic curves. May this be an incentive not only for those seeking to speed up programs in general but also for computer manufacturers to include more advanced extensions that support the increasing demand for cryptography.

Resumen

La criptografía basada en curvas elípticas está dotada de métodos eficientes para la criptografía de llave pública. Investigaciones recientes han mostrado la superioridad de las curvas de Montgomery y de Edwards sobre las curvas de Weierstrass, pues emplean menos operaciones aritméticas. El uso de estas curvas modernas, en cambio, introduce varios desafíos en el diseño de algoritmos criptográficos y expone nuevos blancos de optimización.

Nuestro objetivo principal es proponer optimizaciones algorítmicas y técnicas de implementación para algoritmos criptográficos basados en curvas elípticas. Para acelerar la ejecución de estos algoritmos, nuestro enfoque se basa en el uso de extensiones al conjunto de instrucciones de la arquitectura. Además de aquellas específicas para la criptografía, nosotros usamos las extensiones que siguen el paradigma de cómputo paralelo SIMD (del inglés, *Single Instruction, Multiple Data*). En este modelo, el procesador ejecuta la misma operación sobre un conjunto de datos de forma paralela. Nosotros investigamos cómo aplicar el modelo SIMD en la implementación de algoritmos de curvas elípticas.

Como parte de nuestras contribuciones, diseñamos algoritmos paralelos para la aritmética de cuerpos primos y de curvas elípticas. Diseñamos también un algoritmo de multiplicación escalar para calcular $P + kQ$ y una fórmula optimizada para calcular $3P$ en curvas de Montgomery. Estos algoritmos encontraron aplicabilidad en la criptografía basada en isogenias. Usando extensiones SIMD tales como SSE, AVX y AVX2, desarrollamos implementaciones optimizadas de los siguientes algoritmos criptográficos: X25519, X448, SIDH, ECDH, ECDSA, EdDSA y qDSA. Pruebas de rendimiento muestran que estas implementaciones son más rápidas que implementaciones existentes en el estado del arte.

Nuestro estudio confirma que el uso de extensiones al conjunto de instrucciones de la arquitectura es una herramienta efectiva para optimizar implementaciones de algoritmos criptográficos basados en curvas elípticas. Sea esto un incentivo no solamente para aquellos que desean acelerar programas en general, sino también para que los fabricantes de computadoras incluyan más extensiones avanzadas que respalden la demanda creciente de la criptografía.

List of Figures

2.1.1	Instruction execution in a five-stage pipeline.	37
2.1.2	Execution engine of the Sandy Bridge micro-architecture.	39
2.1.3	Execution engine of the Haswell micro-architecture.	39
2.2.1	Scalar vs vector processing.	43
2.2.2	Relevant AVX2 instructions.	47
2.4.1	Evolution of vector instructions and hardware extensions.	52
3.4.1	Calculation of additions modulo p_{25519}	77
3.4.4	Storing a $\mathbb{F}_{p_{25519}}$ element in 128-bit registers.	82
3.4.14	Scheduling of 128-bit instructions for multiplications.	86
3.4.16	Storing two $\mathbb{F}_{p_{25519}}$ elements in 256-bit registers.	89
3.4.18	Storing four $\mathbb{F}_{p_{25519}}$ elements in 256-bit registers.	91
3.4.19	Storing four $\mathbb{F}_{p_{25519}}$ elements in 512-bit registers.	91
3.5.1	Storing a $\mathbb{F}_{p_{384}}$ element in 128-bit registers.	93
3.5.5	Recursion tree of Karatsuba multiplication for $l = 14$	95
3.5.6	Parallel execution of Karatsuba recursion tree.	96
3.5.10	Storing two $\mathbb{F}_{p_{384}}$ elements in 256-bit registers.	99
3.6.1	Non-constant-time calculation of additions modulo p_{448}	101
3.6.4	Partial reduction modulo p_{448}	103
3.6.7	Storing a $\mathbb{F}_{p_{448}}$ element in 128-bit registers.	105
3.6.14	Storing two $\mathbb{F}_{p_{448}}$ elements in 256-bit registers.	109
3.6.16	Storing four $\mathbb{F}_{p_{448}}$ elements in 256-bit registers.	110
3.7.7	Storing a $\mathbb{F}_{p_{751}}$ element in 128-bit registers.	117
3.7.9	Recursion tree of Karatsuba multiplication for $l = 28$	118
3.7.10	Storing a $\mathbb{F}_{p_{751}^2}$ element in 256-bit registers.	119
4.1.13	Geometric description of the chord-and-tangent rule.	126
4.5.8	Parallel point addition of twisted Edwards curves by Hisil et al.	163
4.5.9	Proposed parallel point addition of twisted Edwards curves.	164
5.3.3	Performance profiling of Ed25519 signature operations.	186
5.3.4	Performance profiling of Ed448 signature operations.	187
5.6.16	SHA-NI instructions for message schedule phase.	205
5.6.21	Two rounds of the SHA256 algorithm.	206
5.6.23	Performance of SHA-256 measured on Zen.	207
5.7.1	Pipeline execution of SHA256RND2 instructions.	210
5.7.2	Timings of multiple-message SHA-256 hashing.	211
5.7.3	Timings of pipelined multiple-message SHA-256 hashing.	212
5.7.5	Performance comparison of XMSS.	213

List of Tables

1.1.1	Comparison of the bit-length of keys of cryptosystems.	23
1.5.1	Technical specifications of the computers used in this research.	31
2.1.4	Comparison of hardware optimizations.	42
3.2.8	Comparison of machine representations of n -bit integers.	68
3.4.2	Instruction counts for 256-bit integer multiplication.	80
3.4.15	Time in clock cycles of $\mathbb{F}_{p_{25519}}$ operations measured on Skylake.	88
3.4.17	Time in clock cycles of two-way $\mathbb{F}_{p_{25519}}$ operations measured on Skylake.	90
3.4.20	Time in clock cycles of four-way $\mathbb{F}_{p_{25519}}$ operations measured on Skylake.	92
3.5.11	Time in clock cycles of $\mathbb{F}_{p_{384}}$ operations measured on Skylake.	100
3.6.3	Instruction counts of 448-bit integer multiplication.	102
3.6.13	Time in clock cycles of $\mathbb{F}_{p_{448}}$ operations measured on Skylake.	108
3.6.15	Time in clock cycles of two-way $\mathbb{F}_{p_{448}}$ operations measured on Skylake.	110
3.6.17	Time in clock cycles of four-way $\mathbb{F}_{p_{448}}$ operations measured on Skylake.	111
3.7.6	Instruction counts of reduction modulo p_{751}	116
3.7.15	Time in clock cycles of $\mathbb{F}_{p_{751}^2}$ operations measured on Skylake.	121
4.3.2	Operation counts of point addition for Weierstrass curves.	141
4.3.9	Operation counts of parallel \mathbb{F}_q -complete formulas.	144
4.4.18	Operation counts of parallel algorithms for Montgomery ladder step.	153
4.4.22	Operation counts of scalar multiplication on the x -line.	156
4.4.25	Operation counts of $\mathbf{x}(P + kQ)$ on the x -line.	158
4.4.35	Operation counts of point tripling on the x -line.	161
4.5.12	Time in clock cycles of point addition on a twisted Edwards curve.	165
4.5.19	Operation counts of fixed-point multiplication on the edwards25519 curve.	168
5.1.1	Suite B of cryptographic algorithms.	171
5.1.6	Timings of ECDH and ECDSA using P-384.	174
5.2.5	Timings of X25519 shared secret.	179
5.2.6	Timings of X448 shared secret.	180
5.2.7	Timings of the key generation phase of X25519 and X448.	181
5.2.8	Timings of the Diffie-Hellman protocol at the 128-bit security level.	182
5.3.5	Timings of Ed25519 and Ed448.	188
5.4.13	Timings of qDSA using the right-to-left fixed-point ladder.	195
5.4.14	Timings of qDSA operations.	195
5.4.15	Timings of qDSA and other digital signature schemes.	196
5.5.4	Timings of $\mathbf{x}(P + [k]Q)$ for SIDH-751.	201
5.5.5	Timings of SIDH-751.	201
5.7.4	Timings of XMSS and XMSS ^{MT} measured on Zen.	213

List of Algorithms

2.3.1	The AES encryption algorithm..	48
3.1.6	Integer addition using full adder circuit.	57
3.1.8	Integer subtraction using full subtractor circuit..	57
3.1.14	Montgomery’s REDC algorithm.	60
3.2.2	Obtaining the radix- 2^w representation of an integer.	66
3.2.4	Obtaining the generalized polynomial representation of an integer.. . . .	67
3.2.12	Integer Addition using Polynomial Representation.	69
3.2.13	Integer Multiplication using Polynomial Representation (operand scanning method).	69
3.2.14	Integer Addition using Redundant Representation.	70
3.2.23	Integer Multiplication using Redundant Representation.	72
3.2.26	Karatsuba Algorithm for Integer Multiplication using Redundant Representation.	73
3.2.28	Digit Size Reduction (sequential)..	74
3.2.29	Digit Size Reduction (parallel)..	75
3.4.12	Adaptation of Mastrovito’s Algorithm for Prime Field Multiplication using Redundant Representation.	84
3.5.7	Reduction modulo p_{384}	96
3.6.2	Constant-time implementation of additions modulo p_{448}	102
3.6.10	Karatsuba multiplication merged with reduction modulo p_{448}	107
3.7.1	Montgomery’s REDC algorithm in radix- 2^w	112
3.7.5	Montgomery’s REDC tuned for $p = 2^a b - 1$ in radix 2^w	115
3.7.13	Multiplication on \mathbb{F}_{p^2}	120
3.7.14	Squaring on \mathbb{F}_{p^2}	121
4.2.1	Left-to-Right Binary Algorithm for Scalar Multiplication.	130
4.2.2	Right-to-Left Binary Algorithm for Scalar Multiplication.	130
4.2.3	Fixed-Window Algorithm for Scalar Multiplication.	131
4.2.4	Conversion of Integers to ω -NAF Representation.	132
4.2.5	The ω -NAF Algorithm for Scalar Multiplication.	132
4.2.6	Conversion of Integers to Signed-Digit Representation..	133
4.2.7	Regular Signed-Digit Algorithm for Scalar Multiplication.	133
4.2.8	Montgomery Ladder for Scalar Multiplication.	134
4.2.9	Joye Ladder for Scalar Multiplication.	134
4.2.10	CSWAP-based Montgomery Ladder Algorithm for Scalar Multiplication.. .	135
4.2.11	CMOV-based Montgomery Ladder Algorithm for Scalar Multiplication. . .	135
4.2.12	CSWAP-based Joye Ladder Algorithm for Scalar Multiplication.	135
4.2.13	CSWAP Implemented with Logic Arithmetic.	136
4.2.14	CMOV Implemented with Logic Arithmetic..	136

4.2.15	Interleaved Algorithm for Multiple-Point Multiplication.	138
4.2.16	Interleaved Algorithm with ω -NAF for Double-Point Multiplication.	139
4.3.3	Two-way \mathbb{F}_q -Complete Point Addition on $E/\mathbb{F}_q: y^2 = x^3 + Ax + B$	142
4.3.4	Two-way \mathbb{F}_q -Complete Point Addition on $E/\mathbb{F}_q: y^2 = x^3 - 3x + B$	142
4.3.5	Two-way \mathbb{F}_q -Complete Point Doubling on $E/\mathbb{F}_q: y^2 = x^3 + Ax + B$	143
4.3.6	Two-way \mathbb{F}_q -Complete Point Doubling on $E/\mathbb{F}_q: y^2 = x^3 - 3x + B$	143
4.3.7	Four-way \mathbb{F}_q -Complete Point Addition on $E/\mathbb{F}_q: y^2 = x^3 + Ax + B$	143
4.3.8	Four-way \mathbb{F}_q -Complete Point Doubling on $E/\mathbb{F}_q: y^2 = x^3 + Ax + B$	144
4.4.11	Montgomery Ladder Algorithm for Scalar Multiplication on the x -Line.	148
4.4.14	Recovering the y -Coordinate of a Point on a Montgomery Curve.	150
4.4.15	Scalar Multiplication of Points on a Montgomery Curve.	150
4.4.16	Two-way Parallel Algorithm for Montgomery Ladder Step.	151
4.4.17	Four-way Parallel Algorithm for Montgomery Ladder Step.	152
4.4.19	Right-to-Left Ladder Algorithm for Scalar Multiplication on the x -Line.	154
4.4.21	Right-to-Left Ladder Algorithm for Fixed-Point Multiplication on the x - Line.	155
4.4.23	Left-to-Right Three-Point Ladder Algorithm for $\mathbf{x}(P + kQ)$	157
4.4.24	Right-to-Left Three-Point Ladder Algorithm for $\mathbf{x}(P + kQ)$	157
4.4.26	Right-to-Left Three-Point Ladder Algorithm for $\mathbf{x}(P + kQ)$ with Fixed- Points.	159
4.4.34	Point Tripling on the x -Line.	161
4.5.10	Two-way Point Addition for Twisted Edwards Curves with $a = -1$	165
4.5.11	Four-way Point Addition for Twisted Edwards Curves with $a = -1$	166
4.5.16	Conversion of Integers to Signed Digits.	167
5.4.12	Unequivocal Verification Procedure for qDSA.	194
5.6.11	Update Function for SHA-256.	203
5.6.22	SHA-256 Update Implemented with SHA-NI.	207

Contents

Foreword	18
Preface	19
1 Introduction	20
1.1 Cryptography	20
1.1.1 Symmetric-Key Cryptography	21
1.1.2 Public-Key Cryptography	21
1.1.3 Elliptic Curve Cryptography	22
1.2 Cryptographic Engineering	24
1.2.1 Security of Implementations	24
1.2.2 Efficiency of Implementations	25
1.2.3 Shifting to Modern Elliptic Curves	25
1.3 Related Works	26
1.3.1 Algorithmic Optimizations	26
1.3.2 Software Implementations of Elliptic Curves	27
1.3.3 Implementations using Hardware Extensions	28
1.4 Research Problem	29
1.4.1 Motivation	29
1.4.2 Problem Statement	30
1.5 Aims and Scope	30
1.5.1 Aims	30
1.5.2 Scope	30
1.6 Contributions	32
1.6.1 Publications	32
1.6.2 Software Libraries	34
1.7 Outline	35
2 Modern Computer Architectures	36
2.1 Optimizations in Computer Architectures	36
2.1.1 Pipeline Execution	37
2.1.2 Superscalar Processors	38
2.1.3 Simultaneous Multi-Threading	39
2.1.4 Parallel Computing	40
2.1.5 Comparison of Optimizations	42
2.2 SIMD Vector Units	43
2.2.1 Vector Instructions of the x86-64 Architecture	44
2.3 Extensions for Cryptography	48
2.3.1 The AES New Instructions	48

2.3.2	The Carry-Less Multiplier	49
2.3.3	Multi-Precision Integer Arithmetic	49
2.3.4	The SHA New Instructions	51
2.3.5	Vectorized AES and Galois Field Extensions	51
2.4	Evolution of Hardware Extensions	52
2.5	Chapter Summary	52
3	Prime Field Arithmetic	54
3.1	Algebraic Structures	54
3.1.1	The Ring of Integers	56
3.1.2	Prime Fields	58
3.1.3	Extension Fields	63
3.2	Operations over Prime Field Elements	65
3.2.1	Machine Representation of Integers	65
3.2.2	Operations using Polynomial Representation	68
3.2.3	Operations using a Redundant Representation	70
3.3	Parallel Calculation of Arithmetic Operations	75
3.4	Arithmetic on $\text{GF}(2^{255} - 19)$	76
3.4.1	Polynomial Representation	77
3.4.2	Redundant Representation	81
3.4.3	Two-way Operations	89
3.4.4	Four-way Operations	91
3.5	Arithmetic on $\text{GF}(2^{384} - 2^{128} - 2^{96} + 2^{32} - 1)$	92
3.5.1	Redundant Representation	93
3.5.2	Two-way Operations	99
3.5.3	Performance Benchmark	99
3.6	Arithmetic on $\text{GF}(2^{448} - 2^{224} - 1)$	100
3.6.1	Polynomial Representation	101
3.6.2	Redundant Representation	104
3.6.3	Two-way Operations	109
3.6.4	Four-way Operations	110
3.7	Arithmetic on $\text{GF}((2^a b - 1)^2)$	111
3.7.1	Improvements on the Reduction Modulo $p = 2^a b - 1$	111
3.7.2	Redundant Representation	117
3.7.3	Two-way Operations for the Quadratic Extension	119
3.7.4	Performance Benchmark	121
3.8	Chapter Summary	122
4	Arithmetic of Elliptic Curves	123
4.1	Background	123
4.1.1	Elliptic Curves	123
4.1.2	The Group of Rational Points	126
4.1.3	Elliptic Curve Discrete Logarithm Problem	128
4.2	Algorithms for Scalar Multiplication	129
4.2.1	Basic Algorithms	129
4.2.2	Algorithms with Regular Execution Pattern	131
4.2.3	Ladder Algorithms	133
4.2.4	Special Cases	136
4.3	Arithmetic of Weierstrass Curves	139

4.3.1	Point Addition Formulas	139
4.3.2	Parallel Complete Addition Formulas	141
4.4	Arithmetic of Montgomery Curves	144
4.4.1	Montgomery Curves	145
4.4.2	The x -Line Variety	146
4.4.3	Parallel Montgomery Ladder Step	150
4.4.4	A Review of Right-To-Left Algorithms	152
4.4.5	A New Three-Point Ladder Algorithm	155
4.4.6	An Optimized Point Tripling Formula	159
4.5	Arithmetic of Twisted Edwards Curves	161
4.5.1	Twisted Edwards Curves	162
4.5.2	Parallel Point Addition	163
4.5.3	Fixed-Point Multiplication	166
4.6	Chapter Summary	169
5	Cryptographic Algorithms and Protocols	170
5.1	Implementation of ECDH and ECDSA with P-384	170
5.1.1	Review of Standard Elliptic Curves	170
5.1.2	Implementation Details	172
5.1.3	Performance Benchmark and Comparison	173
5.2	Implementation of X25519 and X448	175
5.2.1	Review of Diffie-Hellman Protocol on the x -Line	175
5.2.2	Implementation Details	177
5.2.3	Performance Benchmark and Comparison	179
5.3	Implementation of Ed25519 and Ed448	182
5.3.1	Review of Edwards Digital Signature Algorithm	183
5.3.2	Implementation Details	184
5.3.3	Performance Benchmark and Comparison	186
5.4	Implementation of qDSA using Curve25519	188
5.4.1	Review of the Quotient Digital Signature Algorithm	189
5.4.2	Implementation Details	190
5.4.3	Unequivocal Verification Methods	191
5.4.4	Performance Benchmark and Comparison	195
5.5	Implementation of SIDH-751	196
5.5.1	Review of Supersingular Isogeny Diffie-Hellman	197
5.5.2	Implementation Details	199
5.5.3	Performance Benchmark and Comparison	200
5.6	Implementation of SHA-256	202
5.6.1	The SHA-256 Algorithm	202
5.6.2	Implementation Details	203
5.6.3	Performance Benchmark and Comparison	206
5.7	Implementation of XMSS and XMSS ^{MT}	208
5.7.1	Review of Hash-based Signatures	208
5.7.2	Implementation Details	209
5.7.3	Performance Benchmark and Comparison	211
5.8	Chapter Summary	214

6	Conclusions	215
6.1	Concluding Remarks	215
6.2	Retrospective	216
6.2.1	Specialized Literature	216
6.2.2	Programming Languages	216
6.2.3	Usage of AVX2	217
6.2.4	New Instructions	218
6.3	Summary of Contributions	219
6.3.1	Algorithmic Optimizations	219
6.3.2	Implementation Techniques	220
6.4	Future Work	221
	Bibliography	222
A	Research Production	250
A.1	Journal Articles	250
A.1.1	A Faster Implementation of SIDH	250
A.1.2	High-Performance Implementation of ECC	251
A.2	Publications in Conference Proceedings	251
A.2.1	Software Implementation of Prime Fields	251
A.2.2	Curve25519 using AVX2	252
A.2.3	How to Precompute a Ladder	253
A.2.4	Speeding up the P-384 Curve	253
A.2.5	Implementation of qDSA	254
A.2.6	Performance Evaluation of Cryptographic Instructions	255
A.2.7	Vectorization of Hash to Curve Functions	256
A.3	Book Chapters	257
A.3.1	Implementation of Cryptographic Algorithms	257

Foreword

Armando's journey towards obtaining his doctoral degree was long, and I am delighted to have witnessed it from a close distance for most of it. It started during the end of my doctoral degree at UNICAMP and continued after we both left Brazil in search of better opportunities.

Our collaboration began in 2011 when we worked on using the unpronounceable PCLMULQDQ carryless multiplier instruction to speed up software implementations of binary elliptic curves. At the time, Armando was doing his Master's degree in Mexico under the supervision of Francisco Rodríguez-Henríquez, Ph.D., who became a great common friend not long after. I can still remember the excitement of finally seeing binary fields as a first-class citizen in modern vector instruction sets—supported natively by a main computer manufacturer—and then the disappointment of benchmarking the instruction at 14 cycles.

That was our first CHES paper, accepted at the premier conference of our still-growing field. There were other collaborations after that, some of which are included in this thesis. We extended our techniques to the Koblitz curves for LATINCRYPT 2012, got a joint paper accepted to SPACE 2017, and worked on a short course for SBSeg 2015. Writing short course chapters is a tradition in Brazil, as a way for more senior researchers to introduce the field to newcomers. I find it sometimes a joyless and laborious work, but sharing the burden with Armando and co-authors made it better somehow. Meanwhile, the throughput of PCLMULQDQ decreased to just 1 cycle!

Armando has come a long way, and it shows clearly in his thesis. He matured from a graduate student to an experienced cryptographic engineer with an impressive publication record. His contributions in the thesis set speed records for the implementation of many standardized curves used in production for protocols critical to modern society, such as SSH and TLS. The breadth of his work went past beyond Elliptic Curve Cryptography to also impact the emerging field of quantum-safe isogeny-based cryptography, which is, unfortunately, facing some troubled teenage years currently.

I hope the reader can have as much fun reading the thesis, as I had enjoyed collaborating with Armando and witnessing his every new achievement. Since we share our brilliant doctoral advisor Julio López, Ph.D., I guess this makes Armando my academic younger brother. Here he is for many more speed records to come!

Diego F. Aranha, Ph.D.
Associate Professor, Aarhus University

Preface

From my perspective, I feel I started this project a while back ago, even before I joined UNICAMP. One good day, I was a part of a crowd of international students attending the Advanced School of Cryptography (ASCrypto'11) in Atibaia, São Paulo. This event certainly pushed me forward in the right direction. Elliptic curves and bilinear pairings were the trending topics in my circles. To be honest, I was tempted to get involved in cryptanalysis, but I found lots of affinity with UNICAMP's research projects, its faculty staff, and the mighty future that was looming. A couple of emails later and a quick conversation with Prof. Julio Lopez in Santiago, Chile was more than enough to decide to pack my stuff and land in Campinas in 2013 to become a doctoral student.

During my PhD, I have received support in many different ways. So, I would like to express my gratitude to Prof. Julio Lopez, who advises me not only academically but also in daily life. I remember many good conversations that left me with meaningful things. He always showed eagerness to support my research, and I appreciate his patience and trust during the bad days. Needless to say, we share a particular interest in binary curves.

I also want to highlight the leadership and expertise of Prof. Ricardo Dahab, who has fostered a solid cryptography community. Likewise, I admire the tenacity of Diego Aranha, we worked close while he was at UNICAMP. To Francisco Rodríguez, for the support given when attending Latincrypt events. Also, I want to mention the kindness of Fabio Tagnin, David Ott, and Rafael Misoczki, who followed in part this project. To Nick Sullivan, who gives me valuable advice.

I would like to thank my committee members, all my professors, administrative staff, the international office, libraries, dining and other campus facilities of the University of Campinas for the great labor done at the service of national and foreign students. Similar acknowledgments, to the state of São Paulo and the entire Brazilian nation.

I feel fortunate to meet amazing people and colleagues who were partakers of many adventures on my journey. I appreciate and value the friendship of Amanda, Ana, Carlos, Citlali, David, Eduardo, Francisco, Hayato, Israel, Jadisha, Junior, Karina, Kihiro, Leonara, Manuel, Marcelo, Marleny, Patrick, Rafael, Roberto, Sheila, Thomaz, and everyone that I missed mentioning.

Todo mi aprecio a mi querida familia; a Carmen, Armando y Susana. Porque gracias a su ayuda incondicional y motivación hoy podemos celebrar juntos este nuevo logro.

This thesis summarizes lots of efforts for optimizing elliptic curve cryptography. Participating in this project was satisfactory, with no time to get bored as new and clever ideas appear every other day. Parts of this document were written in sunny Campinas and hilly San Francisco. Hope this research helps others to solve problems, and encourage people to continue researching the points that remain unexplored.

Armando Faz Hernández

Chapter 1

Introduction

The simplest communication model involves two honest participants, called Alice and Bob, who want to interchange messages through a public communication channel. There is also a participant, known as Eve, who can eavesdrop the messages transmitted through the channel. In this context:

*How can Alice and Bob communicate confidentially
even in the presence of Eve?*

This is a central problem in the field of cryptography.

1.1 Cryptography

Cryptography is a discipline that studies how to encode information so it remains secret and protected from adversaries. More formally, *cryptography* is the scientific study of techniques for securing digital information, transactions, and distributed computations [170].

One solution cryptography provides to the above problem is data encryption. An encryption algorithm converts a message, also known as *plaintext*, into a *ciphertext*. Alice encrypts a message and send the ciphertext to Bob. Once Bob receives the ciphertext, he uses a decryption algorithm that recovers the original message from the ciphertext. Eve never has access to the original message because only the ciphertext is in transit. So Alice and Bob can communicate confidentially.

An issue of the procedure described above is that the algorithms must be kept secret. If Eve knows the encryption or decryption algorithms, she would be able to participate in the conversation. Hiding the algorithms is, however, not always possible or even practical. A better approach is to shift the secrecy requirements from the algorithms to the secrecy of an additional data, referred to as *keys*. Thus, the encryption and decryption algorithms take a key as an additional input making possible to publicly disclose the algorithms used. If Alice and Bob maintain the key in secrecy, they can interchange encrypted messages without Eve being able to read them.

It is evident that the initial problem becomes harder as adding more details to its description. In the following sections, we incrementally introduce some other issues that allow us to motivate the actual research problem addressed in this thesis. We then present our approach to solve it and show the findings of our study. Let's begin.

1.1.1 Symmetric-Key Cryptography

In the description above, we showed a system that enables two entities to communicate confidentially through an insecure channel. More formally, a *cryptosystem* consists of the encryption and decryption algorithms together with the set of all possible plaintexts, ciphertexts, and keys. It is said a cryptosystem offers a *security level of λ bits* if the best-known attack that breaks it requires a computational effort of $O(2^\lambda)$ operations. Examples of breaking a cryptosystem includes methods that systematically find the secret key or recover the plaintext from a ciphertext without knowledge of the key. The security level of a cryptosystem is strongly related to the size of the key space.

In data encryption algorithms, only those with knowledge of the key are able to encrypt and to decrypt messages. In general, there is no restriction on using different keys for encryption and decryption provided that one key can be easily obtained from the other. Algorithms that use the same key for encrypting and decrypting messages are known as *symmetric-key encryption* algorithms, which are studied by a branch of cryptography known as *symmetric-key cryptography*.

The put in practice of symmetric-key encryption raises some issues concerning to the management of keys. Issues on the operational side include a secure way to distribute keys among the participants. For example, rotation of keys must be performed every time a participant abandons a group of communication. There are also issues regarding the storage of keys. If one-to-one secret communication is needed for a group of n participants, there is required to store n^2 keys securely, which can be cumbersome if the group keeps growing. A trusted key distribution center addresses this issue for fixed-sized groups. Unfortunately, it does not scale to groups with an arbitrary number of participants.

The symmetric-key cryptography assumes that Alice and Bob *share* a secret key, which they use to interchange encrypted messages. One way to agree on the key is, for example, by having an in-person secret conversation; but, in practice, this is not always possible. For this reason, Alice and Bob require a secure way to agree on a secret key through the public communication channel. This requirement of symmetric-key cryptography is not easy to accomplish; at least, not until the discovery of a revolutionary idea that solves this problem and that introduced a new paradigm for cryptography.

1.1.2 Public-Key Cryptography

In 1976, Diffie and Hellman [87], and independently Merkle [192], shown new ideas that led to the origin of a new branch of cryptography called *public-key cryptography*. Specifically, they proposed the use of two personal keys: one of them is made publicly available, and the other key is in the private possession of its owner.

The first algorithm of public-key cryptography is *the Diffie-Hellman protocol* [87]. This protocol allows Alice and Bob to agree on a shared secret through an insecure communication channel. Thus, Alice and Bob can generate a shared key that they use as a secret key in a symmetric-key encryption algorithm. The combination of these two techniques enables confidential communication between participants located remotely.

Another breakthrough in public-key cryptography is the separation of the capabilities of the keys. In public-key data encryption, Alice uses Bob's public key to encrypt a

message sending the ciphertext through the channel. Upon receiving the ciphertext, Bob uses his private key to decrypt it and recovers the message. Unlike symmetric-key encryption, a public key is used only for encryption and a private key only for decryption. This imbalance in the capabilities of keys motivates the name of *asymmetric* cryptography when referring to the public-key cryptography [149, 190, 256].

A fundamental requirement for this asymmetry to work is that it must be computationally infeasible to obtain the private key from the public key. A way to ensure this relies on the assumption that one-way functions exist. A function is *one-way* if it is easy to compute but hard to invert. Although no proof is known for the existence of one-way functions, the hardness of some mathematical problems serves to base one-way functions. A concrete example is the integer factoring problem in number theory. It is easy to calculate the product of two prime integers, but finding the factors from their product is widely believed to be hard.

There exists a well-known cryptosystem based on the hardness of integer factoring. In 1977, Rivest, Shamir, and Adleman [232] proposed a public-key encryption algorithm known as RSA. Let p and q be two primes and $n = pq$, choose an integer $e > 1$ such that $\gcd(e, \phi(n)) = 1$, where $\phi(n) = (p - 1)(q - 1)$; then there exists a unique d such that $ed \equiv 1 \pmod{\phi(n)}$. Now, given (n, e) as the public key, it is hard to find the private key d without the knowledge of $\phi(n)$. The encryption of a message m is performed as $c = m^e \pmod{n}$, whereas decryption is $m = c^d \pmod{n}$. The RSA problem [233] is to recover the message given a ciphertext and the public key. This problem becomes easy if d or the factorization of n is known. Currently, no polynomial time algorithms for integer factoring are known; the best ones have sub-exponential time complexity on the size of the primes. The RSA assumption supports the security of public-key encryption and digital signature schemes massively used in digital communications.

ElGamal [92] proposed the use of the discrete logarithm problem in group theory for basing a one-way function. Given a group with generator g and assuming an efficiently-computable group law, it is easy to calculate group exponentiation, i.e., given g and an integer k to calculate $h = g^k$. The opposite calculation is known as the *discrete logarithm problem* (DLP), which is to find the integer k given g and arbitrary element h such that $h = g^k$. The hardness of the DLP depends on the choice of the group. For example, in the multiplicative group of the integers modulo a prime, the best-known algorithms have sub-exponential time-complexity on the size of the group order. Hence, it is desirable to use a group with a strong DLP and an efficient group law. A group with these properties is found in the theory of elliptic curves.

1.1.3 Elliptic Curve Cryptography

In 1985, Koblitz [171] and Miller [194] independently proposed a way to use elliptic curves in public-key cryptography leading to the *elliptic curve cryptography* (ECC). Specifically, they noted that elliptic curves defined over finite fields allow instantiating groups with a hard discrete logarithm problem. The main advantage of elliptic curve-based cryptosystems is the use of shorter key sizes and more efficient operations than previous cryptosystems such as RSA.

Elliptic curves have a special mathematical structure that allows instantiating a group. The points lying on an elliptic curve are the group elements, and the group law takes two points P and Q on the curve and calculates their addition $P + Q$, which is also a point on the curve. Analogously to the group exponentiation, the scalar multiplication operation multiplies a point P by an integer k , which abbreviates the repeated application of the group law on P to itself $k - 1$ times, and the resulting point is denoted as kP .

As in the general case, one can base a one-way function from this group. Given a generator P of the group and an integer k , it is easy to calculate kP . On the other hand, given two points P and Q , finding an integer k such that $Q = kP$ is believed to be a hard problem, which is known as the *elliptic curve discrete logarithm problem* (ECDLP). The best-known algorithm for solving ECDLP is the Pollard's rho algorithm [226], which has exponential time-complexity on the size of the group order.

Elliptic curve cryptography is efficient with regard to the size of keys. The complexity of the ECDLP has an advantage over the DLP on the multiplicative group of integers modulo a prime. By fixing the effort required for solving these problems, the order of the group is smaller in the case of elliptic curve groups. In practice, this translates on shorter key sizes for equivalent security levels. Table 1.1.1 lists estimates of the key sizes of some symmetric-key and public-key cryptosystems. It is clear the advantage of elliptic curve-based algorithms, specially when moving to higher security levels.

Table 1.1.1: Comparison of the bit-length of keys of cryptosystems.

Security Level (bits)	Symmetric-key Algorithm	Public-key Algorithm		
	AES	RSA	DL-based	ECC-based
128	128	3,072	3,072	256
192	192	7,680	7,680	384
256	256	15,360	15,360	512

Although elliptic curves are the object of study of number theory and algebraic geometry, their use in cryptography attracted so much attention introducing a number of applications. For example, Lenstra [179] used elliptic curves in integer factoring algorithms, a problem that Montgomery [196] also studied giving significant contributions used in contemporary algorithms. Some other applications of elliptic curves include digital signature schemes, protocols for key agreement and key encapsulation, public-key encryption, pairing-based cryptography, zero-knowledge proof systems, and multiparty computation systems. More recently, elliptic curves also found a place in the portfolio of quantum-resistant algorithms leading to the *isogeny-based cryptography*, which relies on the mathematical relations between elliptic curves to base a one-way function. In summary, elliptic curve cryptography has positioned as an efficient way for supporting public-key cryptography in practice.

1.2 Cryptographic Engineering

The fundamentals of cryptography are as relevant as their put in practice. For this reason, special attention must be given to the *cryptographic engineering*, which focuses on the application and practical aspects related to the use of cryptography and its implementation under real-world constraints [105].

The security of implementations is a chief goal to pursue. Translating mathematical formulations into machine instructions is not a straightforward task because it is prone to introduce not only errors, but also security vulnerabilities. Although security always prevails over other goals, it is not the only factor when using a system in practice as some other engineering constraints should be considered too. In certain situations such as scaling up a system or when the computational resources are limited, the computational efficiency becomes a relevant factor.

Security and efficiency are goals often in compromise. For example, suppose an application requires to perform group exponentiation by a secret exponent. One may be tempted to use the fastest algorithm, however, the algorithm could be expose a time variation that depends on the exponent; thus, revealing the secret. In this situation, the fastest algorithm is not always suitable for its direct use in cryptography.

The following sections focus on the implementation of elliptic curve cryptography. We describe some aspects regarding the security and the efficiency of software implementations. We also describe a recent initiative that motivates the need for new elliptic curves.

1.2.1 Security of Implementations

The security of a system is as strong as its weakest link. Breaking a cryptographic algorithm is by far the attacker's target, but breaking its implementation is an easier one. A special class of implementation-specific attacks are the *side-channel attacks* [173]. In this scenario, the attacker tries to learn some secret data while the computer is running. To do so, the attacker measures some physical variables of the computer environment and correlates these measurements with the data processed. The attacker's success highly depends on the behavior of the hardware and the software implementation during execution.

Side-channel attacks are a first-class concern when implementing cryptography. Although hardware implementations are more susceptible to these attacks, software implementations can also be vulnerable as has been exemplified in these works [4, 57, 178]. For this reason, protecting software implementations against side-channel attacks is mandatory to prevent the leakage of secret information. There exist several types of side-channel attacks such as timings attacks, cache-memory attacks, and power analysis attacks.

One type of side-channel attacks are the *timing attacks*. For instance, assume a program executes a time-consuming operation only when a determined bit of a secret key is set, otherwise it performs a faster operation. A timing attack leverages this situation. So an attacker that observes variations in the running time of a program can correlate a longer execution with the bits of the key that are set, thus, learning the secret information. A conventional countermeasure against timing attacks is to ensure the program follows a regular execution pattern when processing secret data, and to ensure that the

latency of operations is independent of the secret values. This programming pattern, also known as a *constant-time execution*, is key for developing secure implementations.

Other well-known techniques exist for protecting implementations against side-channel attacks. For example, secret data must not remain in memory for a long time, only when needed; and after its use, it must be wiped out. Also programs must avoid using secret data for performing bifurcations and to use secret indexes for accessing memory, these patterns in programs are often the target of cache-memory attacks. Hence, the development of cryptographic algorithms must consider all these threats to protect implementations. As the attacks become more powerful, looking for new methods and techniques to protect implementations is an active research topic.

1.2.2 Efficiency of Implementations

Efficiency is a relevant factor in the implementation of algorithms. Generally, *efficiency* is regarded as the proper utilization of the computational resources to accomplish a task without a waste of time and effort. Usually, efficiency is linked to high performance, but the vast diversity of devices and computer architectures brings a variety of metrics for determining the efficiency of implementations.

Latency is the amount of time that an operation takes to be executed. Commonly the unit of latency is the second. Nonetheless, when measuring fine-grained operations, such as machine instructions, the number of clock cycles is used instead. An advantage of using clock cycles is that it allows to make comparisons between computers that have a similar architecture but that run at different clock frequencies. A metric closely related to latency is the *throughput*, which is the number of operations performed by a unit of time, and is usually reported as operations-per-second (or instructions-per-cycle, if measuring the throughput of instructions of a program). Both metrics can be used as indicators of the performance of an implementation.

Memory footprint is another metric used for efficiency that indicates the amount of memory required to perform a task. The memory footprint of a program is often an issue in systems with memory limitations, such as in embedded devices. This is not the case in commodity computer architectures. Depending on the execution environment, memory footprint becomes an influential factor for the efficiency of an implementation.

More recently, the energy consumption of computers has become a concern regarding efficiency. This concern is not particular to the increasing use of battery-powered devices but it also applies to large data centers supporting Internet applications. On the hardware side, modern computer architectures allow the processor to run at a lower power level reducing their energy consumption. However, there is still investigation needed to make programs consume less power, and more tooling for measuring energy consumption.

1.2.3 Shifting to Modern Elliptic Curves

Since their introduction to cryptography, the research on elliptic curves has focused on finding better algorithms, optimizing curve parameters, and proposing improvements for both hardware and software implementations.

In 1999, a set of elliptic curves was recommended for their standardization [200]. Their use for performing cryptographic operations was endorsed by international organizations such as the American National Standards Institute (ANSI) [8, 9], the Institute of Electrical and Electronics Engineers (IEEE) [155], the National Institute of Standards and Technology (NIST) [204], among others. Sometime later, the standardized curves became the subject of controversy after the discovery of trapdoors in a standardized algorithm based on elliptic curves for pseudorandom number generation [249].

The research community responded with an active avalanche of proposals that improve over the standard elliptic curves. Firstly, the parameter generation process of new elliptic curves should not be biased, so all parameters must be chosen through rigid and explicit arguments. A line of research promotes the use of faster elliptic curve forms such as the Montgomery, Hessian, and (twisted) Edwards curves. Also due to the recent efforts on solving DLP over binary fields [166, 241], several proposals have a preference for defining curves over finite fields of large characteristic rather than over small characteristic fields.

Several proposals appeared intending to improve security and efficiency. The Safe-Curves [36] project summarizes some of them. The following is a non-exhaustive list of these proposals:

- Curve25519 [23]
- Brainpool project [191]
- Ed3363 [239]
- GLV/GLS binary curves [142]
- Curve448 [140]
- FourQ [77]
- Hyper-elliptic curves [45]
- Edwards curves [12]
- Curve41417 [28]
- MSR ECCLib [48]
- Isogeny-based cryptography [160]

Each of them presents new parameters for elliptic curves and prime fields showing a certain advantage over the standardized curves. Naturally, there exist trade-offs between all of these proposals when instantiating public-key cryptographic algorithms. A concern these proposals have in common is that they enforce the implementations to include some kind of side-channel protection.

1.3 Related Works

This section summarizes some research related on the study of elliptic curve cryptography. We split the discussion in generic algorithm optimizations, efficient software implementations, and the use of special hardware extensions.

1.3.1 Algorithmic Optimizations

We describe some theoretical advances in prime field and elliptic curve arithmetic that have impacted in the development of elliptic curve cryptography.

Prime Field Arithmetic

We start summarizing some contributions on faster methods for modular multiplication using prime moduli of special shape. Granger and Scott [121] proposed an efficient algorithm for calculating multiplications over \mathbb{F}_p , where $p = 2^{521} - 1$ is a Mersenne prime; thus, they reduced the number of digit multiplications to be calculated. Some other methods are shown in Crandall-Pomerance's book [79] for pseudo-Mersenne moduli, i.e., numbers of the form $p = 2^k - c$ where c is a short number.

Scott [240] revisited the implementation of the arbitrary degree Karatsuba (ADK) multiplier [271]. Scott analyzed the threshold degree at which the ADK is faster than quadratic-complexity methods. As a result, ADK could be worthwhile even for smaller sizes raising the question whether this technique can be helpful for the primes used in elliptic curve cryptography.

Elliptic Curve Arithmetic

Renes, Costello, and Batina [230] showed optimizations on complete formulas for point addition. On the one hand, implementations of elliptic curve arithmetic can now be computed following a regular execution pattern. On the other hand, the complete formulas require more field operations than the non-complete ones, specifically, more field additions and subtractions. There is a need for an alternative field representation that allows performing field additions faster, which consequently can reduce the overhead introduced by complete formulas.

1.3.2 Software Implementations of Elliptic Curves

We review recent software implementations of cryptographic algorithms using different elliptic curve models.

Standardized Elliptic Curves

A vast number of cryptographic libraries support operations over standard elliptic curves. Most of them have poor performance, nonetheless, some libraries have optimized implementations for the most used curves. For example, OpenSSL [263] has highly-optimized code for P-224 and P-256 curves derived, respectively, from contributions by Kasper [169], and Gueron and Krasnov [132]. The main optimization applied to P-256 curve targets field multiplication using Montgomery method. It remains unknown whether a different set of techniques can accelerate operations on these curves.

Binary Elliptic Curves

When running in hardware, operations over binary fields are faster than operations over prime fields. However, for software implementations the main bottleneck is the binary multiplier. Aranha et al. [15] showed how to use permutation instructions as look-up tables for calculating multiplications, squares, and square roots in extensions of binary fields. In 2010, Intel [125] introduced a carry-less multiplication instruction called PCLMULQDQ

that is useful for binary field multiplication. As a result, a positive impact on the performance of binary elliptic curves was observed by several researchers [43, 260]. The NEON instruction of the ARM architecture set also contains a carry-less multiplier, which was used by Câmara et al. [60] for performing binary field operations. The inclusion of a carry-less multiplier exemplifies the benefits associated to the use of special instructions.

Endomorphisms

Researchers have proposed the use of endomorphisms for accelerating elliptic curve arithmetic. The GLV [112] and GLS [111] are multiplication methods that use endomorphisms for calculating scalar multiplication faster. More acceleration is obtained if the curve has two efficient endomorphisms. These methods allow parallel calculation of scalar multiplication. Some previous articles show the use of endomorphisms in software implementations of prime curves [46, 97, 150, 185] and of binary curves [211, 212].

Alternative Curve Models

Joye and Quisquater [164] showed a unified formula for Hessian curves that can prevent against some side-channel attacks. In 2015, a generalization of Hessian curves was introduced [27] improving the operation counts of point addition.

A model for representing hyper-elliptic curves is through Kummer surfaces [113]. Recently, some implementations were developed targeting the 112- and 128-bit security levels [29, 45, 46]. Developing efficient and secure implementations of hyper-elliptic curves is a novel approach for delivering efficient public-key cryptography.

Another approach centers on optimized software implementations using Edwards or Montgomery curves. For example, Curve25519 [23] is a Montgomery curve used for setting speed records of the Diffie-Hellman protocol. Some other researchers followed a similar approach reporting novel implementation techniques that reduce the latency of elliptic curve operations [31, 38, 64, 139, 140].

1.3.3 Implementations using Hardware Extensions

Extensions to the computer architecture have been used to accelerate the implementation of some cryptographic algorithms. Released by Intel in 2010, a new instruction set called AES-NI [125] accelerates execution of the AES data encryption algorithm. According to Gueron's estimates [125], the performance of AES gets improved by an order of magnitude for parallel modes, and two times faster for a sequential mode such as CBC encryption.

Belonging to the AES-NI set, the `PCLMULQDQ` is a new instruction that performs carry-less multiplications. Using this instruction jointly with those for AES, one can implement the AES Galois Counter Mode (GCM), an authenticated encryption algorithm proposed by McGrew and Viega [188]. Timings measured by Gueron and Kounavis [127] showed an improvement of six times faster encryption than implementations using look-up tables. The NEON instruction set of the ARM architecture also has instructions for AES. Similar speedups were reported by Gouvêa et al. [120]. The carry-less multiplier is also used for implementing binary elliptic curves as shown in several research papers [43, 259, 260].

Another important target of optimizations is the multi-precision integer arithmetic. In [126,130,131,134], Gueron and Krasnov showed a series of optimizations for accelerating large-integer multiplications using advanced vector instructions. In [132], they also presented an optimized implementation of the P-256 curve using 128-bit vector instructions for performing prime field arithmetic and querying look-up tables.

A trend for accelerating implementations is using vector parallel processing. Together with the introduction of Curve25519 in 2006, Bernstein [23] suggested the use of floating-point vector instructions. Follow up implementations adopted this suggestion but using instead integer vector instructions [197,198], and Chou [64] extended this work to use 128-bit vector instructions. Later, in 2018, de Valence [84,85] showed speed improvements of elliptic curve arithmetic using 256-bit vector instructions. Hamburg [140] used vector instructions to process a batch of additions required by the Karatsuba multiplication in the Goldilocks curve.

1.4 Research Problem

1.4.1 Motivation

Extensive research efforts have focused on delivering public-key cryptography securely and efficiently. Cryptography based on elliptic curves provides efficient methods using keys shorter than the ones used in RSA and DLP-based algorithms. Despite elliptic curves being endorsed by international standards, a recent line of research proposes new elliptic curves to improve efficiency and preserving high-security guarantees.

With the avalanche of novel elliptic curve proposals, new challenges have appeared. Former investigations focused on optimizations for elliptic curves given in the Weierstrass form; however, there is still room for optimizing operations of alternative elliptic curve models. Algorithms for elliptic curves must likely be adapted, or otherwise reformulated considering the upsides and downsides of each model. New improvements could arise by analyzing the algorithms from theoretical, computational, and practical standpoints. Therefore, the pathway for designing cryptographic algorithms, their implementation, and their put into practice are currently in progress.

From the computational perspective, a compelling approach for improving performance is using extensions to the instruction set architecture. There exist extensions that support the *Single Instruction, Multiple Data* (SIMD) paradigm characterized in Flynn’s taxonomy [107,108] of parallel computers. In this model, a *vector instruction* encodes an operation that is executed over several data units simultaneously. Historically, SIMD processing has been shown effective in the high-performance computing area applied to graphics processing, scientific computing, and mathematical modeling, among others. In the early days, SIMD units were exclusive of large workstations and supercomputers. Nowadays, computer architectures have incorporated SIMD execution units and extensions tailored to accelerate cryptographic algorithms.

1.4.2 Problem Statement

The widespread availability of SIMD execution units in commodity computers, Internet servers, and mobile devices motivates their application to the implementation of cryptographic algorithms. Nonetheless, a few resources explain how to use SIMD units efficiently, and even fewer are dedicated to the case of elliptic curves and cryptography. Moreover, it is unclear to what extent these computational resources can help to improve efficiency.

It is interesting to know how to apply SIMD processing to implementations of elliptic curve cryptography. Especially to the algorithms derived from the recent proposals of elliptic curves and making use of the most advanced vector instructions and other extensions found in contemporary computers. For this reason, it is imperative to investigate how to design new algorithms and data structures (or adapt the existing ones) so that implementations take full advantage of SIMD processing.

1.5 Aims and Scope

Thesis Statement We claim that the execution of algorithms for elliptic curve cryptography can be accelerated through a combination of algorithmic optimizations, implementation techniques, and the use of SIMD processing and other hardware extensions.

1.5.1 Aims

To support this assumption, we investigate algorithmic optimizations and look for implementation techniques for elliptic curve algorithms emphasizing the application of SIMD parallel processing.

An objective of our study to close the gap between theory and practice. For instance, in addition to proposing parallel algorithms, we also cover their implementation in software. We highlight some issues arisen during development and propose some solutions for them. The design of our proposed algorithms considers the capabilities and limitations of the computer architectures studied.

Our research aims to enlighten a pathway for applying SIMD efficiently. Current computer architectures support hundreds of SIMD instructions, which are gradually increasing in the upcoming computer architectures. Part of this research is to give guidance on the use of SIMD instructions, identify some issues and their limitations, and show how to apply them to the algorithms used in elliptic curve cryptography.

1.5.2 Scope

Since the study of elliptic curve cryptography and its implementation involve many dimensions and other engineering aspects, we delimit the scope of our study as follows.

Elliptic Curves

Due to the recent proposals for new elliptic curves, we target the Weierstrass, Montgomery, and (twisted) Edwards curves defined over fields of large prime characteristic.

Cryptographic Algorithms

We consider algorithms of public-key cryptography. We study the following Diffie-Hellman protocols: the ECDH [9] protocol with the P-384 curve, the X25519 [23] and X448 [264] protocols, and the SIDH-751 protocol.

We study the following digital signature schemes: ECDSA [8] with the P-384 curve, Ed25519 [31] and Ed448 [162], qDSA with Curve25519 [231], XMSS and XMSS^{MT} [58,153] schemes. Note that XMSS and XMSS^{MT} are algorithms believed to be quantum-resistant. Digital signatures often use a cryptographic hash function, so we study the SHA-256 [203] hash function.

Hardware

We target the x64 (also known as x86-64) computer architecture [175], which is commonly found in end-user devices such as desktop and laptop computers, as well as in Internet servers and data centers. This architecture includes a bank of 64-bit registers and supports a general-purpose instruction set.

We looked for micro-architectures that implement the x64 architecture. In particular, we looked for those that support the SIMD instruction sets such as SSE, AVX, and AVX2; and extensions for cryptographic applications such as the AESNI, and PCLMULQDQ instructions. We have access to the Haswell, Skylake, SkylakeX, and Kaby Lake micro-architectures from Intel, and to the Zen micro-architecture from AMD. Zen additionally supports the SHA-NI instruction set. In Table 1.5.1, we list the specifications of the processors used in this research.

Table 1.5.1: Technical specifications of the computers used in this research.

Micro-architecture	Processor	Frequency	Instruction Sets
Haswell	Core i7-4770	3.4 GHz	SSE, AVX, AVX2.
Skylake	Core i7-6700K	4.0 GHz	SSE, AVX, AVX2, BMI2, ADX.
Kaby Lake	Core i5-7400	3.0 GHz	SSE, AVX, AVX2, BMI2, ADX.
SkylakeX	Core i7-7820X	3.6 GHz	SSE, AVX, AVX2, BMI2, ADX, AVX-512.
Zen	Ryzen 7 1800X	2.4 GHz	SSE, AVX, AVX2, BMI2, ADX, SHA-NI.

Software

Our software implementations are written in the C programming language. We use special declarations of C functions, called *intrinsics*, that allow accessing to the SIMD and other hardware extensions without writing assembly code. A comprehensive list of intrinsics is provided by Intel at [75]. For efficiency reasons, we sometimes resort to inline assembly language into the C code. We verify source codes are successfully compiled by the GNU C Compiler (gcc) [115], the Intel C Compiler (icc) [74], and the Clang (clang) compiler [262].

Measurement Methodology

We follow the benchmark methodology recommended by Intel in Paoloni’s paper [219]. We use clock cycles instead of seconds as the unit of time, this allows making time comparisons between processors that operate at different clock frequencies. The timings reported are obtained as the median of the time taken to compute a batch of operations of the same type. The comparison tables report codes compiled with the Clang compiler version 5.0.2 using the following compilation flags: “`-O3 -march=native -mtune=native`”.

Several factors interfere with the accuracy and reproducibility of timings. Time measurements are sensitive to the computer (micro-)architecture, the processor’s frequency, and the processor usage. Also, varying the compiler, the compiler version, or the compilation flags results in different binary programs, which could exhibit different performance.

To get accurate measurements, we disable any frequency scaling technology, such as the Intel Turbo Boost or Hyper-Threading, for reducing interference during the execution of benchmarks. To make fair comparisons, we measure publicly available code under the same measurement environment we used to measure our codes. We encourage those wanting to reproduce our experiments to consider all these factors before making comparisons.

1.6 Contributions

Our contributions are a conjunction of several layers of improvements involving theoretical optimizations as well as practical implementation techniques and running software.

Our initial target for optimizations is the prime field arithmetic. We show data structures and representation of numbers that are suitable for processing scalar and vector instructions efficiently. We develop optimized SIMD implementations of prime field operations whose prime modulus is given in a special form. We target prime fields that are suggested in recent proposals of new elliptic curves distinguishing four families of primes.

We propose optimizations for the arithmetic of elliptic curves. For Montgomery curves, we show a new three-point ladder algorithm, an optimized tripling formula, and a parallel implementation of the Montgomery ladder algorithm. For Edwards curves, we show parallel algorithms for point additions and scalar multiplications. For Weierstrass curves, we show parallel algorithms for the complete formulas for point additions. For completeness, we implemented these parallel algorithms applying the SIMD prime field operations.

All of these optimizations are of general interest, and we show their immediate applicability on several elliptic curve cryptography algorithms such as X25519, EdDSA, and others. Some of these algorithms are in track for standardization so they will be used massively for securing Internet communications in the TLS, SSH, and VPN protocols. We also contribute with public-available software libraries that are optimized for SIMD processing, and performance benchmarks provide evidence of their superiority.

1.6.1 Publications

Some of our contributions were published in peer-reviewed academic venues such as scientific journals and conference proceedings of cryptography.

Journal Articles

A Faster Software Implementation of the Supersingular Isogeny Diffie-Hellman Key Exchange Protocol by Armando Faz Hernández, Julio López, Eduardo Ochoa-Jiménez and Francisco Rodríguez-Henríquez. IEEE Transactions on Computers, Nov 2018.

doi: 10.1109/TC.2017.2771535 .

High-performance Implementation of Elliptic Curve Cryptography Using Vector Instructions by Armando Faz Hernández, Julio López, and Ricardo Dahab. ACM Transactions on Mathematical Software (TOMS), Jul 2019. doi: 10.1145/3309759 .

Papers in Conference Proceedings

On Software Implementation of Arithmetic Operations on Prime Fields using AVX2 by Armando Faz Hernández and Julio López. XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, Nov 2014. doi: 10.5753/sbseg.2014.20148 .

Fast Implementation of Curve25519 using AVX2 by Armando Faz Hernández and Julio López. Progress in Cryptology – LATINCRYPT, Sep 2015.

doi: 10.1007/978-3-31922174-8_18 .

Speeding up Elliptic Curve Cryptography on the P-384 Curve by Armando Faz-Hernández, and Julio López. XVI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, Nov 2016. doi: 10.5753/sbseg.2016.19306 .

How to (Pre-)Compute a Ladder by Thomaz Oliveira, Julio López, Hüseyin Hişil, Armando Faz Hernández, and Francisco Rodríguez-Henríquez. Selected Areas in Cryptography, Dec 2017. doi: 10.1007/978-3-319-72565-9_9 .

A Secure and Efficient Implementation of the Quotient Digital Signature Algorithm (qDSA) by Armando Faz Hernández, Hayato Fujii, Diego F. Aranha, and Julio López. Security, Privacy, and Applied Cryptography Engineering (SPACE 2017), Nov 2017.

doi: 10.1007/978-3-319-71501-8_10 .

SoK: A Performance Evaluation of Cryptographic Instruction Sets on Modern Architectures by Armando Faz Hernández, Julio López, and Ana K. D. S. de Oliveira. APKC'18 Proceedings of the 5th ACM on ASIA Public-Key Cryptography Workshop. Jun 2018.

doi: 10.1145/3197507.3197511 .

Generation of Elliptic Curve Points in Tandem by Armando Faz Hernández, and Julio López. XX Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais. Oct 2020. doi: 10.5753/sbseg.2020.19230 .

Book Chapter

Implementação Eficiente e Segura de Algoritmos Criptográficos by Armando Faz Hernández, Roberto Cabral, Diego F. Aranha, and Julio López. XV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais: minicursos. Nov 2015.

ISBN: 978-85-7669-304-8. doi: 10.5753/sbc.9004.8.3 .

1.6.2 Software Libraries

We develop a set of software libraries that show implementation techniques and optimizations of several cryptographic algorithms. Our libraries are available at public repositories released under permissive software licenses, and are also available at an institutional repository: https://gitlab.ic.unicamp.br/ra142685/phd_libs/.

fld-ecc-vec

An optimized implementation of the Ed25519 and Ed448 signature schemes, the X25519 and X448 Diffie-Hellman protocols, and hash to curve functions using AVX2 vector instructions.

<https://github.com/armfazh/fld-ecc-vec>

flor-sidh-x64

An implementation of the SIDH-751 Diffie-Hellman protocol accelerated with BMI2 and ADX instructions.

<https://github.com/armfazh/flor-sidh-x64>

rfc7748_precomputed

An optimized 64-bit implementation of the X25519 and X448 Diffie-Hellman protocols accelerated with BMI2 and ADX instructions.

https://github.com/armfazh/rfc7748_precomputed

nistp384_avx2

A SIMD implementation of the complete addition formulas for the P-384 Weierstrass curve.

https://github.com/armfazh/nistp384_avx2

qdsa_space2017

An optimized 64-bit implementation of the qDSA signature scheme.

<https://github.com/armfazh/qdsa-space17>

flo-shani-aesni

Optimized implementations of SHA-256 using SHANI instructions, and AES and AEGIS using AESNI instructions.

<https://github.com/armfazh/flo-shani-aesni>

Our x64 implementation of X25519 was included in the implementation of the Wireguard protocol [88]. This protocol offers a VPN-like secure communication between remote machines and was recently included in the kernel of Linux. Our x64 implementation of X25519 was formally verified in EverCrypt project by Protzenko et al. [227] proving the correctness of the code. Our implementation techniques for SIDH were adopted by the SIKE [159] project submitted to the Post-Quantum Cryptography Standardization project by NIST [206].

1.7 Outline

In the next chapters we present our findings in a bottom-up manner. We start by reviewing concepts of computer architectures and their hardware extensions. Then we deep dive into the arithmetic of prime fields and its implementation. Building on top of it, we describe the arithmetic of elliptic curves and present some optimizations. After that, we describe the implementation of some cryptographic algorithms based on elliptic curves. Finally, we summarize our conclusions and future work.

In Chapter 2, we give a panorama of current and upcoming hardware optimizations available in computer architectures. We cover some basic concepts about parallel processing and their support in recent computer architectures. Then, we examine the SIMD processing showing several vector instruction sets and highlight some extensions of special interest for cryptography.

In Chapter 3, we review algorithms for performing prime field arithmetic. We start by reviewing data structures for representing large integer numbers and describe how to calculate arithmetic operations with such representations. We introduce a key concept that defines the parallel execution of field operations suitable for a SIMD processing. Then, we detail the implementation of operations over some prime fields of interest covering different families of prime moduli. For each study case, we show a performance benchmark and comparisons.

In Chapter 4, we center our attention to the arithmetic of elliptic curves. We review fundamental concepts of elliptic curves and well-known algorithms for scalar multiplication. Then, we show some algorithmic improvements for Montgomery curves, and review the point addition formulas of the Weierstrass, Montgomery, and twisted Edwards curves. For each curve model, we propose parallel algorithms for point operations and show their implementation using SIMD processing.

In Chapter 5, we give details about the implementation of a selection of cryptographic algorithms and protocols. For each one, we describe techniques for its efficient implementation showing specific optimizations. We present results of performance benchmarks and made comparisons against state-of-the-art implementations.

In Chapter 6, we present the conclusions of this research study. We give a brief retrospective and summarize our contributions. We end this chapter suggesting some paths for future investigation.

In Appendix A, we provide detailed bibliographic information of the publications produced as part of this thesis.

We believe that our research can help broaden the knowledge base of elliptic curve cryptography and the engineering aspects around its implementation.

Chapter 2

Modern Computer Architectures

The design of newer computer architectures often includes optimizations for speeding up the execution of programs. The goal of some optimizations is to execute more instructions per unit of time, and the most recent ones seek to process more data at a time relying on parallel computing. Although most optimizations benefit to generic programs, others target domain-specific applications.

In this chapter, we give a summary of some hardware optimizations found in contemporary computer architectures. Our description is centered on extensions to the instruction set architecture, in particular, those that support parallel computing as well as the extensions tailored for cryptography.

2.1 Optimizations in Computer Architectures

Around the middle of the 1980s decade, the emergence of the Reduced Instruction Set Computer (RISC) was a breakthrough in the design of computer architectures. The RISC design proposes the use of a small set of instructions, this results in short instruction formats leading to a higher hardware utilization [221]. The RISC architecture is more renowned due to the introduction of the pipeline execution, a technique used for increasing the instruction-level parallelism of programs.

RISC-based processors achieve higher throughput by issuing several instructions per clock cycle. The so-called *superscalar processors* contain multiple units to execute instructions, and these instructions can be issued to the pipeline out of the program order through a dynamic pipeline scheduling. Unlike previous processors that execute program's instructions sequentially (in program order), executing them in out-of-order increases the chances to execute instructions in parallel.

Another hardware optimization commonly found in the recent processors is the *simultaneous multi-threading*, which is a high-level technique to share a physical core among multiple logical cores (or execution threads). With this technique, the pipeline interleaves the execution of instructions that belong to different logic cores. This is possible because it is unlikely that data dependencies occur between instructions of different logical cores.

The hardware optimizations mentioned above are, in some sense, unnoticed by the programmer; i.e., a few or no actions are required by the programmer to enable (or disable)

these optimizations. On the other hand, other hardware resources, such as vectorization or multi-core parallelization, require the intervention of the programmer mainly to distribute a workload among the parallel units. Usually, the assignment of tasks to execution units is performed through special functions or keywords added to the source code. In some cases, this is not even required since advanced compilers are able to recognize common execution patterns that are suitable for their execution in parallel.

Other hardware optimizations are in the form of new instructions. The latest micro-architectures have included specialized instructions targeting a wide range of applications, such as text processing, bit manipulation, cryptography, and neural networks processing.

In the following sections we give more details about hardware optimizations available in contemporary processors.

2.1.1 Pipeline Execution

Some computer architectures process instructions using an execution path or a *pipeline* consisting of five stages: fetch, decode, execute, access to memory, and write-back the result [146]. An instruction completes its execution once it has passed through each of these stages sequentially. Assuming each stage takes one cycle, each instruction would take five cycles to complete.

The main idea of a *pipeline execution* is to process several stages of different instructions at once. To do so, the instructions get overlapped to keep the hardware units always busy executing stages of different instructions. When an instruction finishes one stage and goes to the next stage, the execution unit becomes available for the next instruction. Thus, more than one stage is performed at once as shown in Figure 2.1.1. Instructions still take five cycles to complete, but at every cycle one instruction is completed. Although the latency of each instruction remains unaltered, the amount of instructions processed per unit of time, also known as the *throughput* of instructions, increases.

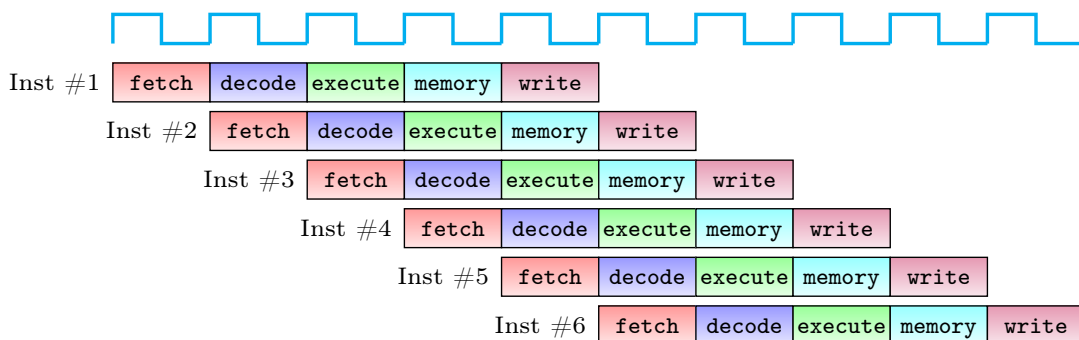


Figure 2.1.1: Instruction execution in a five-stage pipeline.

Pipeline hazards prevent the efficient use of pipelining. The source of these hazards is mainly due to data dependencies between instructions (data hazards), but also they can be originated when execution units are busy (structural hazards). In both types of hazards, the execution of instructions is delayed, and the pipeline stalls until the dependencies are resolved. One way to avoid pipeline stalls is forwarding the result value of an instruction

to the instruction that depends on it; therefore, the latter instruction can use the value immediately without waiting for the pipeline to write the result in the register bank.

A data dependency analysis is needed for benefiting from a pipeline execution. The analysis reveals which instructions of the program can be executed in parallel. To do so, it often requires reordering the instructions of the program. Compilers usually produce an optimized instruction scheduling based on the data dependency analysis of the source code. However, sometimes the compiler cannot identify all the opportunities for optimizations, so the programmer must reorder the instructions manually.

Pipeline execution is a powerful technique for executing programs faster. This hardware optimization was implemented in architectures for supercomputers such as the Z3 [235], Stretch [42], and Illiac II [54]. The pipeline length has grown as long as 31-stages in the Prescott [70] architecture. More recent architectures, like Haswell, operate with pipelines of around 14 to 16 stages [248].

2.1.2 Superscalar Processors

A *superscalar processor* refers to the processors that can issue more than one instruction per clock cycle [146]. Also known as *multiple-issue* architectures, they enhanced the RISC design by replicating execution units, which execute several instructions at once. The availability of several units raises the following question: how to handle more than one instruction per clock cycle? A solution is to extend the pipeline with a set of reservation stations, a set of execution units, and a commit unit. These additional hardware units work together to dynamically scheduling instructions.

The multiple-issue mechanism works as follows. Once an instruction is decoded, it enters into a reservation station waiting to be assigned to an execution unit. Since each execution unit can only perform a subset of all possible instructions, different units have a different amount of instructions to process. When an execution unit becomes available, the instruction on the top of the reservation station is sent for its execution. If another instruction of the same type is waiting in the reservation station, the instruction can only be sent to the execution unit after t clock cycles, where t represents the reciprocal throughput of this instruction. After its completion, the computed value is stored in a buffer, which is part of the commit unit. Finally, the commit unit writes back the values in the buffer to either the registers or the memory.

Note that the order in which the instructions are executed could be different of the program's order. Suppose a program in which a long-latency instruction is followed by several short-latency instructions that do not depend on it. Clearly all these instructions can be sent to other execution units, and since they have a short latency, they will be completed before the long-latency instruction ends storing their results into a buffer. Note that these values cannot be written back until the result of the long-latency instruction is computed. Once the long-latency instruction finishes, the results are committed in the program's order. Hence, in an *out-of-order execution* instructions are executed in out of the program's order but the results are committed to the registers in the program's order.

The execution units vary depending on the micro-architecture. In some Intel architectures, there are *ports* that redirect instructions from the reservation station to an execution

unit. As shown in Figure 2.1.2, the Sandy Bridge micro-architecture has six ports: the ports 0, 1, and 5 handle most of the arithmetic instructions; the ports 2, 3, and 4 manage memory-accessing instructions; and the port 5 is used for branching instructions.

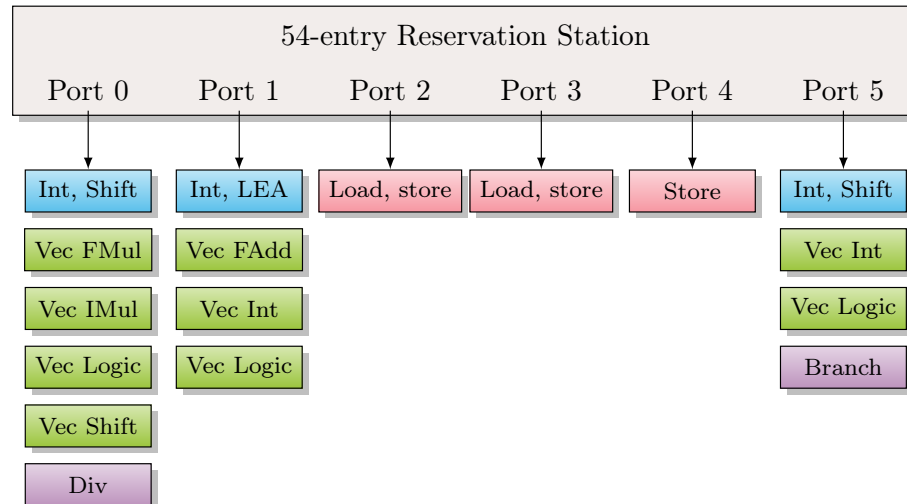


Figure 2.1.2: Execution engine of the Sandy Bridge micro-architecture.

The Haswell micro-architecture, successor of Sandy Bridge, increased the number of ports to eight to achieve better performance. As shown in Figure 2.1.3, Haswell has two ports that handle branching instructions (ports 0 and 6), and one extra that calculates addresses of memory-storing instructions (port 7). Another difference is that Haswell reduced the type of instructions performed by the port 5, which handles arithmetic instructions. The Intel micro-architectures released after Haswell, such as Skylake and Kaby Lake, have a similar execution engine.

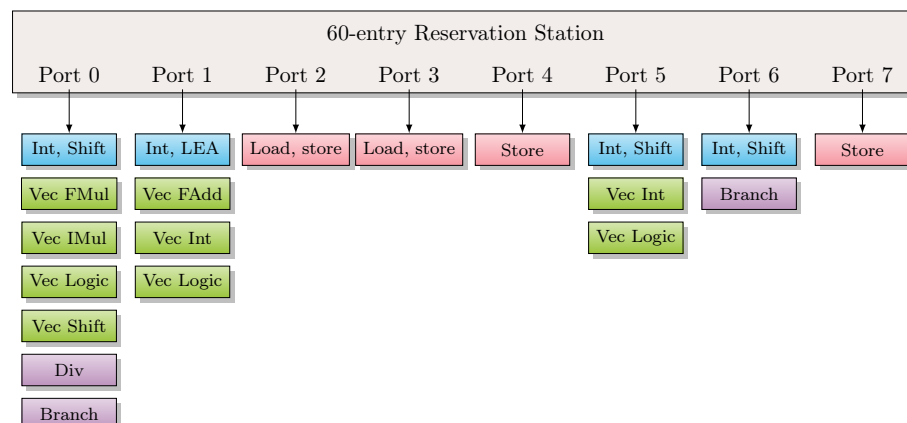


Figure 2.1.3: Execution engine of the Haswell micro-architecture.

2.1.3 Simultaneous Multi-Threading

Multiprogramming is a standard technique used to run several programs concurrently using a single processor [146]. To do so, several programs or threads (light-weight programs) have assigned chunks of processor's time for their execution.

An issue that arises in multiprogramming is the following. While a program is running, it could happen that some execution units of the pipeline stay idle, which can be caused due to a low instruction-level parallelism in the program, or because the program only uses a specific subset of instructions. For example, when a program executes many logic arithmetic instructions but only a few integer instructions, so the program underutilizes the other execution units available.

There is a technique that avoids this issue. The Simultaneous Multi-Threading (SMT) shares the execution units of a physical core among different threads or programs. Thus, the pipeline dynamically schedules instructions that belong to different programs; these instructions have attached a thread identifier that allows committing the result to the correspondent thread. A program running several threads in the same physical core gets an improvement in the execution time on top of using the multiprogramming technique. On the other hand, for single-threaded programs, the use of SMT does not have a significant impact on the running time. Therefore, SMT is intended to increase the thread-level parallelism (TLP) of programs running concurrently.

The SMT has been incorporated in some contemporary architectures. The Intel architectures support SMT through Intel Hyper-Threading, which shares the resources of a physical processor between two logic processors. The Intel Xeon Phi coprocessor extends this support to handle four threads per physical core. Some AMD architectures have a technology known as SMT, which can be found in the Zen micro-architecture.

2.1.4 Parallel Computing

In 1966, Michael Flynn [107, 108] characterized the computer architectures based on how parallelism behaves in the face of instruction and data streams. Thus, every computing model can be classified in the following categories:

SISD Single instruction stream, single data stream.

SIMD Single instruction stream, multiple data streams.

MISD Multiple instruction streams, single data stream.

MIMD Multiple instruction streams, multiple data streams.

Computers from the early days were able to execute one instruction at a time operating over a single unit of data matching the SISD processing. However, the capabilities of computers have improved. Currently, modern computers overlap these categories since they implement multiple features in the same architecture.

The SIMD Paradigm

The SIMD processing one instruction encodes an operation that is executed over a set of data. Implementing SIMD in a computer architecture requires of new instructions and registers that contain several units of data. For this reason, implementations of the SIMD model commonly used the term *vector processing*.

Actual implementations of SIMD firstly appeared in supercomputers. Notable examples around the 1970s decade are the Illiac-IV computer from the Illinois University [53], the CDC Star processor [228], and the ASC processor from Texas Instruments [270]. These computer architectures share similar characteristics such as large registers (each one storing more than 64 floating-point words), multiple functional units, and a configurable set of registers. A different implementation was the Berkeley's Intelligent RAM (IRAM) project [220], which combines vector processing units with the DRAM memory. The Cray supercomputer [237] was another implementation of SIMD processing focusing mostly on high-performance computing and scientific applications.

Vectorization is the process of transforming a program to use vector instructions. In this process, the programmer identifies fragments of the program that are suitable to run in parallel and determines how to pack data into vector registers. Depending on the complexity of the program, vectorization could be as easy as a one-to-one replacement of instructions, or it could be more complex requiring of a detailed analysis of the flow of data to determine the best partition and distribution of the operations. Unlike the hardware optimizations presented in the previous sections, the use of vector instructions requires the programmer to take action.

The MIMD Paradigm

This paradigm refers to the computation of a task across several processing units or cores [146]. In the MIMD processing, the communication between cores is performed using shared memory or through a message passing interface. Unfortunately, both of them introduce notorious performance overheads. In addition, in some platforms, the time taken for accessing to memory is non-uniform; i.e., a core unit observes a different latency for accessing to values stored in different memory locations. The latency mainly depends on the topology of the parallel units. Hence, the time taken in communications and in memory accessing are key factors that must be considered in this model.

Implementing programs following the MIMD model often involves the following steps. First, identify the part of the program that can be parallelized and divide such a workload into independent slices. Then, each slice is assigned to one or more core units for its execution. Finally, partial results of each core unit are collected by a single unit that produces the final result.

The OpenMP library [80] is a tool that helps on the implementation of MIMD programs. The programmer annotates the source code in those parts that can run in parallel. For example, under certain conditions, the iterations of a time-consuming for-loop can be mapped to a set of independent tasks, which are executed by the cores simultaneously. At compilation time, the compiler is hinted by these annotations to produce a parallel program with several threads; each thread contains a fraction of the total workload to be computed. At running time, the parallel program determines the number of available processing cores and distributes the threads to the processing units.

Nowadays computers are equipped with multiple processors. End-user processors have two or four cores, and processors used in Internet servers have more than twelve cores usually. The number of operations performed by a multi-core processor is significantly

large, since each physical core supports pipelining, simultaneous multi-threading, and vector instructions.

Another computer architecture that overlaps both the SIMD and MIMD categories is the Graphics Processing Unit (GPU). Initially intended to accelerate graphics and video processing, more recently the GPU has been extended to support a wider number of applications. This originated to the creation of general-purpose GPU (GPGPU) architectures. In cryptography, GPGPUs are applied to solve some instances of hard problems that are easily parallelizable, for example, the problem of integer factorization [51, 193].

2.1.5 Comparison of Optimizations

In Table 2.1.4, we compare the hardware optimizations presented the previous sections.

The first and second columns indicate whether the programmer or the compiler needs to take some action to benefit from the hardware optimization. Most of the optimizations target generic programs, and because of that, no intervention by the programmer or the compiler is needed. On the other hand, vectorization and multi-core execution require the programmer explicitly indicates which parts of the program run in parallel. Although some advanced compilers can automatically identify parts of the code to be vectorized [74, 115], fine-tuning optimizations must be done by the programmer to get faster execution.

The third column of Table 2.1.4 shows that most of the hardware optimizations are available per physical core. This is clear because most of the programs are written to be run by a single processor. This fact becomes more relevant on cloud computing scenarios, in which a multi-core computer is shared in such a way that its physical cores are assigned to virtual machines. Thus, although each virtual machine runs on a single physical core, it still possible to execute code in parallel using vector instructions.

The last column lists the type of parallelism exploited by each hardware optimization. For generic programs, exploiting the instruction-level parallelism (ILP) is more beneficial since most programmers write code with a low sense of instruction optimizations. On the other hand, vectorization and multi-core execution are focused on accelerating the execution of programs that have a certain degree of data-level parallelism (DLP). Finally, in the cases where many different tasks must be performed, then the thread-level parallelism (TLP) is exploited through simultaneous multi-threading or multi-core execution.

Table 2.1.4: Comparison of hardware optimizations.

	Programmer Intervention	Compiler Intervention	Available in Single/Multiple Processors	Type of Parallelism
Pipeline Execution	No	No	Single	ILP
Superscalar	No	No	Single	ILP
SIMD processing	Yes	Yes	Single	DLP
Multi-Threading	No	No	Single	TLP
Multi-Core	Yes	Yes	Multiple	DLP, TLP

2.2 SIMD Vector Units

A *vector unit* is an extension of the computer architecture that supports the SIMD parallel computing paradigm. Vector units are composed of large registers, called as *vector registers*, that store a fixed number of words; and extensions to the instruction set architecture, also known as *vector instructions*, that operate on vector registers. To distinguish from vector instructions, we refer to the non-vector instructions as *scalar* or native instructions. Following the SIMD paradigm, vector instructions perform the same operation on every word stored in a vector register, as shown in Figure 2.2.1.

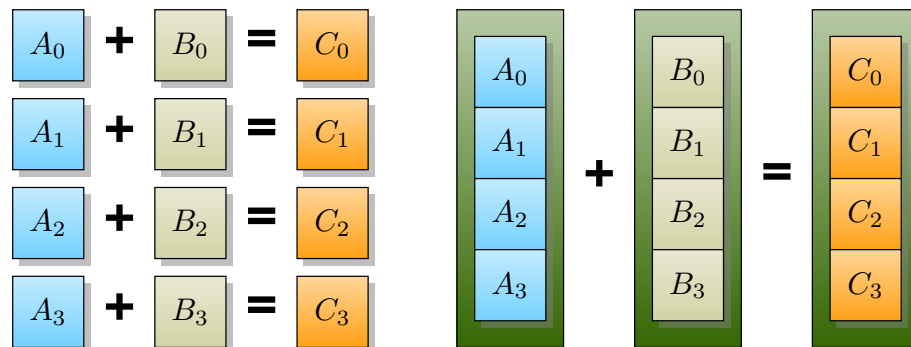


Figure 2.2.1: Scalar vs vector processing. Four additions are performed by either four scalar instructions or by only one vector instruction.

Vector registers are fixed-size units of memory that store several words. Let n be the size in bits of a vector register (usually, $n \geq 64$ and is a power of two), a vector register can be interpreted as a single register of n bits, or as an array of either $\frac{n}{8}$ bytes, $\frac{n}{16}$ words, $\frac{n}{32}$ double-words (or single-precision floating-point numbers), or $\frac{n}{64}$ quad-words (or double-precision floating-point numbers). The exact interpretation is encoded in the vector instruction, which determines the operation to be performed, the number of words in the vector register, as well as the datatype of each word.

One advantage of vector instructions is that they reduce the memory footprint of programs. The binary encoding of one vector instruction shortens the encoding of several scalar instructions. The processor decodes only one vector instruction and avoids decoding the same scalar instruction many times. Having less instructions to decode releases pressure on the instruction decoders.

Vector instructions fetch more data from memory, which tends to be a costly operation. On architectures with a cache memory, every time that a program requests a word from memory, a large portion of data is transferred from memory to a line in the cache; and then, the requested word is moved from the cache to the processor. A reason for doing this is due to the *spatial locality* of data. It is likely that subsequent memory accesses refer to locations close to the previous reference; thus, the requested words are already in the cache reducing the latency of accessing to memory.

Cache memory helps when accessing units of data, but vector units require fetching vectors of data. Note that filling a vector register with data from memory will consume a larger part of the cache line. So subsequent references to memory will require more cache lines. In this situation, spatial locality does not help as much as with scalar access to

memory, unless the architecture has a large bank of cache memory. This explains why accessing memory is a time consuming operation when using a vector unit.

Availability

The SIMD paradigm landed on conventional computers around the middle of the 1990s decade. At that time, the design of computer architectures focused more on extracting parallelism at data level rather than at the instruction level. For this reason, several extensions to the instruction set architecture appeared for supporting the emerging multimedia applications, which heavily focused on sound, image, and video processing [86]. Motivated by these end-user applications, many SIMD instructions have been increasingly added to processors since then. Nowadays the applicability of SIMD instructions has been extended to other domains such as graphics processing, data encryption, advanced image processing, neural networks, and others.

Vector units are present in several computer architectures. For instance, the ARM architecture contains an extension called ASIMD vector unit formerly known as the NEON vector unit [17]. Similarly, the Power ISA architecture supports the AltiVec [110] vector instructions. The x86-64 architecture was extended with a vast number of vector instructions, known as the SSE and AVX instruction sets. Although these extensions are not compatible to each other, they share an equivalent functionality for performing basic arithmetic and logic operations.

2.2.1 Vector Instructions of the x86-64 Architecture

We describe instruction sets of the x86-64 architecture following a chronological order and grouping instructions that operate over equal-sized registers.

The 64-bit Vector Instructions

During the 1990s decade, most of the processors support the x86 architecture and operate over words of 32 bits. In 1997, Intel introduced the MultiMedia eXtensions (MMX) [222, 223], which is a set of 57 vector instructions that operate over a bank of eight 64-bit registers named MM0-MM7. The goal of these larger registers was not to execute 64-bit operations, but performing two 32-bit operations simultaneously.

Later in 1998, AMD released the 3DNow! technology, which enables the support of SIMD processing for integer and floating-point operations [209]. The first implementation of this technology appeared in the AMD-K6 processor and consisted of 21 instructions. Although more extensions appeared, 3DNow! has been deprecated [7].

MMX and 3DNow! share the bank of registers with the floating-point unit (FPU). The processor is allowed to use either the FPU or the MMX/3DNow! unit at a given time, and programs must emit an instruction that toggles between the units. Unfortunately, switching between the units incurred on performance overheads.

The 128-bit Vector Instructions

In 1999, Intel released the *Internet Streaming SIMD Extensions* (SSE) [261] introducing several changes to the micro-architecture. It was included a new bank of eight 128-bit vector registers, called XMM0-XMM7. This bank is independent from the bank used by the FPU unit and has a larger storage capacity. The SSE set contains 70 vector instructions that process up to four 32-bit operations simultaneously.

Since most of the SSE instructions are for single-precision floating-point arithmetic, a second version of SSE, called SSE2, included 144 new vector instructions covering integer arithmetic, and double-precision floating-point arithmetic.

In 2000, AMD launched the AMD64 architecture that allows to perform operations over 64-bit words. This architecture is also known as x64, x86_64, or as a 64-bit architecture. Since its release, most of the processors found in current computers are based on the x64 architecture. In addition, AMD64 also supports the SSE vector unit and increased from eight to sixteen the number of vector registers (XMM0-XMM15).

Although SSE2 instructions can execute more operations than the MMX instructions, in practice the performance observed was roughly the same. The main reason of this poor performance was because the cost of accessing to misaligned data incurs in significant timing penalties. To solve this issue, SSE3 was released in 2004 with instructions that load/store data from unaligned memory addresses. Also, SSE3 included 13 new vector instructions that operate horizontally in the vector register, i.e., they operate using the words within a vector register, as opposed to the previous instructions, which operate (vertically) with words from two different vector registers.

The Supplemental SSE3 instruction set (SSSE3) contains one relevant instruction called PSHUFB. The documentation of this instruction suggests using this instruction to permute bytes within a vector register. However, this instruction can be used to compute other operations. For example, PSHUFB can simultaneously perform sixteen queries on a 16-entry table of bytes, i.e., given two 128-bit vector registers, A and B , obtains $C \leftarrow \text{PSHUFB}(A, B)$ as follows: $c_i \leftarrow a_i[b_i]$ for $i = 0$ to 15 and $0 \leq b_i < 16$. Another interpretation given to this instruction is the calculation of any boolean function $f: \{0, 1\}^4 \rightarrow \{0, 1\}^8$. This versatility makes PSHUFB suitable for many applications.

A shift in the vector instruction design was observed to support more diverse applications. This strategy was even more evident in the SSE4 instruction set. The SSE4 set consists of 54 instructions, the first 47 instructions are known as SSE4.1, and the last seven instructions are known as SSE4.2. Unlike previous instruction sets, SSE4 has not targeted graphics applications, but string processing. Thus, operations such as string comparisons and population count were benefited from special vector instructions. Additionally, SSE4 included an instruction to calculate the CRC-32 error detection code [247]. This trend continued with the inclusion of instructions for cryptography, such as the AES-NI set presented in Section 2.3.

In 2007, AMD released the specification of SSE5 [6], which integrates instructions that perform multiplications combined with additions; for example, $D \leftarrow A \times B + C$; these operations are known as Fused-Multiply and Accumulate (FMA). However, two years later, AMD split the SSE5 instruction set between two sets: the FMA4 instructions

and the eXtended Operations (XOP). The novelty of the XOP set is that includes vector instructions to perform shift rotations, conditional moves, and permutation between words of registers. Both sets were supported since the release of the Bulldozer AMD's micro-architecture, but they were discontinued on the Zen micro-architecture.

The 256-bit Vector Instructions

The next big step to improve the SIMD parallel execution was given in 2008. The main objective of the new vector instructions was to extend the size of registers initially to 256 bits and in future processor generations to 512 bits. Consequently, processors must support lots of new instructions and for this reason, a new instruction coding scheme was introduced, which can handle instructions with up to four operands. These modifications to the instruction set architecture are the Advanced Vector eXtensions (AVX).

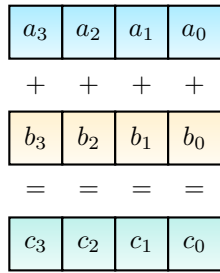
The first implementation of AVX appeared in 2011 with the release of the Sandy Bridge micro-architecture of Intel and the Bulldozer micro-architecture of AMD. Although AVX extended the size of vector registers to 256 bits, the AVX instruction set was not complete since it does not support most of the integer arithmetic operations. For this reason, the Advanced Vector eXtensions 2 (AVX2) were released in 2013, which promoted most of the integer arithmetic operations from 128 bits to 256 bits. Haswell and Zen were the first micro-architectures of, respectively, Intel and AMD that support AVX2.

Extending a 128-bit vectorized code to use 256-bit instructions shows in the general setting an increase in the performance. This increment is in part because the latency of the AVX2 instructions is almost the same as the latency of SSE instructions, which allows increasing the parallelism degree almost linearly. However, Zen exhibits a downgrade on performance when executing AVX2 instructions. This occurs due to the micro-architectural design of Zen, which emulates a 256-bit vector instruction by splitting the workload into two parts, and each part is executed by a 128-bit vector unit sequentially what causes that the latency of AVX2 instructions is twice slower than the latency of SSE instructions. Hence, the performance of an AVX2 program is severely penalized on Zen.

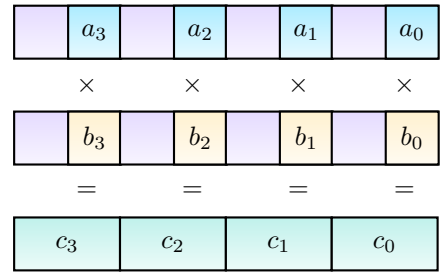
The 512-bit Vector Instructions

Following the plans made in 2008 about extending the SIMD vector unit to operate on vector registers of 512 bits, the Advanced Vector eXtensions 512 (AVX-512) was announced to be implemented on the Skylake micro-architecture; however, Intel delayed the release of AVX-512. Part of this instruction set was supported by the Intel Xeon Phi co-processor, which is a hardware accelerator mainly focused on many-core programming paradigm. The actual implementation of AVX-512 on desktop processors appeared under the codename of Skylake Core X-series.

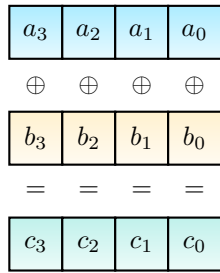
The AVX-512 extensions also increased the bank of registers to 32 512-bit vector registers, which quadruples the size of the bank used by the AVX/AVX2 vector unit. Another novelty of AVX-512 is the inclusion of conditional execution predicates for all the instructions; thus, each vector instruction has attached a bit mask that determines whether or not to execute an operation. Moreover, AVX-512 has a set of instructions dedicated to enabling conditional execution for legacy 128- and 256-bit vector instructions.



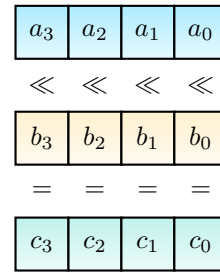
(a) VPADDQ: Adds four 64-bit words.



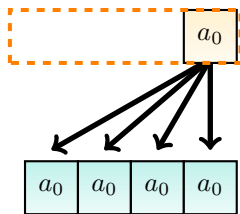
(b) VPMULUDQ: Multiplies four 32-bit words.



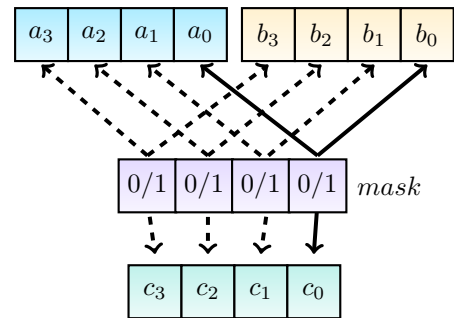
(c) VPXOR: XORs four 64-bit words.



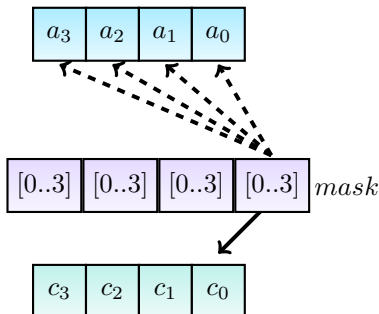
(d) VPSLLVQ: Variable left-shift of four 64-bit words.



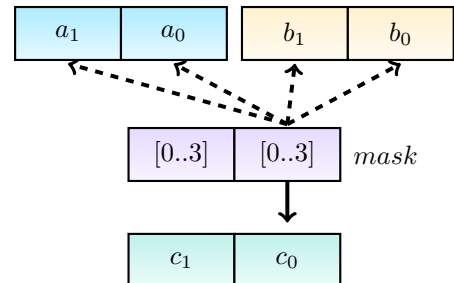
(e) VPBROADCASTD: Broadcast one word into the 64-bit words.



(f) VPBLEND: Conditionally selects between two registers.



(g) VPERMQ: Permutes the 64-bit words of one register.



(h) VPERM2I128: Permutes the 128-bit words of two registers.

Figure 2.2.2: Relevant AVX2 [75] vector instructions used in our implementations.

2.3 Extensions for Cryptography

Together with the inclusion of specialized instructions and technologies to increase the performance of programs, micro-architecture designers also considered the inclusion of instructions that aid in the execution of cryptographic algorithms.

2.3.1 The AES New Instructions

The Advanced Encryption Standard (AES) is a family of symmetric-key block ciphers and is currently the standard algorithm for performing data encryption [201]. AES takes a message of 128 bits and a key k , and produces a ciphertext of 128 bits; where $|k| \in \{128, 192, 256\}$ matching the security level of the algorithm.

We briefly describe the AES algorithm as is shown in Algorithm 2.3.1. First, the AES algorithm uses the secret key to generate a key schedule using the KeyExpansion function. Then, the message is stored in a state structure, which is processed by the following operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. These operations are repeated for a determined number of iterations or rounds. The final value in the state is declared as the ciphertext.

The internal structure of AES is a substitution and permutation network. The SubBytes function is a substitution layer providing a high-degree nonlinear function. The diffusion of bits across the state is performed by the subsequent functions: ShiftRows and MixColumns. Then, the AddRoundKey function combines the state with one of the round keys produced by the KeyExpansion function. The exact definition of all these functions is specified in [201, Section 5]. We suggest to look at the Paar-Pelzl's book [218] for a more didactic description of AES.

Algorithm 2.3.1 The AES encryption algorithm.

Input: M is a 128-bit message, and k is a key such that $(|k|, n_r) \in \{(128, 10), (192, 12), (256, 14)\}$.

Output: C is a 128-bit ciphertext such that $C = \text{AES}_k(M)$.

```

1:  $(K_0, \dots, K_{n_r}) \leftarrow \text{KeyExpansion}(k)$ 
2:  $S \leftarrow \text{AddRoundKey}(M, K_0)$ 
3: for  $i \leftarrow 1$  to  $n_r - 1$  do
4:    $S \leftarrow \text{SubBytes}(S)$ 
5:    $S \leftarrow \text{ShiftRows}(S)$ 
6:    $S \leftarrow \text{MixColumns}(S)$ 
7:    $S \leftarrow \text{AddRoundKey}(S, K_i)$ 
8: end for
9:  $S \leftarrow \text{SubBytes}(S)$ 
10:  $S \leftarrow \text{ShiftRows}(S)$ 
11:  $C \leftarrow \text{AddRoundKey}(S, K_{n_r})$ 
12: return  $C$ 

```

$\left. \begin{array}{l} 4: \\ 5: \\ 6: \\ 7: \end{array} \right\} \text{AESENC}(S, K_i)$
 $\left. \begin{array}{l} 9: \\ 10: \\ 11: \end{array} \right\} \text{AESENCLAST}(S, K_{n_r})$

In 2010, Intel released the AES-NI set, which contains six instructions dedicated to computing parts of the AES algorithm. In particular, the AESENC and AESENCLAST in-

structions encapsulate the operations performed in each round of the AES algorithm. Hence, Algorithm 2.3.1 can be implemented replacing the lines 4-7 by an `AESENC` instruction and the lines 9-11 by an `AESENC` instruction. Regarding performance timings, the resultant AES-NI implementation renders around 9 times faster performance since its execution is entirely performed in hardware.

The AES-NI set is supported on Intel processors since the release of the Westmere micro-architecture, and on AMD processors since the Jaguar micro-architecture. Also, some ARM processors have hardware support for executing AES; however, these extensions are not compatible with the AES-NI set.

2.3.2 The Carry-Less Multiplier

In addition to the AES instructions, the AES-NI instruction set contains the `PCLMULQDQ` instruction, which performs a carry-less multiplication of two 64-bit words. This multiplier calculates the product of the inputs words replacing the integer multiplication and integer addition by the AND and XOR Boolean operations, respectively.

The carry-less multiplier is useful for multiplying elements of a binary extension field. One of the first applications of this multiplier was to implement the AES Galois Counter Mode (GCM) [188], an algorithm for authenticated encryption, which requires calculating operations over an extension of a binary field. Gueron and Kounavis [127] showed that an implementation of AES-GCM using the `PCLMULQDQ` instruction is six times faster than implementations using look-up tables.

Binary field arithmetic is another application of the carry-less instruction. Taverne et al. [259,260] used the carry-less multiplier for implementing binary elliptic curves achieving better timings than implementations of elliptic curves defined over prime fields. Aranha et al. [13] achieved similar improvements for Koblitz binary curves.

Due to the relevance of this instruction, the latency of the carry-less multiplier have reduced from 14 cycles (when it was released) to around four cycles in the most recent architectures benefiting implementations of binary elliptic curves and AES-GCM.

2.3.3 Multi-Precision Integer Arithmetic

Multi-precision integer arithmetic is an essential part of the development public-key cryptography. This type of arithmetic is used to implement prime field operations, as we will discuss this topic extensively in Chapter 3. Due to its complexity, the calculation of multiplications is the most critical operation regarding performance. Although several techniques exist to perform multi-precision integer multiplications, all of them rely on the use of the native word-sized addition and multiplication instructions.

The instruction set of the x64 architecture has a structural issue that limits to increase the performance of integer multiplications. Any processor implementing the x64 architecture has the `FLAGS` register, which contains a set of bit flags that are updated according to the result of any integer operation executed by the processor. For example, both the carry bit (`CF`) and the overflow bit (`OF`) of the `FLAGS` register are set whenever adding two 64-bit integers produces a result greater than 2^{64} . Recalling that the exe-

cution engine has several units for processing more than one instruction simultaneously, then instructions depending on values stored in the `FLAGS` register could cause dependencies that avoid the efficient use of multiple execution units. This behavior is observed on instructions that calculate integer additions (the `ADD` instruction), additions with carry (the `ADC` instruction), and integer multiplications (the `MUL` and `IMUL` instructions).

The `BMI` and the `ADCX/ADOX` extensions are new instruction sets released to optimize the execution of multi-precision integer multiplications.

MULX: A New 64-bit Multiplier

The Bit Manipulation Instruction (`BMI`) is a set of instructions to perform bit operations over 64-bit registers. The `MULX` instruction belongs to `BMI` and is used to calculate 64-bit integer multiplications; however, `MULX` has properties that allow executing multi-precision integer multiplications faster.

The original `MUL` instruction calculates a 64-bit integer multiplication of the first register by the `RDX` register storing the product in the `RDX` and `RAX` registers.

```
1 MUL <reg> # <reg> x RDX -> RDX || RAX
```

Thus, `MUL` overwrites the `RDX` register and updating the `FLAGS` register whenever the product is greater than 2^{64} .

Like the `MUL` instruction, the `MULX` also calculates a 64-bit integer multiplication, however the new multiplier handles the product differently. The new `MULX` instruction uses a three-operand code. The `MULX` instruction multiplies the first operand register by the `RDX` register; then, the product is stored in the second and third operand registers.

```
1 MULX <reg0>, <reg1>, <reg2> # <reg0> x RDX -> <reg2> || <reg1>
```

Unlike `MUL`, the `FLAGS` register is not modified by `MULX`. This property is key for implementing multi-precision multiplications faster.

Some multi-precision multiplication algorithms require executing addition and multiplication instructions sequentially. By using the `MUL` instruction, the `FLAGS` register is modified overwriting the bits required by a previous `ADC` instruction. However, no overwriting happens when the `MULX` instruction is used. Therefore, addition instructions can be combined with `MULX` instructions resulting on a faster execution [217].

ADX: New Addition Instructions

The product-scanning technique, used for multi-precision integer multiplications, requires to process two additions with carry per integer multiplication; one of them is used to add consecutive words of the product, and the other one accumulates the intermediate product into the final product. However, calculating two additions requires to handle two carry bits at the same time, which is not possible using `ADC` instructions.

The `ADX` extensions [217] introduced two new instructions that modify the operation of integer additions. The `ADCX` instruction adds two 64-bit integers storing the last carry bit in the `CF` bit, and the other bits of the `FLAGS` register remain unaltered. The `ADOX` instruction performs analogously to the `ADCX` instruction; however, the carry bit is stored

in the OF bit. Hence, the processor can execute both instructions handling each carry bit separately. The ADX instructions can be used in conjunction with the MULX instruction to implement multi-precision integer multiplications leading to faster execution timings.

2.3.4 The SHA New Instructions

A hash function produces a short, fixed-size output known as the *digest* or *hash value* of an arbitrary-length input data. A *cryptographic hash function* adds the following security requirements: (i) Pre-image resistant: it must be difficult to find an input that produces a given digest; (ii) Second pre-image resistant: it must be difficult to find an input different from a given value such that both have the same digest; and (iii) Collision resistant: it must be difficult to find two different inputs that have the same digest.

There exists several algorithms that implement a cryptographic hash function. Former algorithms such as MD5 and SHA-1 are proven insecure [255, 268], and they should not be used for cryptographic purposes. Instead, the Secure Hash Algorithm (SHA) [203] is a standard that defines a family of cryptographic hash functions. The SHA-256, SHA-384, and SHA-512 are the most popular instances of the SHA-2 standard. There is also a third version of the SHA standard, called SHA-3 [205], that introduces new hash functions and extendable-output functions called SHAKE-128, and SHAKE-256.

Cryptographic hash functions are widely used in digital signature schemes. For instance, the RSA signature scheme signs the hash of messages, so it supports signing arbitrary-length messages. Another application is to verify the integrity of a message. Hash functions are designed in such a way that a minimal variation in the input significantly changes the output digest allowing an easy verification.

In 2013, Intel announced the specification of the SHA-NI instruction set [136]. These instructions offer native support for the most critical internal operations of the SHA-1 and SHA-256 hash functions. In Section 5.6, we show more details about the instructions and the implementation of SHA-256.

Although SHA-NI was introduced in 2013, the first processors supporting SHA-NI were available several years later. In 2016, the Goldmont micro-architecture, tailored for low power-consumption devices, was the first on supporting the SHA-NI set. In 2017, AMD released Zen micro-architecture that supports SHA-NI, AES-NI, and SIMD instructions.

2.3.5 Vectorized AES and Galois Field Extensions

In 2018, Intel announced a set of new extensions for improving the implementation of algorithms for data encryption [73]. The new VAES instruction set is a vectorized approach that extends the AES-NI instructions to execute up to four AES instructions simultaneously. Similarly, the SIMD processing was also applied to the carry-less multiplier by introducing the VPCLMULQDQ instruction.

Another relevant extension is the Galois field New Instructions (GFNI). This set includes instructions that perform operations over the binary field $\mathbb{F}_2[x]/(g(x))$, where $g(x) = x^8 + x^4 + x^3 + x + 1$. These operations are used to implement the AES-GCM authenticated encryption algorithm [202].

2.4 Evolution of Hardware Extensions

More instruction sets will be available in future generations of processors. For example, the IFMA instruction set has instructions that perform fused-multiply and add operations over 52-bit integers; these instructions are of particular interest to implement arbitrary-precision integer arithmetic, which is central to the implementation of the RSA algorithm [133].

We designed Figure 2.4.1 that show a timeline of the release date of several vector instruction sets other hardware extensions. In the figure, the height of the bars represents proportionally the number of instructions belonging to the each set. The color of the bars shows the predominant field of application of each set. becoming more specific As it can be seen, the number of vector instructions increased significantly in the past few years. Also, the applicability of recent hardware extensions is becoming more diverse. For instance, the Neural Network Vector Instructions (VNNI) [234] target the acceleration of deep learning processing, which is a building block in the artificial intelligence field.

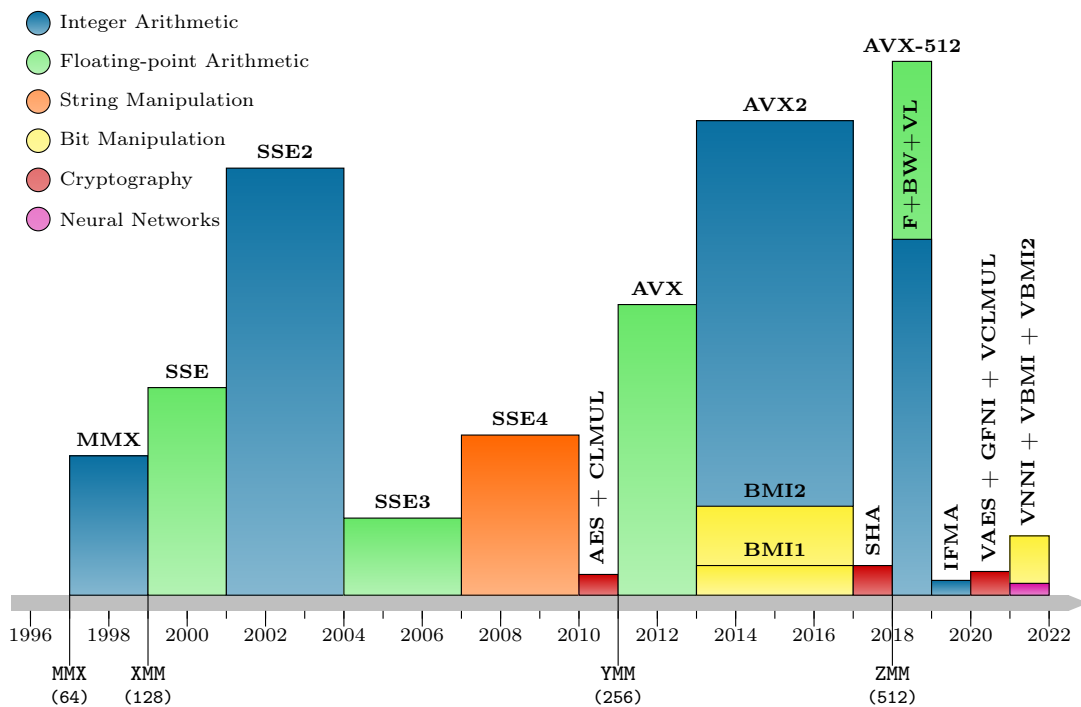


Figure 2.4.1: Evolution of vector instructions and hardware extensions.

2.5 Chapter Summary

Modern computer architectures aim to accelerate the execution of programs using different hardware optimizations. The most commonly available in contemporary processors are pipeline execution, superscalar processors, simultaneous multi-threading, SIMD processing, and multiple-core processing. Although they all have a common goal, which is to run programs faster, they benefit programs to different extents. Some issues and trade-offs appear concerning the efficiency of a given application.

A compelling way of improving performance is processing more data per unit of time. Since the middle of the 1990s decade, end-user processors increasingly added new instructions for SIMD processing. Initially, these instructions targeted programs with computationally intensive floating-point arithmetic. Nowadays, SIMD instructions cover a wide range of applications, such as integer arithmetic, text processing, bit manipulation, neural network acceleration, and cryptography.

The high relevance of information security boosts the inclusion of new extensions for cryptography. Modern processors have instructions that help with data encryption, cryptographic hashing, data integrity, and binary field arithmetic. However, public-key cryptography algorithms still need more support. One basic building block is the prime field arithmetic, a topic we cover in the next chapter. More specifically, we show how to efficiently apply the SIMD instruction sets to arithmetic of prime fields.

Chapter 3

Prime Field Arithmetic

In cryptography, data is manipulated through mathematical operations, mainly using algebraic operations. For this reason, it is convenient to use not only discrete domains (such as the integer numbers), but also finite domains (like a subset of integers) because the operands can be compactly stored and manipulated by computers. In abstract algebra, there are several mathematical objects (algebraic structures) with these properties. Notably the prime fields are elementary on the construction of cryptographic algorithms.

In this chapter, we review concepts about algebraic structures and algorithms for arithmetic operations over prime fields. We propose implementation techniques for performing operations efficiently and using regular execution algorithms. We cover special families of prime moduli identifying the following study cases:

- $p_{25519} = 2^{255} - 19$,
- $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$,
- $p_{448} = 2^{448} - 2^{224} - 1$,
- $p_{751} = 2^{372}3^{239} - 1$ and more generally primes of the form $p = 2^a b - 1$.

Each case is of interest in the recent advances of elliptic curve cryptography.

As shown in the previous chapter, recent computer architectures have included several layers of hardware optimizations promoting SIMD processing. In this chapter, we describe how to use these hardware extensions for executing prime field operations in parallel so to achieve a better utilization of the computer resources.

3.1 Algebraic Structures

It is said that a non-empty set of elements possesses an algebraic structure, if it is possible to define arithmetic operations on the elements of the set. Different algebraic structures emerge by imposing certain conditions on the set or on the properties of the operations. Three algebraic structures are fundamental for our purposes. We reproduce their definitions as given in a classic book about cryptography.

Definition 3.1.1 (§2.162 of [190]). A *group* (G, \star) consists of a set G with a binary operation $\star: G \times G \rightarrow G$ satisfying the following properties:

1. The group operation is associative: $a \star (b \star c) = (a \star b) \star c$ for all $a, b, c \in G$.
2. There exists an element $1 \in G$, known as the identity element, such that $a \star 1 = 1 \star a = a$ for all $a \in G$.
3. For all $a \in G$ there exists an element $a^{-1} \in G$, called the inverse of a , such that $a \star a^{-1} = a^{-1} \star a = 1$.

The group is abelian (or commutative) if the binary operation is commutative, i.e., $a \star b = b \star a$ for all $a, b \in G$. If this is the case, it is common to refer to the group as an additive group, its group operation as addition (denoted by $+$), its identity element is 0 , and the inverse of an element a is denoted as $-a$.

The order of the group, denoted as $\#G$, is equal to the cardinality (or the number of elements) of G . However, the term order has a different meaning when applied to the elements of G . Hence, the order of an element $a \in G$ is the least positive integer $k = \text{ord}(a)$ such that $a^k = 1$ provided that k exists, otherwise the order is ∞ . For any $a \in G$, the set of powers of a generates a cyclic group $H = \langle a \rangle$ of order $\#H = \text{ord}(a)$, which is a subgroup of G . Lagrange's theorem states that for every subgroup H of G , it holds that the order of H divides the order of G .

There exist other algebraic structures that consider more than one binary operation.

Definition 3.1.2 (§2.175 of [190]). A *ring* $(R, +, \times)$ consists of a set R together with two binary operations, denoted as $+$ (addition) and \times (multiplication), satisfying:

1. $(R, +)$ is an abelian group with identity denoted by 0 .
2. The operation \times is associative: $a \times (b \times c) = (a \times b) \times c$, for all $a, b, c \in R$.
3. There exists a multiplicative identity denoted by 1 such that $1 \neq 0$ and $a \times 1 = 1 \times a = a$ for all $a \in R$.
4. The operation \times is distributive over $+$, i.e., $a \times (b + c) = (a \times b) + (a \times c)$ and $(b + c) \times a = (b \times a) + (c \times a)$ for all $a, b, c \in R$.

The ring is commutative if $a \times b = b \times a$ for all $a, b \in R$. An element $a \in R$ is called a unit or an invertible element if there exists an element $b \in R$ such that $a \times b = 1$. If it exists, b is called the multiplicative inverse of a and is denoted by $b = a^{-1}$.

A subgroup I of R is called a *left ideal* of R if for all $r \in R$ and all $x \in I$, it holds that $rx \in I$. A *right ideal* is defined such that $xr \in I$. If I is both a left ideal and a right ideal, then it is called a *two-sided ideal*, or simply an *ideal* of R . An ideal I is maximal if for any ideal J of R with $I \subseteq J$, either $J = I$ or $J = R$.

The fact that some elements of the ring have no multiplicative inverse allows identifying another algebraic structure.

Definition 3.1.3 (§2.181 of [190]). A *field* is a commutative ring $(F, +, \times)$ in which every non-zero element has multiplicative inverse. The characteristic of a field, denoted as $\text{char}(F)$, is the least integer $k \geq 1$ such that $\sum_{i=1}^k 1 = 0$ (if the sum is never equal to 0 , the characteristic of the field is 0).

Fields that have a finite number of elements are also known as *Galois fields*, named in honor of Évariste Galois. The number of elements in the field is called its *order* and is equal to p^m , where p is a prime number and $m \geq 1$. A *prime field* is when $m = 1$. Conversely, for any prime power p^m , there exists a unique (up to isomorphism) finite field of order p^m , which is unambiguously denoted as \mathbb{F}_{p^m} or $\text{GF}(p^m)$. A subset E of a field F is a subfield of F if E is itself a field with the same binary operations. Hence, F is an extension field of E .

We describe some instances of algebraic structures that are of particular interest.

3.1.1 The Ring of Integers

The integer numbers (\mathbb{Z}) form a ring structure using the conventional operations of addition (+) and multiplication (\times). We describe the calculation of these operations in terms of bit operations restricting to the positive integers.

Representation

The *binary representation* of $a \in \mathbb{Z}^+$ is the unique sequence of bits $(b_{k-1}, \dots, b_0)_2$ such that $a = \sum_{i=0}^{k-1} b_i 2^i$, where k is the size (in bits) of a defined as

$$|a| = \begin{cases} 1, & \text{if } a = 0, \\ \lfloor \log_2(a) \rfloor + 1, & \text{otherwise.} \end{cases} \quad (3.1.4)$$

It is said that a is a k -bit integer if $|a| = k$.

Let $n \geq k = |a|$, the *n -bit representation* of a is defined as its binary representation prefixed by $n - k$ leading zeros, i.e., $(b_{n-1}, \dots, b_0)_2$ setting $b_i = 0$, for $|a| \leq i < n$.

Integer Addition

The integer addition can be defined in terms of bit operations. Let x and y be integers of one bit size, then $x + y$ is represented by two bits, the addition bit s and the (output) carry bit c such that $x + y = 2c + s$. More generally, the addition with (an input) carry is performed using a full adder circuit defined as

$$\text{FullAdder: } \{0, 1\}^3 \rightarrow \{0, 1\}^2 \\ (x, y, c_{\text{in}}) \mapsto (c_{\text{out}}, s) = \left((x \wedge y) \oplus (c_{\text{in}} \wedge (x \oplus y)), x \oplus y \oplus c_{\text{in}} \right). \quad (3.1.5)$$

If c_{out} produced by a full adder is fed as the c_{in} of another full adder, a two-bit full adder is obtained. Using this strategy repeatedly, one can calculate additions of integers of arbitrary size as shown in Algorithm 3.1.6.

Observe that Algorithm 3.1.6 could return a number of $n + 1$ bits, which happens, for example, when adding two n -bit integers, and thus, the n -th bit of their sum is non-zero.

Algorithm 3.1.6 Integer addition using full adder circuit.

Input: $(a_{n-1}, \dots, a_0)_2$ and $(b_{n-1}, \dots, b_0)_2$, the n -bit representation of integers $a, b \geq 0$ such that $|a|, |b| \leq n$.

Output: $(c_n, \dots, c_0)_2$, the $(n + 1)$ -bit representation of $a + b$.

```

1:  $z \leftarrow 0$ 
2: for  $i \leftarrow 0$  to  $n - 1$  do
3:    $(z, c_i) \leftarrow \text{FullAdder}(a_i, b_i, z)$ 
4: end for
5:  $c_n \leftarrow z$ 
6: return  $(c_n, \dots, c_0)_2$ 

```

Integer Subtraction

Analogously to addition, integer subtraction can be defined using bit operations. A subtraction with borrow is performed using a full subtractor circuit defined as

$$\text{FullSubtractor: } \{0, 1\}^3 \rightarrow \{0, 1\}^2$$

$$(x, y, b_{\text{in}}) \mapsto (b_{\text{out}}, d) = \left((\neg x \wedge y) \oplus (b_{\text{in}} \wedge (\neg x \oplus y)), x \oplus y \oplus b_{\text{in}} \right). \quad (3.1.7)$$

This circuit calculates $x - y - b_{\text{in}} = 2b_{\text{out}} + d$, where the resulting value must be interpreted in two's complement. Under this representation, b_{out} is non-zero if the result is negative. If the borrowed bits are chained between circuits, one can subtract integers of arbitrary size as shown in Algorithm 3.1.8.

Algorithm 3.1.8 Integer subtraction using full subtractor circuit.

Input: $(a_{n-1}, \dots, a_0)_2$ and $(b_{n-1}, \dots, b_0)_2$, the n -bit representation of integers $a, b \geq 0$ such that $|a|, |b| \leq n$.

Output: $(c_n, \dots, c_0)_2$, the $(n + 1)$ -bit representation of $a - b$.

```

1:  $z \leftarrow 0$ 
2: for  $i \leftarrow 0$  to  $n - 1$  do
3:    $(z, c_i) \leftarrow \text{FullSubtractor}(a_i, b_i, z)$ 
4: end for
5:  $c_n \leftarrow z$ 
6: return  $(c_n, \dots, c_0)_2$ 

```

Integer Multiplication

Let a and b be n -bit integers, their product $c = a \times b$ is calculated as

$$a \times b = a \times \left(\sum_{j=0}^{n-1} b_j 2^j \right) = \sum_{j=0}^{n-1} (a \times b_j) 2^j \quad (3.1.9)$$

$$= \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} a_i 2^i \times b_j 2^j = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} (a_i \times b_j) 2^{i+j} = \sum_{k=i+j} c_k 2^k.$$

The last equality follows since the product $a \times b$ represented in binary is $(c_{2n-2}, \dots, c_0)_2$, where $c_k = \sum_{k=i+j} a_i \times b_j$, for all $0 \leq i, j < n$.

Well-known algorithms for calculating integer multiplications have quadratic time-complexity on the size of the operands. That is, calculating the product of n -bit integers takes $O(n^2)$ operations. Sub-quadratic time-complexity algorithms exist such as the Karatsuba [168] algorithm. This algorithm takes $O(n^{\log_2 3})$ operations outperforming quadratic-complexity algorithms for big-enough operands.

Integer Squaring

Calculating squares is a special case of multiplication that occurs when both operands are equal. Let a be an n -bit integer, a^2 is calculated faster than a multiplication as follows

$$\begin{aligned} a^2 &= \left(\sum_{j=0}^{n-1} a_j 2^j \right)^2 = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} a_i 2^i \times a_j 2^j \\ &= \sum_{i=0}^{n-1} a_i^2 2^{2i} + 2 \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} (a_i \times a_j) 2^{i+j}. \end{aligned} \tag{3.1.10}$$

From the last equality, one can derive a method for integer squaring. First, each bit is squared leading to a partial product, and then, the product of bits with different indexes is multiplied by two and added to the first calculation. Although squaring and multiplication have the same complexity, squaring has smaller constants behind the big- O notation, it requires around one half of the operations of a multiplication, which is something that implementations usually leverage.

3.1.2 Prime Fields

This section shows how to construct a prime field from the ring of the integer numbers.

Let p be a prime and $p\mathbb{Z}$ be a two-sided ideal of \mathbb{Z} , which contains all the multiples of p . The \sim symbol denotes an equivalence relation between two elements $i, j \in \mathbb{Z}$ such that $i \sim j$ iff $i - j \in p\mathbb{Z}$. Thus, the equivalence class of an element $a \in \mathbb{Z}$ is $[a] = \{a + kp : k \in \mathbb{Z}\}$. The quotient ring $\mathbb{Z}/p\mathbb{Z}$ is defined as the set of all equivalence classes under \sim and the cardinality of this set is p . Since $p\mathbb{Z}$ is a maximal ideal, then $(\mathbb{Z}/p\mathbb{Z}, +, \times)$ is a finite field, denoted as \mathbb{F}_p , of characteristic p , whose binary operations are defined as $+$: $([a], [b]) \mapsto [a + b]$, and \times : $([a], [b]) \mapsto [a \times b]$.

The canonical projection is a function that given $a \in \mathbb{Z}$ obtains the correspondent equivalence class $[a] \in \mathbb{Z}/p\mathbb{Z}$. Any integer belonging to an equivalence class can be used as the representative of the class; however, usually the representative of $[a]$ is chosen to be $r \in [a]$ that is the unique integer $0 \leq r < p$ such that $a = kp + r$ for $k \in \mathbb{Z}$. Note that r is also the remainder of the division of a by p , which is denoted as $r = a \bmod p$. The reduction modulo p is a projection used to define unique representatives of \mathbb{F}_p .

Definition 3.1.11 (Canonical Representatives of \mathbb{F}_p). Let $\mathbb{F}_p = (\mathbb{Z}/p\mathbb{Z}, +, \times)$ be a prime field of p elements. The *canonical representatives* of the elements of \mathbb{F}_p are given by the set of integers $\{0, 1, \dots, p-1\}$, where 0 is the identity element of $+$, and 1 is the identity element of \times .

The canonical representation gives us a first approach to represent and operate prime field elements. Arithmetic operations in \mathbb{F}_p correspond to modular arithmetic with prime modulus p . In this arithmetic, operations are equal to the conventional integer arithmetic, but are followed by a reduction modulo p .

There exist several efficient algorithms for reduction modulo p such as the Barrett's [21], Blakley's [41], and Montgomery's [195] methods. All these methods are generic with respect to an arbitrary modulus. However, simpler algorithms for reduction exist if the modulus holds certain properties. Section 3.1.2 details families of prime moduli with faster methods for reduction modulo p .

Prime Field Addition and Subtraction

The addition of $a, b \in \mathbb{F}_p$ obtains $c = a + b \in \mathbb{Z}$ and then reduces c modulo p . After calculating the integer addition, only one of these cases can occur:

1. if $c \geq p$, one must subtract p from c ,
2. otherwise, $c < p$ and c corresponds to the element $c = a + b \in \mathbb{F}_p$.

Thus, addition in \mathbb{F}_p is calculated as an integer addition followed by an integer subtraction only if the result is greater than p .

For subtractions in \mathbb{F}_p , first calculate $d = a - b \in \mathbb{Z}$; then d could be either positive, in which case $d = a - b \in \mathbb{F}_p$; or negative, in this case adding p to d gives the canonical representative.

In both cases, the reduction modulo p is calculated faster than an integer division since only an extra addition or subtraction was calculated.

On the one hand, these methods, as described above, present an irregular execution pattern. Note that some extra operations are performed only when the integer operation gives a result that is not a canonical representative. On the other hand, this irregularity is easy to handle following secure software development techniques. For example, performing always the extra operations regardless whether or not the result is a canonical representative, and conditionally selecting the correct result.

Prime Field Multiplication

Prime field multiplications are calculated in two steps: an integer multiplication followed by a reduction modulo p . However, the reduction modulo p is more complicated because the size of the integer product is twice as large as the size of the input operands. Thus, the reduction procedure must be able to handle double-sized inputs. The next section describes a generic algorithm that calculates reductions modulo p efficiently.

Montgomery Arithmetic

Montgomery [195] showed an efficient method for modular arithmetic that does not require divisions by arbitrary integers. Instead, all divisions are by powers of two, which are easily computed with binary operations.

Montgomery's method requires that elements of \mathbb{F}_p be pre-processed before being operated. Let R be a positive integer such that $\gcd(p, R) = 1$ and $R > p$, then given $a \in \mathbb{F}_p$ define $\bar{a} = aR \bmod p$. The calculation of additions and subtractions remains as usual since $\bar{a} \pm \bar{b} = aR \pm bR = (a \pm b)R = \overline{a \pm b}$. However, the multiplication $\bar{a} \times \bar{b} = abR^2$ requires of dividing this product by R to obtain $\overline{a \times b}$.

Montgomery's REDC algorithm, shown in Algorithm 3.1.14, calculates $\overline{a \times b}$ from $T = abR^2 < Rp$. The core operation of REDC is adding a multiple of p to T such that the result is a multiple of R , i.e., there exists an integer q such that

$$T + qp = RT' . \quad (3.1.12)$$

Hence, $T' = (T + qp)/R$ is an integer congruent to $TR^{-1} \bmod p$. The value of q is derived from Equation (3.1.12) as

$$\begin{aligned} q &= (RT' - T)/p \\ &= -T/p \bmod R \\ &= (T \bmod R)(-p^{-1} \bmod R) \bmod R . \end{aligned} \quad (3.1.13)$$

It is guaranteed that the second factor exists whenever $\gcd(p, R) = 1$, and this factor is a constant value for each p . Since $0 \leq T + qp \leq 2Rp$, the value of T' is bounded as $0 \leq T' < 2p$. This means that if $T' \geq p$, subtracting p from T' results in $T' = TR^{-1} \bmod p$.

Algorithm 3.1.14 Montgomery's REDC algorithm [195].

Constants: Define R such that $R > p$ and $\gcd(p, R) = 1$. Define $p' = -p^{-1} \bmod R$.

Input: T , an integer such that $0 \leq T < Rp$.

Output: T' , an integer such that $T' = TR^{-1} \bmod p$.

- 1: $q \leftarrow (T \bmod R)p' \bmod R$
 - 2: $T' \leftarrow (T + qp)/R$
 - 3: **if** $T' \geq p$ **then**
 - 4: $T' \leftarrow T' - p$
 - 5: **end if**
 - 6: **return** T'
-

Algorithm 3.1.14 requires to calculate integer divisions and residues modulo R ; however, if R is a power of two, these operations are easily performed in the binary representation. One motivation behind REDC algorithm is calculating divisions by two modulo p . REDC generalizes this operation when R is a power of two.

REDC has a regular execution pattern except by the last subtraction. To remedy this issue, Walter [266] proposed a parameter selection that avoids the subtraction when a series of consecutive multiplications are performed. Using such a parametrization, REDC algorithm is a suitable choice for calculating reductions modulo p in constant time.

The Montgomery reduction admits different implementation techniques. Some of them reduce the number of operations performed and others modify the order in which operations are computed. Some examples of these techniques are: the SOS, FIOS, CIOS, FIPS, and CIHS methods described by Koç et al. [172], the hybrid scanning technique [137], and some hybrid variants of the methods presented above [181].

Multiplicative Inverse

The extended Euclidean algorithm (EEA) for integers can be adapted to calculate multiplicative inverses. Given $a, p \in \mathbb{Z}$, EEA calculates (d, a', p') such that $d = \gcd(a, p) = aa' + pp'$. Note that if $0 \leq a < p$ and p is prime, then $\gcd(a, p) = aa' + pp' = 1$, and reducing this equation modulo p , it follows that $aa' \equiv 1 \pmod{p}$ implying that $a' = a^{-1}$. Therefore, EEA can be modified to keep track of a' (as d and p' are not required) to calculate multiplicative inverses in \mathbb{F}_p .

Unfortunately, some operations required by EEA are dependent on the size of the inputs leading to a non-regular execution pattern. Although some countermeasures can be added to EEA, see for example Bos' work [44], the implementation effort is elevated.

An alternative way to calculate multiplicative inverses relies on the Fermat's little theorem. This theorem states that if p is a prime number then for any integer a , it holds that $a^p \equiv a \pmod{p}$. Thus, $a^{-1} \equiv a^{p-2} \pmod{p}$, which can be calculated using an exponentiation by a fixed constant.

In the literature, there are several algorithms that perform exponentiation using a regular execution pattern, avoiding the vulnerabilities exhibited by EEA. In fact, since the exponent depends on p , which is usually a public parameter, shorter addition chains can be used for speeding up this exponentiation.

Itoh and Tsujii [156] introduced an efficient algorithm to calculate inverses in \mathbb{F}_{2^m} . Relying on the Lagrange's theorem, the inverse of $a \in \mathbb{F}_{2^m} \setminus \{0\}$ is given as $a^{-1} = a^{2^m-2}$. Itoh-Tsujii's algorithm performs this exponentiation as $a^{-1} = (\alpha_{m-1})^2$, where $\alpha_x = a^{2^x-1}$ for some positive integer x . The term α_{m-1} is obtained from the following set of field elements $S = \{\alpha_{c_1}, \dots, \alpha_{c_s}\}$, where the sequence (c_1, \dots, c_s) represents an addition chain of length s such that $c_1 = 1$ and $c_s = m-1$. The elements of S are obtained constructively using the rule $\alpha_{c_i} = (\alpha_{c_x})^{2^{c_y}} \alpha_{c_y}$, where $c_i = c_x + c_y$ for $1 \leq x, y < i \leq s$. Calculating the entire set S takes $s-1$ field multiplications and $c_y(s-1)$ field squarings. For this reason, the shorter addition chain for $c_s = m-1$, the lesser number of elements in the set S , and the lesser number of operations required. Itoh-Tsujii's method can also be adapted to prime fields. The method saves a number of operations when the modulus is close to a power of two.

Regarding addition chains, Clift [67] provides a list of the lengths of all shortest addition chains for the integers lesser than 2^{31} . Flammenkamp [106] provides an online service that calculates the shortest addition chain of integers lesser than 2^{27} . These tools are useful for implementing Itoh-Tsujii's algorithm on prime fields. Once an addition chain is chosen, the operations of the Itoh-Tsujii method follow a regular execution pattern, which is a desirable property in secure software development.

Special Families of Moduli

Some cryptographic algorithms offer certain flexibility on selecting of the prime numbers that define a field. The main restriction is the relation between the security level and the size of the primes. However, after fulfilling this requirement, prime modulus can be chosen in such a way that it allows optimizations on arithmetic operations. In this section, we describe some advantages of well-known families of moduli.

Montgomery-Friendly Primes. The REDC algorithm performs a reduction modulo p using a constant value R , and as noted by Montgomery, by setting R as a power of two, the divisions by R can be performed as bit shifts. However, this is not the only trick that can be used in this algorithm. Special formats on the prime modulus are commonly known as Montgomery-friendly primes, since they allow saving some operations in the REDC algorithm. The following are some examples.

- Optimal prime fields [123]. Assuming that p is represented using w -bit words, the primes of the form $p = u2^k + v$, such that $u, v < 2^w$ and $k > 0$, are used to reduce the number of multiplications of the REDC algorithm. The key observation is that apart of the least- and the most-significant words, the remainder words are equal to zero, then multiplications by zero can be omitted. This kind of primes were used for optimizing implementations of elliptic curve algorithms [65, 183, 274].
- Acar and Shumow [1] presented a set of primes that avoid the calculation and storage of the constant $p' = -p^{-1} \pmod R$. Thus, whenever $p^2 \equiv 1 \pmod R$ and $\gcd(p, R) = 1$, then $p' = -p \pmod R$.
- Gueron and Krasnov [132] defined a property that allows to characterize some Montgomery-friendly primes.

Definition 3.1.15 (k -Montgomery-friendly prime [132, Def. 1]). Let p be a prime and k a positive integer, if $-p^{-1} \equiv 1 \pmod{2^k}$, then p is a k -Montgomery-friendly modulus.

Using k -Montgomery-friendly primes saves n word multiplications of REDC algorithm.

- A prime p such that $p = 2^i 3^j + 1$, for $i, j > 0$, is known as a Pierpont prime [224]. Similarly, if $p = 2^i 3^j - 1$, then p is a Pierpont prime of the second kind. The latter case was also classified in the class 1 by Erdős and Selfridge [138, Def. A18]. It can be noted that any Pierpont prime $p = 2^i 3^j - 1$ is also an i -Montgomery-friendly prime according to Definition 3.1.15. This indicates that if $i > w$, where w is the word size of the instruction set architecture, then the Montgomery reduction can be performed using fewer operations. This fact was also noticed by Bos and Friedberger [49] using different notation.

The use of these primes enables a faster calculation of the Montgomery's REDC algorithm.

Mersenne Numbers. The k -th Mersenne number has the form $M_k = 2^k - 1$ for some positive integer k . As of December 2018, there are known only 51 Mersenne numbers that have been proved to be primes [272].

Let p be a n -bit Mersenne prime and let a be the product of two n -bit numbers, thus $0 < a < 2^{2n}$. The operation $c = a \pmod p$ is performed as the addition $c = a_0 + a_1$, where $a_0 = a \pmod{2^n}$ and $a_1 = \lfloor a/2^n \rfloor$. If $c > p$, then the same procedure can be applied one more time. Therefore, the reduction modulo p has linear time-complexity on the size of

the prime, unlike general reduction algorithms, such as Montgomery REDC algorithm, which have quadratic time-complexity.

In cryptography, the prime M_{521} is used to define standardized elliptic curve algorithms. In [5, 121, 246], several optimizations were proposed targeting the arithmetic operations on $\mathbb{F}_{M_{521}}$. Also, the prime M_{128} is used to construct a quadratic extension of $\mathbb{F}_{M_{128}}$, which is used to define an elliptic curve called FourQ [77].

Pseudo-Mersenne Numbers. The existence of Mersenne primes is scarce, nonetheless the primes of the form $p = 2^n - c$, where c is a small number, are abundant. These numbers are known as pseudo-Mersenne primes and admit fast reduction modulo p as described in Crandall-Pomerance's book [79].

Let a be the product of two n -bit numbers, then $0 \leq a < 2^{2n}$ and the number $d = a \bmod p$ can be calculated as $d = a_0 + a_1c$, where $a_0 = a \bmod 2^n$ and $a_1 = \lfloor a/2^n \rfloor$. After calculating that addition, d could be greater than p ; in this case, apply a few times the same rule on d until obtaining $0 \leq d < p$. This operation has also a linear time complexity on the size of p , but it is slightly slower than a reduction modulo a Mersenne prime due to the multiplications by c . The cost of this last multiplication can be reduced if c is chosen as a small number or as having a low Hamming weight. Also, c can be chosen as $c < 2^w$, where w is the word size of the instruction set architecture, to save word multiplications.

Generalized Mersenne Numbers. Solinas [251] showed a family of moduli known as the generalized Mersenne numbers that admit fast reduction. The modulus p is expressed as a polynomial $f(t) \in \mathbb{Z}[t]$, where $t = 2^k$ for some integer k .

Let a be the number to be reduced modulo $p = f(t)$, the reduction is performed as $c = a \bmod p = \sum_j B_j 2^j$ where B_j are j linear combinations defined as $B_j = b_{0,j}a_0 + b_{1,j}a_1 + \dots + b_{n-1,j}a_{n-1}$ such that $b_{i,j}$ is a bit value and a_i are the digits of a in base t . Using this representation, a search of primes can be performed using several conditions that ensure the reduction takes a lower number of operations.

Primes of this form are used in standardized elliptic curves [200]. For example, the prime numbers $p_{192} = 2^{192} - 2^{64} - 1$, $p_{224} = 2^{224} - 2^{96} - 1$, $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$, and $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$ were chosen using Solinas's method.

More recently, Hamburg [141] proposed the use of the prime $p_{448} = 2^{448} - 2^{224} - 1$ to perform efficient prime field operations. This number can also be seen as a generalized Mersenne number by setting $t = 2^{224}$, thus $p_{448} = f(t) = t^2 - t - 1$. Observe that one of the roots of this polynomial is the golden ratio φ .

3.1.3 Extension Fields

Constructing Extension Fields

An extension field is a field that can be constructed from a given field. Similarly to the construction of prime fields, an extension field can be constructed from a ring and a maximal ideal on it. The quotient ring of them fulfills the properties of a finite field.

Let K be a field of characteristic p and order $q = p^n$ for some $n > 0$. and let $K[x]$ be the ring of polynomials on variable x and coefficients on K . This ring is the one used to construct a field of order q^m .

First, choose a non-constant monic polynomial $f(x) \in K[x]$ of degree $m > 1$ such that $f(x)$ be irreducible on K , i.e., $f(x)$ cannot be expressed as a product of non-constant polynomials of degree lesser than m . Let $(f(x))$ denote an ideal of $K[x]$ containing all the polynomials that are multiples of $f(x)$. Hence, there exists an equivalence relation \sim on the elements of $K[x]$. Given $g_0(x), g_1(x) \in K[x]$, $g_0(x) \sim g_1(x)$ iff $g_0(x) - g_1(x) \in (f(x))$. The equivalence class of $g(x) \in K[x]$ is $[g(x)] = \{g(x) + h(x)f(x) : h(x) \in K[x]\}$. Using this relation, the quotient ring $K[x]/(f(x))$ corresponds to the set of equivalence classes of \sim and its cardinality is q^m .

Since $f(x)$ is irreducible over K and $(f(x))$ is an ideal maximal of $K[x]$, it follows that $L = (K[x]/(f(x)), +, \times)$ is a field of characteristic p and order q^m . The representation of its elements are all the polynomials of degree lesser than m . Moreover, L is an extension field, or an extension of the field K , and this relation is denoted as L/K . It is said that K is the base (or ground) field of L .

Given $g(x), h(x) \in L$, the addition of elements is defined as the addition of polynomials $g(x) + h(x)$ where the coefficient operations are performed in K . On the other hand, the product of elements in L is defined as the polynomial $r(x) = g(x) \times h(x) \bmod f(x)$ such that $0 \leq \deg(r(x)) < m$.

A Quadratic Extension Field

We exemplify the construction of an extension field of order p^2 from a field of prime order p . We denote the quadratic extension of \mathbb{F}_p as \mathbb{F}_{p^2} . This extension has been used in the GLS method [111] and the FourQ curve [77], and more recently, it has also been used in the supersingular isogeny setting.

An extension field of order p^2 can be efficiently constructed if $p \equiv 3 \pmod{4}$ as follows. Let $\mathbb{F}_p[\tau]$ be the ring of polynomials in τ with coefficients on \mathbb{F}_p and $f(\tau) = \tau^2 + 1 \in \mathbb{F}_p[\tau]$. Note that $\tau^2 + 1$ can be factored as $(\tau - \sqrt{-1})(\tau + \sqrt{-1})$; however, -1 is a quadratic non-residue on \mathbb{F}_p indicating that $\sqrt{-1} \notin \mathbb{F}_p$; hence, $\tau^2 + 1$ is irreducible over \mathbb{F}_p . Finally, since $(\tau^2 + 1)$ is a maximal ideal, this results on that $\mathbb{F}_{p^2} = (\mathbb{F}_p[\tau]/(\tau^2 + 1), +, \times)$ is a field of order p^2 .

The elements of \mathbb{F}_{p^2} are represented by polynomials $a_1\tau + a_0$ such that $a_0, a_1 \in \mathbb{F}_p$. Let $a_1\tau + a_0$ and $b_1\tau + b_0$ two elements of \mathbb{F}_{p^2} , addition is calculated as

$$(a_1\tau + a_0) + (b_1\tau + b_0) = (a_1 + b_1)\tau + (a_0 + b_0), \quad (3.1.16)$$

and subtraction as

$$(a_1\tau + a_0) - (b_1\tau + b_0) = (a_1 - b_1)\tau + (a_0 - b_0). \quad (3.1.17)$$

Hence, to calculate one addition (subtraction) on \mathbb{F}_{p^2} are required two additions (subtractions) on the base field \mathbb{F}_p . The multiplication of elements of \mathbb{F}_{p^2} is performed as a polynomial multiplication followed by a reduction modulo $\tau^2 + 1$. The polynomial multi-

plication $(a_1\tau + a_0) \times (b_1\tau + b_0)$ is equal to $a_1b_1\tau^2 + (a_1b_0 + a_0b_1)\tau + a_0b_0 \in \mathbb{F}_p[\tau]$. Note that $\tau^2 \equiv -1 \pmod{\tau^2 + 1}$, so we obtain $(a_1b_0 + a_0b_1)\tau + (a_0b_0 - a_1b_1) \in \mathbb{F}_{p^2}$. As can be seen, calculating a multiplication in the extension field requires four base field multiplications. However, it can be reduced to three multiplications using the Karatsuba identity as

$$(a_1\tau + a_0) \times (b_1\tau + b_0) = [(a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0]\tau + (a_1b_1 - a_0b_0). \quad (3.1.18)$$

The square of $a_1\tau + a_0 \in \mathbb{F}_{p^2}$ is calculated as

$$\begin{aligned} (a_1\tau + a_0)^2 &= (2a_0a_1)\tau + (a_0^2 - a_1^2), \\ &= (2a_0a_1)\tau + (a_0 + a_1)(a_0 - a_1). \end{aligned} \quad (3.1.19)$$

The first formula requires two squarings and one multiplication on the base field, and the second one takes two multiplications on the base field. The choice of the formula depends on the cost of multiplying versus squaring. Finally, the multiplicative inverse of $a_1\tau + a_0 \in \mathbb{F}_{p^2} \setminus \{0\}$ is given as

$$(a_1\tau + a_0)^{-1} = \frac{-a_1}{a_0^2 + a_1^2}\tau + \frac{a_0}{a_0^2 + a_1^2}. \quad (3.1.20)$$

This operation takes two squares, two multiplications, and one inverse on the base field.

3.2 Operations over Prime Field Elements

3.2.1 Machine Representation of Integers

The size of the numbers used in cryptographic algorithms is around a few hundreds of bits; nonetheless, current processors operate over integers of 64 bits. Because of that, multi-precision software libraries are used to handle numbers of that magnitude. For example, the GNU Multi-Precision (GMP) [122] library is one of the most used software libraries that supports operations over arbitrary-precision integers. It is common that cryptographic algorithms fix a set of parameters for a given security level. Under this assumption, developing a multi-precision software library can be specialized to operate over numbers of a fixed size enabling a number of optimizations.

We now describe two representations suitable for operating large integer numbers.

Polynomial Representation

Multi-precision libraries habitually use a representation known as the radix- 2^w representation, which splits integers into digits of w bits, where w is the size of the words operated by the instruction set architecture.

Definition 3.2.1 (Polynomial Representation, or radix- 2^w). Let w be the processor's word size and a be an n -bit positive integer, the *polynomial representation* of a is a sequence of integers $A = (a_{l-1}, \dots, a_0)_{2^w}$ such that $a = \sum_{i=0}^{l-1} a_i 2^{iw}$ and $0 \leq a_i < 2^w$, where $l = \lceil n/w \rceil$.

One advantage of this representation is that elements require a compact memory-footprint, since the digits of sequence can be stored using l registers of w bits. This compact representation is convenient when large integers must be stored or transmitted.

In the literature, some authors use interchangeably the following terms for referring to the elements of a sequence: digits, words, chunks, or limbs. In this work, we decided to use the term *digits* because we are restricting to use only positive integers, and to use the term *word* for referring to the data stored in a machine register. Algorithm 3.2.2 shows steps to follow for obtaining the polynomial representation of an integer.

Algorithm 3.2.2 Obtaining the radix- 2^w representation of an integer.

Input: (a, w) , where a is an n -bit positive integer and w is a positive integer.

Output: (a_{l-1}, \dots, a_0) is the polynomial representation of a .

```

1:  $l \leftarrow \lceil n/w \rceil$ 
2: for  $i \leftarrow 0$  to  $l - 1$  do
3:    $a_i \leftarrow a \bmod 2^w$ 
4:    $a \leftarrow \lfloor a/2^w \rfloor$ 
5: end for
6: return  $(a_{l-1}, \dots, a_0)$ 

```

Generalized Polynomial Representation

One possible generalization of the radix- 2^w representation is to consider w as a non-integer number. This modification brings a representation which is formally specified as follows.

Definition 3.2.3 (Generalized Polynomial Representation). Let w be the processor's word size and a be an n -bit positive integer, given $\rho \in \mathbb{R}^+$ such that $0 < \rho \leq w$, the *generalized polynomial representation* of a is a sequence of integers $A = (a_{l-1}, \dots, a_0)$ such that $a = \sum_{i=0}^{l-1} a_i 2^{\lceil i\rho \rceil}$, and $0 \leq a_i < 2^{\beta_i}$, where $l = \lceil n/\rho \rceil$, and $\beta_i = \lceil (i+1)\rho \rceil - \lceil i\rho \rceil$.

We want to highlight some relevant properties of the generalized polynomial representation. First, observe that the polynomial representation is a special case setting $\rho = w$. When $\rho < w$, the maximum size of the digits (β_i) reduces while the number of digits (l) increases in comparison to the polynomial representation. For this reason, the selection of ρ introduces a trade-off between the number of digits and the maximum size of digits.

Additionally, when ρ is not an integer, the maximum size of digits (β_i) is not uniform across the sequence because the size of the i -th digit can be as large as β_i bits. Conversely to the polynomial representation, in which the maximum size of digits is 2^w .

A sequence of l digits can be stored in l registers of w bits. Note that by storing the digit a_i , it only uses the β_i least-significant bits of a w -bit register, meanwhile the remainder $w - \beta_i$ bits are set to 0. When $\rho = w$, registers have no extra room for storing more bits, then it is said that the digit *saturates* the register; and this is one reason for which the polynomial representation is also known as a *saturated representation*. On the other hand, when $\rho < w$, the digits do not populate the whole register, and this case is commonly known as an *unsaturated representation*. Algorithm 3.2.4 shows steps for obtaining the generalized polynomial representation of an integer.

Algorithm 3.2.4 Obtaining the generalized polynomial representation of an integer.

Input: (a, ρ) , where a is an n -bit positive integer and $\rho \in \mathbb{R}^+$.

Output: (a_{l-1}, \dots, a_0) is the generalized polynomial representation of a .

```

1:  $l \leftarrow \lceil n/\rho \rceil$ 
2: for  $i \leftarrow 0$  to  $l - 1$  do
3:    $\beta_i \leftarrow \lceil (i + 1)\rho \rceil - \lceil i\rho \rceil$ 
4:    $a_i \leftarrow a \bmod 2^{\beta_i}$ 
5:    $a \leftarrow \lfloor a/2^{\beta_i} \rfloor$ 
6: end for
7: return  $(a_{l-1}, \dots, a_0)$ 

```

Redundant Representation

From the generalized polynomial representation, we derive a redundant representation, in which integers are represented in a non-unique way. Note that by storing digits into w -bit registers, digits can safely grow up to $2^w - 1$ without overflowing the register. If this behavior is allowed, the maximum size of digits increases from β_i to 2^w , and as a result, the integers could have more than one sequence of digits that represents them. We formally state this idea defining an equivalence relation between sequences of digits.

Given $\rho \in \mathbb{R}^+$, and let $A = (a_{l_a-1}, \dots, a_0)$ and $B = (b_{l_b-1}, \dots, b_0)$, be two sequences of digits of length l_a and l_b , respectively, such that $0 \leq a_i, b_j < 2^w$ for $0 \leq i < l_a$ and $0 \leq j < l_b$. Note that we relax the condition about the digits (i.e., $0 \leq a_i, b_j < 2^w$) in contrast to the generalized polynomial representation, in which digits must be smaller than 2^{β_i} . Then, it is easy to prove that \sim is an equivalence relation such that $A \sim B$ iff there exists a positive integer k such that

$$\sum_{i=0}^{l_a-1} a_i 2^{\lceil i\rho \rceil} = k = \sum_{j=0}^{l_b-1} b_j 2^{\lceil j\rho \rceil}. \quad (3.2.5)$$

Thus, it can be said that A and B are equivalent representations of k . This equivalence relation allows us to define more formally a redundant representation as follows.

Definition 3.2.6 (Redundant Representation). Given $\rho \in \mathbb{R}^+$, the *redundant representation* of a refers to any sequence of positive integers that belongs to the equivalence class $[A]$, where $A = (a_{l-1}, \dots, a_0)$ is the generalized polynomial representation of a . A canonical representative of an equivalence class is the sequence of digits that holds the conditions of the generalized polynomial representation.

Example 3.2.7. Let $\rho = 3$ and $w = 4$ (informally, this means registers can store hexadecimal digits while we work on a *redundant* octal radix), it can be seen that $A = (2, 7)$ and $B = (1, 15)$ are equivalent sequences as both evaluate to $23 = 2 \times 2^3 + 7 = 1 \times 2^3 + 15$.

An advantage of using a redundant representation is that digits can grow a few bits without overflowing registers. For instance, in a saturated arithmetic, digits are likely to overflow a register after an addition operation due to the carry bit generation. However, this is not the case when working on a redundant, unsaturated representation, since the

extra bits in each register support a bounded increment of the digit's size. This slight difference enables a parallel calculation of operations, as we will show later in the section devoted for arithmetic operations. Table 3.2.8 shows a summary of the parameters used by the representations described above.

Table 3.2.8: Comparison of machine representations of n -bit integers.

Representation	l (digits)	Maximum size of digits (bits)	Unique Representative
Binary	n	1	Yes
Hexadecimal	$\lceil n/16 \rceil$	16	Yes
Polynomial or radix- 2^w	$\lceil n/w \rceil$	w	Yes
Generalized Polynomial	$\lceil n/\rho \rceil$	$\max_{0 \leq i < l} \beta_i$	Yes
Redundant	$\lceil n/\rho \rceil$	w	No

The length of a sequence is the number of digits in a sequence. But, the *size of a sequence* $A = (a_{l-1}, \dots, a_0)$ is given by the size of its largest digit, and is defined as

$$|A| = \max(|a_{l-1}|, \dots, |a_0|). \quad (3.2.9)$$

Proposition 3.2.10. If A is the generalized representation of an integer, then $|A| \leq \lceil \rho \rceil$.

Proof. From Equation (3.2.9), it is known that $|A| = \max(|a_i|) \leq \max(\beta_i)$ for $0 \leq i < l$ and from Definition 3.2.3, we have $\beta_i = \lceil (i+1)\rho \rceil - \lceil i\rho \rceil$; then, we must find an upper bound for β_i . We start with this fact

$$\lceil i\rho \rceil + \lceil \rho \rceil - 1 \leq \lceil (i+1)\rho \rceil \leq \lceil i\rho \rceil + \lceil \rho \rceil. \quad (3.2.11)$$

Then, subtracting $\lceil i\rho \rceil$ to this inequality, we have $\lceil \rho \rceil - 1 \leq \beta_i \leq \lceil \rho \rceil$, which indicates that for any $0 \leq i < l$, the value of $\beta_i \leq \lceil \rho \rceil$, which implies $|A| \leq \lceil \rho \rceil$. \square

3.2.2 Operations using Polynomial Representation

This section describes how to perform arithmetic operations over large integer numbers using a polynomial representation.

Addition

The computer architecture has an instruction, called ADC, that performs integer addition with carry of w -bit integers, i.e., given a, b two w -bit and a bit z the ADC instruction calculates $(x, c) \leftarrow \text{ADC}(a, b, z)$, where $c = a + b$ is a w -bit integer and x is the carry bit.

The ADC instruction is used to add sequences of digits following Algorithm 3.2.12. This algorithm exhibits inherent sequential execution, since the digit c_{i+1} depends on the carry bit produced by the addition of a_i and b_i . Hence, the carry bit z introduces a loop-carried dependency that limits the parallel execution of additions.

Algorithm 3.2.12 Integer Addition using Polynomial Representation.

Input: A and B , two sequences of digits of length l .

Output: C and z , a sequence of digits of length l , such that $C = A + B$ and z is the carry bit.

```

1:  $z \leftarrow 0$ 
2: for  $i \leftarrow 0$  to  $l - 1$  do
3:    $(z, c_i) \leftarrow \text{ADC}(a_i, b_i, z)$ 
4: end for
5: return  $C = (c_0, \dots, c_{l-1})$  and  $z$ 

```

Multiplication

Algorithm 3.2.13 shows the operand-scanning method to multiply integers in a polynomial representation. The multiplication of w -bit integers, is denoted as $(x, y) \leftarrow a \times b$, where x and y represent, respectively, the w most- and least-significant bits of $a \times b$.

Algorithm 3.2.13 Integer Multiplication using Polynomial Representation (operand scanning method).

Input: A and B , two sequences of digits of length l .

Output: C , a sequence of digits of length $2l$, such that $C = A \times B$.

```

1:  $z \leftarrow 0$ 
2: for  $j \leftarrow 0$  to  $l - 1$  do
3:    $(x, y) \leftarrow a_0 \times b_j$ 
4:    $c_j \leftarrow y + z$ 
5:    $z \leftarrow x$ 
6: end for
7:  $c_l \leftarrow z$ 
8: for  $i \leftarrow 1$  to  $l - 1$  do
9:    $z \leftarrow 0$ 
10:  for  $j \leftarrow 0$  to  $l - 1$  do
11:     $(x, y) \leftarrow a_i \times b_j$ 
12:     $c_{i+j} \leftarrow c_{i+j} + y + z$ 
13:     $z \leftarrow x$ 
14:  end for
15:   $c_{i+l} \leftarrow z$ 
16: end for
17: return  $C = (c_0, \dots, c_{2l-1})$ 

```

This algorithm is known as the operand-scanning method because the digits of B are scanned sequentially whereas one digit of A is fixed. Note that the internal loop of Algorithm 3.2.13 calculates one multiplication and two additions, one of these additions is used to propagate the highest part of the previous product and the second one is used to accumulate the product into the output register. Therefore, Algorithm 3.2.13 requires l^2 digit multiplications and $2l^2 - l$ digit additions to multiply two sequences of length l .

Other multiplication techniques exist. The product-scanning technique also known as the Comba multiplier [69], the operand-caching technique [154], the consecutive operand-caching technique [242], the full operand-caching technique [243], and the reverse product

scanning [182]. These methods target different optimization goals, for example, for reducing the number of digit additions or the number of load and stores, or for improving the usage of registers. However, all of them calculates l^2 digit multiplications. Unless a distinction is needed, we refer to any of these methods as quadratic complexity algorithms.

3.2.3 Operations using a Redundant Representation

This section shows how to calculate arithmetic operations using a redundant representation of integers specified in Definition 3.2.6.

Addition

Let A and B be redundant sequences, the addition $C = A + B$ is performed digit-by-digit following Algorithm 3.2.14. An advantage of this algorithm is that the digits of C are calculated without dependencies between digits, unlike the addition of sequences in the polynomial representation. Thus, the addition of integers exhibits a larger degree of parallelism by using a redundant representation.

Algorithm 3.2.14 Integer Addition using Redundant Representation.

Input: A and B , two sequences of digits of length l , such that $0 \leq |A|, |B| < w$.

Output: C , a sequence of digits of length l , such that $C = A + B$ and $0 \leq |C| \leq w$.

```

1: for  $i \leftarrow 0$  to  $l - 1$  do
2:    $(x, c_i) \leftarrow \text{ADC}(a_i, b_i, 0) = a_i + b_i$ 
3: end for
4: return  $C = (c_{l-1}, \dots, c_0)$ 

```

Beware that this algorithm restricts the size of the input operands to be strictly lesser than w bits, which ensures that $x = 0$, i.e., the carry bit produced by the ADC instruction is always equal to 0, and because of that, it can be safely discarded. On the other hand, if the size of the input sequences is w , then, $x = 1$ for some i ; and if this occurs, Algorithm 3.2.14 will fail because the carry bit must be propagated to the next digit, which is a task that is not performed by it.

The following proposition can be used to find the maximum the number of consecutive digit additions that can be performed before overflowing a register.

Proposition 3.2.15. Given an integer $n \in \mathbb{Z}^+$, let a be an integer such that $|a| = n$ and define $c = \sum_1^{2^k} a$, then $|c| = n + k$ bits for an integer $k \geq 0$.

Proof. We want to prove that the addition of 2^k integers of n bits results in a number of $n + k$ bits for $k \geq 0$. Note that $c = 2^k a$, then $|c| = \lfloor \log_2(2^k a) \rfloor + 1 = \lfloor k + \log_2(a) \rfloor + 1$, and since k is integer then $|c| = k + \lfloor \log_2(a) \rfloor + 1 = n + k$ as claimed. \square

Assume that A is a sequence such that $|A| \leq \lceil \rho \rceil$, then according to Proposition 3.2.15, A can be added with itself $2^k - 1$ times, where $k = w - \lceil \rho \rceil$, without overflowing the w -bit registers. This result is relevant since it tells us that for each bit added in the extra room of an unsaturated representation, the number of consecutive additions using Algorithm 3.2.14 is doubled. Hence, if a high-level operation requires of many consecutive additions, one can reduce the value of ρ for supporting this workload.

Multiplication

The objective of this section is to explain how to calculate integer multiplications avoiding any carry propagation or overflowing. To achieve these properties, one must guarantee certain bounds on the size of input operands and registers. This approach differs from the calculation performed by Algorithm 3.2.13, which splits digit products and accumulates each part in separated digits. Conversely, in this setting, digit products are accumulated without dependencies, which increases the degree of parallelism of multiplications. We now describe a procedure we followed to derive a set of parameters and an integer multiplication method that hold these requirements.

Given two sequences A and B of length l , the sequence $C = A \times B = (c_{2l-2}, \dots, c_0)$ is calculated as

$$c_{i+j} = \sum_{j=0}^{l-1} \sum_{i=0}^{l-1} a_i \times b_j. \quad (3.2.16)$$

The first restriction we impose to calculate this operation is that the digits of C must fit in w -bit registers, because this avoids the propagation of bits between digits during integer multiplication. As a consequence, one must guarantee that the registers have enough space to accumulate double-sized digit products. To that end, one can reduce the size of input operands. So we want to find tighter bounds on the size of operands that allows us to precisely determine the capacity of registers for performing this calculation.

We want to determine the size of the sequence $C = A \times B$ assuming that $|A|, |B| \leq \lceil \rho \rceil$. First, we must know the size of each digit in the products, which is given by the following proposition that bounds the size of the product of two integers. With this information, we can calculate the size of the digits of C .

Proposition 3.2.17. If a and b are positive integers, then it follows that the size of $a \times b$ is bounded as $|a| + |b| - 1 \leq |a \times b| \leq |a| + |b|$.

Proof. From Equation (3.1.4) we have

$$|a \times b| = \lfloor \log_2(a \times b) \rfloor + 1 = \lfloor \log_2(a) + \log_2(b) \rfloor + 1, \quad (3.2.18)$$

then, it is known that $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor \leq \lfloor x \rfloor + \lfloor y \rfloor + 1$ is valid for $x, y \in \mathbb{R}$; so, we can apply this relation to obtain

$$\lfloor \log_2(a) \rfloor + \lfloor \log_2(b) \rfloor \leq \lfloor \log_2(a) + \log_2(b) \rfloor \leq \lfloor \log_2(a) \rfloor + \lfloor \log_2(b) \rfloor + 1; \quad (3.2.19)$$

finally, adding one and using Equation (3.1.4), it follows that

$$|a| + |b| - 1 \leq |a \times b| \leq |a| + |b|, \quad (3.2.20)$$

which proves the proposition. \square

With this relation, we know the size of each digit product is bounded as $|a_i \times b_j| \leq 2 \lceil \rho \rceil$. Recall that the digits of C are generated by accumulating some of these digit products; in fact, c_{l-1} is the digit that accumulates more products, since there are l pairs (i, j) that hold $i + j = l - 1$. Let's calculate the size of c_{l-1} which gives us the size of C .

The digit c_{l-1} accumulates l digit products. From Proposition 3.2.15, we know that adding 2^k integers of n bits produces an integer of $n+k$ bits. Then, if we set $k = \lfloor \log_2(l) \rfloor$, we have that $|c_{l-1}| = 2\lceil \rho \rceil + k$, and finally, we can conclude that $|C| \leq 2\lceil \rho \rceil + |l| - 1$.

Finally, we want to find a tight bound for ρ assuming that the digits of C are stored in registers of w bits, i.e., $|C| \leq w$. Using the previous result, the following relation gives us such a bound

$$\lceil \rho \rceil \leq \frac{w - |l| + 1}{2}. \quad (3.2.21)$$

This relation is a strong condition used to choose the value of ρ , otherwise, a loss of precision or overflowing can be experienced.

Another condition for the value of ρ comes from the instruction set architecture. Assume that w' is an integer that represents the size of the integer multiplier available in the instruction set architecture, i.e., the multiplication instruction calculates the product of w' -bit integers. Usually, $w' = w$ in most architectures; however, others could have a smaller multiplier, i.e., $w' \leq w$. For instance, in AVX2 most of the arithmetic operations can be performed for 64 bit words, except for multiplications, which calculates products of 32 bits; thus, $w' = 32 < 64 = w$. This micro-architectural issue introduces the following condition on the value of ρ , specifically

$$\lceil \rho \rceil < w' \leq w. \quad (3.2.22)$$

Therefore, the implementation of integer multiplications using redundant representation must select a value of ρ that satisfies bounds given in Equations (3.2.21) and (3.2.22). With these conditions at hand, we present Algorithm 3.2.23 showing the operand-scanning method to calculate the product of two sequences of length l .

Algorithm 3.2.23 Integer Multiplication using Redundant Representation.

Input: A and B , two sequences of digits of length l and size $|A|, |B| \leq \lceil \rho \rceil$.

Output: $C = A \times B$, a sequence of digits of length $2l - 1$ and size $|C| \leq 2\lceil \rho \rceil + |l| - 1$.

```

1: for  $i \leftarrow 0$  to  $2l - 2$  do
2:    $c_i \leftarrow 0$ 
3: end for
4: for  $i \leftarrow 0$  to  $l - 1$  do
5:   for  $j \leftarrow 0$  to  $l - 1$  do
6:      $c_{i+j} \leftarrow c_{i+j} + a_i \times b_j$ 
7:   end for
8: end for
9: return  $C = (c_{2l-2}, \dots, c_0)$ 

```

Karatsuba Multiplication

The Karatsuba algorithm [168] is a divide and conquer strategy for multiplying polynomials of degree $2n$ at the cost of three polynomial multiplications of degree n plus some polynomial additions. We apply the Karatsuba algorithm to multiply sequences of digits as reduces the total number of digit multiplications. Given $A = (a_{l-1}, \dots, a_0)$

and $B = (b_{l-1}, \dots, b_0)$, define $n = \lceil l/2 \rceil$ and $A_1 = (a_{l-1}, \dots, a_n)$, $A_0 = (a_{n-1}, \dots, a_0)$, $B_1 = (b_{l-1}, \dots, b_n)$, and $B_0 = (b_{n-1}, \dots, b_0)$; the sequence $C = A \times B$ is calculated as

$$\begin{aligned} X &= A_0 \times B_0, \\ Y &= A_1 \times B_1, \\ Z &= (A_0 + A_1) \times (B_0 + B_1), \\ C &= X + 2^n(Z - X - Y) + 2^{2n}Y. \end{aligned} \tag{3.2.24}$$

The last equality is also known as the Karatsuba identity. One can apply Karatsuba multiplication to calculate X , Y , and Z recursively. Although the recursion can continue until reaching sequences of length one, it usually stops earlier at a point in which multiplying short-length sequences is performed faster with a different method.

Bernstein [25] observed that some additions can be saved during the recombination of products. The refined Karatsuba identity gives an alternate way for calculating C from X , Y , and Z as

$$C = (1 - 2^n)(X - 2^n Y) + 2^n Z. \tag{3.2.25}$$

The idea is to calculate $X - 2^n Y$ first, and then add it to the first position and subtract it from n -th position, and then add Z in the n -th position to produce the output sequence.

Algorithm 3.2.26 shows how to multiply sequences of digits using the Karatsuba method together with the refined identity. We use this algorithm when l is (close to) a power of two.

Algorithm 3.2.26 Karatsuba Algorithm for Integer Multiplication using Redundant Representation.

Input: A and B , two sequences of digits of length l and size $|A|, |B| \leq \lceil \rho \rceil$.

Output: $C = A \times B$, a sequence of digits of length $2l$ and size $|C| \leq 2\lceil \rho \rceil + |l| - 1$.

```

1:  $n \leftarrow \lceil l/2 \rceil$ 
2:  $A_1 \leftarrow (a_{l-1}, \dots, a_n)$ 
3:  $A_0 \leftarrow (a_{n-1}, \dots, a_0)$ 
4:  $B_1 \leftarrow (b_{l-1}, \dots, b_n)$ 
5:  $B_0 \leftarrow (b_{n-1}, \dots, b_0)$ 
6:  $X \leftarrow A_0 \times B_0$ 
7:  $Y \leftarrow A_1 \times B_1$ 
8:  $Z \leftarrow (A_0 + A_1) \times (B_0 + B_1)$ 
9: for  $i \leftarrow 0$  to  $n - 1$  do
10:    $r_i \leftarrow x_{i+n} - y_i$ 
11:    $c_i \leftarrow x_i$ 
12:    $c_{i+n} \leftarrow r_i - x_i + z_i$ 
13:    $c_{i+2n} \leftarrow y_{i+n} - r_i$ 
14:    $c_{i+3n} \leftarrow y_{i+n}$ 
15: end for
16: return  $C = (c_{2l-1}, \dots, c_0)$ 

```

Digit Size Reduction

Suppose we want to calculate $A \times B \times C$. To do that, calculate $D = A \times B$ using Algorithm 3.2.23; however, one cannot proceed to multiply $D \times C$ using Algorithm 3.2.23 since $|D| > \lceil \rho \rceil$. Thus, there is a need of an operation that shortens the size of each digit without increasing the number of digits of a sequence. In other words, we must be able to find an equivalent sequence D' , such that $|D'| \leq \lceil \rho \rceil$, so we can multiply $D' \times C$ with Algorithm 3.2.23 to get the final result.

Definition 3.2.27 (Digit Size Reduction). Given a sequence A of length l such that $|A| \leq w$, the *digit size reduction* of A , denoted as $\text{DSR}(A)$, derives an equivalent sequence A' of length l such that $A \sim A'$ and $|A'| \leq \lceil \rho \rceil$.

The in-place, sequential algorithm for digit size reduction of A works as follows. Initially, the first digit a_0 is split in two parts, the first part contains the β_0 (cf. Definition 3.2.3) least-significant bits of a_0 , and the second part has the remainder bits. Then, a_0 is updated with the first part, and the second part is added to a_1 . This process is commonly known as a *propagation of bits*, in which the most-significant bits of one digit are added to the next significant digit. After propagating a_0 , this operation is performed to the remainder digits in increasing index order. Note that the last digit will produce a number $z2^{\lceil l\rho \rceil}$, for some integer z , that must be reduced modulo p and added to the sequence A' . Algorithm 3.2.28 shows the steps to follow for digit size reduction.

Algorithm 3.2.28 Digit Size Reduction (sequential).

Input: A , a sequence of digits of length l such that $|A| \leq w$.

Output: $C = \text{DSR}(A)$, a sequence of digits of length l such that $C \sim A$ and $|C| \leq \lceil \rho \rceil$.

```

1:  $C \leftarrow A$ 
2:  $z \leftarrow 0$ 
3: for  $i \leftarrow 0$  to  $l - 1$  do
4:    $\beta_i \leftarrow \lceil (i + 1)\rho \rceil - \lceil i\rho \rceil$ 
5:    $c_i \leftarrow c_i + z$ 
6:    $z \leftarrow \lfloor c_i / 2^{\beta_i} \rfloor$ 
7:    $c_i \leftarrow c_i \bmod 2^{\beta_i}$ 
8: end for
9:  $Z \leftarrow \text{GenPolyRepr}(z2^{\lceil l\rho \rceil} \bmod p)$  //Algorithm 3.2.4
10:  $C \leftarrow C + Z$ 
11: return  $C = (c_{l-1}, \dots, c_0)$ 

```

In lines 8-9 of Algorithm 3.2.28, the value $z2^{\lceil l\rho \rceil}$ is reduced modulo p and added to the output sequence. This task is efficiently accomplished if the prime modulus has a special form that allows fast reduction. Also, for efficiency purposes, it is desirable that the reduction modulo p can be performed using a redundant representation.

Algorithm 3.2.28 exhibits sequential execution since there is a loop-carried dependency that avoids a parallel execution. Nonetheless, there exist alternative implementations of digit size reduction. For example, Algorithm 3.2.29 shows how to propagate bits independently removing the loop-carried dependency. This change increases the degree of parallelism of this operation enabling a parallel execution.

Algorithm 3.2.29 Digit Size Reduction (parallel).

Input: A , a sequence of digits of length l such that $|A| \leq k \leq w$ for some integer k .

Output: $C = \text{DSR}(A)$, a sequence of digits of length l such that $C \sim A$ and $|C| \leq \max(\lceil \rho \rceil, \lceil k - \rho \rceil) + 1$.

```

1: for  $i \leftarrow 0$  to  $l - 1$  do
2:    $\beta_i \leftarrow \lceil (i + 1)\rho \rceil - \lceil i\rho \rceil$ 
3:    $x_i \leftarrow a_i \bmod 2^{\beta_i}$ 
4:    $y_i \leftarrow \lfloor a_i / 2^{\beta_i} \rfloor$ 
5: end for
6:  $c_0 \leftarrow x_0$ 
7: for  $i \leftarrow 1$  to  $l - 1$  do
8:    $c_i \leftarrow x_i + y_{i-1}$ 
9: end for
10:  $Z \leftarrow \text{GenPolyRepr}(y_{l-1} 2^{\lceil l\rho \rceil} \bmod p)$  //Algorithm 3.2.4
11:  $C \leftarrow C + Z$ 
12: return  $C = (c_{l-1}, \dots, c_0)$ 

```

Although both methods reduce the size of digits, one of them offers a tighter bound on the size of the output sequence. Both algorithms take a sequence A such that $|A| \leq k$, where $\lceil \rho \rceil < k \leq w$. On the one hand, Algorithm 3.2.28 calculates $C \sim A$ such that $|C| \leq \lceil \rho \rceil$; this bound is guaranteed because it propagates the most significant bits of every digit sequentially. On the other hand, the size of sequence calculated by Algorithm 3.2.29 is $|C| \leq \max(\lceil \rho \rceil, \lceil k - \rho \rceil) + 1$, i.e., the size of C depends on the size of the input sequence.

3.3 Parallel Calculation of Arithmetic Operations

High-level operations, such as elliptic curve arithmetic, require to calculate many prime field operations and it is often the case that some of these operations have no data dependencies between them. This situation motivates the use of parallel execution units to reduce the execution time of high-level operations. In this section, we describe the case of parallel prime field arithmetic.

In our study case, we use vector units (described in Section 2.2) for calculating arithmetic operations in parallel. The central idea of this implementation technique is to store the digits of several prime field elements in vector registers, and to use vector instructions for calculating arithmetic operations simultaneously.

We now introduce a notation that allows us referring to prime field elements that are ready to be operated by vector instructions. Although it could be trivial in the mathematical sense, our notation is full of meaning for implementation purposes, and because of that, we formally state the following.

Definition 3.3.1 (An n -way operand). Let A_0, \dots, A_{n-1} be n field elements, where each of them is represented by l digits following Definition 3.2.6. An n -way operand, denoted as $\langle A_0, \dots, A_{n-1} \rangle$, refers to a distribution of nl digits in a set of vector registers.

Note that this distribution is not unique and is implementation-dependent due to several factors including: (i) the size of p and the value of ρ , which both determine

the number of digits l , (ii) the size of vector registers, (iii) the capabilities of vector instructions, (iv) the number of parallel calculations n .

For efficiency, it is vital to choose a distribution suitable for calculating operations as fast as possible. Regarding the value of n , it seems to be completely determined by the size of the vector registers, but this is not always the case. For example, a 128-bit vector register can be seen as a parallel unit able to process two 64-bit operations in parallel; then, a natural choice is to store $n = 2$ elements. However although this approach can be easily extended to wider vector registers, i.e., storing $n = 4$ elements on 256-bit vector registers, we will show that there are more efficient ways to use these registers. Therefore, all of these factors must be taken in consideration to find efficient distributions.

Once n -way operands are defined, let's define operations on them too.

Definition 3.3.2 (An n -way field operation). Given a k -arity operation \star , we refer to *n -way field operation* as the task of operating \star over k n -way input operands and producing one n -way output operand. It must hold that none of these operations has a data dependency with other operation, i.e. they are pair-wise independent.

Given $\langle A_0, \dots, A_{n-1} \rangle$ and $\langle B_0, \dots, B_{n-1} \rangle$, an n -way addition is calculated as

$$\langle A_0, \dots, A_{n-1} \rangle + \langle B_0, \dots, B_{n-1} \rangle \mapsto \langle A_0 + B_0, \dots, A_{n-1} + B_{n-1} \rangle, \quad (3.3.3)$$

an n -way multiplication as

$$\langle A_0, \dots, A_{n-1} \rangle \times \langle B_0, \dots, B_{n-1} \rangle \mapsto \langle A_0 \times B_0, \dots, A_{n-1} \times B_{n-1} \rangle, \quad (3.3.4)$$

and analogously, the digit size reduction (DSR) is performed as

$$\text{DSR: } \langle A_0, \dots, A_{n-1} \rangle \mapsto \langle \text{DSR}(A_0), \dots, \text{DSR}(A_{n-1}) \rangle. \quad (3.3.5)$$

To process a large number of arithmetic operations, the elements are stored in vector registers; then, parallel arithmetic operations are calculated on them; and finally, the result is converted back to a canonical representation. This implementation technique is successful whenever the performance of n -way operations is superior to calculate n single operations sequentially.

We found different trade-offs on the implementation of these operations for the prime fields of interest. In the rest of this chapter, we provide more details for each case.

3.4 Arithmetic on $\text{GF}(2^{255} - 19)$

This section shows the implementation of arithmetic operations of the prime field $\mathbb{F}_{p_{25519}}$, where $p_{25519} = 2^{255} - 19$. We report two implementations based on the following representations: using a polynomial representation setting $w = 64$, and using a redundant representation setting $\rho = 25.5$.

3.4.1 Polynomial Representation

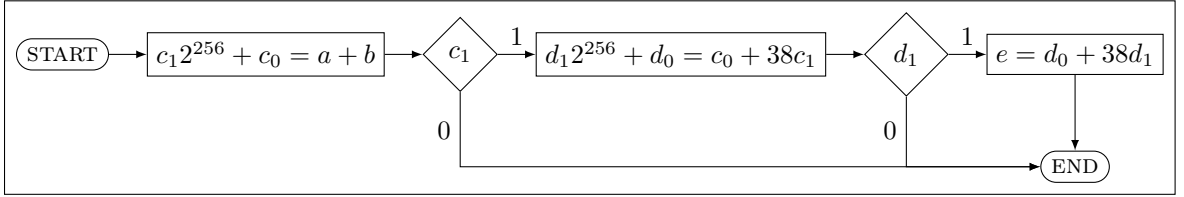
This implementation targets the use of native 64-bit instructions, so we set $w = 64$. Since p_{25519} is a number of 255 bits, we use sequences of $l = 4$ digits to represent prime field elements. Thus, a sequence $A = (a_3, a_2, a_1, a_0)$ is stored in an array of four 64-bit words.

Now, we describe implementation details of the arithmetic operations.

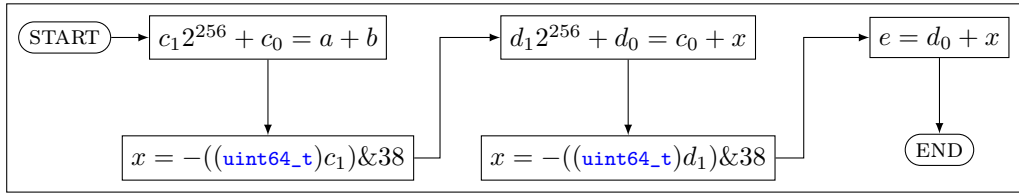
Addition using ADD and ADC Instructions

Given $A = (a_3, a_2, a_1, a_0)$ and $B = (b_3, b_2, b_1, b_0)$ be two sequences representing elements of \mathbb{F}_p , then $a = \sum_0^3 a_i 2^{64i}$ and $b = \sum_0^3 b_i 2^{64i}$ are numbers of at most 256 bits. Thus, to calculate $a + b \pmod p$, one must determine how to handle the carry bit produced by the integer addition $a + b$.

The addition of a and b , namely c , could be larger than 2^{256} . If this is the case, a carry bit will be generated, then $c = c_1 2^{256} + c_0$, where c_1 is the carry bit and c_0 are the 256 least-significant bits of c . Then, since $c_1 2^{256} \equiv 38c_1 \pmod{2^{255} - 19}$ it follows that $d = c_0 + 38c_1$. However, there are some cases in which d could be greater than 2^{256} . Observe that for $2^{256} - 38 \leq c_0 < 2^{256}$, then $2^{256} \leq d < 2^{256} + 38$, and because of that $d = d_1 2^{256} + d_0$. Once again another addition must be performed to reduce $d_1 2^{256}$, namely $e = d_0 + 38d_1$. Note that e is never large than 2^{256} , since d_0 is a small number, more specifically $0 \leq d_0 < 38$. Figure 3.4.1a shows graphically this description.



(a) Non-constant time algorithm.



(b) Constant time algorithm.

Figure 3.4.1: Calculation of additions modulo p_{25519} .

The calculation of additions modulo p_{25519} as described in Figure 3.4.1a exhibits an irregular execution pattern. Observe that there are early termination conditions (when the carry bits are 0) that cause the addition could take different amount of time depending on the value of the inputs. For this reason, we must implement additions in such a way the program executes always the same number of operations regardless the value of the carry bits.

To achieve a regular execution pattern, the addition must always add either 0 or 38 depending on the value of the carry bit. Given $b \in \{0, 1\}$, we must construct a 64-bit

word containing either $x = 38$ if $b = 1$, or $x = 0$ otherwise. We can use the following C code to accomplish this task.

```
1 x = -((uint64_t)b) & 38;
```

Alternatively, the same task can be performed using a conditional move instruction `CMOV`.

```
1 mov $0, ax
2 mov $38, cx
3 cmovc ax, cx
```

The `CMOV` instruction will assign 38 to `CX` if the carry bit from the `FLAGS` register is set, otherwise, `CX` is not modified.

Either alternative allows that the addition of integers modulo p_{25519} is performed using a constant time algorithm. As shown in Figure 3.4.1b, a regular execution pattern is achieved by removing the conditionals and always executing the same number of operations. Specifically, the first two additions take eight `ADC` instructions and one more to calculate e ; in total, adding two 256-bit numbers takes nine `ADC` instructions.

Additions using `ADX` Instructions

Additions modulo p_{25519} can be calculated using the `ADX` instructions. The main change that is noticed in the use of the `ADCX/ADOX` instructions is that they always calculate the addition with carry using the carry/overflow bit from the `FLAGS` register. Because of that, these bits must be cleared before starting a series of additions.

The instruction `CLC` sets the `CF` bit to 0; unfortunately, there is no analogous instruction for clearing the `OF` bit. Then, it is required an operation that produces 0 as output setting the destination register to 0, which clears both the `CF` and `OF` flags. One way to perform that is through the `XOR` instruction.

```
1 xor ax, ax
```

Also, it can be used the `SUB` instruction.

```
1 sub ax, ax
```

There are other instructions that have the same effect.

Although there are required extra instructions for clearing bits, these instructions are not executed entirely. Instruction idioms are instructions for which the result of the operation is known in advance. Some advanced processors recognize these instruction idioms and perform the minimum micro-operations to simulate the effect caused by them. Thus, the overhead of adding instruction idioms is negligible.

Subtraction

The method used to subtract two 256-bit numbers mimics the procedure followed by addition. For subtractions, the addition with carry (`ADC` instructions) is replaced by subtraction with borrow (`SBB` instructions).

Integer Multiplication using MULX and ADX Instructions

One feature of the MULX instruction is that allows to specify the destination registers that store the product calculated. Another relevant feature of this instruction is that it does not overwrite the input RDX register. Thus, once RDX was loaded with some value, this register can be used by subsequent multiplications, reducing the number of loads from memory. This operation is particularly useful in the operand-scanning multiplication method, where one digit of the first operand is multiplied by all the digits from the second operand. For this reason, we applied the operand-scanning method to calculate integer multiplications.

The implementation of integer multiplications of 256-bit integers, a and b , follows Algorithm 3.2.13, which multiplies sequences of digits A and B of length four. The first for-loop schedules four MULX instructions, which calculate the values $(x_i, y_i) \leftarrow a_0 \times b_i$ for $0 \leq i < 4$ resulting in eight registers, where the 64 most-significant bits of each product are propagated resulting in five 64-bit words representing the number $a_0 \times b$

$$(d_{0,4}, d_{0,3}, d_{0,2}, d_{0,1}, d_{0,0}) \leftarrow (x_3, x_2, x_1, x_0, 0) + (0, y_3, y_2, y_1, y_0).$$

Then, $c_0 \leftarrow d_{0,0}$ is stored in memory since no more operations are performed with it, and the last four words remain in registers to be used later. The second for-loop of Algorithm 3.2.13 iterates over the words a_j , for $1 \leq j < 4$, to calculate $(x_i, y_i) \leftarrow a_j \times b_i$ for $0 \leq i < 4$ and from these words, it obtains

$$(d_{j,4}, d_{j,3}, d_{j,2}, d_{j,1}, d_{j,0}) \leftarrow (x_3, x_2, x_1, x_0, 0) + (y_4, y_3, y_2, y_1, y_0).$$

After that, these words must be added to the words obtained in the previous iteration as

$$(d_{j,4}, d_{j,3}, d_{j,2}, d_{j,1}, d_{j,0}) \leftarrow (d_{j,4}, d_{j,3}, d_{j,2}, d_{j,1}, d_{j,0}) + (0, d_{j-1,4}, d_{j-1,3}, d_{j-1,2}, d_{j-1,1}).$$

Like before, $c_j \leftarrow d_{j,0}$ is stored in memory since no more operations are performed on it. After performing the second for-loop, the words $(c_7, c_6, c_5, c_4) \leftarrow (d_{3,4}, d_{3,3}, d_{3,2}, d_{3,1})$ are stored in memory completing the calculation of the integer multiplication.

We applied several optimizations to improve the performance of this implementation. First, we maintain all the intermediate values in registers; otherwise, some overheads appear by spilling some registers to memory. Fortunately, it was possible to store intermediate products in registers since the length of the sequences is only four digits.

A second optimization applied relies on observing that the execution of MULX instructions does not interfere with the calculation of addition instructions. Hence, we can schedule addition instructions in between of a series of multiplication instructions. Observe that once the products $(x_i, y_i) \leftarrow a_j \times b_i$ and $(x_{i+1}, y_{i+1}) \leftarrow a_j \times b_{i+1}$ have been calculated, the addition $x_i + y_{i+1}$ can be scheduled before to the execution of the product $(x_{i+2}, y_{i+2}) \leftarrow a_j \times b_{i+2}$. This scheduling can be repeated for the subsequent calculations; thus, the addition and multiplication instructions get interleaved. This schedule of instructions runs faster because multiplication instructions do not alter the carry bit used by additions. Be aware that this optimization is only valid by using the MULX instruc-

tion, which (unlike the IMULQ/MULQ instructions) does not modify the bits of the FLAGS register that are used by the addition instructions.

A third optimization is derived from the second one when using the ADX instructions. The ADCX and ADOX instructions calculate two multi-precision additions without interference between them. That is, two series of additions with carry can be performed simultaneously as they use, respectively, the CF and OF bits as the carry bit. This is useful in the second for-loop, where one multi-precision addition propagates the 64 most-significant bits of $(x_i, y_i) \leftarrow a_j \times b_i$, and another one accumulates the values $d_{j,i} \leftarrow d_{j,i} + d_{j,i-1}$. Thus, these additions are calculated using a series of ADCX and ADOX instructions and they are scheduled between MULX instructions. This optimization is only valid by using the MULX instruction in conjunction with the ADX instructions.

These optimizations lead to three implementations of the 256-bit integer multiplier. Table 3.4.2 summarizes the instruction counts of them. By using MULX, the number of move instructions is reduced significantly allowing a better utilization of the registers. On the other hand, using ADCX/ADOX instructions requires some additional move instructions to set registers to zero, but without modifying the FLAGS register; thus, explicit MOV instructions are used instead of instruction idioms, like the XOR instruction.

Table 3.4.2: Instruction counts for 256-bit integer multiplication.

Implementation	Multiplication	Addition	Load	Store	Move
MULQ + ADC	16	34	20	8	24
MULX + ADC	16	31	20	8	0
MULX + ADX	16	31	20	8	7

Reduction Modulo $p = 2^{255} - 19$

After calculating integer multiplications a sequence of eight digits is generated, which must be reduced modulo p to obtain a shorter sequence of length four. Let $A = (a_7, \dots, a_0)$ be the sequence to be reduced, it holds that $(a_7, a_6, a_5, a_4, 0, 0, 0, 0) \sim (38) \times (a_7, a_6, a_5, a_4)$ since $2^{256} \equiv 38 \pmod{p_{25519}}$. Thus, the reduction calculates

$$(c_4, c_3, c_2, c_1, c_0) \leftarrow (a_3, a_2, a_1, a_0) + (38) \times (a_7, a_6, a_5, a_4),$$

after that, c_4 must be reduced modulo p_{25519} applying the same strategy

$$(d_4, d_3, d_2, d_1, d_0) \leftarrow (c_3, c_2, c_1, c_0) + (38) \times (c_4).$$

However, this last addition could generate a carry bit $d_4 \in \{0, 1\}$. When $a_i = 2^{64} - 1$ for $0 \leq i < 8$, it follows that $c_4 = 38$; and $d_4 = 1$, if $2^{256} - 38^2 \leq k < 2^{256}$ for $k = \sum_{i=0}^3 c_i 2^{64i}$; otherwise $d_4 = 0$. Hence, the reduction of A modulo p_{25519} is

$$A' = (d_3, d_2, d_1, d_0 + 38d_4),$$

where the addition $d_0 + 38d_4$ does not generate carry regardless the value of d_4 . All these operations must be implemented in constant time.

We also provide three implementations depending whether the MULX and ADX instructions are available in the target machine. In the implementations that use MULX, the reduction procedure uses five multiplications, fourteen additions, one conditional move, eight loads, and four store instructions.

Multiplicative Inverse

The multiplicative inverse of an element in $a \in \mathbb{F}_{2^{255-19}}$ is given as

$$a^{-1} = a^{2^{255}-21} = a^{2^{255}-32+11} = \left(a^{2^{250}-1}\right)^{2^5} a^{11} = (\alpha_{250})^{2^5} a^{11}. \quad (3.4.3)$$

The exponentiation $a^{2^{250}-1}$ is calculated using the Itoh-Tsujii's method [156] as follows. Let $\alpha_x = a^{2^x-1}$, we calculate α_{250} looking for an addition chain (c_1, \dots, c_s) such that $c_1 = 5$ and $c_s = 250$. We started from α_5 instead of α_1 , since some intermediate values used to calculate a^{11} can be shared to obtain $\alpha_5 = a^{31}$ at the cost of three multiplications and four squares. The addition chain for α_{250} is shown in the following table.

i	c_x	c_y	$c_i = c_x + c_y$	$S \leftarrow S \cup \{\alpha_{c_i} = (\alpha_{c_x})^{2^{c_y}} \alpha_{c_y}\}$
-	-	-	5	$\{\alpha_5\}$
1	5	5	10	$\{\alpha_5, \alpha_{10}\}$
2	10	10	20	$\{\alpha_5, \alpha_{10}, \alpha_{20}\}$
3	20	20	40	$\{\alpha_5, \alpha_{10}, \alpha_{20}, \alpha_{40}\}$
4	40	10	50	$\{\alpha_5, \alpha_{10}, \alpha_{20}, \alpha_{40}, \alpha_{50}\}$
5	50	50	100	$\{\alpha_5, \alpha_{10}, \alpha_{20}, \alpha_{40}, \alpha_{50}, \alpha_{100}\}$
6	100	100	200	$\{\alpha_5, \alpha_{10}, \alpha_{20}, \alpha_{40}, \alpha_{50}, \alpha_{100}, \alpha_{200}\}$
7	200	50	250	$\{\alpha_5, \alpha_{10}, \alpha_{20}, \alpha_{40}, \alpha_{50}, \alpha_{100}, \alpha_{200}, \alpha_{250}\}$

In each row, the algorithm includes $\alpha_{c_i} = (\alpha_{c_x})^{2^{c_y}} \alpha_{c_y}$ to S taking one multiplication and c_y squares. So, calculating a multiplicative inverse takes 11 multiplications and 254 squares.

3.4.2 Redundant Representation

The redundant representation as stated in Definition 3.2.6 requires to select $\rho < w$, then we will describe some conditions that allows to select a proper value for ρ .

First of all, we assume that $w = 64$, since the vector registers can operate up to four 64-bit integer operations simultaneously. However, the PMULUDQ instruction multiplies 32-bit integers, and for this reason, we must consider that $w' = 32$. Based on these parameters and relying on the relations given in Equations (3.2.21) and (3.2.22), we are free to chose any ρ value holding $\rho \leq 29$.

We explored across different values for ρ . For instance, by setting $\rho = 26$ and $l = 10$, the procedure that performs the reduction modulo p_{25519} requires more multiplications by powers of two. Note that after multiplication some digits have 2^{260} as a factor. This factor can be replaced by 19×2^5 after reduction modulo p_{25519} ; however, these extra five

bits increase the size of the output digits. Alternatively, by setting $\rho = \frac{255}{9}$ and $l = 9$, the integer multiplication is calculated with fewer digit multiplications. Although that $\lceil \frac{255}{9} \rceil \leq 29$ satisfying the condition above, this choice reduces the extra room used for carry bits, and as a consequence, the digit products will overflow registers during the reduction modulo p_{25519} . We will see it is more efficiently to set $l = 10$, because any digit with factor 2^{255} , it will have factor 19, which is the shortest factor.

Setting $\rho = \frac{255}{10} = 25.5$ and $l = 10$ leads to work with sequences of ten digits, which are stored in five 128-bit vector registers. Figure 3.4.4 shows that each of these registers stores two digits in a 64-bit word. As the size of digits is at most 26 bits, then the red part represents the room of space until reaching 32 bits, and the blue part represents the remainder 32 bits. Due to ρ is not integer, the size of the digits is not uniform. The size is 26 bits for even-indexed digits and is 25 bits for odd-indexed digits.

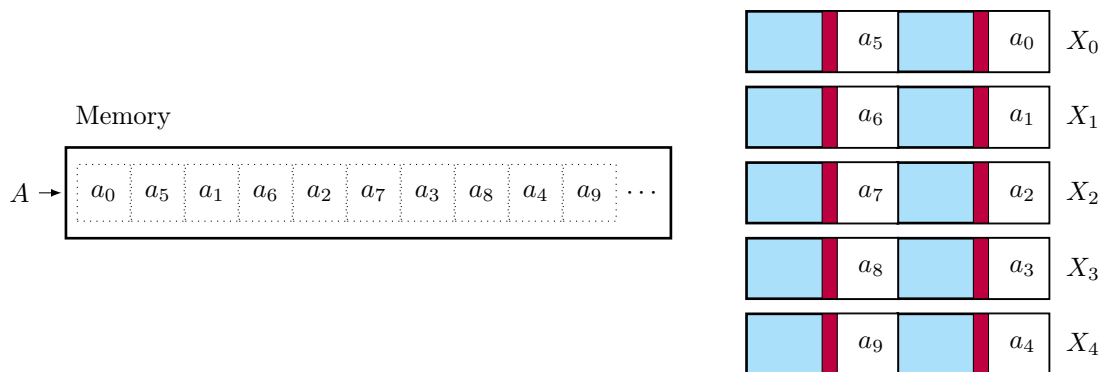


Figure 3.4.4: A sequence of digits $A = (a_9, \dots, a_0)$ representing $a \in \mathbb{F}_{p_{25519}}$ is loaded from memory into five 128-bit registers X_0, \dots, X_4 .

Addition

The addition of sequences must be performed digit-by-digit, then we use three PADDQ instructions to each digit from the input sequences. Each digit has several bits for storing the carry bits produced by additions. In this case, no reduction is performed, since the size of the digits will increase in at most one bit. Hence, several additions can be performed consecutively without overflowing registers. Let A and B be sequences stored in five 128-bit registers, we use five PADDQ instructions to calculate $C = A + B$ as

$$\begin{aligned} [c_5 \ c_0] &\leftarrow [a_5 \ a_0] + [b_5 \ b_0] \\ [c_6 \ c_1] &\leftarrow [a_6 \ a_1] + [b_6 \ b_1] \\ [c_7 \ c_2] &\leftarrow [a_7 \ a_2] + [b_7 \ b_2] \\ [c_8 \ c_3] &\leftarrow [a_8 \ a_3] + [b_8 \ b_3] \\ [c_9 \ c_4] &\leftarrow [a_9 \ a_4] + [b_9 \ b_4]. \end{aligned}$$

Subtraction

Replacing the PADDQ by PSUBQ instructions, we can perform subtraction between sequences of digits; however, the digits could be lesser than zero and according to the redundant representation (see Definition 3.2.6) all the digits must be positive.

One way to subtract sequences and producing positive digits is by adding a multiple of p_{25519} . Let A and B be the sequences to be subtracted, we perform $C = A - B + P$, where P is a redundant sequence of p_{25519} . We carefully select a sequence P such that $|c_i| \geq 0$ for $0 \leq i < 10$. If we assume that $|A|, |B| \leq \lceil \rho \rceil = 26$, then $|P| = 27$ and

$$P = (0x3fffffe, 0x7fffffe, 0x3fffffe, 0x7fffffe, 0x3fffffe, \quad (3.4.5) \\ 0x7fffffe, 0x3fffffe, 0x7fffffe, 0x3fffffe, 0x7ffffda).$$

Multiplication

Since p_{25519} is a pseudo-Mersenne prime, the reduction can be merged with the integer multiplication resulting in a single procedure for calculating prime field multiplications. Let A and B be two sequences of ten digits, $C = A \times B = (c_9, \dots, c_0)$ is calculated as

$$\begin{aligned} c_0 &\leftarrow a_0b_0 + 38a_9b_1 + 19a_8b_2 + 38a_7b_3 + 19a_6b_4 + 38a_5b_5 + 19a_4b_6 + 38a_3b_7 + 19a_2b_8 + 38a_1b_9 \\ c_1 &\leftarrow a_1b_0 + a_0b_1 + 19a_9b_2 + 19a_8b_3 + 19a_7b_4 + 19a_6b_5 + 19a_5b_6 + 19a_4b_7 + 19a_3b_8 + 19a_2b_9 \\ c_2 &\leftarrow a_2b_0 + 2a_1b_1 + a_0b_2 + 38a_9b_3 + 19a_8b_4 + 38a_7b_5 + 19a_6b_6 + 38a_5b_7 + 19a_4b_8 + 38a_3b_9 \\ c_3 &\leftarrow a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 + 19a_9b_4 + 19a_8b_5 + 19a_7b_6 + 19a_6b_7 + 19a_5b_8 + 19a_4b_9 \\ c_4 &\leftarrow a_4b_0 + 2a_3b_1 + a_2b_2 + 2a_1b_3 + a_0b_4 + 38a_8b_5 + 19a_8b_6 + 38a_7b_7 + 19a_6b_8 + 38a_5b_9 \\ c_5 &\leftarrow a_5b_0 + a_4b_1 + a_3b_2 + a_2b_3 + a_1b_4 + a_0b_5 + 19a_9b_6 + 19a_8b_7 + 19a_7b_8 + 19a_6b_9 \\ c_6 &\leftarrow a_6b_0 + 2a_5b_1 + a_4b_2 + 2a_3b_3 + a_2b_4 + 2a_1b_5 + a_0b_6 + 38a_9b_7 + 19a_8b_8 + 38a_7b_9 \\ c_7 &\leftarrow a_7b_0 + a_6b_1 + a_5b_2 + a_4b_3 + a_3b_4 + a_2b_5 + a_1b_6 + a_0b_7 + 19a_9b_8 + 19a_8b_9 \\ c_8 &\leftarrow a_8b_0 + 2a_7b_1 + a_6b_2 + 2a_5b_3 + a_4b_4 + 2a_3b_5 + a_2b_6 + 2a_1b_7 + a_0b_8 + 38a_9b_9 \\ c_9 &\leftarrow a_9b_0 + a_8b_1 + a_7b_2 + a_6b_3 + a_5b_4 + a_4b_5 + a_3b_6 + a_2b_7 + a_1b_8 + a_0b_9. \end{aligned} \quad (3.4.6)$$

In this formulation, the intermediate products $a_i b_j$ are accumulated in c_k for $0 \leq i, j < 10$ and $k = i + j \bmod 10$. We identify three cases in this computation:

- When $i + j \geq 10$, the products $a_i b_j$ are multiplied by 19 as they had 2^{255} as a factor.
- When both i, j are odd, the products $a_i b_j$ are multiplied by two. To see this, note that there exist integers q, r such that $i = 2q + 1$ and $j = 2r + 1$, and the following is true

$$\begin{aligned} a_i 2^{\lceil 25.5i \rceil} \times b_j 2^{\lceil 25.5j \rceil} &= a_i b_j 2^{\lceil 25.5i \rceil + \lceil 25.5j \rceil} \\ &= a_i b_j 2^{\lceil 25.5(2q+1) \rceil + \lceil 25.5(2r+1) \rceil} \\ &= 2a_i b_j 2^{51(q+r \bmod 10) + 51}. \end{aligned} \quad (3.4.7)$$

The c_k digit represents the number

$$\begin{aligned} c_k 2^{\lceil 25.5k \rceil} &= c_k 2^{\lceil 25.5(i+j \bmod 10) \rceil} \\ &= c_k 2^{\lceil 25.5(2q+1+2r+1 \bmod 10) \rceil} \\ &= c_k 2^{51(q+r \bmod 10) + 51}. \end{aligned} \quad (3.4.8)$$

These equations explain the multiplication by two, which appears because ρ is not an integer.

- The remainder cases, products have no factor and are directly accumulated in c_k .

Some products are multiplied by 38 because the first two cases apply. Equation (3.4.6) takes 100 digit multiplications, 30 multiplications by 19, 15 multiplications by 38, and 10 multiplications by two. The size of C is bounded as $|C| \leq 2\lceil\rho\rceil + |l| + |38| = 61$ bits. Thus, there is still room (three bits) on the top of 64-bit registers to perform additions to C before overflowing the register.

We analyzed efficient ways for implementing prime field multiplication using vector instructions. In particular, we look for an algorithm that produces a large number of independent multiplications in order to take advantage of the high throughput of the PMULUDQ instruction.

The Mastrovito multiplier [187] is a hardware technique used for multiplication over binary field extensions, and in this context, the central idea of this algorithm is to calculate a series of bit multiplications simultaneously. To that end, at every iteration one of the inputs is multiplied by x and reduced modulo $f(x)$, where $f(x)$ is the irreducible polynomial defining the extension field. Inspired by the design of Mastrovito multiplier, we adapted it for computing a prime field multiplications as described next.

Let A and B defined as above. We perform $C = A \times B$ as a vector-matrix product $\text{Circ}(A) \times B$ such that B and C are column vectors and $\text{Circ}(A)$ is a circulant matrix generated by a column-vector A as

$$\text{Circ}(A) = [A, \pi(A), \pi^2(A), \dots, \pi^{l-1}(A)], \quad (3.4.9)$$

where π is a function defined as

$$(a_{l-1}, \dots, a_0)^T \mapsto (a_{l-2}, \dots, a_0, 0)^T + (a_{l-1} \bmod p)^T. \quad (3.4.10)$$

Then, the product sequence C is calculated as

$$C = A \times [b_0] + \pi(A) \times [b_1] + \dots + \pi^{l-1}(A) \times [b_{l-1}]. \quad (3.4.11)$$

This algorithm iterates over the digits of B (like in the operand-scanning method) to multiply them with every digit of A using l independent multiplications. After that, A is updated using the π function, which shifts the digits of A by one position to the next power of two, and the last digit is reduced modulo p . This process is repeated until all the digits of B have been multiplied. See Algorithm 3.4.12 for a formal description.

Algorithm 3.4.12 Adaptation of Mastrovito's Algorithm for Prime Field Multiplication using Redundant Representation.

Input: A and B , two sequences of digits of length l and size $|A|, |B| \leq \lceil\rho\rceil$.

Output: $C = A \times B$, a sequence of digits of length $2l - 1$ and size $|C| < |l| + 2\lceil\rho\rceil + 1$.

- 1: $C \leftarrow A \times (b_0)$
 - 2: **for** $i \leftarrow 1$ **to** $l - 1$ **do**
 - 3: $A \leftarrow \pi(A)$
 - 4: $C \leftarrow C + A \times (b_i)$
 - 5: **end for**
 - 6: **return** $C = (c_{l-1}, \dots, c_0)$
-

Let's instantiate Algorithm 3.4.12 with p_{25519} . First, ten digit multiplications are used to process $A \times [b_0]$. After that, at every iteration of the loop, this algorithm schedules ten digit multiplications and ten digit additions to accumulate $A \times [b_i]$ into C . Notice that each digit multiplication is independent to each other, like in the Mastrovito multiplier; hence, these multiplications can be executed as a series of consecutive PMULUDQ instructions. We note that scheduling independent instructions is favorable for reducing the latency of the prime field multiplication.

The role of π is to shift the position of the digits of one operands, and then, to reduce modulo p . For p_{25519} , the π function is defined as

$$(a_9, a_8, a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)^T \mapsto (a_8, a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0, 19a_9)^T. \quad (3.4.13)$$

It is evident that π can be computed in-place by using permutation instructions and one multiplication by 19. Moreover, the digits of A can be reordered through register renaming, which is a task that is usually performed by the compiler at a negligible cost.

One issue that arises by using $\rho = 25.5$ is that some intermediate products must be multiplied by two. Unfortunately, the Mastrovito multiplier does not calculate these multiplications. We show two approaches that solve this issue. First, we can use two independent accumulators for C ; thus, one of them is used to store the products that must be multiplied by two; and at the end of the execution of the Mastrovito multiplier, these two accumulators are added to obtain C . A second approach is to multiply the digits of A either by b_i or $2b_i$ according to the index value.

Now, we describe the implementation of the prime field multiplication using 128-bit registers. Due to the issue described above, we implemented the Mastrovito multiplier using two accumulators X and Y , which are calculated from A and B as Figure 3.4.14 shows. The calculation of π over the five registers representing A requires to permute and multiply by 19 the first register, meanwhile the remaining registers are not modified. After that, some products of Y must be multiplied by two, and we perform these multiplications using bit shifts (\ll). Thus, the sequence $C = A \times B$ is calculated as

$$\begin{aligned} [c_5 \ c_0] &\leftarrow [x_5 \ x_0] + [y_5 \ y_0] \ll [0 \ 1] \\ [c_6 \ c_1] &\leftarrow [x_6 \ x_1] + [y_6 \ y_1] \ll [1 \ 0] \\ [c_7 \ c_2] &\leftarrow [x_7 \ x_2] + [y_7 \ y_2] \ll [0 \ 1] \\ [c_8 \ c_3] &\leftarrow [x_8 \ x_3] + [y_8 \ y_3] \ll [1 \ 0] \\ [c_9 \ c_4] &\leftarrow [x_9 \ x_4] + [y_9 \ y_4] \ll [0 \ 1]. \end{aligned}$$

Hence, a prime field multiplication takes 50 multiplications (PMULUDQ), 45 additions (PADDQ), 5 bit shifts (PSLLI), and 9 multiplications by 19. The latter multiplications can be performed either using two additions and two shifts to the left, i.e., $19x = (x \ll 4) + (x \ll 2) + x$; or using one PMULUDQ instruction. In our experiments, it's faster to perform an extra multiplication because the instruction has a high-throughput when is combined with the other multiplications.

$i = 0$ $[x_5 \ x_0] \leftarrow [a_5 \ a_0] \times [b_0 \ b_0]$ $[x_6 \ x_1] \leftarrow [a_6 \ a_1] \times [b_0 \ b_0]$ $[x_7 \ x_2] \leftarrow [a_7 \ a_2] \times [b_0 \ b_0]$ $[x_8 \ x_3] \leftarrow [a_8 \ a_3] \times [b_0 \ b_0]$ $[x_9 \ x_4] \leftarrow [a_9 \ a_4] \times [b_0 \ b_0] .$	$i = 1$ $[y_5 \ y_0] \leftarrow [a_4 \ 19a_9] \times [b_1 \ b_1]$ $[y_6 \ y_1] \leftarrow [a_5 \ a_0] \times [b_1 \ b_1]$ $[y_7 \ y_2] \leftarrow [a_6 \ a_1] \times [b_1 \ b_1]$ $[y_8 \ y_3] \leftarrow [a_7 \ a_2] \times [b_1 \ b_1]$ $[y_9 \ y_4] \leftarrow [a_8 \ a_3] \times [b_1 \ b_1] .$
$i = 2$ $[x_5 \ x_0] += [a_3 \ 19a_8] \times [b_2 \ b_2]$ $[x_6 \ x_1] += [a_4 \ 19a_9] \times [b_2 \ b_2]$ $[x_7 \ x_2] += [a_5 \ a_0] \times [b_2 \ b_2]$ $[x_8 \ x_3] += [a_6 \ a_1] \times [b_2 \ b_2]$ $[x_9 \ x_4] += [a_7 \ a_2] \times [b_2 \ b_2] .$	$i = 3$ $[y_5 \ y_0] += [a_2 \ 19a_7] \times [b_3 \ b_3]$ $[y_6 \ y_1] += [a_3 \ 19a_8] \times [b_3 \ b_3]$ $[y_7 \ y_2] += [a_4 \ 19a_9] \times [b_3 \ b_3]$ $[y_8 \ y_3] += [a_5 \ a_0] \times [b_3 \ b_3]$ $[y_9 \ y_4] += [a_6 \ a_1] \times [b_3 \ b_3] .$
$i = 4$ $[x_5 \ x_0] += [a_1 \ 19a_6] \times [b_4 \ b_4]$ $[x_6 \ x_1] += [a_2 \ 19a_7] \times [b_4 \ b_4]$ $[x_7 \ x_2] += [a_3 \ 19a_8] \times [b_4 \ b_4]$ $[x_8 \ x_3] += [a_4 \ 19a_9] \times [b_4 \ b_4]$ $[x_9 \ x_4] += [a_5 \ a_0] \times [b_4 \ b_4] .$	$i = 5$ $[y_5 \ y_0] += [a_0 \ 19a_5] \times [b_5 \ b_5]$ $[y_6 \ y_1] += [a_1 \ 19a_6] \times [b_5 \ b_5]$ $[y_7 \ y_2] += [a_2 \ 19a_7] \times [b_5 \ b_5]$ $[y_8 \ y_3] += [a_3 \ 19a_8] \times [b_5 \ b_5]$ $[y_9 \ y_4] += [a_4 \ 19a_9] \times [b_5 \ b_5] .$
$i = 6$ $[x_5 \ x_0] += [19a_9 \ 19a_4] \times [b_6 \ b_6]$ $[x_6 \ x_1] += [a_0 \ 19a_5] \times [b_6 \ b_6]$ $[x_7 \ x_2] += [a_1 \ 19a_6] \times [b_6 \ b_6]$ $[x_8 \ x_3] += [a_2 \ 19a_7] \times [b_6 \ b_6]$ $[x_9 \ x_4] += [a_3 \ 19a_8] \times [b_6 \ b_6] .$	$i = 7$ $[y_5 \ y_0] += [19a_8 \ 19a_3] \times [b_7 \ b_7]$ $[y_6 \ y_1] += [19a_9 \ 19a_4] \times [b_7 \ b_7]$ $[y_7 \ y_2] += [a_0 \ 19a_5] \times [b_7 \ b_7]$ $[y_8 \ y_3] += [a_1 \ 19a_6] \times [b_7 \ b_7]$ $[y_9 \ y_4] += [a_2 \ 19a_7] \times [b_7 \ b_7] .$
$i = 8$ $[x_5 \ x_0] += [19a_7 \ 19a_2] \times [b_8 \ b_8]$ $[x_6 \ x_1] += [19a_8 \ 19a_3] \times [b_8 \ b_8]$ $[x_7 \ x_2] += [19a_9 \ 19a_4] \times [b_8 \ b_8]$ $[x_8 \ x_3] += [a_0 \ 19a_5] \times [b_8 \ b_8]$ $[x_9 \ x_4] += [a_1 \ 19a_6] \times [b_8 \ b_8] .$	$i = 9$ $[y_5 \ y_0] += [19a_6 \ 19a_1] \times [b_9 \ b_9]$ $[y_6 \ y_1] += [19a_7 \ 19a_2] \times [b_9 \ b_9]$ $[y_7 \ y_2] += [19a_8 \ 19a_3] \times [b_9 \ b_9]$ $[y_8 \ y_3] += [19a_9 \ 19a_4] \times [b_9 \ b_9]$ $[y_9 \ y_4] += [a_0 \ 19a_5] \times [b_9 \ b_9] .$

Figure 3.4.14: Scheduling of 128-bit vector instructions to calculate prime field multiplications.

Digit Size Reduction

Given a sequence C such that $|C| \geq \lceil \rho \rceil$ we want to obtain an equivalent sequence D such that $C \sim D$ and $|D| \leq \lceil \rho \rceil$. In the previous section, we presented two algorithms for implementing this operation, and in this section we describe their implementation using vector registers.

Algorithm 3.2.28 splits each digit in two parts adding the most significant part to the next digit. This propagation of bits starts from the first digit and continues along the digits of the sequence. At the end, one digit is generated, which must be reduced modulo p_{25519} and added to the output sequence.

For the implementation of Algorithm 3.2.28, we assume that the sequence C is stored into five 128-bit registers. Given a digit c_i , we calculate $x_i \leftarrow c_i \bmod 2^{\beta_i}$ and $y_i \leftarrow \lfloor c_i / 2^{\beta_i} \rfloor$ using one mask (PAND instruction) and one right bit-shift (PSRLI instruction) on c_i . Note that, for example, the propagation of digits from the digit c_0 to c_1 happens at the same time as the propagation of bits of the digit c_5 to c_6 ; and continuing analogously, we propagate until reaching the last pair of digits $[c_9, c_5]$ as follows

$$\begin{array}{ccc}
 \begin{array}{c} i = 0, i = 5 \\ [c_5 \ c_0] \rightarrow [x_5 \ x_0] \ [y_5 \ y_0] \\ [c_6 \ c_1] \\ \hline [d_5 \ d_0] \ [c_6 \ c_1]. \end{array} &
 \begin{array}{c} i = 1, i = 6 \\ [c_6 \ c_1] \rightarrow [x_6 \ x_1] \ [y_6 \ y_1] \\ [c_7 \ c_2] \\ \hline [d_6 \ d_1] \ [c_7 \ c_2]. \end{array} &
 \begin{array}{c} i = 2, i = 7 \\ [c_7 \ c_2] \rightarrow [x_7 \ x_2] \ [y_7 \ y_2] \\ [c_8 \ c_3] \\ \hline [d_7 \ d_2] \ [c_8 \ c_3]. \end{array} \\
 \\
 \begin{array}{c} i = 3, i = 8 \\ [c_8 \ c_3] \rightarrow [x_8 \ x_3] \ [y_8 \ y_3] \\ [c_9 \ c_4] \\ \hline [d_8 \ d_3] \ [c_9 \ c_4]. \end{array} &
 \begin{array}{c} i = 4, i = 9 \\ [c_9 \ c_4] \rightarrow [x_9 \ x_4] \ [y_9 \ y_4] \\ \hline [d_9 \ d_4] \ . \end{array} &
 \end{array}$$

As a result of this last propagation, the pair $[y_9, y_4]$ of new digits is generated. Then, y_9 is multiplied by 19 (reduced modulo p_{25519}) and added to d_0 ; and y_4 is added to d_5 .

$$[d_5 \ d_0] \leftarrow [d_5 \ d_0] + [y_4 \ 19y_9] .$$

Finally, we propagate d_0 to d_1 and also d_5 to d_6 .

$$\begin{array}{c} [d_5 \ d_0] \rightarrow [x'_5 \ x'_0] \ [y'_5 \ y'_0] \\ [d_6 \ d_1] \\ \hline [d_5 \ d_0] \ [d_6 \ d_1] . \end{array}$$

Although this implementation is processed sequentially, we still benefit from the use of vector registers by performing the propagation of two digits at each step.

In contrast to Algorithm 3.2.28, Algorithm 3.2.29 calculates more operations in parallel. The parallel algorithm calculates the values x_i and y_i from the input digits, however, these values can be calculated simultaneously. Then, the algorithm propagates the most significant bits to the next digit. Here, y_9 must be reduced modulo p_{25519} and added to the output sequence.

The implementation of Algorithm 3.2.29 assumes that C is stored into five 128-bit registers. First, the values x_i and y_i are calculated from the digits c_i for all $0 \leq i < 10$ as

$$\begin{aligned}
[c_5 \ c_0] &\rightarrow [x_5 \ x_0] \ [y_5 \ y_0] \\
[c_6 \ c_1] &\rightarrow [x_6 \ x_1] \ [y_6 \ y_1] \\
[c_7 \ c_2] &\rightarrow [x_7 \ x_2] \ [y_7 \ y_2] \\
[c_8 \ c_3] &\rightarrow [x_8 \ x_3] \ [y_8 \ y_3] \\
[c_9 \ c_4] &\rightarrow [x_9 \ x_4] \ [y_9 \ y_4] .
\end{aligned}$$

Then, we propagate the most significant bits in each digit and assign $d_0 \leftarrow x_0$ and $d_5 \leftarrow x_5$.

$$\begin{array}{cccccc}
[x_5 \ x_0] & [y_5 \ y_0] & & & & \\
& [x_6 \ x_1] & [y_6 \ y_1] & & & \\
& & [x_7 \ x_2] & [y_7 \ y_2] & & \\
& & & [x_8 \ x_3] & [y_8 \ y_3] & \\
& & & & [x_9 \ x_4] & [y_9 \ y_4] \\
\hline
[d_5 \ d_0] & [d_6 \ d_1] & [d_7 \ d_2] & [d_8 \ d_3] & [d_9 \ d_4] & .
\end{array}$$

Finally, we accumulate y_4 into d_5 , and $19y_9$ into d_0 .

$$[d_5 \ d_0] \leftarrow [d_5 \ d_0] + [y_4 \ 19y_9] .$$

This procedure runs faster than the previous one because most of the operations do not have dependencies between them. In both, implementations, the multiplication by 19 was performed using two bit shifts and two additions. The following table shows the operation counts of these two implementations.

Implementation	Addition (PADDQ)	Bit Shift (PSLLI/PSRLI)	Logic (PAND)
Algorithm 3.2.28	8	8	6
Algorithm 3.2.29	7	7	5

Performance Comparison between Representations

We measured the latency of the arithmetic operations and compared the timings of both implementations. Table 3.4.15 shows the performance timings measured on Skylake.

Table 3.4.15: Time in clock cycles of $\mathbb{F}_{p_{25519}}$ operations measured on Skylake.

Implementation	ISA	A	M	S	Digit Size Reduction		I
					Sequential	Parallel	
Radix-2 ⁶⁴	x64	7	43	41	–	–	11,600
Redundant $\rho = 25.5$	AVX2	5	46	37	20	10	14,000

At a first glance, the cycle counts of the implementation using a redundant representation are slower faster for additions and squares, also the timings for multiplications are quite competitive in comparison with the timings of the implementation using polynomial representation. However, we noticed some extra overheads by using the redundant representation.

In the redundant representation the size of digits increases after calculating multiplications or squares. This overhead in calculations accounts for 20 cycles; meanwhile in the polynomial representation, this operations is not required. An example of this overhead is shown in the time to calculate a multiplicative inverse; this operation requires to calculate many consecutive squares; thus, a digit size reduction must be executed in between of consecutive square calculations. Then, although the squaring operation takes 37 cycles in the vector implementation, there are also required 20 additional cycles for reducing size of digits, resulting in 57 cycles per squaring, which in contrast to the 41 cycles that takes the implementation using the polynomial representation.

This comparisons show that the vector implementation using 128-bit vector registers is slightly slower for calculating arithmetic operations. This is partially caused because an extra overhead must be considered by using the redundant representation, since the size of digits must be reduced before a multiplication or a squaring is calculated. Another reason of this performance difference is due to the size of the integer multiplier, even calculating two 32-bit multiplications in parallel, the performance of this instruction is lower to the performance of the 64-bit multiplier. As a result, the implementation using radix- 2^{64} achieves better timings for calculating (single) prime field operations.

3.4.3 Two-way Operations

One can accelerate the calculation of operations by increasing their throughput rather than reducing their latency. To do so, we rely on the notion of n -way operations, introduced in Section 3.3, and develop *two-way operations* by extending the 128-bit vector implementation from the previous section.

In the implementation of two-way prime field operations, we store one sequence in each 128-bit part of the 256-bit vector registers. Thus, given two sequences of digits $A = (a_9, \dots, a_0)$ and $B = (b_9, \dots, b_0)$, the pair $\langle A, B \rangle$ represents the distribution of digits into five 256-bit registers as Figure 3.4.16 shows.

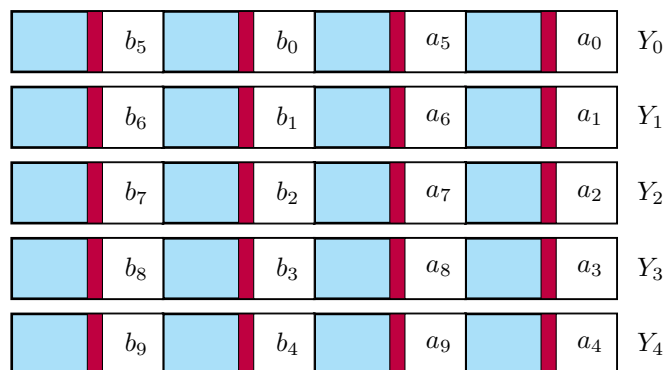


Figure 3.4.16: The notation $\langle A, B \rangle$ represents the distribution of digits of two sequences A and B stored into five 256-bit registers.

The implementation of the arithmetic of two-way operations follows the description given in the previous sections. In essence, we use 256-bit registers as two units of 128 bits, and each unit is in charge to calculate one prime field operation. For this to work, the operations must be calculated independently from each other.

Sometimes it is required to swap the contents of $\langle A, B \rangle$ to obtain $\langle B, A \rangle$. We implement this operation using five `VPERMQ` instructions, which moves 64-bit words within a 256-bit register. We want to highlight that `VPERMQ` has a higher latency than other integer operations. For this reason, swapping words within a 256-bit register results in overheads that negatively impact on performance.

Another common operation between pairs is to combine sequences of digits. Given $\langle A, B \rangle$ and $\langle C, D \rangle$, we want to obtain either $\langle A, D \rangle$ or $\langle C, B \rangle$. These pairs can be calculated using five `BLEND` instructions, which are faster than `VPERMQ` instructions. This difference on performance is due to the latency of `BLEND` instructions is one cycle and also because there are three execution units for processing them; thus, the processor can execute three `BLEND` instructions per cycle. Therefore, combining two pairs is faster than swapping the sequences stored in 256-bit registers.

Performance Benchmark

Table 3.4.17 lists the timings measured on a Skylake for executing two-way operations. The first two rows are equal to the rows of Table 3.4.15. The third row lists the timings of two-way operations. The last row shows the acceleration factor calculated as $2T_{\text{single}}/T_{\text{two-way}}$, where T_{single} is the minimum time to compute an operation either using x64 or SSE instructions, and $T_{\text{two-way}}$ is the time taken by a two-way operation.

Table 3.4.17: Time in clock cycles of two-way $\mathbb{F}_{p_{25519}}$ operations measured on Skylake.

Operations	Radix	ISA	A	M	S	Digit Size Reduction	
						Sequential	Parallel
Single	Radix-2 ⁶⁴	x64	7	43	41	–	–
	$\rho = 25.5$	SSE	5	46	37	20	10
Two-way	$\rho = 25.5$	AVX2	6	47	37	21	11
			1.66×	1.83×	2.00×	1.90×	1.82×

Extending the processing of operations from 128-bit to 256-bit registers scales almost linearly. This is because most of the integer instructions have the same latency on either register size. Differences on the latency of these two implementations are evident in operations with high memory-accessing instructions. Moving values from/to the memory to/from vector registers requires a higher latency when dealing with larger vectors.

We optimize the calculation of the digit size reduction. Recall that the calculation of digit size reduction using Algorithm 3.2.28 is a sequential process, and the same instruction scheduling is used for calculating $\text{DSR}(\langle A, B \rangle)$. However, the processor can handle execute more instructions by issuing these instructions to other execution units. Hence, we implemented a function that calculates both $\text{DSR}(\langle A, B \rangle)$ and $\text{DSR}(\langle C, D \rangle)$, such a function interleaves the calculation of these operations. For example, the first step of the reduction propagates the digits $[b_5, b_0, a_5, a_0]$ to $[b_6, b_1, a_6, a_1]$, and at the same time the digits $[d_5, d_0, c_5, c_0]$ to $[d_6, d_1, c_6, c_1]$; this processing leverages the capabilities of a superscalar processor, which can execute more than one vector instruction using several

execution units. We denote as $\text{DSR2}(\langle A, B \rangle, \langle C, D \rangle)$ to this optimized function. Regarding performance timings, the DSR2 function takes 23 clock cycles, which is $1.73\times$ faster than calculating two $\text{DSR}(\langle A, B \rangle)$ operations consecutively.

In Table 3.4.17, we do not report timings for calculating multiplicative inverses. This operation is scarcely used and it is an inherently sequential operation that does not take advantage of the use of two-way operations. Although we could implement a two-way inverse, it is more efficient to calculate a batch of n inverses using the simultaneous inversion method [143, Section 2.26], which requires one inversion and $3(n - 1)$ multiplications.

3.4.4 Four-way Operations

We found two ways to implement four-way operations. One way is using 512-bit registers as four units of 128 bits, as shown in Figure 3.4.19. Another approach, and the one we follow, is using 256-bit registers as four units of 64 bits. So, $\langle A, B, C, D \rangle$ denotes four sequences $A, B, C,$ and D stored into a 256-bit register as shown in Figure 3.4.18.

We developed functions that perform arithmetic operations in parallel. In this case, we store a tuple using ten 256-bit registers. The scheduling of instructions for multiplications follows Algorithm 3.4.12; thus at every iteration, ten PMULUDQ instructions are executed independently and their products are accumulated into ten destination registers. The calculation of π function is performed using one PMULUDQ instruction to multiply by 19, meanwhile the reordering of digits is performed by means of register renaming, which is handled at compilation time.

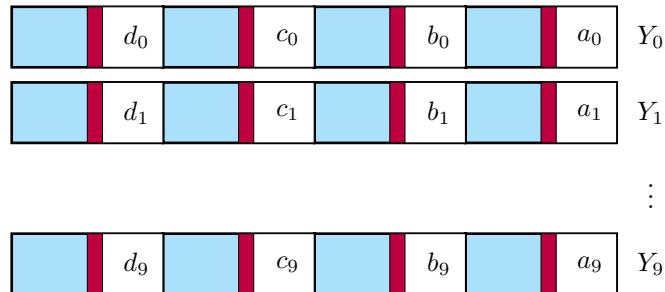


Figure 3.4.18: Digit distribution of $\langle A, B, C, D \rangle$ to perform four-way prime field operations using 256-bit registers.

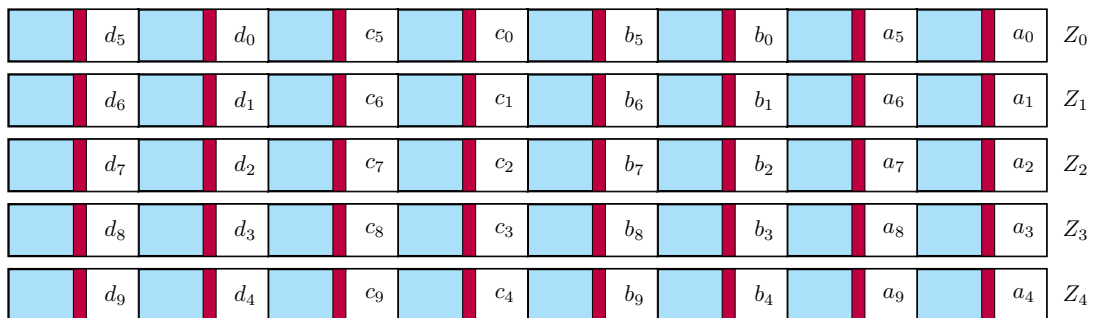


Figure 3.4.19: Digit distribution of $\langle A, B, C, D \rangle$ to perform four-way prime field operations using 512-bit registers.

Performance Benchmark

Table 3.4.20 lists the timings measured on Skylake for executing prime field operations. The first four rows are equal to the rows of Table 3.4.17. The fifth row lists the timings of four-way operations. The last row shows the acceleration factor calculated as $4T_{\text{single}}/T_{\text{four-way}}$, where T_{single} is the minimum time to compute an operation either using x64 or SSE instructions, and $T_{\text{four-way}}$ is the time taken by a four-way operation.

Table 3.4.20: Time in clock cycles of four-way $\mathbb{F}_{p_{25519}}$ operations measured on Skylake.

Operations	Radix	ISA	A	M	S	Digit Size Reduction	
						Sequential	Parallel
Single	Radix-2 ⁶⁴ $\rho = 25.5$	x64	7	43	41	–	–
		SSE	5	46	37	20	10
Two-way	$\rho = 25.5$	AVX2	6 1.66×	47 1.83×	37 2.00×	21 1.90×	11 1.82×
Four-way	$\rho = 25.5$	AVX2	11 1.81×	84 2.05×	52 2.84×	34 2.35×	16 2.50×

The performance timings of the four-way prime field arithmetic show an increase on the throughput of the operations. For example, calculating four multiplications takes 84 cycles using the four-way operation whereas, using four multiplications using the radix-2⁶⁴ implementation takes $4 \times 43 = 172$ cycles; thus, the four-way multiplications are twice faster. However, the four-way implementation do not scale linearly with the number of units, since it was expected an acceleration factor close to the ideal factor ($4\times$).

One reason that explains this loss of performance is because operating over larger tuples requires more live registers than the other implementations. Processors supporting AVX2 have only sixteen 256-bit vector registers, so the compiler must issue instructions to spill registers to memory more frequently. Since the latency of accessing memory using large registers is high, the performance of the four-way operations is downgraded.

3.5 Arithmetic on $\text{GF}(2^{384} - 2^{128} - 2^{96} + 2^{32} - 1)$

In this section, we cover the case of a prime modulus that belongs to the family of Generalized Mersenne numbers [251]. These primes admit an efficient procedure for reduction modulo p . The prime field studied is $\mathbb{F}_{p_{384}}$ where $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$, which is specified in standards for elliptic curve cryptography. We now describe the implementation of arithmetic operations in this field.

The choice of a multi-precision representation of integers is key for achieving good performance. By using a polynomial representation, modular additions propagate some carry bits through all the digits of the number; however, this propagation of bits is a dependency chain that avoids calculating operations in parallel. Besides, the presence of large dependency chains reduces the performance of the operations significantly. As a

corollary, it follows that reducing such dependencies is crucial for improving the performance. Conversely, using a redundant representation guarantees that the addition of two elements will not overflow the registers, which enables parallel processing of digits and leads to an immediate application of vector instructions.

3.5.1 Redundant Representation

The redundant representation as stated in Definition 3.2.6 requires setting ρ . Although there are several alternatives, we selected ρ following these criteria. First of all, recall that Equation (3.2.22) must be satisfied to use the `PMULUDQ` AVX2 instruction. Initially, we considered setting $\rho = 30$ leading to sequences of $l = 13$ digits; however, digits will not have enough room to store carry bits, which does not satisfy $\rho \leq 29.5$ according to Equation (3.2.21). The next options are to choose $\rho = \{28, 29\}$ as either case leads to work with sequences of $l = 14$ digits. We selected $\rho = 28$ because registers will have more room for storing carry bits after calculating a multiplication. Finally, we discarded the case of $\rho < 28$ as it increases the length of sequences and the number of instructions for performing arithmetic operations.

Let $\rho = 28$, an element $a \in \mathbb{F}_{p_{384}}$ is represented as any sequence $A = (a_{13}, \dots, a_0)$ of length $l = 14$ such that $a = \sum_{i=0}^{13} 2^{28i} a_i \bmod p_{384}$, as shown in Figure 3.5.1.

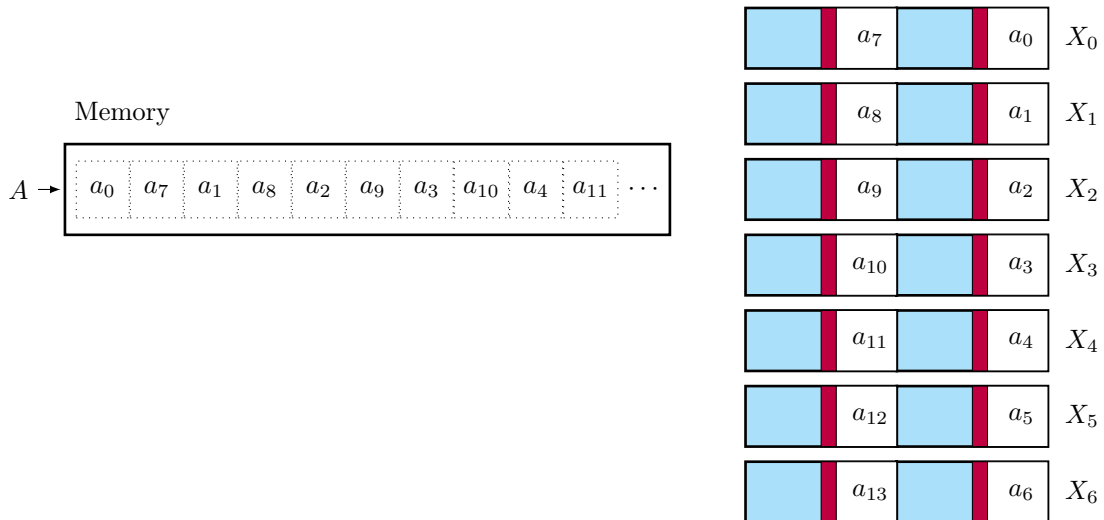


Figure 3.5.1: A sequence of digits $A = (a_{13}, \dots, a_0)$ representing $a \in \mathbb{F}_{p_{384}}$ is loaded from memory into seven 128-bit registers X_0, \dots, X_6 .

Addition

Given $A = (a_{13}, \dots, a_0)$ and $B = (b_{13}, \dots, b_0)$, the addition $C = A + B$ is calculated as $C = (a_{13} + b_{13}, \dots, a_0 + b_0)$. These additions are calculated using seven `PADDQ` instructions.

Subtraction

For subtracting sequences, we must ensure that all digits of the resultant sequence be positive. To do that, we perform the subtraction of $A = (a_{13}, \dots, a_0)$ minus $B = (b_{13}, \dots, b_0)$

as $C = A - B + P$, where P is a redundant sequence of p_{384} such that $|P| \leq 30$, so we set

$$\begin{aligned}
P &= (p_{13}, \dots, p_0) \\
&= (0x101ffffe, 0x1ffffffe, 0x1ffffffe, 0x1ffffffe, 0x1ffffffe, \\
&\quad 0x1ffffffe, 0x1ffffffe, 0x1ffffffe, 0x1ffffffe, 0x1efdffe, \\
&\quad 0x1efdffe, 0x1ffffffe, 0x2000101e, 0x1ffffefe).
\end{aligned} \tag{3.5.2}$$

In this sequence, $|p_1| = 30$ whereas the size of the remaining digits is 29 bits; the selection of this sequence is due to the size of the second digit increases faster than the size of the other digits when performing a reduction modulo p_{384} .

Multiplication

We evaluate two alternatives for multiplying prime field elements. First, we implemented the Mastrovito multiplier, which merges the integer multiplication with the reduction modulo p ; and we also implemented the Karatsuba multiplier, which calculates the integer multiplication, followed by the reduction modulo p .

Multiplication using Mastrovito Multiplier

Recalling that the Mastrovito multiplier enables the calculation of several digit multiplications without dependencies, which is suitable to leverage the high throughput of the PMULUDQ instruction. In this setting, this algorithm calculates the multiplication of the sequence A times B as $C = A \times B = \sum_{i=0}^{13} \pi^i(A) \times b_i$, where π is defined as

$$(a_{13}, \dots, a_0) \mapsto (a_{12}, \dots, a_0, 0) + R, \tag{3.5.3}$$

and R is a sequence of digits representing the number $2^{\lceil l\rho \rceil} a_{13} = 2^{392} a_{13}$. Reducing this number modulo p_{384} , it follows that $2^{392} a_{13} \equiv (2^{136} + 2^{104} - 2^{40} + 2^8) a_{13} \pmod{p_{384}}$. An initial approach is $R = (2^{24} a_{13}, 2^{20} a_{13}, 0, -2^{12} a_{13}, 2^8 a_{13})$; however, since $|a_{13}| \leq 28$ then $|R| \leq 52$, which avoids performing a subsequent multiplication of digits with the 32-bit PMULUDQ multiplication instruction. For this reason, we looked for a sequence R that ensures $|\pi^i(A)| \leq 32$ for $0 \leq i < 14$, and we arrived to a better approach: $R = (r_5, \dots, r_0)$, where

$$\begin{aligned}
r_5 &= \lfloor a_{13}/2^4 \rfloor, & r_4 &= \lfloor a_{13}/2^8 \rfloor + 2^{24}(a_{13} \bmod 2^4), \\
r_3 &= 2^{20}(a_{13} \bmod 2^8), & r_2 &= -\lfloor a_{13}/2^{16} \rfloor, \\
r_1 &= \lfloor a_{13}/2^{20} \rfloor - 2^{12}(a_{13} \bmod 2^{16}), & r_0 &= 2^8(a_{13} \bmod 2^{20}).
\end{aligned} \tag{3.5.4}$$

Thus $|R| \leq 28$ and if we consider that $|A| \leq 28$, then the size of $\pi(A)$ is at most 29 bits. We experimentally verified that if $|A| \leq 28$, then $|\pi^i(A)| \leq 30$ for $0 \leq i < 14$. This last relation allows performing digit multiplications using the PMULUDQ instruction. Therefore, we implemented Mastrovito's multiplication method following Algorithm 3.4.12.

The operations required by π were implemented using only logic and bit shifting instructions. That is using the PAND and PSRLQ/PSLLQ vector instructions. We optimized the implementation of π noticing that whenever the shift displacement is multiple of

eight, we use instead byte permutation instructions (PSHUF8 instruction). This permutation instruction enters to an execution unit that is different to the ones used for shift instructions. We use this equivalence of instructions to relieve pressure in one of the units and to better distribute the workload among other execution units. This optimization technique allows increasing the instruction-level parallelism of this task which results on reducing the execution time of the function.

Multiplication using Karatsuba Multiplier

Since $l = 14$ is even, we can split the input sequences by half and calculate three multiplications of sequences of seven digits. These multiplications can be performed again using the Karatsuba multiplier. In Figure 3.5.5, we show the recursion tree used to calculate multiplications of sequences of fourteen digits. Each node is annotated with n , which represents the multiplication of sequences of length n . Each internal node is calculated using Karatsuba algorithm splitting in three nodes. The leaves of the tree represent the end of the recursion, where the product of these sequences is calculated using the schoolbook method, more specifically, using the operand-scanning method.

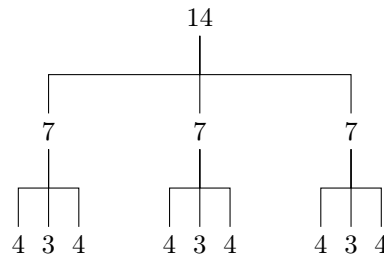


Figure 3.5.5: Recursion tree of Karatsuba multiplication of sequences of length $l = 14$. The leaves of the tree represent multiplications of sequences of length three and four using the schoolbook multiplication.

Given $A = (a_{13}, \dots, a_0)$ and $B = (b_{13}, \dots, b_0)$, the Karatsuba multiplication algorithm sets $n = 7$ and splits the inputs into shorter sequences $A_1 = (a_{13} \dots, a_7)$, $A_0 = (a_6 \dots, a_0)$, $B_1 = (b_{13} \dots, b_7)$, and $B_0 = (b_6 \dots, b_0)$. Then, it calculates $X = A_0 \times B_0$, $Y = A_1 \times B_1$, and $Z = (A_0 + A_1) \times (B_0 + B_1)$. Note that the recursion trees of these multiplications are identical, and more importantly, they are independent of each other, which enables a parallel execution of the operations. For instance, one can construct the tuple $\langle A_0, B_0, A_0 + A_1 \rangle$ and multiply it by $\langle A_1, B_1, B_0 + B_1 \rangle$ to obtain $\langle X, Y, Z \rangle$ executing this operation entirely in parallel using, for example, 256-bit vector registers. However, we do not follow such approach because inserting (and extracting) values into the high part of a 256-bit register is a time-consuming operation that slows down the execution of the multiplication.

For the third recursion tree, we execute the first two sub-trees in parallel as is shown in Figure 3.5.6. Once we calculate X , Y , and Z the product $C = A \times B$ is obtained using the refined Karatsuba identity from Equation (3.2.25). Finally, C will have 28 digits which must be reduced modulo p_{384} to obtain an equivalent sequence of 14 digits.

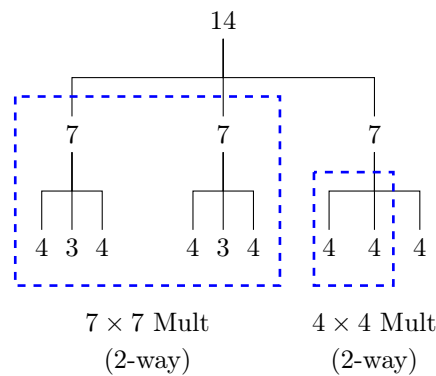


Figure 3.5.6: Parallel execution of the Karatsuba recursion tree for $l = 14$. The recursion levels inside the rectangles are processed using a 2-way $n \times n$ digit multiplication function using 128-bit vector instructions.

Reduction Modulo p_{384}

This section describes a procedure to reduce A modulo p_{384} , where A is the sequence resulting from calculating an integer multiplication using the Karatsuba algorithm. In such a case, the input sequence A has length 28 and $|A| \leq 60$ bits.

Our reduction method is an iterative algorithm that shortens the sequence A one digit at a time. For $i \geq 14$, the digit a_i of A represents the number $2^{28i}a_i$ which is equivalent to $2^{28i}a_i \equiv (2^{136} + 2^{40} - 2^{40} + 2^8)2^{28(i-14)}a_i \pmod{p_{384}}$. By using this equivalence, we can replace the last digit of the sequence, and accumulate the correspondent sequence into the remaining digits. This equivalence is applied fourteen times until it remains fourteen digits. Algorithm 3.5.7 shows the steps to follow for reducing modulo p_{384} .

Algorithm 3.5.7 Reduction modulo p_{384} .

Input: A , a sequence of length 28.

Output: C , a sequence of length 14 such that $C \sim A$.

- 1: $C \leftarrow A$
 - 2: **for** $i \leftarrow 0$ **to** 13 **do**
 - 3: $x \leftarrow c_{27-i}$
 - 4: $c_{19-i} \leftarrow c_{19-i} + \lfloor x/2^{32} \rfloor$
 - 5: $c_{18-i} \leftarrow c_{18-i} + \lfloor x/2^{36} \rfloor$
 - 6: $c_{17-i} \leftarrow c_{17-i} + 2^{24}(x \bmod 2^{32})$
 - 7: $c_{16-i} \leftarrow c_{16-i} + 2^{20}(x \bmod 2^{36}) - \lfloor x/2^{44} \rfloor$
 - 8: $c_{15-i} \leftarrow c_{15-i} + \lfloor x/2^{48} \rfloor$
 - 9: $c_{14-i} \leftarrow c_{14-i} - 2^{12}(x \bmod 2^{44})$
 - 10: $c_{13-i} \leftarrow c_{13-i} + 2^8(x \bmod 2^{48})$
 - 11: **end for**
 - 12: **return** $C = (c_{13}, \dots, c_0)$
-

By analyzing Algorithm 3.5.7, there exists a series of loop-carried dependencies in the execution of the main loop. Assume that the main loop is processing the i -th iteration for replacing the digit c_{27-i} , then it can be seen that the digits c_{13-i} to c_{19-i} are updated with parts of the digit c_i ; thus the digits c_{13-i} to c_{19-i} depend on c_i . Using the same

argument, the digits c_{6-i} to c_{12-i} depend on the digit c_{20-i} . However, we observed that the loop-carried dependencies do not interfere with the calculation of the digits derived by reducing c_{27-i} and c_{20-i} . Hence, the reduction of this digits runs in parallel.

The parallel execution of Algorithm 3.5.7 can reduce two digits at the same time. First, the input digits must be stored into 128-bit registers as shown below.

$$\begin{array}{l}
[c_{27} \ c_{20}] \rightarrow [c_{19} \ c_{12}] \ [c_{18} \ c_{11}] \ [c_{17} \ c_{10}] \ [c_{16} \ c_9] \ [c_{15} \ c_8] \ [c_{14} \ c_7] \ [c_{13} \ c_6] \\
[c_{26} \ c_{19}] \rightarrow [c_{18} \ c_{11}] \ [c_{17} \ c_{10}] \ [c_{16} \ c_9] \ [c_{15} \ c_8] \ [c_{14} \ c_7] \ [c_{13} \ c_6] \ [c_{12} \ c_5] \\
\vdots \\
[c_{21} \ c_{14}] \rightarrow [c_{13} \ c_6] \ [c_{12} \ c_5] \ [c_{11} \ c_4] \ [c_{10} \ c_3] \ [c_9 \ c_2] \ [c_8 \ c_2] \ [c_7 \ c_0] .
\end{array}$$

The leftmost pair represents the digits that are reduced modulo p_{384} updating the value stored into the pairs on the right. Thus, each row represents the calculation of two reductions modulo p_{384} in parallel. After seven iterations, the input sequence will have fourteen digits, which are already stored as $\langle A_0, A_1 \rangle$ into 128-bit vector registers.

The implementation of this reduction procedure is analogous to the one used for implementing the π function. In this case, Algorithm 3.5.7 requires performing divisions and reductions modulo a power of two, operations that are straightforwardly implemented using bit shifting and logic instructions. As before, some of these operations can be alternatively implemented using byte permutation instructions (PSHUF instruction). The processor executes them as fast as logical operations, and more importantly, the execution unit that processes PSHUF is different to the one that processes bit shifting instructions.

Digit Size Reduction

This operation reduces the size of each digit of a given sequence to be lesser than or equal to 28 bits. In the reduction modulo p_{384} , every time a digit is reduced some parts of it are added to the reduced sequence. However since $2^{384} \bmod p_{384}$ has one negative power of 2, some bits must be subtracted instead. This can be seen in the Steps 7 and 9 of Algorithm 3.5.7. These subtractions cause, in some cases, that the reduced sequence has negative digits; nonetheless, although such a sequence also represents the same integer as the non-reduced one, we restrict the redundant representation to work with positive digits (cf. Definition 3.2.6). Thus, to remove negative digits in an input sequence A , we add a redundant sequence P , which is a multiple of p_{384} such that $60 < |P| \leq 62$, specifically

$$\begin{aligned}
P &= (p_{13}, \dots, p_0) \\
&= (0x1ffffefe00000000, 0x2000101e00000000, 0x1ffffffe00000000, \\
&\quad 0x1efdffe00000000, 0x1efdffe00000000, 0x1ffffffe00000000, \\
&\quad 0x1ffffffe00000000, 0x1ffffffe00000000, 0x1ffffffe00000000, \\
&\quad 0x1ffffffe00000000, 0x1ffffffe00000000, 0x1ffffffe00000000, \\
&\quad 0x1ffffffe00000000, 0x101ffffe00000000) .
\end{aligned} \tag{3.5.8}$$

Once the sequence was updated as $A \leftarrow A + P$, we can perform the digit size reduction.

Recall that a sequence A is stored into 128-bit vector registers as A_0, A_1 ; then we have registers containing the digits $[a_0, a_7], \dots, [a_6, a_{13}]$. We start the propagation of digits

from $[a_6, a_{13}]$. Thus, the most significant bits of a_6 are propagated to a_7 , whereas the most significant bits of a_{13} are reduced using $2^{392}x \equiv (2^{136} + 2^{104} - 2^{40} + 2^8)x \pmod{p_{384}}$. After performing this propagation and reduction, we propagate the most significant bits of a_0 and a_7 into, respectively, the digits a_1 and a_8 . We continue with this propagation of bits until reaching again a_6 and a_{13} . Finally, we propagate the most significant bits of a_6 and reduce the most significant bits of a_{13} , which ensures that $|A| \leq 28$. The propagation of digits is performed in parallel since digits are stored in 128-bit vector registers.

Multiplicative Inverse

Let $a \in \mathbb{F}_{p_{384}}$, the multiplicative inverse of a is calculated as

$$\begin{aligned}
a^{-1} &= a^{2^{384} - 2^{128} - 2^{96} + 2^{32} - 3} \\
&= \left[a^{2^{288} - 2^{32} - 1} \right]^{2^{96}} \left[a^{2^{32} - 3} \right] \\
&= \left[a^{2^{288} - 2^{33} + 2^{32} - 1} \right]^{2^{96}} \left[a^{2^{32} - 2^2 + 2^1 - 1} \right] \\
&= \left[\left(a^{2^{255} - 1} \right)^{2^{33}} \left(a^{2^{32} - 1} \right) \right]^{2^{96}} \left[\left(a^{2^{30} - 1} \right)^{2^2} \left(a^{2^1 - 1} \right) \right] \\
&= \left[(\alpha_{255})^{2^{33}} T \alpha_2 \right]^{2^{96}} [T \alpha_1], \text{ where } T = (\alpha_{30})^{2^2} \text{ and } \alpha_x = a^{2^x - 1}.
\end{aligned} \tag{3.5.9}$$

We calculate a large part of this exponentiation using the Itoh-Tsujii algorithm [156]. To calculate α_{255} , we looked for the shortest addition chain (c_1, \dots, c_s) , such that $c_1 = 1$ and $c_s = 255$.

i	c_x	c_y	$c_i = c_x + c_y$	$S \leftarrow S \cup \{ \alpha_{c_i} = (\alpha_{c_x})^{2^{c_y}} \alpha_{c_y} \}$
-	-	-	1	$\{ \alpha_1 \}$
1	1	1	2	$\{ \alpha_1, \alpha_2 \}$
2	2	1	3	$\{ \alpha_1, \alpha_2, \alpha_3 \}$
3	3	3	6	$\{ \alpha_1, \alpha_2, \alpha_3, \alpha_6 \}$
4	6	6	12	$\{ \alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12} \}$
5	12	3	15	$\{ \alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{15} \}$
6	15	15	30	$\{ \alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{15}, \alpha_{30} \}$
7	30	30	60	$\{ \alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{15}, \alpha_{30}, \alpha_{60} \}$
8	60	60	120	$\{ \alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{15}, \alpha_{30}, \alpha_{60}, \alpha_{120} \}$
9	120	120	240	$\{ \alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{15}, \alpha_{30}, \alpha_{60}, \alpha_{120}, \alpha_{240} \}$
10	240	15	255	$\{ \alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{15}, \alpha_{30}, \alpha_{60}, \alpha_{120}, \alpha_{240}, \alpha_{255} \}$

We leverage that fact that the term α_{30} is part of the addition chain since it is used to calculate the multiplicative inverse. In total, calculating multiplicative inverses takes 14 multiplications and 385 squares.

3.5.2 Two-way Operations

The previous section shows how to perform prime field operations using the 128-bit vector instructions. Such an implementation leverages that some internal prime field operations can be performed in parallel among the digits of the sequence. This section shows the extension to 256-bit registers to process two-way prime field operations.

Given a pair of sequences A and B , we use seven 256-bit registers to store the tuple $\langle A, B \rangle$ as Figure 3.5.10 shows.

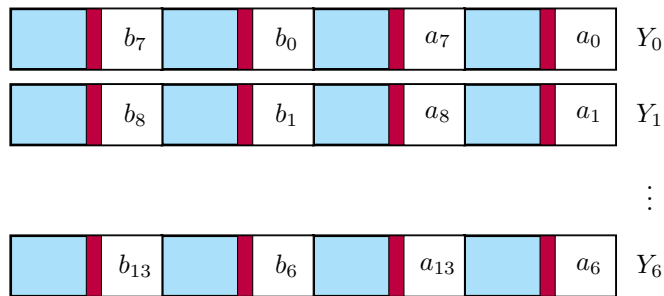


Figure 3.5.10: Distribution of the digits of $\langle A, B \rangle$ into seven 256-bit vector registers.

The arithmetic operations are performed using 256-bit AVX2 instructions, and for implementing the algorithms described in the previous section, the digits are processed analogously. With the exception of multiplicative inverses, the implementation of the arithmetic operations follows the parallel strategy described in Section 3.3.

3.5.3 Performance Benchmark

We report a performance benchmark of the vectorized implementation of the arithmetic operations on $\mathbb{F}_{p_{384}}$.

Since multiplication is a critical operation of the implementation, we want to determine which multiplication algorithm offers a better performance. We measured the time taken to calculate multiplications between sequences of fourteen digits and obtained the following timings.

Implementation	Latency (cycles)	Code Size (bytes)
Mastrovito Method	181	2,314
Karatsuba Method	173	3,428

We experimentally verified that multiplications run around 5% faster when using the Karatsuba algorithm. The memory footprint of the Karatsuba implementation is $1.48\times$ the size of the Mastrovito's implementation. If a software library wants to target a low memory consumption, the implementation of Mastrovito multiplier becomes the preferred choice for calculating multiplications. We opted for using the fastest implementation, so the Karatsuba algorithm is the one used in our implementation.

Comparison with Related Works

We looked for software libraries that perform arithmetic operations using 64-bit instructions. The OpenSSL library [263] has support for calculating operations on $\mathbb{F}_{p_{384}}$ but the code is not optimized for this field. This library uses generic multi-precision integer arithmetic and uses the Montgomery REDC algorithm for calculating reductions modulo p_{384} . We measured the OpenSSL’s functions: `BN_mod_add` for additions, `BN_mod_mul_montgomery` for multiplications, and `BN_mod_exp_mont_consttime` for multiplicative inverses. We also found an optimized 64-bit library for $\mathbb{F}_{p_{384}}$ developed by McMillion [189]. This is a Go library with the prime field arithmetic written in assembly. These two 64-bit implementations are the baseline used in comparisons. Table 3.5.11 shows the timings of operations measured on Skylake.

Table 3.5.11: Time in clock cycles of $\mathbb{F}_{p_{384}}$ operations measured on Skylake.

Operations	Radix	ISA	A	M	S	DSR	I	Reference
Single	Radix-2 ⁶⁴	x64	106	298	271	–	124,000	OpenSSL [263]
	Radix-2 ⁶⁴	x64	18	162	162	–	116,500	McMillion [189]
	$\rho = 28$	SSE	5	173	145	34	75,100	This work
Two-way	$\rho = 28$	AVX2	7 1.43×	178 1.82×	150 1.93×	36 1.88×	–	This work

It is clear that the optimized assembly implementation surpasses OpenSSL’s implementation. For example, the timings for squaring are almost twice as fast, but it is not the case for inversions. McMillion’s implementation uses a generic square-and-multiply method to calculate inverses, whereas OpenSSL uses a fixed-size window exponentiation method. In contrast, our vectorized implementation calculates inverses 35% and 39% faster than the baseline implementations. This is due to we use an exponentiation method that requires fewer operations. For field additions, the 64-bit implementations always perform a reduction modulo p . In the vectorized implementation, additions are executed in parallel and the reduction modulo p is only calculated after multiplications.

Regarding the implementation of two-way operations, we achieve significant improvements by using 256-bit vector instructions. The last row of Table 3.5.11 summarizes the acceleration factor calculated as $2T_{\text{single}}/T_{\text{two-way}}$, where T_{single} is the minimum time to compute a prime field operation either using 64-bit or SSE instructions, and $T_{\text{two-way}}$ is the time taken by a two-way operation. As can be seen, almost all arithmetic operations get acceleration factors close to the ideal factor ($2\times$).

3.6 Arithmetic on $\text{GF}(2^{448} - 2^{224} - 1)$

This section shows the implementation of arithmetic operations of the prime field $\mathbb{F}_{p_{448}}$, where $p_{448} = 2^{448} - 2^{224} - 1$. This modulus is an example of a Generalized Mersenne prime, but also resembles the polynomial of the golden ratio. Moreover, this prime field is used to define elliptic curves with efficient arithmetic. We describe two optimized implementations

based on the following representations: using a polynomial representation setting $w = 64$, and using a redundant representation setting $\rho = 28$.

3.6.1 Polynomial Representation

We implement arithmetic operations for $\mathbb{F}_{p_{448}}$ using the native 64-bit instruction set. Prime field elements are represented as sequences of digits in radix- 2^{64} , i.e., $w = 64$ following Definition 3.2.1. So, an element $a \in \mathbb{F}_{p_{448}}$ is represented by $A = (a_6, \dots, a_0)$, which stored in an array of $l = 7$ words of 64-bits.

Addition

Adding two prime field elements requires performing an addition with carry for every digit of the sequence. This operation produces one carry bit, which must be reduced modulo p_{448} . We want to emphasize that the reduction modulo p_{448} must be implemented using a regular execution pattern to fulfill the requirements of secure software development.

Now, we show the operations required for adding a and b , two 448-bit numbers, to obtain c , a 448-bit number that is equivalent to $c \equiv a + b \pmod{p_{448}}$. First, calculate $d \leftarrow a + b$, this number can be written as $d_1 2^{448} + d_0$, where $d_0 = d \bmod 2^{448}$ and $d_1 = \lfloor d/2^{448} \rfloor$. After calculating the addition, if the carry bit $d_1 = 0$, then the operation ends by setting $c \leftarrow d$; otherwise, if $d_1 = 1$, then d_1 must be reduced modulo p_{448} and added to d_0 , i.e., $e \leftarrow d_0 + d_1(2^{224} + 1)$. However, e could be greater than 2^{448} generating and additional carry bit. Thus, it follows $e = e_1 2^{448} + e_0$, where $e_0 = e \bmod 2^{448}$ and $e_1 = \lfloor e/2^{448} \rfloor$. Observe that if $e_1 = 0$, then the addition ends by setting $c \leftarrow e$; otherwise, e_1 is different to zero whenever $2^{448} - 2^{224} - 1 \leq d_0 < 2^{448}$, which causes that e is bounded as $2^{448} \leq e < 2^{448} + 2^{224} + 1$. Finally, reduce e_1 modulo p_{384} and calculate $c \leftarrow e_0 + e_1(2^{224} + 1)$. In this operation, the calculation of c does not produce carry bit, since $0 \leq e_0 < 2^{224} + 1$; thus $0 \leq c < 2^{448}$. A graphic description of this algorithm is shown in Figure 3.6.1.

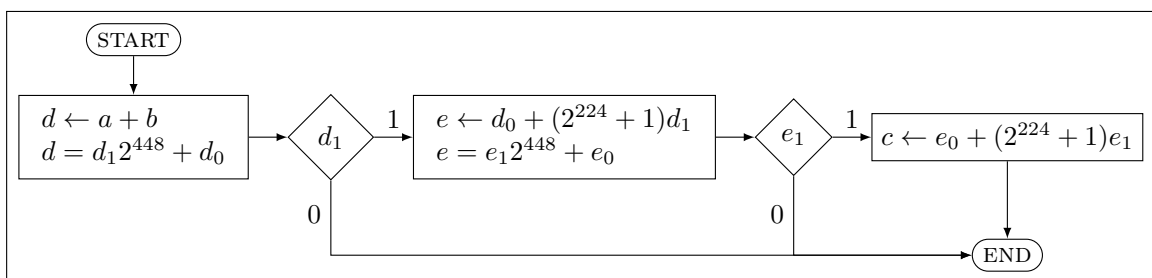


Figure 3.6.1: Non-constant-time calculation of additions modulo p_{448} .

The previous method as it was described does not run in constant time. To implement these operations in constant-time, we use conditional instructions to handle the carry bit and always perform a fixed number of operations. After performing an addition with carry operation, the resultant carry bit is used to conditionally load 2^{32} into a register, which is performed through the CMOV instruction. Algorithm 3.6.2 shows the steps followed to calculate the addition modulo p_{448} using a regular execution pattern.

Algorithm 3.6.2 Constant-time implementation of additions modulo p_{448} .

Input: A and B , two sequences of seven digits of 64-bits representing $a, b \in \mathbb{F}_{p_{448}}$.

Output: C , a sequence of seven digits representing $c = a + b \bmod p_{448}$.

$(d_7, \dots, d_0) \leftarrow \text{AddCarry}((a_6, \dots, a_0), (b_6, \dots, b_0), 0)$

$x \leftarrow 0$

$x \leftarrow \text{CMOV}(d_7, 2^{32})$

$(e_7, \dots, e_0) \leftarrow \text{AddCarry}((d_6, \dots, d_0), (x, 0, 0, 0), d_7)$

$x \leftarrow 0$

$x \leftarrow \text{CMOV}(e_7, 2^{32})$

$(c_6, \dots, c_0) \leftarrow \text{AddCarry}((e_6, \dots, e_0), (x, 0, 0, 0), e_7)$

return $C \leftarrow (c_6, \dots, c_0)$

Subtraction

The subtraction of prime field elements is calculated using an analogous implementation as the one used for calculating additions. Thus, all the addition with carry operations (ADC instruction) were replaced by subtraction with borrow operations (SUB/SBB instruction).

Multiplication

We decompose the calculation of prime field multiplication in an integer multiplication followed by a reduction modulo p_{448} .

For implementing integer multiplication, we use the operand scanning method, because this method is highly compatible with the use of MULX instructions and ADX addition instructions. We developed a 448-bit multiplier that calculates a 896-bit product following Algorithm 3.2.13 and we provide three implementations using a combination of MULX and ADX instructions, the MULX instruction (without ADX instructions), and the MULQ instruction (without MULX and ADX instructions).

Table 3.6.3 shows the instruction counts of our implementations of the 448-bit integer multiplier. The application of the MULX instruction reduces the number of addition instructions. Moreover, by using ADX instructions, the number of memory accesses reduces significantly.

Table 3.6.3: Instruction counts of 448-bit integer multiplication.

Implementation	Multiplication	Addition	Load	Store	Move
MULQ + ADC	49	134	98	56	49
MULX + ADC	49	97	98	56	0
MULX + ADX	49	97	56	14	13

Reduction Modulo p_{448}

The implementation of the reduction modulo p_{448} assumes the input is a 896-bit number stored in fourteen words $C = (c_{13}, \dots, c_0)$ and produces as output a shorter sequence $D = (d_6, \dots, d_0)$ such that $C \sim D$.

The special format of p_{448} allows obtaining a fast reduction method. Observe that the most significant words of C can be reduced modulo p_{448} as Figure 3.6.4 shows.

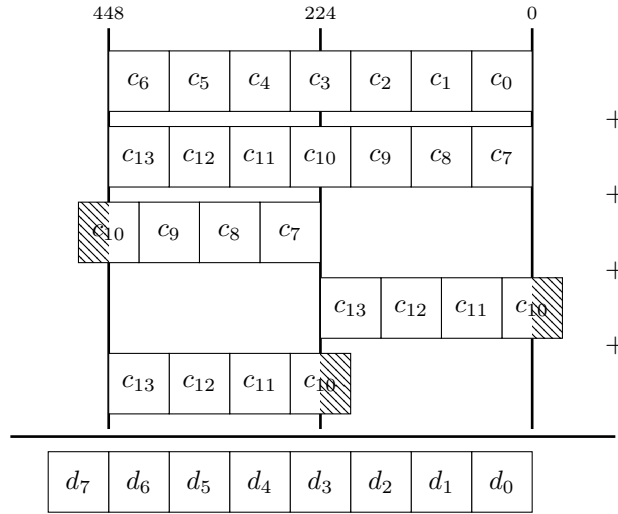


Figure 3.6.4: Partial reduction modulo p_{448} from fourteen 64-bit words $C = (c_{13}, \dots, c_0)$ to eight words $D = (d_7, \dots, d_0)$. The shadowed areas over c_{10} represents that only 32 bits of c_{10} are considered for the addition. This reduction holds $C \sim D$.

In the implementation we calculate $D = (d_7, \dots, d_0) = (c_6, \dots, c_0) + E + F$, where E and F are defined as

$$\begin{aligned}
 E &= (2c_{13}, 2c_{12}, 2c_{11}, 2c_{10} - (c_{10} \bmod 2^{32}), c_9, c_8, c_7), \\
 F &= (\lfloor c_9/2^{32} \rfloor + c_{10} \bmod 2^{32}, \lfloor c_8/2^{32} \rfloor + c_9 \bmod 2^{32}, \lfloor c_7/2^{32} \rfloor + c_8 \bmod 2^{32}, \\
 &\quad \lfloor c_{13}/2^{32} \rfloor + c_7 \bmod 2^{32}, \lfloor c_{12}/2^{32} \rfloor + c_{13} \bmod 2^{32}, \lfloor c_{11}/2^{32} \rfloor + c_{12} \bmod 2^{32}, \\
 &\quad \lfloor c_{10}/2^{32} \rfloor + c_{11} \bmod 2^{32}).
 \end{aligned}$$

This operation can generate one extra word with the carry bits of the addition. The word d_7 must be reduced modulo p_{448} by performing $D = (d_6, \dots, d_0) + (2^{32}d_7, 0, 0, d_7)$ and this last addition does not produce carry bits.

Multiplicative Inverse

Let $a \in \mathbb{F}_{2^{448-2^{224}-1}}$ and define $\alpha_x = a^{2^x-1}$, the multiplicative inverse of a is given as

$$a^{-1} = a^{2^{448}-2^{224}-3} = \left((\alpha_{223})^{2^{223}} \alpha_{222} \right)^2 \alpha_1. \tag{3.6.5}$$

We use the Itoh-Tsujii [156] method to calculate α_{223} and α_{222} efficiently. To calculate α_{223} , we look for a short addition chain (c_1, \dots, c_s) setting $c_1 = 1$ and $c_s = 223$. Fortunately, the shortest addition chain also calculates α_{222} as an intermediate value, and it is shown in the following table.

i	c_x	c_y	$c_i = c_x + c_y$	$S \leftarrow S \cup \{\alpha_{c_i} = (\alpha_{c_x})^{2^{c_y}} \alpha_{c_y}\}$
-	-	-	1	$\{\alpha_1\}$
1	1	1	2	$\{\alpha_1, \alpha_2\}$
2	2	1	3	$\{\alpha_1, \alpha_2, \alpha_3\}$
3	3	3	6	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_6\}$
4	6	6	12	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}\}$
5	12	12	24	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{24}\}$
6	24	3	27	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{24}, \alpha_{27}\}$
7	27	27	54	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{24}, \alpha_{27}, \alpha_{54}\}$
8	54	54	108	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{24}, \alpha_{27}, \alpha_{54}, \alpha_{108}\}$
9	108	3	111	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{24}, \alpha_{27}, \alpha_{54}, \alpha_{108}, \alpha_{111}\}$
10	111	111	222	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{24}, \alpha_{27}, \alpha_{54}, \alpha_{108}, \alpha_{111}, \alpha_{222}\}$
11	222	1	223	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_{12}, \alpha_{24}, \alpha_{27}, \alpha_{54}, \alpha_{108}, \alpha_{111}, \alpha_{222}, \alpha_{223}\}$

In each row, the set S contains a new element $\alpha_{c_i} = (\alpha_{c_x})^{2^{c_y}} \alpha_{c_y}$, which is calculated with one multiplication and c_y squares. Once α_{223} is calculated, the multiplicative inverse of a takes 13 multiplications and 447 squares in total.

Merging Square-Root and Inversion Calculation

Like the multiplicative inverse, the square-roots of a prime field element could be calculated raising to a power that depends on the prime modulus. In the particular case when $p \equiv 3 \pmod{4}$, the square-roots of $a \in \mathbb{F}_p$ can be obtained as $\pm\sqrt{a} = \pm a^{\frac{p+1}{4}}$.

Given $u, v \in \mathbb{F}_p$ and $v \neq 0$, the calculation of $x = \pm\sqrt{\frac{u}{v}} \in \mathbb{F}_p$ can be performed faster by merging the calculation of inversion and square-root. Thus, defining x as

$$\begin{aligned}
 x &= \left(\frac{u}{v}\right)^{\frac{p+1}{4}} = u^{\frac{p+1}{4}} v^{-\frac{p+1}{4}} \\
 &= u^{\frac{p-3+4}{4}} v^{(p-1)-\frac{p+1}{4}} = u^{\frac{p-3}{4}+1} v^{\frac{3p-5}{4}} \\
 &= u v (uv^3)^{\frac{p-3}{4}}.
 \end{aligned} \tag{3.6.6}$$

This calculation has two cases: if $x^2 v = u$, then $x = \pm\sqrt{\frac{u}{v}}$; otherwise, the square-root does not exist. In RFC-8032 [162], the same calculation is performed as $x = u^3 v (u^5 v^3)^{\frac{p-3}{4}}$; however, Equation (3.6.6) is faster since it saves two multiplications and one square.

For the case of $\mathbb{F}_{p_{448}}$, the exponentiation $a^{\frac{p-3}{4}}$ can be calculated using an analogous method as for calculating multiplicative inverses. It follows that $a^{\frac{p-3}{4}} = (\alpha_{223})^{2^{223}} \alpha_{222}$. Note that this factor appears in the calculation of inverses, so we can reuse the same addition chain for both purposes reducing the code size of the software library.

3.6.2 Redundant Representation

The redundant representation, stated in Definition 3.2.6, requires to select a value of ρ for representing elements of the prime field. According to the bounds defined in Equations (3.2.21) and (3.2.22), the size of digits must be lower than $w = 32$ for enabling the

use of vector instructions. Since p_{448} is a 448-bit number, we opt for splitting the numbers in $l = 16$ digits of size $\rho = 28$ bits because this selection sets the size of sequences to be a power of two, which is particularly useful for implementing Karatsuba multiplication. Setting $\rho < 28$ increases the length of sequences, which incurs in more digit multiplications and larger storage requirements. Therefore, an element $a \in \mathbb{F}_{p_{448}}$ is represented by any sequence $A = (a_{15}, \dots, a_0)$ of length $l = 16$ such that $a \equiv \sum_{i=0}^{15} 2^{28i} a_i \pmod{p_{448}}$. The digits of a sequence are stored into 128-bit vector registers as shown in Figure 3.6.7.

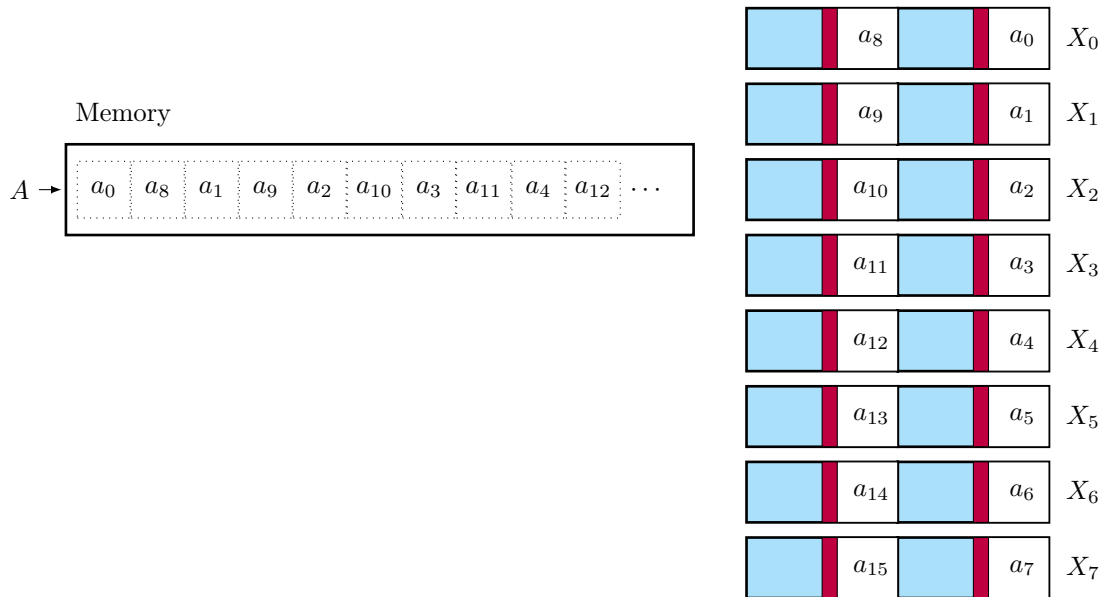


Figure 3.6.7: A sequence of digits $A = (a_{15}, \dots, a_0)$ representing $a \in \mathbb{F}_{p_{448}}$ is stored into eight 128-bit registers X_0, \dots, X_7 .

Addition

The addition of sequences $C = A + B$ is performed by scheduling eight 128-bit vector additions PADDQ as

$$\begin{aligned}
 [c_8 \ c_0] &\leftarrow [a_8 \ a_0] + [b_8 \ b_0] \\
 [c_9 \ c_1] &\leftarrow [a_9 \ a_1] + [b_9 \ b_1] \\
 [c_{10} \ c_2] &\leftarrow [a_{10} \ a_2] + [b_{10} \ b_2] \\
 [c_{11} \ c_3] &\leftarrow [a_{11} \ a_3] + [b_{11} \ b_3] \\
 [c_{12} \ c_4] &\leftarrow [a_{12} \ a_4] + [b_{12} \ b_4] \\
 [c_{13} \ c_5] &\leftarrow [a_{13} \ a_5] + [b_{13} \ b_5] \\
 [c_{14} \ c_6] &\leftarrow [a_{14} \ a_6] + [b_{14} \ b_6] \\
 [c_{15} \ c_7] &\leftarrow [a_{15} \ a_7] + [b_{15} \ b_7].
 \end{aligned}$$

Although the addition of digits could produce some carry bits, these bits are contained inside of the 64 bit registers. So, no carry propagation is required for calculating additions.

Subtraction

Unlike the calculation of additions, the calculation of subtractions can produce some negative digits. Although a sequence with negative digits is an equivalent sequence of the

subtraction of sequences, we restricted digits to be always positive. To avoid calculating negative digits, we perform the subtraction as $C = A - B + P$, where P is a redundant sequence of p_{448} such that $|p_i| = 29$ for $0 \leq i < 16$, more specifically

$$\begin{aligned}
 P = (p_{15}, \dots, p_0) = & (0x1ffffffe, 0x1ffffffe, 0x1ffffffe, 0x1ffffffe, \\
 & 0x1ffffffe, 0x1ffffffe, 0x1ffffffc, 0x3ffffffc, \\
 & 0x1ffffffe, 0x1ffffffe, 0x1ffffffe, 0x1ffffffe, \\
 & 0x1ffffffe, 0x1ffffffe, 0x1ffffffe, 0x1ffffffe).
 \end{aligned} \tag{3.6.8}$$

Using this auxiliary sequence, the subtraction of digits is always positive assuming that the size of the input digits is $\rho = 28$ bits. This operation is implemented using eight vector additions (PADDQ) and eight vector subtractions (PSUBQ).

Multiplication

We implemented two multiplication algorithms the Mastrovito multiplier and the Karatsuba algorithm, and compared their performance.

The Mastrovito multiplier has the main advantage that several multiplications can be issued independently, which can be suitable for taking advantage of the throughput of the PMULUDQ instruction. Recall that for multiplying $C = A \times B$, the sequence C is obtained as $C = \text{Circ}(A) \times B$, where $\text{Circ}(A) = [A, \pi(A), \dots, \pi^{l-1}(A)]$. To instantiate this algorithm, we define

$$\begin{aligned}
 \pi : & (a_{15}, a_{14}, a_{13}, a_{12}, a_{11}, a_{10}, a_9, a_8, a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) \\
 \mapsto & (a_{14}, a_{13}, a_{12}, a_{11}, a_{10}, a_9, a_8, a_7 + a_{15}, a_6, a_5, a_4, a_3, a_2, a_1, a_0, a_{15}).
 \end{aligned} \tag{3.6.9}$$

As can be seen, π rotates the digits of the input sequence and updates the eighth digit with $a_7 + a_{15}$. The rotation of digits can be efficiently performed using register renaming, which is accomplished at compilation time at no cost during execution. Thus, the calculation of π reduces to calculate one digit addition.

One negative result of the implementation of the Mastrovito multiplier is that it requires lots of memory accesses, which are translated in performance penalties. For example, to store C and A are required 32 live registers, but since the processor only has sixteen this causes that some registers be spilled to memory more often, which downgrades the performance of this implementation significantly.

As a better alternative, we implemented the Karatsuba [168] multiplication method instead. The main advantage of this algorithm when is instantiated with this prime field is that elements have a larger number of words making suitable the use of this sub-quadratic time-complexity algorithm. Moreover, since the number of digits is a power of two, then the Karatsuba algorithm can partition the sequences evenly.

We experimentally determined the optimal recursion level that minimizes the running time of the integer multiplication and obtained the following timings.

Recursion Level	0	1	2	3	4
Clock Cycles	250	164	143	163	183

The zero recursion level represents the time taken by the execution of the schoolbook multiplication method. It can be seen that by increasing the recursion level, the time to compute integer multiplication is gradually reduced until a point in which the cost of performing integer additions dominates the total cost of the multiplication. According to our experiments, we found that the inflection point was reached at the third recursion level. Hence, we implement Karatsuba algorithm using two recursion levels and the remainder multiplications are calculated using the schoolbook method. Algorithm 3.6.10 shows the steps followed in the first recursion level of the Karatsuba multiplier.

Algorithm 3.6.10 Karatsuba multiplication merged with reduction modulo p_{448} .

Input: $A = (a_0, \dots, a_{15})$ and $B = (b_0, \dots, b_{15})$, two sequences of digits.

Output: $C = (c_0, \dots, c_{15})$, a sequence of digits such that $C = A \times B$.

Integer Multiplication

- 1: $(s_0, \dots, s_7) \leftarrow (a_0 + a_8, \dots, a_7 + a_{15})$
- 2: $(t_0, \dots, t_7) \leftarrow (b_0 + b_8, \dots, b_7 + b_{15})$
- 3: $(x_0, \dots, x_{15}) \leftarrow \text{Karatsuba}_8((a_0, \dots, a_7), (b_0, \dots, b_7))$
- 4: $(y_0, \dots, y_{15}) \leftarrow \text{Karatsuba}_8((a_8, \dots, a_{15}), (b_8, \dots, b_{15}))$
- 5: $(z_0, \dots, z_{15}) \leftarrow \text{Karatsuba}_8((s_0, \dots, s_7), (t_0, \dots, t_7))$

Karatsuba recombination merged with reduction modulo p_{448}

- 6: **for** $i \leftarrow 0$ **to** 7 **do**
 - 7: $c_i \leftarrow x_i + y_i + z_{i+8} - x_{i+8}$
 - 8: $c_{i+8} \leftarrow y_{i+8} + z_i + z_{i+8} - x_i$
 - 9: **end for**
 - 10: **return** $C = (c_0, \dots, c_{15})$
-

Reduction modulo p_{448}

It is possible to merge integer multiplication and reduction modulo p_{448} . The prime p_{448} resembles the minimal polynomial $\varphi^2 - \varphi - 1$ of the golden ratio $\varphi = \frac{a+b}{a} = \frac{a}{b}$ setting $\varphi = 2^{224}$. Hamburg [141] pointed out this fact and showed that given $A = A_0 + 2^{224}A_1$ and $B = B_0 + 2^{224}B_1$, the product $C = A \times B$ is calculated with three multiplications as

$$\begin{aligned} C &= A \times B = (A_0 + 2^{224}A_1) \times (B_0 + 2^{224}B_1) \\ &= (A_0B_0 + A_1B_1) + 2^{224}((A_0 + A_1)(B_0 + B_1) - A_0B_0). \end{aligned} \quad (3.6.11)$$

We go one step further to optimize the calculation of C . We use the common sub-expression elimination technique to perform the reduction modulo p_{448} . We can split the product A_0B_0 in a pair of sequences X_0 and X_1 such that $A_0B_0 = X_0 + 2^{224}X_1$. Analogously, we define $A_1B_1 = Y_0 + 2^{224}Y_1$, and $(A_0 + A_1)(B_0 + B_1) = Z_0 + 2^{224}Z_1$. Thus, C is calculated as

$$C = C_0 + 2^{224}C_1 = (X_0 + Y_0 + Z_1 - X_1) + 2^{224}(Y_1 + Z_0 + Z_1 - X_0). \quad (3.6.12)$$

Lines 6-9 of Algorithm 3.6.10 calculate the Karatsuba recombination merged with the reduction modulo p_{448} from the partial products calculated by the Karatsuba recursion.

Digit Size Reduction

Since $\rho = 28$, given a sequence C such that $|C| \geq 28$ we want to obtain an equivalent sequence D such that $C \sim D$ and $|D| \leq 28$. In Section 3.2.3, we presented two algorithms for implementing this operation, and in this section we describe their implementation using vector registers.

The implementation of the sequential version (described in Algorithm 3.2.28) requires to propagate the most-significant bits of each digit to the next digit in the sequence. Since digits are stored into 128-bit registers as Figure 3.6.7 shows, every time we propagate bits from the digit a_i to the digit a_{i+1} , we are also propagating bits from the digit a_{i+8} to the digit a_{i+9} for $0 \leq i < 7$ as follows

$$[a_8 \ a_0] \rightarrow [a_9 \ a_1] \rightarrow [a_{10} \ a_2] \rightarrow [a_{11} \ a_3] \rightarrow [a_{12} \ a_4] \rightarrow [a_{13} \ a_5] \rightarrow [a_{14} \ a_6] \rightarrow [a_{15} \ a_7] .$$

The propagation of bits of the digit a_7 to a_8 is a special case, since at the same time the most significant bits of the digit a_{15} must be reduced modulo p_{448} . To do that, we add the most-significant bits of a_{15} denoted by x to the digit a_0 and a_8 . Finally, this last addition can result into a digit greater than 2^{28} . To avoid that, we perform two propagation steps from the digits a_0 and a_8 as

$$[a_8 + x \ a_0 + x] \rightarrow [a_9 \ a_1] \rightarrow [a_{10} \ a_2] .$$

Unlike the sequential version, the parallel version (described in Algorithm 3.2.29) allows to calculate more operations simultaneously. We implement Algorithm 3.2.29 using eight PAND, eight PSRLI, and nine PADDQ vector instructions.

Performance Comparison between Representations

Table 3.6.13 reports timings of arithmetic operations using 64-bit scalar instructions and 128-bit vector instructions.

Table 3.6.13: Time in clock cycles of $\mathbb{F}_{p_{448}}$ operations measured on Skylake.

Implementation	ISA	A	M	S	Digit Size Reduction		I
					Sequential	Parallel	
Radix- 2^{64}	x64	14	98	94	–	–	43,100
Redundant $\rho = 28$	AVX2	6	87	78	25	14	33,400

For this prime field, the implementation of arithmetic operations renders better performance using vector instructions. Part of this improvement is due to the representation of elements. Recalling that the redundant representation allows to perform some operations in parallel, this case is observed in the calculation of additions where a vector addition is twice as fast than using 64-bit instructions.

Regarding the calculation of multiplications, the performance exhibited by using the Karatsuba algorithm saves some clock cycles with respect to use the schoolbook multiplication. This implementation shows that for sequences of sixteen digits is worthwhile to apply Karatsuba algorithm instead of using a quadratic time-complexity algorithm.

The same argument applies to the squaring operation. We want to remark that the ratio S/M is 0.95 for the 64-bit implementation and is 0.89 for the 128-bit implementation. As a reference, the ratio is usually around 0.8 in generic implementations. Thus, these numbers indicate that in both implementations the time for calculating squares is closer to the time consumed by multiplications.

As a result of the improved performance of squaring using vector instructions, this improvement also reduces the cost of calculating multiplicative inverses. From the numbers on Table 3.6.13, it can be seen that inversion is 22% faster when using the vector implementation. This is a notable example in which the vector instructions do accelerate an inherently sequential operation.

3.6.3 Two-way Operations

We implement functions that calculate two-way prime field operations. Following the implementation techniques described in the last section, we extended the calculation of one operation using 128-bit instructions to calculate two operations using 256-bit instructions. Let A and B be two sequences, then the tuple $\langle A, B \rangle$ is stored into eight 256-bit vector registers as Figure 3.6.14 shows.

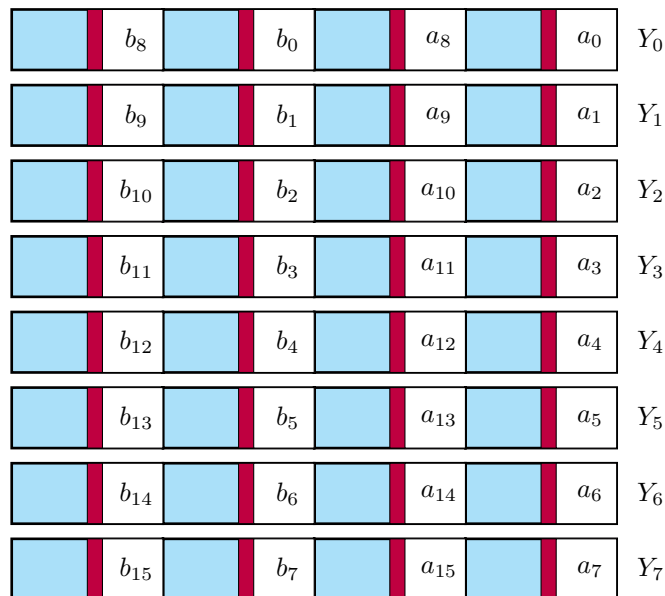


Figure 3.6.14: The notation $\langle A, B \rangle$ represents the distribution of digits of two sequences A and B stored into eight 256-bit registers.

The implementation of two-way prime field operations allows increasing the throughput for calculating operations. We leverage this fact to calculate more operations but taking almost the same latency as performing only one operation. As the operations are performed inside the lanes of 128 bits, it avoids moving data between these two lanes. In our experiments, we observed a performance penalty when using permutation instructions that move data across the 128-bit lanes. Therefore, to achieve better efficiency using two-way prime field operations, the schedule of operations must avoid in as much as possible to move data between lanes.

Performance Benchmark

Table 3.6.15 shows the timings of the prime field operations measured in Skylake. The first two rows have exactly the same entries as the ones displayed in Table 3.6.13. The last row shows the speedup factor yielded by the use of two-way prime field operations and is calculated as $2T_{\text{single}}/T_{\text{two-way}}$, where T_{single} is the minimum time to compute a prime field operation either using x64 or SSE instructions, and $T_{\text{two-way}}$ is the time taken by a two-way operation.

Table 3.6.15: Time in clock cycles of two-way $\mathbb{F}_{p_{448}}$ operations measured on Skylake.

Operations	Radix	ISA	A	M	S	Digit Size Reduction	
						Sequential	Parallel
Single	Radix- 2^{64}	x64	14	98	94	–	–
	$\rho = 28$	SSE	6	87	78	25	14
Two-way	$\rho = 28$	AVX2	9 $1.33\times$	114 $1.52\times$	86 $1.81\times$	25 $2.00\times$	14 $2.00\times$

Extending the implementation of arithmetic operations to the use of 256-bit vectors resulted on significant speedups on the timings for calculating two-way prime field operations. In particular, the speedup factor obtained for multiplications is $1.52\times$, which means that calculating two multiplication in parallel is 34% faster than executing two consecutive calls to the single operation implementation. Moreover, calculating squares is 44% faster. These numbers can improve the timings of workloads that require parallel calculations of prime field operations.

3.6.4 Four-way Operations

The four-way arithmetic operations uses 256-bit vector registers. In this case, each 64-bit word of the 256-bit vector register stores one digit of four sequences. Let A , B , C , and D be four sequences, the tuple $\langle A, B, C, D \rangle$ denotes the distribution of their digits as shown in Figure 3.6.16.

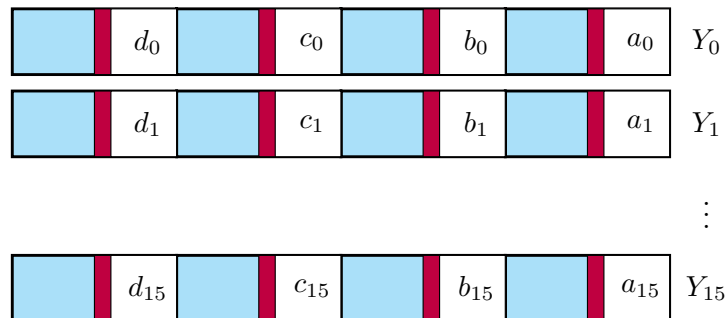


Figure 3.6.16: Digit distribution of $\langle A, B, C, D \rangle$ to perform four-way prime field operations using 256-bit registers.

Performance Benchmark

Table 3.6.17 shows the timings of four-way operations measured on Skylake. The last row of the table summarizes the acceleration factor calculated as $4T_{\text{single}}/T_{\text{four-way}}$, where T_{single} is the minimum time to compute a prime field operation either using x64 or SSE instructions, and $T_{\text{four-way}}$ is the time taken by a four-way operation.

The cycle counts of the four-way operations reveal that a significant portion of time can be saved in comparison to single operations. For example, the calculation of a four-way multiplication takes 149 clock cycles, which is 57% faster than using the single implementation four times. The implementation of four-way operations increases the throughput of workloads that calculate several prime field operations independently.

Table 3.6.17: Time in clock cycles of four-way $\mathbb{F}_{p_{448}}$ operations measured on Skylake.

Operations	Radix	ISA	A	M	S	Digit Size Reduction	
						Sequential	Parallel
Single	Radix-2 ⁶⁴ $\rho = 28$	x64	14	98	94	–	–
		SSE	6	87	78	25	14
Two-way	$\rho = 28$	AVX2	9 1.33×	114 1.52×	86 1.81×	25 2.00×	14 2.00×
Four-way	$\rho = 28$	AVX2	18 1.33×	149 2.33×	105 2.97×	40 2.50×	25 2.24×

3.7 Arithmetic on $\text{GF}((2^a b - 1)^2)$

We study the case of implementing operations over a field of characteristic $p = 2^a b - 1$. These primes received lots of attention after its use on isogeny-based cryptography because supersingular elliptic curves are defined over \mathbb{F}_{p^2} .

In this section, we first show a multi-precision implementation of the REDC algorithm for reduction modulo p and present some optimizations derived from the format of these primes. Second, we describe implementation techniques for calculating arithmetic operations over the quadratic extension field.

3.7.1 Improvements on the Reduction Modulo $p = 2^a b - 1$

Multi-precision Implementation of REDC

The Montgomery REDC algorithm [195] can be extended for operating on large integers using a multi-precision representation. In this section, we show how to apply REDC to numbers represented in radix-2^w. Nonetheless, the optimizations derived from the use of special primes are also applicable to other representations with few modifications. We denote with l the number of digits required to represent n -bit integers in radix-2^w following Definition 3.2.1.

We briefly detail how to implement REDC under this representation. First, we set $R = 2^{lw}$ so we can reduce any number T modulo p such that $0 \leq T < Rp$. To do that, we iteratively apply the core operation from Equation (3.1.12). In this step, we calculate a multiple of T that can be evenly divided by 2^w to obtain a number congruent to $T/2^w \pmod{p}$. By repeating this process l times, one obtain $T/2^{lw} \pmod{p}$, which can be seen as processing l successive multiplications by $2^{-w} \pmod{p}$. Finally, we subtract p if this number is larger than p . This procedure is shown in Algorithm 3.7.1 and is equivalent to the multi-precision REDC algorithm given by Montgomery.

Algorithm 3.7.1 Montgomery’s REDC algorithm in radix- 2^w .

Constants: Define $R = 2^{lw}$ such that $R > p$ and $\gcd(2^w, p) = 1$, and $p' = -p^{-1} \pmod{2^w}$.

Input: T , an integer such that $0 \leq T < Rp$.

Output: T' , an integer such that $T' = TR^{-1} \pmod{p}$.

```

1:  $T' \leftarrow T$ 
2: for  $i \leftarrow 0$  to  $l - 1$  do
3:    $q \leftarrow (T' \pmod{2^w})p' \pmod{2^w}$ 
4:    $T' \leftarrow (T' + qp)/2^w$ 
5: end for
6: if  $T' \geq p$  then
7:    $T' \leftarrow T' - p$ 
8: end if
9: return  $T'$ 

```

We summarize the cost of REDC counting the number of w -bit (digit) multiplications. Each iteration of Algorithm 3.7.1 calculates one digit multiplication in line 3 and l digit multiplications in line 4; thus, the total cost of REDC is $l^2 + l$ digit multiplications.

The cost of performing a prime field multiplication using REDC takes $2l^2 + l$ digit multiplications. From them, l^2 multiplications calculate the integer multiplication, so more than half of the cost is due to REDC.

Several efforts for reducing the total number of digit multiplications in prime field multiplications have been proposed. The KCM method [124] implements REDC combining the Karatsuba and Comba techniques, so prime field multiplications take $\frac{7}{4}l^2 + l$ digit multiplications. Seo et al. [245] proposed a hybrid Karatsuba reduction that requires $\frac{7}{8}l^2 + l$ digit multiplications. Although both methods rely on the Karatsuba algorithm (a sub-quadratic time-complexity method), these methods are well-suited only when l is larger than certain threshold. For instance, they are advantageous on architectures with a short word size, e.g., in 8-bit or 16-bit architectures makes the size of l be larger. However, in 64-bit architectures, quadratic complexity algorithms usually perform better for some particular cases.

A different approach for speeding up REDC is relying on the use of primes with special properties. In the next section, we present the case of the primes used in isogeny-based cryptography and show new optimizations for implementing REDC efficiently.

Properties of REDC using $p = 2^a b - 1$

Section 3.1.2 presents some methods that leverage special primes for reducing the execution time of the REDC algorithm. In this section, we detail some well-known optimizations that are applicable when using primes of the form $p = 2^a b - 1$. We highlight key properties that lead us to improvements on the implementation of REDC.

An invariant of the REDC algorithm when $p = 2^a b - 1$ is that $p' = -p^{-1} \pmod{2^x}$ is always $p' = 1$ for any $0 \leq x \leq a$ as can be seen

$$\begin{aligned}
 pp^{-1} &\equiv 1 \pmod{2^x} \\
 (2^a b - 1)p^{-1} &\equiv 1 \pmod{2^x} \\
 2^x 2^{a-x} b p^{-1} - p^{-1} &\equiv 1 \pmod{2^x} \\
 -p^{-1} &\equiv 1 \pmod{2^x}.
 \end{aligned} \tag{3.7.2}$$

Therefore, if $a \geq w$, the multiplication in line 3 of Algorithm 3.7.1 is not performed anymore saving l digit multiplications of the $l^2 + l$ multiplications required in the general case. In practice, this fact was leveraged by Gueron and Krasnov [132], who accelerated the calculation of REDC using $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. Koziel et al. [176] also pointed out this fact in their implementation of SIDH.

Another improvement can be derived from Equation (3.1.12). Assuming $a \geq w$, the special form of the prime allows T' to be calculated as

$$\begin{aligned}
 T' &= (T + qp)/2^w \\
 &= (T + p'(T \bmod 2^w)p)/2^w \\
 &= (T + T_0(2^a b - 1))/2^w, \quad \text{where } T_0 = T \bmod 2^w \\
 &= (2^w T_1 + T_0 + 2^a b T_0 - T_0)/2^w, \quad \text{where } T_1 = \lfloor T/2^w \rfloor \\
 &= T_1 + 2^{a-w} b T_0.
 \end{aligned} \tag{3.7.3}$$

The dominant operation for calculating T' is a $1 \times |b|$ integer multiplication. This operation is cheaper than a $1 \times l$ integer multiplication calculated in the arbitrary-prime case. For example, assuming that b fits in $l/2$ digits, the number of digit multiplications is reduced from l^2 to $\frac{1}{2}l^2$. This fact was observed by Costello et al. [78] in their implementation of SIDH. Additionally, note that the same trick can be applied several times, say B times, as long as $a \geq Bw$. This is an equivalent observation as the one reported by Bos and Friedberger [49]. We independently arrived to a similar result when looking for the elimination of loop-carried dependencies that appear in REDC. We now explain our approach, and then we will later connect it with Bos and Friedberger's observations.

Avoiding Loop-carried Dependencies in REDC

The REDC algorithm has a data dependency that prevents calculating some operations independently. More specifically, the dependency is present on calculating q in each iteration of Algorithm 3.7.1. When T' is updated with $(T' + qp)/2^w$, the value of q for the next iteration can not be calculated before T' is updated. Hence, there exists a loop-carried dependency that avoids the calculation of q for several iterations independently.

Avoiding such a dependency is relevant since it enables more flexibility on calculating operations of REDC. Without such a dependency, the programmer can use a better scheduling of instructions that are executed faster on some architectures. Also note that multi-precision integer multiplication has no loop-carried dependencies on running values. This explains why there are several ways of implementing integer multiplications, but fewer for REDC.

We observe that the loop-carried dependency on REDC can be partially avoided when $p = 2^a b - 1$. First, we know from Equation (3.7.2) that $p' = 1$; therefore, we can efficiently perform divisions by $2^a \bmod p$, and the value of T' is alternatively updated as

$$\begin{aligned}
 T' &= (T + qp)/2^a \\
 &= (T + p'(T \bmod 2^a)p)/2^a \\
 &= (T + T_0(2^a b - 1))/2^a, \quad \text{where } T_0 = T \bmod 2^a \\
 &= (2^a T_1 + T_0 + 2^a b T_0 - T_0)/2^a, \quad \text{where } T_1 = \lfloor T/2^a \rfloor \\
 &= T_1 + bT_0.
 \end{aligned} \tag{3.7.4}$$

A division by $2^a \bmod p$ is calculated as one $a \times |b|$ integer multiplication, which can be performed faster as we explain below.

We show how to break some data dependencies in Algorithm 3.7.1 by observing the following. In the original multi-precision algorithm, the step size is w bits (processing one digit at a time). On the other hand, using this shape of primes, we can process more bits (up to a rather than w bits) per iteration. So one can perform larger steps, of any size up to a bits. More specifically, since $a \geq Bw$ for some $B > 0$, the value of q for the first B iterations is known in advance because it only depends on $T_0 = T \bmod 2^{Bw}$. In other words, there is no loop-carried dependency in B consecutive iterations.

All of these observations were discovered working in collaboration with CINVESTAV IPN's researchers and landed on an improved implementation of REDC that is tuned for this family of primes. The algorithm was published in the TC 2017 paper [103] [\(P\)](#), and is shown in Algorithm 3.7.5.

Algorithm 3.7.5 requires to set B , a parameter that indicates the step size of the multi-precision REDC. Setting $B = 1$ leads to the original algorithm that processes one digit per iteration. The value $\lambda_0 = \lfloor l/B \rfloor$ refers to the number of steps of size B , and the value $\lambda_1 = l \bmod B$ denotes the number of steps of size one. It holds that $a \geq Bw > \lambda_1 w \geq 0$.

The cost of Algorithm 3.7.5 is summarized as follows. First, b is a fixed value that is stored in $|b|_w = \lfloor |b|/w \rfloor + 1$ digits. Line 5 calculates bq , where q has B digits; thus, the product takes $B|b|_w$ digit multiplications. This product is repeated λ_0 times, which amounts to $\lambda_0 B|b|_w$ digit multiplications. Line 9 calculates bq , where q has λ_1 digits, so it takes $\lambda_1 |b|_w$ digit multiplications. In total, Algorithm 3.7.5 takes $\lambda_0 B|b|_w + \lambda_1 |b|_w = l \times |b|_w$ digit multiplications.

We perform the products by powers of two using bit-shifting. Let a_1 and a_0 be the quotient and the remainder of a/w , respectively. We calculate $2^{a-Bw}bq = 2^{(a_1-B)w} \cdot 2^{a_0}bq$ and $2^{a-\lambda_1 w}bq = 2^{(a_1-\lambda_1)w} \cdot 2^{a_0}bq$ shifting bq to the left a_0 bits. The shifts by multiples of w are performed by renaming registers at no cost. This last observation was independently pointed out by Bos-Friedberger [49] as the shifted technique (SH). Alternatively, the

Algorithm 3.7.5 Montgomery’s REDC tuned for $p = 2^ab - 1$ in radix 2^w .

Constants: Let $p = 2^ab - 1$, define $R = 2^{lw} > p$ and an integer B such that $B > 0$ and $a \geq Bw$.

Input: T , an integer such that $0 \leq T < Rp$.

Output: T' , an integer such that $T' = TR^{-1} \pmod{p}$.

```

1:  $T' \leftarrow T$ 
2:  $\lambda_0 \leftarrow \lfloor l/B \rfloor$  and  $\lambda_1 \leftarrow l \pmod{B}$ 
3: for  $i \leftarrow 1$  to  $\lambda_0$  do
4:    $q \leftarrow T' \pmod{2^{Bw}}$ 
5:    $T' \leftarrow \lfloor T'/2^{Bw} \rfloor + 2^{a-Bw}bq$ 
6: end for
7: if  $\lambda_1 \neq 0$  then
8:    $q \leftarrow T' \pmod{2^{\lambda_1w}}$ 
9:    $T' \leftarrow \lfloor T'/2^{\lambda_1w} \rfloor + 2^{a-\lambda_1w}bq$ 
10: end if
11: if  $T' \geq p$  then
12:    $T' \leftarrow T' - p$ 
13: end if
14: return  $T'$ 

```

factor 2^{a_0} can be absorbed while performing the product bq removing the shifting, however in some cases, this extra factor can increase the number of digit multiplications.

Another relevant aspect of Algorithm 3.7.5 is that allows performing bigger steps during the calculation of REDC. The algorithm can calculate up to B iterations without dependencies between them, i.e., processing Bw bits per iteration. Contrarily, the original REDC algorithm proceeds in small steps since it processes only w bits per iteration, and cannot go further due to the presence of data dependencies. Each big step requires an $m \times m'$ integer multiplication, whereas a small step requires a $1 \times m$ integer multiplication. Programmers can leverage this difference since there exist more ways of scheduling instructions for calculating $m \times m'$ multiplications than for $1 \times m$ multiplications.

What it remains is a way to find the optimal value for B . Our initial recommendation is using larger values for B because it would allow more flexibility to the programmer to implement the big steps. Alternatively, one can measure the performance of an actual implementation. So we proceed with it, and provide experimental evidence to determine to what extent our recommendation applies.

Performance Benchmark

We implemented Algorithm 3.7.5 for $p_{751} = 2^{372}3^{239} - 1$ covering three variants corresponding to whether or not the architecture supports the BMI2 and ADX instructions. Table 3.7.6 lists the instruction counts and the latency of reduction modulo p_{751} .

Observe that the latency decreases as B increases, and this holds regardless which instructions are used. These measurements corroborate that selecting larger values for B results in time savings. For example, taking as a baseline the implementation with $B = 1$ and MULQ+ADC, the reduction takes 281 cycles. Setting $B = 4$, it takes 218 cycles, which is 22% faster. Similar speedups are observed using MULX and ADX instructions.

Table 3.7.6: Instruction counts of reduction modulo p_{751} .

Implementation	B	Instruction Set	Instruction Counts				Latency ¹
			MUL	ADD	MOV	Other	
This work	1	MULQ+ADC	84	251	204	8	281
		MULX+ADC	84	191	24	8	232
		MULX+ADX	84	191	24	8	230
	2	MULQ+ADC	84	289	207	10	244
		MULX+ADC	84	257	27	10	208
		MULX+ADX	84	149	27	16	187
	3	MULQ+ADC	84	301	210	10	227
		MULX+ADC	84	281	34	10	210
		MULX+ADX	84	137	34	18	193
	4	MULQ+ADC	84	307	210	10	218
		MULX+ADC	84	292	36	10	191
		MULX+ADX	84	130	42	17	162
	$4 + (\text{SH})^2$	MULQ+ADC	72	265	186	46	204
		MULX+ADC	72	253	36	46	189
		MULX+ADX	72	118	36	55	156
Bos-Friedberger [49]	1	MULQ+ADC	84	332	157	41	254
	2	MULQ+ADC	84	358	202	61	275
	$1 + (\text{SH})^2$	MULQ+ADC	72	299	223	86	240

¹ Entries are clock cycles measured on Skylake.² SH stands for the shifted technique [49].

Comparison with Related Works

Bos and Friedberger [49] presented timings of a 64-bit implementation of REDC. However, they do not obtain performance advantages by increasing the value of B . As part of their performance results, they observed better timings setting $B = 1$ than $B = 2$, which at a first glance, it could indicate that larger values of B do not offer performance advantages. Contrary to that, we showed that our optimized 64-bit implementations reduce their execution time as B gets bigger (as shown in Table 3.7.6).

In a follow up work by Bos and Friedberger [50], they revisited the case of reduction modulo p_{751} targeting 32-bit ARM v7 architecture. In their implementation, they set $l = 24$, $w = 32$, and $B = 12$, which is equivalent to implement REDC using $\lambda_0 = 2$ big steps (according our notation). These big steps calculate integer multiplications faster using advanced multiply-and-accumulate instructions available in the ARM architecture.

Seo et al. [244] implemented SIDH targeting optimizations for ARM architectures. For the ARM v8 64-bit architecture, they implemented REDC setting $B = 4$, which accelerates the execution of reduction modulo p .

All of these independent works obtain performance improvements on REDC when using larger values of B , which confirms our hypothesis about the choice of its value.

Future Work

A different approach is using Karatsuba-based algorithms, since it saves a number of digit multiplications improving the performance of Algorithm 3.7.5 on architectures with small word size. Although we did not evaluate the performance of this variant in these architectures, this task can be turned on a follow up work for those interested on optimizing implementations on smaller architectures. The performance-critical part of this strategy is implementing Karatsuba multiplication efficiently.

3.7.2 Redundant Representation

We describe a vector implementation of arithmetic operations over \mathbb{F}_p and its quadratic extension. As a study case, we use the prime $p_{751} = 2^{372}3^{239} - 1$ due to its first use in the optimized implementation of the SIDH protocol [78].

We use a redundant representation as stated in Definition 3.2.6. According to the bounds given in Equations (3.2.21) and (3.2.22), ρ must be lower than $w' = 32$ for enabling the use of vector instructions. Since p_{751} is a 751-bit number, we opt for using $\rho = 27$ resulting in sequences of $l = 28$ digits. Also, since l is multiple of four, this eases the application of two recursion levels of the Karatsuba multiplication algorithm.

An element $a \in \mathbb{F}_{p_{751}}$ is represented by any sequence $A = (a_{27}, \dots, a_0)$ of length $l = 28$ such that $a \equiv \sum_{i=0}^{27} 2^{27i} a_i \pmod{p_{751}}$. The digits of a given sequence are stored into 128-bit vector registers using the distribution shown in Figure 3.7.7.

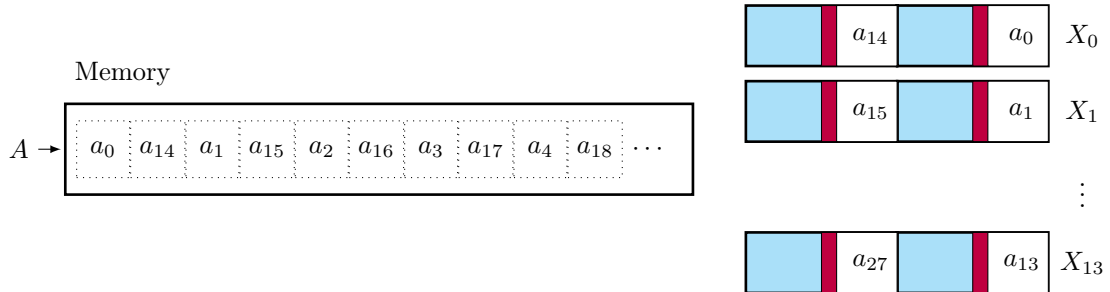


Figure 3.7.7: A sequence of digits $A = (a_{27}, \dots, a_0)$ representing $a \in \mathbb{F}_{p_{751}}$ is stored into fourteen 128-bit registers X_0, \dots, X_{13} .

Addition

The addition of sequences $C = A + B$ is performed by scheduling fourteen 128-bit vector additions PADDQ as follows

$$\begin{aligned} [c_{14} \ c_0] &\leftarrow [a_{14} \ a_0] + [b_{14} \ b_0] \\ [c_{15} \ c_1] &\leftarrow [a_{15} \ a_1] + [b_{15} \ b_1] \\ &\vdots \\ [c_{27} \ c_{13}] &\leftarrow [a_{27} \ a_{13}] + [b_{27} \ b_{13}]. \end{aligned}$$

The addition of digits can produce some carry bits, which are contained inside of 64-bit registers meaning that no carry propagation is required for calculating additions provided the input digits are reduced.

Subtraction

Unlike the calculation of additions, the calculation of subtractions can produce some negative digits. We restricted digits to be always positive by performing $C = A - B + P$, where P is a redundant sequence of p_{751} such that $|p_i| = 28$ for $0 \leq i < 28$; so we set

$$\begin{aligned}
 P = (p_{27}, \dots, p_0) = & (0xdfc\text{baa}7, 0x9f71\text{c}0\text{d}, 0x89484\text{fb}, 0x\text{deeb}718, \\
 & 0xd0\text{ac}569, 0x845\text{cb}24, 0x\text{ba}40426, 0x\text{a}619\text{f}5\text{a}, \\
 & 0xd7\text{d}0\text{eda}, 0x\text{a}959\text{b}19, 0x89\text{fbe}62, 0x\text{db}8\text{fb}24, \\
 & 0xd0\text{a}93\text{ef}, 0x\text{f}8\text{a}8\text{ee}9, 0x\text{ffffffe}, 0x\text{ffffffe}, \\
 & 0x\text{ffffffe}, 0x\text{ffffffe}, 0x\text{ffffffe}, 0x\text{ffffffe}, \\
 & 0x\text{ffffffe}, 0x\text{ffffffe}, 0x\text{ffffffe}, 0x\text{ffffffe}, \\
 & 0x\text{ffffffe}, 0x\text{ffffffe}, 0x\text{ffffffe}, 0x\text{ffffffc}0).
 \end{aligned} \tag{3.7.8}$$

Using this auxiliary sequence, the subtraction of digits is always positive assuming that the size of the input digits is $\rho = 27$ bits. This operation takes 14 PADDQ and 14 PSUBQ.

Multiplication

As the sequences have $l = 28$ digits, we process their multiplication with three products of sequences of $l/2 = 14$ digits using Karatsuba, as shown in Figure 3.7.9. We implement the reduction modulo p_{751} following Algorithm 3.7.5 and setting $B = 2$ to process two REDC iterations, one per 64-bit line of the 128-bit register.

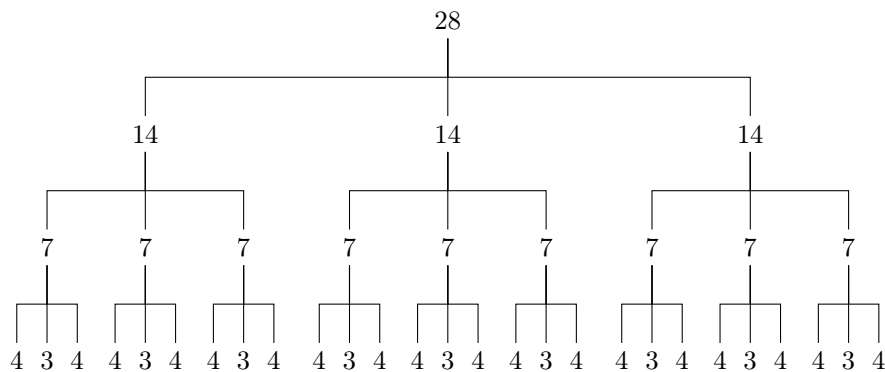


Figure 3.7.9: Recursion tree of Karatsuba multiplication of sequences of length $l = 28$. The leaves of the tree represent multiplications of sequences of length three and four using the schoolbook multiplication method.

Digit Size Reduction

Multiplying sequences of length $l = 28$ results in a sequence of $2l - 1 = 55$ digits. After calculating REDC algorithm (which clears $l = 28$ digits) the sequence gets reduced to $l - 1 = 27$ digits. However the size of each digit is increased by double, i.e., $\approx 2\rho$. At this point, one must perform a digit size reduction for obtaining an equivalent sequence with shorter digit size $\approx \rho$.

In Section 3.2.3, we present Algorithm 3.2.28 for implementing digit size reduction. This algorithm requires to propagate the most-significant bits of each digit to the next digit in the sequence. Since digits are stored into 128-bit registers as Figure 3.7.7 shows, every time we propagate bits from the digit a_i to the digit a_{i+1} , we are also propagating bits from the digit a_{i+14} to the digit a_{i+15} for $0 \leq i < 14$ as follows

$$[a_{14} \ a_0] \rightarrow [a_{15} \ a_1] \rightarrow [a_{16} \ a_2] \rightarrow \dots \rightarrow [a_{25} \ a_{11}] \rightarrow [a_{26} \ a_{12}] \rightarrow [a_{27} \ a_{13}] .$$

After performing these operations, there is required one last propagation of bits, the one that goes from a_{13} to a_{14} .

Multiplicative Inverse

Let $p = 2^a b - 1$ and assuming that $a > 2$ and $b > 0$, then it holds that $p \equiv 3 \pmod{4}$. In this setting, we can obtain the inverse of $x \in \mathbb{F}_p$ by finding an optimized chain for the exponent $k = \frac{p-3}{4}$ and calculating $x^{-1} = (x^k)^{2^2} x$.

3.7.3 Two-way Operations for the Quadratic Extension

In this section, we show the vector implementation of operations over the quadratic extension field using 256-bit registers and AVX2 instructions. We followed the construction presented in Section 3.1.3 to build a quadratic extension field assuming that $a > 2$ and $b > 0$ in $p = 2^a b - 1$, and thus having that $p \equiv 3 \pmod{4}$. These assumptions are valid for the primes used in the SIDH protocol.

The parallel strategy we followed for the implementation of field operations is similar to the one we used for implementing operations on \mathbb{F}_p for $p \in \{p_{255}, p_{384}, p_{448}\}$ described in the previous sections. However, there exists a slight difference, since we will leverage the use of 256-bit registers to store two prime field elements, but these elements correspond to the coefficients of the polynomial used to represent an element in the extension. Thus, the parallelism offered by AVX2 vector instructions is applied internally in the calculation of one operation over the quadratic extension field.

Following the notation given in Section 3.1.3, we represent an element $a_1\tau + a_0 \in \mathbb{F}_{p^2}$ by storing their coefficients as a two-way operand $\langle a_0, a_1 \rangle$ such that each coefficient is represented as sequence of l digits in the redundant representation given in the previous section. For the case of p_{751} , the machine representation is depicted in Figure 3.7.10.

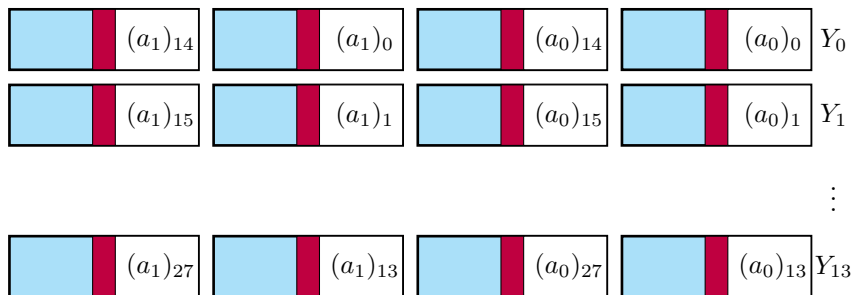


Figure 3.7.10: The notation $\langle a_0, a_1 \rangle$ represents the distribution of digits of an element $a_1\tau + a_0 \in \mathbb{F}_{p_{751}^2}$ stored into fourteen 256-bit registers.

Addition and Subtraction

Given $a_1\tau + a_0, b_1\tau + b_0 \in \mathbb{F}_{p^2}$, the calculation of their addition or subtraction is performed as polynomial operations, i.e., they are operated coefficient-wise. We store the coefficients of elements in the lower and higher part of a 256-bit register because this enables a natural extension from 128- to 256-bit vector instructions. Hence, by following the notation given in Section 3.3, we can express the addition (or subtraction) of elements in the quadratic extension as performing a two-way addition (or subtraction) of their coefficients; which is denoted as $\langle c_0, c_1 \rangle = \langle a_0, a_1 \rangle + \langle b_0, b_1 \rangle$. Since we are using a redundant representation, the carry bits produced by adding digits are not propagated allowing the calculation of several consecutive additions without propagation of bits.

Multiplication

We showed in Section 3.1.3 that the product of $a_1\tau + a_0$ and $b_1\tau + b_0$ can be calculated with four multiplications in the base field or with three multiplications if we use the Karatsuba identity. In our implementation, we opted by performing four multiplications since they are evenly distributed over the two-way operations. The two-way calculation of the REDC algorithm on $\langle a_0, a_1 \rangle$ is defined as

$$\text{REDC}: \quad \langle a_0, a_1 \rangle \mapsto \langle \text{REDC}(a_0), \text{REDC}(a_1) \rangle. \quad (3.7.11)$$

Similarly, the digit size reduction, described in the previous section, is defined analogously

$$\text{DSR}: \quad \langle a_0, a_1 \rangle \mapsto \langle \text{DSR}(a_0), \text{DSR}(a_1) \rangle. \quad (3.7.12)$$

Hence, Algorithms 3.7.13 and 3.7.14 show the steps needed for multiplying and squaring elements of the quadratic extension field.

Algorithm 3.7.13 Multiplication on \mathbb{F}_{p^2} .

Input: $\langle a_0, a_1 \rangle$ and $\langle b_0, b_1 \rangle$ such that $a_1\tau + a_0, b_1\tau + b_0 \in \mathbb{F}_{p^2}$.

Output: $\langle c_0, c_1 \rangle$ such that $c_1\tau + c_0 = (a_1\tau + a_0) \times (b_1\tau + b_0) \in \mathbb{F}_{p^2}$.

- 1: Construct $\langle a_0, a_0 \rangle$, $\langle a_1, a_1 \rangle$, and $\langle b_1, b_0 \rangle$.
 - 2: $\langle a_0b_0, a_0b_1 \rangle \leftarrow \langle a_0, a_0 \rangle \times \langle b_0, b_1 \rangle$
 - 3: $\langle a_1b_1, a_1b_0 \rangle \leftarrow \langle a_1, a_1 \rangle \times \langle b_1, b_0 \rangle$
 - 4: $\langle c_0, c_1 \rangle = \langle a_0b_0 - a_1b_1, a_0b_1 + a_1b_0 \rangle \leftarrow \langle a_0b_0, a_0b_1 \rangle \pm \langle a_1b_1, a_1b_0 \rangle$
 - 5: $\langle c_0, c_1 \rangle \leftarrow \text{REDC}(\langle c_0, c_1 \rangle)$ //Equation (3.7.11)
 - 6: $\langle c_0, c_1 \rangle \leftarrow \text{DSR}(\langle c_0, c_1 \rangle)$ //Equation (3.7.12)
 - 7: **return** $\langle c_0, c_1 \rangle$
-

Multiplicative Inverse

The SIDH protocol scarcely calculates inverses. Because of that, there is low advantage of executing in parallel their internal operations. Our implementation reorders the coefficients and performs the base field operations using 128-bit instructions.

Algorithm 3.7.14 Squaring on \mathbb{F}_{p^2} .

Input: $\langle a_0, a_1 \rangle$ such that $a_1\tau + a_0 \in \mathbb{F}_{p^2}$.

Output: $\langle c_0, c_1 \rangle$ such that $c_1\tau + c_0 = (a_1\tau + a_0)^2 \in \mathbb{F}_{p^2}$.

1: Construct $\langle a_0, a_0 \rangle$, $\langle a_1, a_1 \rangle$, and $\langle a_1, 0 \rangle$.

2: $\langle W, X \rangle \leftarrow \langle a_0, a_0 \rangle - \langle a_1, 0 \rangle$

3: $\langle Y, Z \rangle \leftarrow \langle a_1, a_1 \rangle + \langle a_0, a_1 \rangle$

4: $\langle c_0, c_1 \rangle \leftarrow \langle W, X \rangle \times \langle Y, Z \rangle$

5: $\langle c_0, c_1 \rangle \leftarrow \text{REDC}(\langle c_0, c_1 \rangle)$

//Equation (3.7.11)

6: $\langle c_0, c_1 \rangle \leftarrow \text{DSR}(\langle c_0, c_1 \rangle)$

//Equation (3.7.12)

7: **return** $\langle c_0, c_1 \rangle$

3.7.4 Performance Benchmark

Using the implementation techniques presented in the previous sections, we developed an AVX2 implementation of arithmetic operations on the quadratic extension field for the prime p_{751} , and in this section, we compare its performance with other optimized implementations. Table 3.7.15 shows timings, measured in a Skylake processor, of some arithmetic operations. The first column lists the timings of the SIDH v2 implementation [78] and the second column lists the timings of the SIKE implementation [159]. The most relevant difference between them is that the former implements multiplication operations using MULQ instructions, whereas the latter uses MULX+ADX instructions along with the implementation techniques shown in Section 3.7.1. The third column shows the timings of our implementation.

Table 3.7.15: Time in clock cycles of $\mathbb{F}_{p_{751}^2}$ operations measured on Skylake.

Domain	Operation	SIDH v2 [78]	SIKE [159]	This work
		MULQ	MULX+ADX	PMULUDQ
\mathbb{Z}	Multiplication	248	187	192
	REDC	197	147	135
	Digit size reduction	\emptyset	\emptyset	20
\mathbb{F}_{p^2}	Addition	80	79	16
	Subtraction	65	64	22
	Multiplication	1,452	957	1,223
	Squaring	1,012	772	816

From the table, it is clear that our implementation improves the calculation of additions and subtractions. This happens due to the use of a redundant representation that helps to reduce the latency of these operations significantly. For integer multiplications, our vectorized implementation is faster than the implementation using native instructions. The fastest is the implementation that use advanced multi-precision instructions MULX and ADX. The difference becomes larger on multiplication and squaring over the quadratic extension. Another interesting observation is that the cost of squaring is $1S = 0.8M$ for 64-bit implementations, whereas for vectorized implementation is $1S = 0.6M$, which is closer to the theoretical ratio and makes more convenient trading multiplications by squares.

Future Work

More investigation is needed for performing integer multiplications faster. We observed that this prime size requires a large number of registers. This causes registers be spilled to memory more often, which is time consuming because loading vector registers is more costly than loading 64-bit registers.

We investigated only the case of p_{751} letting other prime sizes for future work. Adj et al. [3] showed that SIDH can be instantiated with 448-bit primes. This prime size is more friendly with vector instructions as the operands are shorter and require fewer registers. Another possible venue is to explore the primes used in the CSIDH [62] protocol.

3.8 Chapter Summary

We showed techniques for implementing prime field operations efficiently. A relevant factor is to find a representation of large integers that allows operating over several words in parallel and with fewer dependencies. We described a redundant representation that enables the use of AVX2 vector instructions. During the implementation, we found some restrictions on the parametrization of the representation partially derived by some AVX limitations, such as the width of the vector integer multiplier.

We showed vectorized implementations of prime field operations. In our measurements, we observed that our vectorized implementations increase the throughput of the operations with a shorter increase in their latency. Larger speedups are achievable in workloads where many field operations are calculated in parallel. In the next chapter, we show how to use parallel prime field operations for processing elliptic curve operations in parallel.

Chapter 4

Arithmetic of Elliptic Curves

Elliptic curves have special properties that shine in the field of cryptography. In particular, the points on an elliptic curve have a group structure with an intractable discrete logarithm problem. Moreover, small instances of elliptic curves can achieve larger security levels, which in practice results in short cryptographic keys.

In this chapter, we review some mathematical concepts of elliptic curves and describe their arithmetic operations. We show some well-known algorithms for scalar multiplication. Our study covers three different curve models: the Weierstrass, Montgomery, and twisted Edwards curves. For each of them, we describe some algorithmic optimizations and propose parallel algorithms for point addition and scalar multiplication. These algorithms build on top of the prime field operations described in the previous chapter.

4.1 Background

We review some concepts needed for defining elliptic curves. We also cover the description of the algebraic group generated from the points of an elliptic curve, and show the hard problem associated to this group that is of interest for cryptography.

4.1.1 Elliptic Curves

The goal of this section is to present the definition of an elliptic curve. To that end, we start with a special equation that will help us to land on a standard curve equation known as the Weierstrass equation. Then, we describe certain properties that this equation must satisfy to become an elliptic curve.

The Weierstrass Equation

Let K be a finite field, and E be a family of curves parametrized by constant values $a_1, a_2, a_3, a_4, a_6 \in K$ defined as

$$E: \quad y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6. \quad (4.1.1)$$

It is said that E is defined over K , denoted as E/K , whenever both its parameters and its solutions are elements of K .

Each curve E has associated two invariants. The value Δ is the discriminant of E and indicates whether the curve has singularities (points at which both partial derivatives are equal to 0). If $\text{char}(K) \neq 2$, the discriminant is

$$\begin{aligned}\Delta &= -b_2^2 b_8 - 8b_4^3 - 27b_6^2 + 9b_2 b_4 b_6, \text{ where} \\ b_2 &= a_1^2 + 4a_2, \\ b_4 &= 2a_4 + a_1 a_3, \\ b_6 &= a_3^2 + 4a_6, \text{ and} \\ b_8 &= a_1^2 a_6 + 4a_2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 - a_4^2.\end{aligned}\tag{4.1.2}$$

When $\Delta = 0$, E is singular; otherwise, E is a smooth curve, usually referred as a non-singular or ordinary curve. The other is called the j -invariant of E and is defined as

$$j = \frac{(b_2^2 - 24b_4)^3}{\Delta},\tag{4.1.3}$$

which indicates that curves with the same j -invariant are in the same isomorphism class. Given $j' \in K$ there exists a curve E with j -invariant equal to j' (See [250, Prop. 1.4]).

After an admissible change of variables, Equation (4.1.1) can be transformed to an equivalent simplified equation:

$$E/K: \begin{cases} y^2 + xy = x^3 + ax^2 + b, & \text{if } \text{char}(K) = 2, \\ y^2 = x^3 + ax^2 + b, & \text{if } \text{char}(K) = 3, \\ y^2 = x^3 + ax + b, & \text{if } \text{char}(K) \neq 2, 3. \end{cases}\tag{4.1.4}$$

The first two cases represent curves defined over fields of small characteristic and are known as binary and ternary curves. In the third case, the curve is defined over fields of large characteristic, and its equation is historically known as *the Weierstrass equation*.

From here on, we will work with non-singular curves defined over fields of large characteristic. So, $K = \mathbb{F}_q$ is a finite field of characteristic $p > 3$, where $q = p^m$ and $m \geq 1$.

Rational Points

The solutions of a Weierstrass equation can be described using algebraic sets. Let $\mathbb{A}^2(\mathbb{F}_q)$ be an affine space over \mathbb{F}_q , and $\mathbb{F}_q[x, y]$ be the ring of polynomials in two variables with coefficients in \mathbb{F}_q , the points of $\mathbb{A}^2(\mathbb{F}_q)$ that evaluated in $f(x, y) = y^2 - x^3 - ax - b \in \mathbb{F}_q[x, y]$ give zero are an algebraic set denoted as

$$\{(x, y) \in \mathbb{A}^2(\mathbb{F}_q) : f(x, y) = 0\}.\tag{4.1.5}$$

This set contains the affine points of E/\mathbb{F}_q corresponding to the solutions over \mathbb{F}_q of the Weierstrass equation $y^2 = x^3 + ax + b$.

The projective space $\mathbb{P}^n(\mathbb{F}_q)$ is the quotient set of non-zero elements in $(\mathbb{F}_q)^{n+1}$ under an equivalence relation \sim . Under this relation, $(X_0, \dots, X_n) \sim (Y_0, \dots, Y_n)$ if it exists $\lambda \in \mathbb{F}_q \setminus \{0\}$ such that $X_i = \lambda Y_i$ for $0 \leq i \leq n$. The equivalence classes are commonly referred as projective points and are denoted as $(X_0 : \dots : X_n)$.

The affine space $\mathbb{A}^n(\mathbb{F}_q)$ can be embedded into the projective space $\mathbb{P}^n(\mathbb{F}_q)$ as

$$\begin{aligned} \mathbb{A}^n(\mathbb{F}_q) &\rightarrow \mathbb{P}^n(\mathbb{F}_q) \\ (x_0, \dots, x_{n-1}) &\mapsto (x_0 : \dots : x_{n-1} : 1), \end{aligned} \quad (4.1.6)$$

with inverse

$$\begin{aligned} \mathbb{P}^n(\mathbb{F}_q) &\rightarrow \mathbb{A}^n(\mathbb{F}_q) \\ (X_0 : \dots : X_{n-1} : X_n) &\mapsto (X_0/X_n, \dots, X_{n-1}/X_n) \text{ for } X_n \neq 0. \end{aligned} \quad (4.1.7)$$

The projective points for which $X_n = 0$ are known as points at infinity.

The points on a Weierstrass curve can also be described as projective algebraic sets over $\mathbb{P}^2(\mathbb{F}_q)$. This can be achieved by embedding the affine points of the Weierstrass curve into $\mathbb{P}^2(\mathbb{F}_q)$ using $(x, y) \mapsto (x : y : 1)$. The projective version of the Weierstrass equation is obtained by changing the variables $x = X/Z$ and $y = Y/Z$, and clearing denominators resulting in an homogeneous polynomial $f \in \mathbb{F}_q[X, Y, Z]$ of degree three.

Definition 4.1.8 (\mathbb{F}_q -rational points of E). Given a Weierstrass curve E , define the homogeneous polynomial $f(X, Y, Z) = Y^2Z - X^3 - aXZ^2 - bZ^3 \in \mathbb{F}_q[X, Y, Z]$. The \mathbb{F}_q -rational points of E is the algebraic set denoted as

$$E(\mathbb{F}_q) = \{(X, Y, Z) \in \mathbb{P}^2(\mathbb{F}_q) : f(X, Y, Z) = 0\}, \quad (4.1.9)$$

which contains the projection of all affine points of E and one point at infinity denoted as $\mathcal{O} = (0 : 1 : 0)$.

As proved in [250, Theorem 1.1], Hasse showed that $\#E(\mathbb{F}_q)$ is bounded as

$$|\#E(\mathbb{F}_q) - q - 1| \leq 2\sqrt{q}. \quad (4.1.10)$$

Therefore, the number of points on $E(\mathbb{F}_q)$ is as large as the size of \mathbb{F}_q .

Elliptic Curve Definition

Definition 4.1.11 (Elliptic Curve [250]). An *elliptic curve* is a pair (E, \mathcal{O}) , where E is a non-singular curve of genus¹ one and $\mathcal{O} \in E$.

The standard example of an elliptic curve is given by a non-singular Weierstrass curve E together with \mathcal{O} as the distinguished point. Note that the genus of the curve is equal to one because the Weierstrass curve is non-singular ($\Delta \neq 0$) and its projective equation has degree equal to three.

Well-known forms of elliptic curves are Montgomery [196], Edwards [90], and twisted Edwards [26], the Jacobi quartic [40], Kummer [114], Hessian [66], and twisted Hessian [27] curves. Each of these curves has an equation (or set of equations) used for its definition, but each model can be represented by a Weierstrass curve equation. However, the opposite is not true, not all Weierstrass curves can be mapped to each model.

¹The genus of a plane curve of degree d is calculated as $g = \frac{1}{2}(d-1)(d-2) - s$, where s is the number of singularities.

4.1.2 The Group of Rational Points

The \mathbb{F}_q -rational points (described in Definition 4.1.8) satisfy all the properties, given in Definition 3.1.1, of an algebraic group. Thus, there exist a binary operation such that given two points calculates a third point in the set. We describe such an operation by stating first a law of composition.

Law of Composition in the Rational Points

The chord-and-tangent rule is a geometric method that given two points on a Weierstrass curve obtains a third point also lying on the curve. This geometric construction provides a law of composition in the \mathbb{F}_q -rational points of a Weierstrass curve.

Definition 4.1.12 (Chord-and-Tangent Rule [250, Proposition 2.1]). Let E be a Weierstrass curve and $P, Q \in E$. Take L as the line through P and Q (if $P = Q$, take L as the tangent line to E at P), and let R be the third point of intersection of L with E . Let L' be the line through R and \mathcal{O} . Then L' intersects E at R , \mathcal{O} , and a third point, which is denoted as $P + Q$. See Figure 4.1.13.

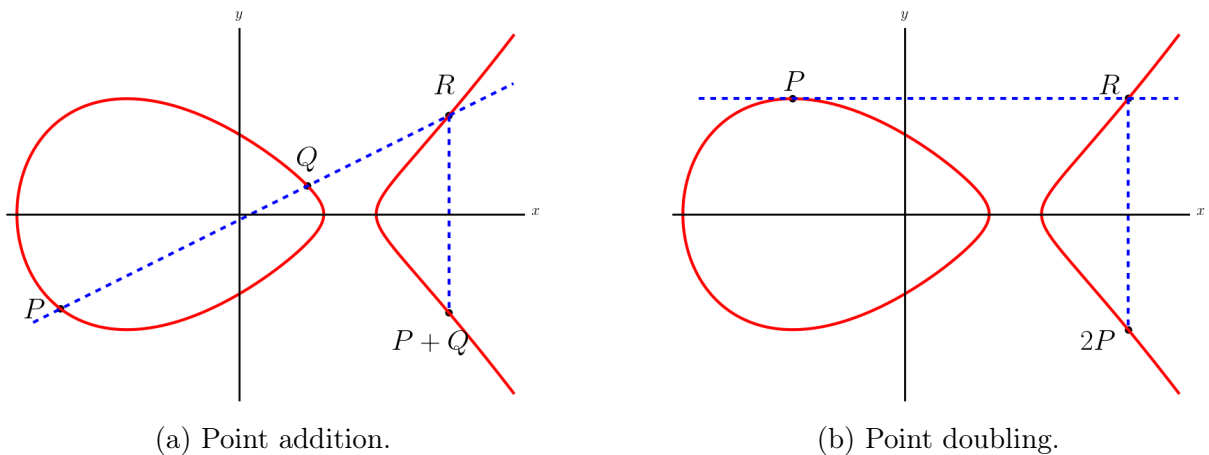


Figure 4.1.13: Geometric description of the chord-and-tangent rule.

Using this law of composition, the \mathbb{F}_q -rational points satisfy all the properties of a commutative group (as proved in Silverman's book [250, Proposition 2.2]). Thus, $(E(\mathbb{F}_q), +)$ is an abelian group with identity \mathcal{O} . The operation of calculating $P + Q$ from P and Q is generally known as a *point addition*, but its special case when $P = Q$ is known as *point doubling* and is denoted by $2P$.

The chord-and-tangent rule can also be described algebraically as a set of rational functions that depend on the coordinates of the points P and Q , and on the parameters of the Weierstrass curve. Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be two affine points on $E(\mathbb{F}_q)$, the coordinates of $P + Q$ are calculated as

$$x_{P+Q} = \left(\frac{y_Q - y_P}{x_Q - x_P} \right)^2 - x_P - x_Q, \text{ and } y_{P+Q} = \left(\frac{y_Q - y_P}{x_Q - x_P} \right) (x_P - x_{P+Q}) - y_P, \quad (4.1.14)$$

whenever $P \neq \pm Q$; otherwise, if $P = -Q$, $P + Q = \mathcal{O}$; and if $P = Q$, the coordinates of $2P$ are

$$x_{2P} = \left(\frac{3x_P^2 + A}{2y_P} \right)^2 - 2x_P, \text{ and } y_{2P} = \left(\frac{3x_P^2 + A}{2y_P} \right) (x_P - x_{2P}) - y_P. \quad (4.1.15)$$

The computational cost of evaluating Equation (4.1.14) is $1\mathbf{M} + 1\mathbf{S} + 6\mathbf{A} + 1\mathbf{I}$ field operations and Equation (4.1.15) takes $1\mathbf{M} + 2\mathbf{S} + 8\mathbf{A} + 1\mathbf{I}$ field operations.

It must be noted that a single formula cannot calculate the composition law of any two \mathbb{F}_q -rational points. In this case, the point addition formula is said to be *incomplete*. For example, the formula in Equation (4.1.14) is incomplete because it is not defined whenever P and Q have the same x -coordinate and requiring of an alternative formula for handling these exceptional cases.

Bosma and Lenstra [52] proved that the minimum set of addition formulas that is *complete* has cardinality two. This means that one formula can be used in replacement when the other fails on calculating a point addition. Kohel et al. [16, 174] noted that one of the formulas is sufficient to add any $P, Q \in E(\mathbb{F}_q)$ provided that the y -coordinate of $P - Q$ be non-zero. Thus, one can use a single formula for adding points provided that its exceptional points did not appear as operands of the addition.

More generally, if the exceptional points of a formula are not defined in \mathbb{F}_q (e.g., the exceptional point has coordinates in an extension field), the formula operates on points with coordinates in \mathbb{F}_q , and thus, no exceptions will be encountered. In this situation, it is said that the formula is \mathbb{F}_q -complete.

Scalar Multiplication

Given a point $P \in E(\mathbb{F}_q)$, it is possible to generate other points on the curve by calculating repeated addition of a point, i.e., $P + P$, $P + P + P$, and so on. The repeated application of point addition is a common operation that is formalized below.

Definition 4.1.16 (Scalar multiplication). Given a point $P \in E(\mathbb{F}_q)$ and an integer m , the scalar multiplication, denoted as mP , is defined as

$$(m, P) \mapsto \begin{cases} \mathcal{O}, & \text{if } m = 0 \text{ or } P = \mathcal{O}, \\ mP = \underbrace{P + P + \cdots + P}_{m \text{ operands}}, & \text{if } m > 0, \\ -m(-P), & \text{if } m < 0. \end{cases} \quad (4.1.17)$$

A point $P \neq \mathcal{O}$ can generate other points calculating mP for any integer; however, by proceeding in this way, P only generates a finite number of different points before generating \mathcal{O} . More formally, the order of P is the smallest integer $\text{ord}(P) = r > 0$ such that $rP = \mathcal{O}$. This implies that every point P generates a subgroup of $E(\mathbb{F}_q)$, which is denoted as $\langle P \rangle \subseteq E(\mathbb{F}_q)$. The group structure of $E(\mathbb{F}_q)$ is isomorphic to $\mathbb{Z}/n_1\mathbb{Z} \times \mathbb{Z}/n_2\mathbb{Z}$, where n_1 and n_2 are the unique integers such that n_2 divides n_1 , and n_2 divides $q - 1$.

Isogenies

Another general operation between the points of elliptic curves are called isogeny maps.

Definition 4.1.18 (Isogeny). Let (E_0, O_{E_0}) and (E_1, O_{E_1}) be two elliptic curves, an isogeny is a non-constant rational map $\phi: E_0 \rightarrow E_1$ satisfying $\phi(O_{E_0}) = O_{E_1}$.

The kernel of an isogeny are the set of points that are mapped to the identity of the codomain curve.

For any integer m , there exists an isogeny called multiplication-by- m defined as

$$\begin{aligned} [m]: E(\mathbb{F}_q) &\rightarrow E(\mathbb{F}_q) \\ P &\mapsto mP. \end{aligned} \tag{4.1.19}$$

Its kernel is known as the m -torsion subgroup of $E(\mathbb{F}_q)$ and is denoted as

$$E[m] = \{P \in E(\mathbb{F}_q) : mP = \mathcal{O}\}. \tag{4.1.20}$$

Let p be the characteristic of \mathbb{F}_q , and for any integer $m > 0$ such that $p \nmid m$, the group structure of $E[m](\mathbb{F}_q) \simeq \mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/m\mathbb{Z}$. Moreover, elliptic curves over a field of characteristic p are classified, for all $r > 0$, as

$$E[p^r](\overline{\mathbb{F}_q}) \simeq \begin{cases} \mathbb{Z}/p^r\mathbb{Z}, & E \text{ is ordinary,} \\ \{\mathcal{O}\}, & E \text{ is supersingular.} \end{cases} \tag{4.1.21}$$

For $p \geq 5$, the number of points of a supersingular curve $E(\mathbb{F}_p)$ is equal to $p + 1$.

4.1.3 Elliptic Curve Discrete Logarithm Problem

In the group of points of an elliptic curve, there exists a one-way function (a function that is easy to compute, but hard to invert) that is useful for cryptographic purposes. In one direction, calculating scalar multiplications is easy, whereas in the other direction, calculating discrete logarithms is a hard task.

Definition 4.1.22 (Elliptic Curve Discrete Logarithm Problem (ECDLP)). Let $G \in E(\mathbb{F}_q)$ be a point generating the subgroup $\langle G \rangle \subseteq E(\mathbb{F}_q)$ of order n . Given a point $P \in \langle G \rangle$, the elliptic curve discrete logarithm problem is to find an integer k such that $0 \leq k < n$ and $P = kG$.

The best-known algorithms that solve ECDLP are the Pohlig-Hellman [225] and the Pollard's rho [226] algorithms. The latter has a time-complexity $O(\sqrt{d})$ where d is the largest prime factor of n . Hence, the hardest instances of ECDLP are those in which n is a prime number. For example, if n is an m -bit prime number, solving ECDLP requires $O(2^{m/2})$ operations.

The difficulty to solve ECDLP allows constructing public-key cryptography using elliptic curves. To target a security level of m bits requires to instantiate an elliptic curve²

²In addition to the size of the group, other aspects must be considered to generate strong instances of elliptic curve groups, see SafeCurves project [36].

whose largest (sub)group has order 2^{2m} . Regarding the scalability of a cryptosystem, the parameters of ECDLP grow slower than the parameters of RSA-based cryptosystems. This means that by increasing the security level of a cryptosystem, smaller elliptic curves achieve equivalent security as using larger RSA-parameters. In practice, this translates in the use of smaller keys for equivalent security levels.

4.2 Algorithms for Scalar Multiplication

We now review several algorithms for scalar multiplication. The algorithms presented take an integer $k \in \mathbb{Z}$ and a point $P \in E(\mathbb{F}_q)$ to calculate $kP \in E(\mathbb{F}_q)$. Let $r = \#E(\mathbb{F}_q)$ be the order of the group, so k is bounded to the interval $0 \leq k < r$. Let n be the size in bits of r , then k is stored in memory using an n -bit binary representation. The time-complexity of algorithms for scalar multiplication is $O(n)$. The algorithms described in this section apply generally. Thus, no special assumptions are considered on the point representation or on the elliptic curve model.

4.2.1 Basic Algorithms

We describe some well-known methods for calculating scalar multiplications. They have in common that the number of operations strongly depends on the value of the scalar so its execution pattern is not regular.

Binary Methods

The double-and-add multiplication algorithm is analogous to the square-and-multiply method for exponentiation. This method, shown in Algorithm 4.2.1, initializes an accumulator point Q with \mathcal{O} and iteratively scans the bits of the scalar k from the most-significant bit to the least-significant bit. For each bit of the scalar, the algorithm doubles Q , and then it adds P to Q only if the current bit is set. This method is also known as a *left-to-right* algorithm due to the order in which the bits of the scalar are evaluated.

A variant of the double-and-add algorithm is obtained by scanning the bits in reverse order. The *right-to-left* method, shown in Algorithm 4.2.2, initializes an accumulator point Q with \mathcal{O} . Then, for every bit of the scalar, it accumulates P on Q only if the bit is set, and doubles P unconditionally. As can be noted, there is a duality between them.

The number of operations of these algorithms is determined as follows. Both algorithms conditionally perform a point addition when a bit is set. More specifically, there are calculated $H_w(k)$ point additions, where H_w is the Hamming weight function. In average, calculating a scalar multiplication takes $n/2$ point additions and n point doublings.

The binary methods rely on the binary representation of the scalar, however, the representation of scalars can also be extended to other bases and numeral systems as shown in the next sections.

Algorithm 4.2.1 Left-to-Right Binary Algorithm for Scalar Multiplication.

Input: $P \in E(\mathbb{F}_q)$ and k is a positive integer.

Output: $kP \in E(\mathbb{F}_q)$.

- 1: Let $(k_{n-1}, \dots, k_0)_2$ be the n -bit representation of k .
 - 2: $Q \leftarrow \mathcal{O}$
 - 3: **for** $i \leftarrow n - 1$ **to** 0 **do**
 - 4: $Q \leftarrow 2Q$
 - 5: **if** $k_i = 1$ **then**
 - 6: $Q \leftarrow Q + P$
 - 7: **end if**
 - 8: **end for**
 - 9: **return** Q
-

Algorithm 4.2.2 Right-to-Left Binary Algorithm for Scalar Multiplication.

Input: $P \in E(\mathbb{F}_q)$ and k is a positive integer.

Output: $kP \in E(\mathbb{F}_q)$.

- 1: Let $(k_{n-1}, \dots, k_0)_2$ be the n -bit representation of k .
 - 2: $Q \leftarrow \mathcal{O}$
 - 3: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
 - 4: **if** $k_i = 1$ **then**
 - 5: $Q \leftarrow Q + P$
 - 6: **end if**
 - 7: $P \leftarrow 2P$
 - 8: **end for**
 - 9: **return** Q
-

Fixed-Window Method

A first generalization of the previous methods is switching from radix-two (binary) to radix- 2^w , which results on the 2^w -ary scalar multiplication method. This method is also known as the fixed-window method since it is parametrized by an integer $w \geq 1$, the window parameter, that is used to represent the scalar k in radix- 2^w . Thus, given $k > 0$, there are $t = \lceil n/w \rceil$ digits $(k_{t-1}, \dots, k_0)_{2^w}$ such that $k = \sum k_i 2^{wi}$ and $k_i \in \{0, \dots, 2^w - 1\}$.

The fixed-window method, shown in Algorithm 4.2.3, follows a similar execution pattern as the one in Algorithm 4.2.1. In fact, the left-to-right binary algorithm is a special case of the fixed-window method setting $w = 1$. Algorithm 4.2.3 first initializes an accumulator point Q with \mathcal{O} and scans the digits of the scalar from the most-significant to the least-significant digit. For each digit k_i , the algorithm performs w consecutive point doublings on the accumulator and adds $k_i P$ to Q only if k_i is non-zero. It is easy to show that after scanning all the digits of k , the accumulator Q will be equal to kP .

A refinement of this method takes in account that given a point P , calculating its inverse $-P$ is fast. This property is not always valid in other groups; for example, in $(\mathbb{Z}/n\mathbb{Z})^\times$, the calculation of modular inverses is significantly more expensive than calculating modular multiplications. Hence, the scalar is encoded as a sequence of signed digits. Setting $t = \lceil n/w \rceil$, there are t signed digits $(k_{t-1}, \dots, k_0)_{2^w}$ such that $k = \sum k_i 2^{iw}$ and each digit $k_i \in \{-2^{w-1}, \dots, 2^{w-1} - 1\}$ is either odd or zero. To calculate scalar multiplications, Algorithm 4.2.3 follows the same approach as before; however, when a negative digit is found, the inverse of the point is accumulated instead.

The window parameter is used to tweak the number of point additions to be performed. As can be seen, the algorithm calculates $tw \approx n$ point doublings whereas the number of point additions averages to $n/2^w$. So the number of point additions gets reduced as the value of the window w increases. On the other hand, the space of all possible digits grows

exponentially on w , which is relevant when adding the multiples $k_i P$ to the accumulator. Even if these multiples are precomputed and stored in memory, the memory footprint easily gets bigger as w increases. Therefore, the window parameter is commonly used for adjusting the trade-off between the time and memory requirements of the algorithm.

Algorithm 4.2.3 Fixed-Window Algorithm for Scalar Multiplication.

Input: $P \in E(\mathbb{F}_q)$, k is a positive integer, and $w \geq 1$ is the window parameter.

Output: $kP \in E(\mathbb{F}_q)$.

- 1: Let $(k_{t-1}, \dots, k_0)_{2^w}$ be the 2^w -ary (signed digit) representation of k .
 - 2: $Q \leftarrow \mathcal{O}$
 - 3: **for** $i \leftarrow t - 1$ **to** 0 **do**
 - 4: $Q \leftarrow 2^w Q$
 - 5: **if** $k_i \neq 0$ **then**
 - 6: $Q \leftarrow Q + k_i P$
 - 7: **end if**
 - 8: **end for**
 - 9: **return** Q
-

The Non-Adjacent Form

The non-adjacent form (NAF) is a signed-digit representation that minimizes the number of non-zero digits of the scalar. More generally, the ω -NAF representation [252] of an integer k is the unique sequence of digits $(k_{l-1}, \dots, k_0)_{\omega\text{-NAF}}$ such that $k = \sum_{i=0}^{l-1} k_i 2^i$. The digits of an ω -NAF representation hold the following properties: $k_{l-1} \neq 0$, every non-zero k_i is an odd integer in the set $\{-2^{\omega-1}, \dots, 2^{\omega-1} - 1\}$, and $l \leq n + 1$, where n is the size in bits of k . Algorithm 4.2.4 shows how to obtain the ω -NAF representation of an integer.

The ω -NAF scalar multiplication method mimics the left-to-right binary method as shown in Algorithm 4.2.5. The main loop of this algorithm calculates, in average, $\frac{l}{\omega+1}$ point additions and l point doublings. Thus, as ω gets larger, the number of point additions gets reduced, whereas the table T_P grows exponentially. Optimal values for ω are usually found experimentally per case basis.

4.2.2 Algorithms with Regular Execution Pattern

In elliptic curve cryptography, scalars represent secret values commonly. The scalar multiplication methods shown in the previous section expose several sources of information that are directly related to the value of the scalar. Some of these sources are, for example, physical variables that can be measured during the execution of the algorithm. Attackers can use this information for recovering secret values. So, it is important implementations do not reveal secret values during its execution.

To prevent leaking secrets, a secure way of implementing algorithms is following a regular execution pattern. That is, the number of operations must not depend on secret values. We show that some multiplication algorithms already possess an inherent regular execution pattern with respect to the scalar. These algorithms are preferred in cryptography as they help to prevent exposure of secret information.

Algorithm 4.2.4 Conversion of Integers to ω -NAF Representation.

Input: ω and k , two integers such that $\omega \geq 2$ and $k > 0$.

Output: $(k_{l-1}, \dots, k_0)_{\omega\text{-NAF}}$, the ω -NAF representation of k .

```

1:  $i \leftarrow 0$ 
2: while  $k \geq 1$  do
3:   if  $k$  is odd then
4:      $k_i \leftarrow (k \bmod 2^\omega) - 2^{\omega-1}$ 
5:      $k \leftarrow k - k_i$ 
6:   else
7:      $k_i \leftarrow 0$ 
8:   end if
9:    $k \leftarrow k/2$ 
10:   $i \leftarrow i + 1$ 
11: end while
12:  $l \leftarrow i$ 
13: return  $(k_{l-1}, \dots, k_0)$ 

```

Algorithm 4.2.5 The ω -NAF Algorithm for Scalar Multiplication.

Input: $P \in E(\mathbb{F}_q)$, and ω and k , two integers such that $\omega \geq 2$ and $k > 0$.

Output: $kP \in E(\mathbb{F}_q)$.

```

1: Let  $(k_{l-1}, \dots, k_0)_{\omega\text{-NAF}}$  be the  $\omega$ -NAF representation of  $k$ . //Algorithm 4.2.4
2:  $T_P[i] \leftarrow (2i + 1)P$ , for  $0 \leq i < 2^{\omega-2}$ .
3:  $Q \leftarrow \mathcal{O}$ 
4: for  $i \leftarrow l - 1$  to 0 do
5:    $Q \leftarrow 2Q$ 
6:    $j \leftarrow (|k_i| - 1)/2$ 
7:   if  $k_i > 0$  then
8:      $Q \leftarrow Q + T_P[j]$ 
9:   else if  $k_i < 0$  then
10:     $Q \leftarrow Q - T_P[j]$ 
11:   end if
12: end for
13: return  $Q$ 

```

Regular Signed-Digit Multiplication

A well-known algorithm that has a regular execution pattern is the one proposed by Joye and Tunstall [165]. This method relies on converting the scalar to a signed-digit representation that exhibits a deterministic pattern of non-zero digits. Given an integer $w \geq 2$, and an odd integer k such that $0 < k < 2^n$, define $l = \lceil \frac{n}{w-1} \rceil$. Algorithm 4.2.6 converts k to a sequence of $l + 1$ signed digits (k_l, \dots, k_0) such that $k = \sum_{i=0}^l k_i 2^{(w-1)i}$ and $k_i \in \{\pm 1, \dots, \pm 2^{w-1} - 1\}$ is odd.

The regular signed-digit algorithm, shown in Algorithm 4.2.7, is a left-to-right method that calculates a scalar multiplication using $w - 1$ point doublings and one point addition per each digit in the signed representation of the scalar. Due to the scalars are bounded to 2^n , the length of the signed representation is fixed, and consequently, the total number of operations is constant.

Note that the signed-digit conversion only works for odd scalars. The even scalars can be handled by converting $k' = k + 1$ and updating the final point $Q \leftarrow Q - P$ at the end of the algorithm. If this correction is also performed using a regular execution pattern, this method is suitable for calculating scalar multiplications when k is a secret value.

Algorithm 4.2.6 Conversion of Integers to Signed-Digit Representation.

Input: (n, k, w) , integers such that $n \geq 1$; k is odd and $0 \leq k < 2^n$; and $w \geq 2$.

Output: (k_l, \dots, k_0) , the signed-digit representation of k .

```

1:  $l \leftarrow \lceil n/(w-1) \rceil$ 
2: for  $i \leftarrow 0$  to  $l-1$  do
3:    $k_i \leftarrow (k \bmod 2^w) - 2^{w-1}$ 
4:    $k \leftarrow (k - k_i)/2^{w-1}$ 
5: end for
6:  $k_l \leftarrow k$ 
7: return  $(k_l, \dots, k_0)$ 

```

Algorithm 4.2.7 Regular Signed-Digit Algorithm for Scalar Multiplication.

Input: $P \in E(\mathbb{F}_q)$, and (n, k, w) are integers such that $n \geq 1$; k is odd and $0 \leq k < 2^n$; and $w \geq 2$.

Output: $kP \in E(\mathbb{F}_q)$.

```

1: Let  $(k_l, \dots, k_0)$  be the signed-digit representation of  $k$ . //Algorithm 4.2.6
2:  $T_P[i] \leftarrow (2i+1)P$  for  $0 \leq i < 2^{w-2}$ 
3:  $Q \leftarrow \mathcal{O}$ 
4: for  $i \leftarrow l-1$  to  $0$  do
5:    $Q \leftarrow 2^{w-1}Q$ 
6:    $j \leftarrow (|k_i| - 1)/2$ 
7:    $Q \leftarrow Q + \text{sgn}(k_i)T_P[j]$ 
8: end for
9: return  $Q$ 

```

4.2.3 Ladder Algorithms

Ladder multiplication algorithms are of primary interest since they exhibit a regular execution pattern regardless the value of the inputs. Although some non-regular algorithms can achieve regularity by introducing additional (dummy) operations, in ladder methods all operations are effective. We describe two well-known examples of ladder algorithms.

Montgomery [196] described a left-to-right multiplication method known as the Montgomery ladder. It is shown in Algorithm 4.2.8, and starts initializing two accumulators, and for each bit of the scalar, it calculates one point doubling and one point addition. The bit of the scalar indicates which accumulator must be doubled, and the other one accumulates the result of the point addition.

Joye's ladder, shown in Algorithm 4.2.9, is a right-to-left multiplication algorithm proposed by Joye [163]. Like in Montgomery ladder, there are two accumulator points that are updated every iteration by one point doubling and one point addition. However, in this ladder algorithm, the bit of the scalar determines which accumulator is doubled before is added with the other point. Due to their similitude, several works [119, 267] pointed out a relation of duality between them.

As it can be seen, the pattern of operations is regular in both algorithms. In every iteration, it always calculates one point doubling and one point addition regardless the value of the bits of the scalar. Consequently, the number of operations remains constant depending on n only. For this reason, the ladder algorithms show an inherent regular execution pattern useful for calculating scalar multiplications when the scalar is secret.

Algorithm 4.2.8 Montgomery Ladder for Scalar Multiplication.

Input: $P \in E(\mathbb{F}_q)$ and k is an integer such that $0 \leq k < 2^n$.

Output: $kP \in E(\mathbb{F}_q)$.

```

1: Let  $(k_{n-1}, \dots, k_0)_2$  be the  $n$ -bit representation of  $k$ .
2:  $R_0 \leftarrow \mathcal{O}$ ,  $R_1 \leftarrow P$ 
3: for  $i \leftarrow n - 1$  to  $0$  do
4:   if  $k_i = 0$  then
5:      $R_1 \leftarrow R_0 + R_1$ 
6:      $R_0 \leftarrow 2R_0$ 
7:   else
8:      $R_0 \leftarrow R_0 + R_1$ 
9:      $R_1 \leftarrow 2R_1$ 
10:  end if
11: end for
12: return  $R_0$ 

```

Algorithm 4.2.9 Joye Ladder for Scalar Multiplication.

Input: $P \in E(\mathbb{F}_q)$ and k is an integer such that $0 \leq k < 2^n$.

Output: $kP \in E(\mathbb{F}_q)$.

```

1: Let  $(k_{n-1}, \dots, k_0)_2$  be the  $n$ -bit representation of  $k$ .
2:  $R_0 \leftarrow \mathcal{O}$ ,  $R_1 \leftarrow P$ 
3: for  $i \leftarrow 0$  to  $n - 1$  do
4:   if  $k_i = 0$  then
5:      $R_1 \leftarrow 2R_1$ 
6:      $R_1 \leftarrow R_1 + R_0$ 
7:   else
8:      $R_0 \leftarrow 2R_0$ 
9:      $R_0 \leftarrow R_0 + R_1$ 
10:  end if
11: end for
12: return  $R_0$ 

```

Secure Implementation of Ladder Algorithms

Although ladder multiplication algorithms have an intrinsic countermeasure against timing attacks due to their regularity, this is not enough to prevent against other types of side-channel attacks. Now, we describe some countermeasures and primitives that allow hardening the implementation of these algorithms.

Secure software development forbids the use of *if-then-else* statements whenever the branching condition depends on a secret value. The reasoning behind this restriction is that compilers usually implement control statements using branching instructions; this leads to secret-dependent branches that are easily detectable at running time. Because of that a secure implementation must replace secret-dependent branches with countermeasures that reduce the surface of this type of attacks.

Conditional execution is a programming technique that replaces an operation depending on a secret value, such as a bifurcation, by a set of arithmetic and/or logic operations that execute the same operation. Applying this technique to if-then-else statements is a common countermeasure to remove branches that depend on secret data. For efficiency, the body of an if-then-else statement must be as short as possible, for example, consisting of a single instruction or a variable assignment.

Conditional execution of variable assignments is covered by two well-known primitives: conditional swap and conditional move. Let $x, y \in \{0, 1\}^n$ be bit strings of length n , and $b \in \{0, 1\}$ be a secret. The *conditional swap*, $\text{CSWAP}(x, y, b)$, is a function that interchanges the values of x and y if $b \neq 0$; otherwise, the values remain unaltered. The *conditional move*, $\text{CMOV}(x, y, b)$, sets x with y if $b \neq 0$; otherwise, the values remain unaltered.

We show how to use these conditional execution primitives for implementing Montgomery ladder (see Algorithms 4.2.10 and 4.2.11) and Joye ladder (see Algorithm 4.2.12). In all the cases, the if-then-else statements are removed and the content of variables is updated by conditional primitives. Thus, elliptic curve operations are always performed on fixed memory locations and without the use of branching instructions.

Algorithm 4.2.10 CSWAP-based Montgomery Ladder Algorithm for Scalar Multiplication.

Input: $P \in E(\mathbb{F}_q)$ and k is an integer such that $0 \leq k < 2^n$.

Output: $kP \in E(\mathbb{F}_q)$.

- 1: Let $(k_{n-1}, \dots, k_0)_2$ be the n -bit representation of k , and define $k_n = 0$.
 - 2: $R_0 \leftarrow \mathcal{O}$, $R_1 \leftarrow P$.
 - 3: **for** $i \leftarrow n - 1$ **to** 0 **do**
 - 4: $b \leftarrow k_i \oplus k_{i+1}$
 - 5: $(R_0, R_1) \leftarrow \text{CSWAP}(R_0, R_1, b)$
 - 6: $(R_0, R_1) \leftarrow (2R_0, R_0 + R_1)$
 - 7: **end for**
 - 8: $(R_0, R_1) \leftarrow \text{CSWAP}(R_0, R_1, k_0)$
 - 9: **return** R_0
-

Algorithm 4.2.11 CMOV-based Montgomery Ladder Algorithm for Scalar Multiplication.

Input: $P \in E(\mathbb{F}_q)$ and k is an integer such that $0 \leq k < 2^n$.

Output: $kP \in E(\mathbb{F}_q)$.

- 1: Let $(k_{n-1}, \dots, k_0)_2$ be the n -bit representation of k , and define $k_n = 0$.
 - 2: $R_0 \leftarrow \mathcal{O}$, $R_1 \leftarrow P$.
 - 3: **for** $i \leftarrow n - 1$ **to** 0 **do**
 - 4: $b \leftarrow k_i \oplus k_{i+1}$
 - 5: $R_2 \leftarrow \text{CMOV}(R_0, R_1, b)$
 - 6: $(R_0, R_1) \leftarrow (2R_2, R_0 + R_1)$
 - 7: **end for**
 - 8: $R_0 \leftarrow \text{CMOV}(R_0, R_1, k_0)$
 - 9: **return** R_0
-

Algorithm 4.2.12 CSWAP-based Joye Ladder Algorithm for Scalar Multiplication.

Input: $P \in E(\mathbb{F}_q)$ and k is an integer such that $0 \leq k < 2^n$.

Output: $kP \in E(\mathbb{F}_q)$.

- 1: Let $(k_{n-1}, \dots, k_0)_2$ be the n -bit representation of k , and define $k_{-1} = 0$.
 - 2: $R_0 \leftarrow \mathcal{O}$, $R_1 \leftarrow P$.
 - 3: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
 - 4: $b \leftarrow k_i \oplus k_{i-1}$
 - 5: $(R_0, R_1) \leftarrow \text{CSWAP}(R_0, R_1, b)$
 - 6: $(R_0, R_1) \leftarrow (2R_0, R_0 + R_1)$
 - 7: **end for**
 - 8: $(R_0, R_1) \leftarrow \text{CSWAP}(R_0, R_1, k_{n-1})$
 - 9: **return** R_0
-

Implementing CSWAP and CMOV Primitives

The security of these algorithms relies on the existence of conditional primitives that must be implemented without leaking any information related to the value of b . The most common way to implement these conditional primitives is using logic arithmetic instructions. This approach relies on the construction of an n -bit mask whose value depends on b and is used to select one of two n -bit values. Assuming a two's complement representation of integers, the mask can be generated storing b in an n -bit register and

subtracting this register from zero, or equivalently, performing an integer negation on b . Thus, the mask will contain all-zeros if $b = 0$, or all-ones if $b = 1$.

Algorithms 4.2.13 and 4.2.14 show, respectively, the implementation of CSWAP and CMOV using logic arithmetic operations. The implementation runs without branches and executes the same operations regardless the value of b , which turns the implementation into a constant-time, regular execution code.

Algorithm 4.2.13 CSWAP Implemented with Logic Arithmetic.

Input: $x, y \in \{0, 1\}^n$ and $b \in \{0, 1\}$.

Output: $(x', y') = \begin{cases} (x, y), & \text{if } b = 0; \\ (y, x), & \text{otherwise.} \end{cases}$

- 1: $m \leftarrow 0 - b$ //Using two's complement.
 - 2: $d \leftarrow m \wedge (x \oplus y)$
 - 3: $x' \leftarrow x \oplus d$
 - 4: $y' \leftarrow y \oplus d$
 - 5: **return** (x', y')
-

Algorithm 4.2.14 CMOV Implemented with Logic Arithmetic.

Input: $x, y \in \{0, 1\}^n$ and $b \in \{0, 1\}$.

Output: $x' = \begin{cases} x, & \text{if } b = 0; \\ y, & \text{otherwise.} \end{cases}$

- 1: $m \leftarrow 0 - b$ //Using two's complement.
 - 2: $x' \leftarrow (\neg m \wedge x) \oplus (m \wedge y)$
 - 3: **return** x'
-

We identify two advantages on the use of CMOV compared to CSWAP. First, the execution of a conditional move is faster than a conditional swap because CMOV overwrites only one of the inputs; this turns out to be relevant when working with large bit vectors. Second, as it can be seen in Algorithm 4.2.11 the point addition of Line 6 can be performed before the CMOV, since CMOV only selects the point that will be doubled. This observation was independently identified during the implementation of Kummer curves [29], where their authors obtained savings by reordering operations.

Although using logic operations is a good countermeasure protecting against timing attacks, it is not enough to prevent other attacks. The use of bit masks can be vulnerable against power analysis attacks on certain architectures. Nascimento et al. [199] showed an attack to an implementation of Montgomery ladder running on an AVR micro-controller. The attack recovers the value of the mask by correlating power consumption variations with the all-zeros or all-ones stored in the mask. Thus, mask-based implementations could not be the best approach in all the cases.

We describe an alternative approach as an additional countermeasure against power analysis attacks. More specifically, we recommend the use of native processor instructions for implementing CSWAP and CMOV. The release of Pentium Intel's micro-architecture introduced CMOVCC, which is an instruction that performs a conditional move between registers according to a selection bit available in the FLAGS register. Implementing CSWAP and CMOV with CMOVCC instruction has the advantage that no (large) bit-masks are required preventing exposure of secret bits.

4.2.4 Special Cases

All the previous methods handle the general case of scalar multiplication, i.e., there are no assumptions about the inputs. When inputs hold certain properties some optimizations

can be applied. This section lists some particular cases of scalar multiplication that appear in cryptographic algorithms.

Fixed-Point Multiplication

Fixed-point multiplication refers to the calculation of a scalar multiplication in which the input point is known in advance (or fixed) and the scalar is an arbitrary secret value. In cryptographic algorithms, the input point is usually the generator of the elliptic curve group. Since this point is fixed for all computations, scalar multiplications can be calculated faster using more efficient algorithms.

There exists a vast collection of articles for fixed-point multiplications [55, 96, 104, 139, 144, 145, 180]. The general approach of these algorithms relies on the pre-computation of multiples of the known point. These points are calculated offline and stored in read-only memory. At running time, the multiplication algorithm fetches some of these multiples from memory using indexes derived from the scalar. This evaluation method calculates fixed-point multiplications faster than generic scalar multiplication algorithms.

The use of pre-computation introduces several trade-offs between the performance and the memory footprint of an implementation. To speed up the implementation one must use more points but it also requires more memory for their storage. The trade-off between time and space can be adjusted depending on several factors such as the execution environment and the targeted architecture.

Compounded to that, the use of pre-computed tables brings another constraint regarding security due to the scalar is secret. Specifically, accessing memory using secret indexes is easily detectable by an attacker that monitors the effects produced in the hierarchy of cache memory. These attacks are known as cache attacks [22]. A trivial yet effective countermeasure is to fetch every entry of the table and to conditionally select the entry corresponding of the index queried. Thus, the attacker has no clue on what value was used in the calculation. Hence, increasing size of precomputed tables can downgrade performance due to the cost of retrieving values securely.

Multiple-Point Multiplication

Some cryptographic protocols require to calculate a linear combination of points, i.e., to calculate $\sum_{i=0}^{t-1} k_i P_i$ for an integer $t > 1$. This operation is generally known as multi-exponentiation, but in the elliptic curve case, is also known as multiple-point multiplication. Depending on the application scalars could be secret values.

If scalars are not secret, the interleaving method [112] can be used for multiple-point multiplication. Algorithm 4.2.15 merges the evaluation of t instances of the left-to-right binary algorithm. Here all the point additions are performed on a single accumulator point, while point doublings are shared for all the scalar multiplications. Assuming scalars are in the range $0 \leq k_i < 2^n$, the total number of operations is $\sum_{i=0}^{t-1} H_w(k_i)$ point additions and n point doublings, where H_w is the Hamming weight function.

Algorithm 4.2.15 Interleaved Algorithm for Multiple-Point Multiplication.

Input: A set of t pairs (k_i, P_i) , where k_i is an integer such that $0 \leq k_i < 2^n$, and $P_i \in E(\mathbb{F}_q)$ for $0 \leq i < t$.

Output: $\sum_{i=0}^{t-1} k_i P_i \in E(\mathbb{F}_q)$.

```

1: Let  $(k_{i(n-1)}, \dots, k_{i(0)})_2$  be the  $n$ -bit binary representation of  $k_i$  for  $0 \leq i < t$ .
2:  $Q \leftarrow \mathcal{O}$ 
3: for  $j \leftarrow n - 1$  to  $0$  do
4:    $Q \leftarrow 2Q$ 
5:   for  $i \leftarrow 0$  to  $t - 1$  do
6:     if  $k_{i(j)} \neq 0$  then
7:        $Q \leftarrow Q + P_i$ 
8:     end if
9:   end for
10: end for
11: return  $Q$ 

```

Double-Point Multiplication

Double-point multiplication refers to a particular case of multiple-point multiplication that appears during the verification of digital signatures [8, 31]. In this setting, a double-point multiplication stands for the calculation of the point $k_0P + k_1Q$ restricted to the following conditions: P is an arbitrary point, Q is a point known in advance, k_0, k_1 are arbitrary scalars, and none of the inputs represents a secret value. These conditions allow accelerating the execution time of this operation by using standard optimization techniques, such as using pre-computed tables, applying scalar conversions, and using algorithms with non-regular execution patterns.

Algorithm 4.2.16 shows a method, described in [143, Alg. 3.51], for double-point multiplication. This method combines the interleaved algorithm [112] with the use of the ω -NAF numeral system [252]. An advantage of this method is that the size of pre-computed tables is independent for each point, which allows increasing the ω value that corresponds to the known point.

So far we have covered generic algorithms for scalar multiplications. We reviewed some algorithms with a regular execution pattern and their implementation. We also showed special operations on points that often appear in cryptographic protocols. Now, we center our attention in specific models of elliptic curves starting with the Weierstrass curves, which are the standard form of any elliptic curve.

Algorithm 4.2.16 Interleaved Algorithm with ω -NAF for Double-Point Multiplication.

Precompute: Calculate $T_Q[j] = (2j + 1)Q$ for all $0 \leq j < 2^{\omega_Q - 2}$.

Input: k_0, k_1 are positive integers, and $P, Q \in E(\mathbb{F}_q)$ such that Q is a known point, and ω_P, ω_Q are positive integers such that $\omega_P, \omega_Q \geq 2$.

Output: $k_0P + k_1Q \in E(\mathbb{F}_q)$.

- 1: Calculate $T_P[j] = (2j + 1)P$ for all $0 \leq j < 2^{\omega_P - 2}$.
- 2: Let $(k_{0(s_0-1)}, \dots, k_{0(0)})_{\omega_P\text{-NAF}}$ be the ω -NAF representation of k_0 . //Algorithm 4.2.4
- 3: Let $(k_{1(s_1-1)}, \dots, k_{1(0)})_{\omega_Q\text{-NAF}}$ be the ω -NAF representation of k_1 . //Algorithm 4.2.4
- 4: $s \leftarrow \max(s_0, s_1)$
- 5: Define $k_{0(i)} \leftarrow 0$, for $s_0 \leq i < s$; and $k_{1(i)} \leftarrow 0$, for $s_1 \leq i < s$.
- 6: $R \leftarrow \mathcal{O}$
- 7: **for** $i \leftarrow s - 1$ **to** 0 **do**
- 8: $R \leftarrow 2R$
- 9: **if** $k_{0(i)} \neq 0$ **then**
- 10: $j \leftarrow (|k_{0(i)}| - 1)/2$
- 11: $R \leftarrow R + \text{sgn}(k_{0(i)})T_P[j]$
- 12: **end if**
- 13: **if** $k_{1(i)} \neq 0$ **then**
- 14: $j \leftarrow (|k_{1(i)}| - 1)/2$
- 15: $R \leftarrow R + \text{sgn}(k_{1(i)})T_Q[j]$
- 16: **end if**
- 17: **end for**
- 18: **return** R

4.3 Arithmetic of Weierstrass Curves

This section describes the implementation of the point addition law for elliptic curves in the Weierstrass form $y^2 = x^3 + Ax + B$ over fields of characteristic $p > 3$. We review point addition formulas in the projective space, including well-known coordinate systems and the recently-optimized complete formulas. We also present parallel algorithms for the complete formulas and their implementation with SIMD instructions.

4.3.1 Point Addition Formulas

The geometric description of the group addition law can also be expressed algebraically as rational functions. The affine formulas given by Equations (4.1.14) and (4.1.15) are an example of this equivalence. Since formulas are quotients of polynomials, point addition formulas must calculate expensive field inverse operations. To avoid calculating inverses, point operations can be performed on a projective space.

Incomplete Projective Formulas

The affine points on a Weierstrass curve are embedded into \mathbb{P}^2 as $(x, y) \mapsto (x : y : 1)$ and $\mathcal{O} \mapsto (0 : 1 : 0)$; and its inverse map is $(X : Y : Z) \mapsto (X/Z, Y/Z)$ and $(0 : 1 : 0) \mapsto \mathcal{O}$. Using the formulas given in [66, 68], adding two projective points takes $12\mathbf{M} + 2\mathbf{S} + 7\mathbf{A}$ field operations, and point doublings take $5\mathbf{M} + 6\mathbf{S} + 1\mathbf{M}_A + 12\mathbf{A}$ field operations.

Alternatively, the Jacobian system of coordinates [66, 194] refers to a projective embedding of the Weierstrass curve. In this setting, the affine points are mapped to a weighted projective space as $(x, y) \mapsto (x : y : 1) \in \mathbb{P}^2$ and $\mathcal{O} \mapsto (1 : 1 : 0)$. The inverse mapping is $(X : Y : Z) \mapsto (X/Z^2, Y/Z^3)$ when $Z \neq 0$ and $(1 : 1 : 0) \mapsto \mathcal{O}$. By applying this map to the Weierstrass curve E/\mathbb{F}_q , it results in the homogeneous polynomial $f(X, Y, Z) = Y^2 - X^3 - AXZ^4 - BZ^6$ that is used to define the \mathbb{F}_q -rational points of E .

The arithmetic of points in Jacobian coordinates is performed without inverses in the base field. Using the formulas given by Cohen et al. [68], point additions are calculated using $12\mathbf{M} + 4\mathbf{S} + 7\mathbf{A}$ field operations, and point doublings take $3\mathbf{M} + 6\mathbf{S} + 1\mathbf{M}_A + 13\mathbf{A}$ field operations, where \mathbf{M}_A is the cost of multiplying by the curve parameter A .

Although its efficiency, neither the projective nor the Jacobian formulas are not complete (like the affine formulas), since they fail on calculating point additions for some exceptional points.

Complete Formulas

Bosma and Lenstra [52] proved that a set of complete formulas for point addition has cardinality two, i.e., one of the formulas can add the exceptional points of the other formula. However, Kohel et al. [16, 174] noted that one of these formulas is \mathbb{F}_q -complete provided that $E(\mathbb{F}_q)$ has no points of order two. Thus, the formula will never encounter an exceptional points if only adds points with coordinates in \mathbb{F}_q . The fact that this \mathbb{F}_q -complete formula can add any point on the curve is relevant for secure software development.

The \mathbb{F}_q -complete addition formula works on a homogeneous projective space \mathbb{P}^2 . In this setting, the affine points are mapped as $(x, y) \mapsto (x : y : 1) \in \mathbb{P}^2$ and $\mathcal{O} \mapsto (0 : 1 : 0)$. The inverse map is $(X : Y : Z) \mapsto (X/Z, Y/Z)$ and $(0 : 1 : 0) \mapsto \mathcal{O}$. Given two projective points $P, Q \in E(\mathbb{F}_q)$ with coordinates, respectively, $(X_P : Y_P : Z_P)$ and $(X_Q : Y_Q : Z_Q)$; the coordinates of $P + Q = (X_{P+Q} : Y_{P+Q} : Z_{P+Q})$ are calculated as

$$\begin{aligned}
 r_0 &= X_P Y_Q + X_Q Y_P, & r_1 &= Y_P Y_Q, \\
 r_2 &= X_P Z_Q + X_Q Z_P, & r_3 &= Z_P Z_Q, \\
 r_4 &= 3B r_3, & r_5 &= Y_P Z_Q + Y_Q Z_P, \\
 r_6 &= 3X_P X_Q + A r_3, & r_7 &= A X_P X_Q + 3B r_2 - A^2 r_3, \\
 X_{P+Q} &= r_0(r_1 - A r_2 - r_4) - r_5 r_7, & Y_{P+Q} &= (r_1 + A r_2 + r_4)(r_1 - A r_2 - r_4) + r_6 r_7, \\
 Z_{P+Q} &= r_5(r_1 + A r_2 + r_4) + r_0 r_6.
 \end{aligned} \tag{4.3.1}$$

Renes et al. [230] showed optimized algorithms for evaluating Equation (4.3.1) taking $12\mathbf{M} + 4\mathbf{M}_A + 2\mathbf{M}_B + 23\mathbf{A}$ field operations. Table 4.3.2 compares several formulas for calculating addition of points on a Weierstrass curve.

As seen in the table, the \mathbb{F}_q -complete formula takes the same number of multiplications as the Jacobian formula. However, it requires a larger number of field additions, which could become a significant overhead if their cost is expensive. Renes et al. [230] measured the performance of the Elliptic Curve Diffie-Hellman (ECDH) protocol by replacing the Jacobian formulas by the complete formulas in the OpenSSL library. In their experiment, they observed $1.4\times$ slow-down factor when the complete formulas are used.

Table 4.3.2: Operation counts of point addition for Weierstrass curves.

\mathbb{F}_q -complete	Projective Coordinates	Point Addition					Point Doubling				
		M	S	M_A	M_B	A	M	S	M_A	M_B	A
No	Homogeneous	12	2	0	0	7	5	6	1	0	12
No	Jacobian	12	4	0	0	7	3	6	1	0	13
Yes ¹	Homogeneous	12	0	4	2	23	8	3	4	2	15

¹ Provided that the curve has no points of order two.

In spite of the loss of performance, complete formulas have a valuable property since they offer natural protection against side-channel attacks. This fact motivated us to investigate on strategies for reducing the execution time of complete point addition. In the next section, we describe our proposal that relies on parallel computing.

4.3.2 Parallel Complete Addition Formulas

The field operations of the complete formulas have certain degree of parallelism that can be exploited using SIMD parallel computing. In the formula given in Equation (4.3.1), several field operations have no dependencies between them allowing their parallel execution. Motivated by the SIMD parallel scheduling of point additions given by Aoki [11], Izu-Takagi [157], and Longa-Miri [184] we looked for a parallel scheduling of the complete formula that distributes field operations across two and four units.

We rearrange the internal operations of the point addition formula according to the following criteria. First, we distributed operations in such a way that, at each step, all units calculate the same arithmetic operation. For simplicity, we considered additions and subtractions equivalent. In a few cases, the symbol \emptyset appears for indicating that no computation is performed in that unit. We also observed that sharing data between units could be an expensive operation in some architectures. For example, in the AVX2 vector unit permutations are costly. Hence, our distribution of operations minimizes the amount of values are interchanged between units.

The set of algorithms described in this section list series of n -way prime field operations (described at Section 3.3) enabling their direct implementation on parallel units. Algorithms 4.3.3 and 4.3.5 describe a two-way scheduling of field operations for performing point addition and point doubling, respectively. Some operations can be saved if the curve parameter $A = -3 \in \mathbb{F}_q$. If this is the case, Algorithms 4.3.4 and 4.3.6 list a two-way scheduling of field operations for point addition and point doubling, respectively.

We go further by proposing a scheduling of operations when four parallel units are available. Algorithms 4.3.7 and 4.3.8 list the operations for point addition and point doubling, respectively. Our scheduling distributes operations in such a way that increases the utilization of the four units.

Algorithm 4.3.3 Two-way \mathbb{F}_q -Complete Point Addition on $E/\mathbb{F}_q: y^2 = x^3 + Ax + B$.

Input: $(X_P: Y_P: Z_P)$ and $(X_Q: Y_Q: Z_Q)$ are coordinates of $P, Q \in E(\mathbb{F}_q)$.

Output: $(X_R: Y_R: Z_R)$ are coordinates of $R = P + Q \in E(\mathbb{F}_q)$.

UNIT 1	UNIT 2
1: $l_0 \leftarrow X_P + Y_P$	$r_0 \leftarrow X_P + Z_P$
2: $l_1 \leftarrow X_Q + Y_Q$	$r_1 \leftarrow X_Q + Z_Q$
3: $l_2 \leftarrow l_0 \times l_1$	$r_2 \leftarrow r_0 \times r_1$
4: $l_3 \leftarrow Y_P \times Y_Q$	$r_3 \leftarrow Z_P \times Z_Q$
5: $l_4 \leftarrow Y_P + Z_P$	$r_4 \leftarrow l_3 + r_3$
6: $l_5 \leftarrow Y_Q + Z_Q$	\emptyset
7: $l_6 \leftarrow l_4 \times l_5$	$r_6 \leftarrow X_P \times X_Q$
8: $l_7 \leftarrow l_3 + r_6$	$r_7 \leftarrow r_3 + r_6$
9: $l_8 \leftarrow l_2 - l_7$	$r_8 \leftarrow r_2 - r_7$
10: $l_9 \leftarrow 3B \times r_3$	$r_9 \leftarrow 3B \times r_8$
11: $l_{10} \leftarrow A \times r_3$	$r_{10} \leftarrow A \times r_8$
12: $l_{11} \leftarrow A \times l_{10}$	$r_{11} \leftarrow A \times r_6$
13: $l_{12} \leftarrow l_9 + r_{10}$	$r_{12} \leftarrow r_9 + r_{11}$
14: $l_{13} \leftarrow l_3 + l_{12}$	\emptyset
15: $l_{14} \leftarrow l_3 - l_{12}$	$r_{14} \leftarrow l_6 - r_4$
16: $l_{15} \leftarrow 3r_6$	\emptyset
17: $l_{16} \leftarrow l_{15} + l_{10}$	$r_{16} \leftarrow r_{12} - l_{11}$
18: $l_{17} \leftarrow l_{14} \times l_8$	$r_{17} \leftarrow r_{14} \times r_{16}$
19: $l_{18} \leftarrow l_{16} \times r_{16}$	$r_{18} \leftarrow l_{16} \times l_8$
20: $l_{19} \leftarrow l_{14} \times l_{13}$	$r_{19} \leftarrow r_{14} \times l_{13}$
21: $X_R \leftarrow l_{17} - r_{17}$	\emptyset
22: $Y_R \leftarrow l_{19} + l_{18}$	$Z_R \leftarrow r_{19} + r_{18}$
23: return $(X_R: Y_R: Z_R)$	

Algorithm 4.3.4 Two-way \mathbb{F}_q -Complete Point Addition on $E/\mathbb{F}_q: y^2 = x^3 - 3x + B$.

Input: $(X_P: Y_P: Z_P)$ and $(X_Q: Y_Q: Z_Q)$ are coordinates of $P, Q \in E(\mathbb{F}_q)$.

Output: $(X_R: Y_R: Z_R)$ are coordinates of $R = P + Q \in E(\mathbb{F}_q)$.

UNIT 1	UNIT 2
1: $l_0 \leftarrow X_P + Y_P$	$r_0 \leftarrow X_Q + Y_Q$
2: $l_1 \leftarrow X_P + Z_P$	$r_1 \leftarrow Y_P + Z_P$
3: $l_2 \leftarrow X_Q + Z_Q$	$r_2 \leftarrow Y_Q + Z_Q$
4: $l_3 \leftarrow X_P \times X_Q$	$r_3 \leftarrow Y_P \times Y_Q$
5: $l_4 \leftarrow l_0 \times r_0$	$r_4 \leftarrow Z_P \times Z_Q$
6: $l_5 \leftarrow l_1 \times l_2$	$r_5 \leftarrow r_1 \times r_2$
7: $l_6 \leftarrow 3l_3$	$r_6 \leftarrow 3r_4$
8: $l_7 \leftarrow l_3 + r_4$	\emptyset
9: $l_8 \leftarrow l_5 - l_7$	\emptyset
10: $l_9 \leftarrow B \times l_8$	$r_9 \leftarrow B \times r_4$
11: $l_{10} \leftarrow l_9 - l_3$	$r_{10} \leftarrow l_8 - r_9$
12: $l_{11} \leftarrow l_{10} - r_6$	\emptyset
13: $l_{12} \leftarrow 3l_{11}$	$r_{12} \leftarrow 3r_{10}$
14: $l_{13} \leftarrow l_6 - r_6$	$r_{13} \leftarrow r_3 - r_{12}$
15: \emptyset	$r_{14} \leftarrow r_3 + r_{12}$
16: $l_{15} \leftarrow l_3 + r_3$	$r_{15} \leftarrow r_3 + r_4$
17: $l_{16} \leftarrow l_4 - l_{15}$	$r_{16} \leftarrow r_5 - r_{15}$
18: $l_{17} \leftarrow l_{16} \times r_{14}$	$r_{17} \leftarrow r_{16} \times l_{12}$
19: $l_{18} \leftarrow r_{14} \times r_{13}$	$r_{18} \leftarrow l_{12} \times l_{13}$
20: $l_{19} \leftarrow l_{16} \times l_{13}$	$r_{19} \leftarrow r_{16} \times r_{13}$
21: $X_R \leftarrow l_{17} - r_{17}$	$Y_R \leftarrow l_{18} + r_{18}$
22: $Z_R \leftarrow l_{19} + r_{19}$	\emptyset
23: return $(X_R: Y_R: Z_R)$	

Algorithm 4.3.5 Two-way \mathbb{F}_q -Complete Point Doubling on $E/\mathbb{F}_q: y^2 = x^3 + Ax + B$.

Input: $(X_P: Y_P: Z_P)$ are coordinates of $P \in E(\mathbb{F}_q)$.

Output: $(X_{2P}: Y_{2P}: Z_{2P})$ are coordinates of $2P \in E(\mathbb{F}_q)$.

UNIT 1	UNIT 2
1: $l_0 \leftarrow 2Z_P$	$r_0 \leftarrow 2X_P$
2: $l_1 \leftarrow Z_P^2$	$r_1 \leftarrow X_P^2$
3: $l_2 \leftarrow Y_P \times l_0$	$r_2 \leftarrow A \times l_1$
4: $l_3 \leftarrow l_2 + l_2$	$r_3 \leftarrow r_1 + r_2$
5: $l_4 \leftarrow l_3 + l_3$	$r_4 \leftarrow r_1 - r_2$
6: $l_5 \leftarrow Y_P^2$	$r_5 \leftarrow Y_P \times r_0$
7: $Z_{2P} \leftarrow l_4 \times l_5$	$r_6 \leftarrow X_P \times l_0$
8: $l_7 \leftarrow A \times r_6$	$r_7 \leftarrow A \times r_4$
9: $l_8 \leftarrow 3B \times l_1$	$r_8 \leftarrow 3B \times r_6$
10: $l_9 \leftarrow l_7 + l_8$	$r_9 \leftarrow r_7 + r_8$
11: $l_{10} \leftarrow l_5 + l_9$	$r_{10} \leftarrow r_1 + r_1$
12: $l_{11} \leftarrow l_5 - l_9$	$r_{11} \leftarrow r_3 + r_{10}$
13: $l_{12} \leftarrow r_5 \times l_{11}$	$r_{12} \leftarrow r_9 \times r_{11}$
14: $l_{13} \leftarrow l_{10} \times l_{11}$	$r_{13} \leftarrow r_9 \times l_2$
15: $X_{2P} \leftarrow l_{12} - r_{13}$	$Y_{2P} \leftarrow r_{12} + l_{13}$
16: return $(X_{2P}: Y_{2P}: Z_{2P})$	

Algorithm 4.3.6 Two-way \mathbb{F}_q -Complete Point Doubling on $E/\mathbb{F}_q: y^2 = x^3 - 3x + B$.

Input: $(X_P: Y_P: Z_P)$ are coordinates of $P \in E(\mathbb{F}_q)$.

Output: $(X_{2P}: Y_{2P}: Z_{2P})$ are coordinates of $2P \in E(\mathbb{F}_q)$.

UNIT 1	UNIT 2
1: \emptyset	$r_0 \leftarrow Y_P^2$
2: $l_1 \leftarrow 2X_P$	$r_1 \leftarrow 2Y_P$
3: $l_2 \leftarrow l_1 \times Y_P$	$r_2 \leftarrow r_1 \times Z_P$
4: $l_3 \leftarrow l_1 \times Z_P$	$r_3 \leftarrow r_2 \times r_0$
5: $l_4 \leftarrow X_P^2$	$r_4 \leftarrow Z_P^2$
6: $l_5 \leftarrow 3l_3$	$r_5 \leftarrow 3r_4$
7: $l_6 \leftarrow B \times l_5$	$r_6 \leftarrow B \times r_5$
8: $l_7 \leftarrow 3l_4$	$r_7 \leftarrow 3r_5$
9: $l_8 \leftarrow r_5 - l_7$	$r_8 \leftarrow r_6 - l_5$
10: $l_9 \leftarrow l_7 - l_6$	$r_9 \leftarrow r_0 - r_8$
11: $l_{10} \leftarrow l_9 + r_7$	$r_{10} \leftarrow r_0 + r_8$
12: $l_{11} \leftarrow r_2 \times l_{10}$	$r_{11} \leftarrow l_8 \times l_{10}$
13: $l_{12} \leftarrow r_9 \times l_2$	$r_{12} \leftarrow r_9 \times r_{10}$
14: $X_{2P} \leftarrow l_{11} + l_{12}$	$Y_{2P} \leftarrow r_{11} + r_{12}$
15: $Z_{2P} \leftarrow 4r_3$	\emptyset
16: return $(X_{2P}: Y_{2P}: Z_{2P})$	

Algorithm 4.3.7 Four-way \mathbb{F}_q -Complete Point Addition on $E/\mathbb{F}_q: y^2 = x^3 + Ax + B$.

Input: $(X_P: Y_P: Z_P)$ and $(X_Q: Y_Q: Z_Q)$ are coordinates of $P, Q \in E(\mathbb{F}_q)$.

Output: $(X_{P+Q}: Y_{P+Q}: Z_{P+Q})$ are coordinates of $P + Q \in E(\mathbb{F}_q)$.

UNIT 1	UNIT 2	UNIT 3	UNIT 4
1: $e_0 \leftarrow Y_P + X_Q$	$f_0 \leftarrow Y_P + Z_Q$	$g_0 \leftarrow X_P + Z_P$	$h_0 \leftarrow X_Q + Z_Q$
2: $e_1 \leftarrow X_P \times X_Q$	$f_1 \leftarrow Y_P \times Y_Q$	$g_1 \leftarrow Z_P \times Z_Q$	$h_1 \leftarrow g_0 \times h_0$
3: $e_2 \leftarrow Y_Q + X_Q$	$f_2 \leftarrow Y_Q + Z_Q$	$g_2 \leftarrow e_1 + e_1$	$h_2 \leftarrow e_1 + g_1$
4: $e_3 \leftarrow f_1 + g_1$	$f_3 \leftarrow e_1 + f_1$	$g_3 \leftarrow g_2 + e_1$	$h_3 \leftarrow h_1 - h_2$
5: $e_4 \leftarrow e_0 \times e_2$	$f_4 \leftarrow f_0 \times f_2$	$g_4 \leftarrow A \times g_1$	$h_4 \leftarrow 3B \times h_3$
6: $e_5 \leftarrow 3B \times g_1$	$f_5 \leftarrow A \times e_1$	$g_5 \leftarrow A \times g_4$	$h_5 \leftarrow A \times h_3$
7: $e_6 \leftarrow e_4 - f_3$	$f_6 \leftarrow f_4 - e_3$	$g_6 \leftarrow e_5 + h_5$	$h_6 \leftarrow f_5 + h_4$
8: $e_7 \leftarrow f_1 - g_6$	$f_7 \leftarrow f_1 + g_6$	$g_7 \leftarrow g_3 + g_4$	$h_7 \leftarrow h_6 - g_5$
9: $e_8 \leftarrow e_6 \times e_7$	$f_8 \leftarrow e_7 \times f_7$	$g_8 \leftarrow e_6 \times g_7$	\emptyset
10: $e_9 \leftarrow f_6 \times h_7$	$f_9 \leftarrow g_7 \times h_7$	$g_9 \leftarrow f_6 \times f_7$	\emptyset
11: $X_{P+Q} \leftarrow e_8 - e_9$	$Y_{P+Q} \leftarrow f_8 + f_9$	$Z_{P+Q} \leftarrow g_8 + g_9$	\emptyset
12: return $(X_{P+Q}, Y_{P+Q}, Z_{P+Q})$			

Algorithm 4.3.8 Four-way \mathbb{F}_q -Complete Point Doubling on $E/\mathbb{F}_q: y^2 = x^3 + Ax + B$.

Input: $(X_P: Y_P: Z_P)$ are coordinates of $P \in E(\mathbb{F}_q)$.

Output: $(X_{2P}: Y_{2P}: Z_{2P})$ are coordinates of $2P \in E(\mathbb{F}_q)$.

UNIT 1	UNIT 2	UNIT 3	UNIT 4
1: $e_0 \leftarrow X_P^2$	$f_0 \leftarrow Y_P^2$	$g_0 \leftarrow Z_P^2$	$h_0 \leftarrow Y_P \times Z_P$
2: $e_1 \leftarrow X_P \times Z_P$	$f_1 \leftarrow X_P \times Y_P$	$g_1 \leftarrow A \times g_0$	$h_1 \leftarrow 3B \times g_0$
3: $e_2 \leftarrow 2e_1$	$f_2 \leftarrow 2f_1$	$g_2 \leftarrow e_0 - g_1$	$h_2 \leftarrow 2h_0$
4: $e_3 \leftarrow 3B \times e_2$	$f_3 \leftarrow A \times e_2$	$g_3 \leftarrow A \times g_2$	$h_3 \leftarrow f_0 \times h_2$
5: $e_4 \leftarrow 2e_0$	$f_4 \leftarrow e_0 + g_1$	$g_4 \leftarrow f_3 + h_1$	$h_4 \leftarrow 2h_3$
6: $e_5 \leftarrow e_3 + g_3$	$f_5 \leftarrow e_4 + f_4$	$g_5 \leftarrow f_0 - g_4$	$h_5 \leftarrow f_0 + g_4$
7: $e_6 \leftarrow f_2 \times g_5$	$f_6 \leftarrow h_2 \times e_5$	$g_6 \leftarrow e_5 \times f_5$	$h_6 \leftarrow g_5 \times h_5$
8: $X_{2P} \leftarrow e_6 - g_6$	$Y_{2P} \leftarrow f_6 + h_6$	$Z_{2P} \leftarrow 2h_4$	\emptyset
9: return $(X_{2P}: Y_{2P}: Z_{2P})$			

Table 4.3.9 shows the operation counts of our parallel algorithms. As it can be seen, the number of operations reduces as the number of units increases. For the two-way formulas, the number of multiplications gets halved. Hence, parallel implementations could reduce the cost of a point operation by half whenever the cost of two-way multiplications is faster than the cost of two sequential multiplications, i.e., $1 < 2M/M_2 \leq 2$. On the other hand, the four-way scheduling for point operations absorbed multiplication by constants as general multiplications. More optimizations can be performed once the elliptic curve constants are fixed.

Table 4.3.9: Operation counts of parallel \mathbb{F}_q -complete formulas.

Operation	Parallel Units	Field Operations ¹	Formula
Point Addition	Single	$12M + 4M_A + 2M_{3B} + 23A$	[230, Alg. 1]
	2-way	$6M_2 + 2(M_A)_2 + 1(M_{3B})_2 + 14A_2$	Algorithm 4.3.3
	4-way	$5M_4 + 6A_4$	Algorithm 4.3.7
Point Doubling	Single	$8M + 3S + 4M_A + 2M_{3B} + 15A$	[230, Alg. 3]
	2-way	$5M_2 + 1S_2 + 1(M_A)_2 + 1(M_{3B})_2 + 7A_2$	Algorithm 4.3.5
	4-way	$4M_4 + 4A_4$	Algorithm 4.3.8

¹ M_A and M_{3B} denote the cost of multiplying by the curve parameters A and $3B$, respectively.

4.4 Arithmetic of Montgomery Curves

Montgomery [196] proposed a special parametrization of elliptic curves for accelerating the Elliptic Curve Method (ECM) for factorizing integers. For the purposes of ECM, it is critical to calculate scalar multiplications as fast as possible, and this is one of the cornerstone advantages of Montgomery's parametrization. Due to their efficiency, Montgomery curves are also used in a large number of applications beyond the scope of integer factorization.

In this section, we describe the remarkable properties of Montgomery curves that allow calculating elliptic curve operations in an efficient way. We revisit several algorithms for operations on the x -line variety such as the Montgomery ladder used for scalar multiplication, and its right-to-left variants. We also show a new algorithm for the calculation of $P + kQ$ using a three-point ladder technique, and we present an optimized formula for point tripling.

4.4.1 Montgomery Curves

Let \mathbb{F}_q be a finite field of odd characteristic, a Montgomery curve over \mathbb{F}_q is defined as

$$E/\mathbb{F}_q: \quad By^2 = x^3 + Ax^2 + x, \quad (4.4.1)$$

such that $A, B \in \mathbb{F}_q$ and $B(A^2 - 4) \neq 0$. By inspection, one can verify that $\mathcal{T} = (0, 0)$ is a point on any Montgomery curve. This curve is non-singular and its j -invariant is

$$j(E) = \frac{256(A^2 - 3)^3}{A^2 - 4}. \quad (4.4.2)$$

The affine formulas for calculating additions of points on a Montgomery curve are given as follows. Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be two points on a Montgomery curve such that $P, Q \neq \mathcal{O}$, $x_P x_Q \neq 0$, and $x_P \neq x_Q$; then $P + Q = (x_{P+Q}, y_{P+Q})$ is calculated as

$$\lambda = \frac{y_P - y_Q}{x_P - x_Q}, \quad x_{P+Q} = B\lambda^2 - A - x_P - x_Q, \quad y_{P+Q} = (2x_P + x_Q + A)\lambda - B\lambda^3 - y_P. \quad (4.4.3)$$

For those points $P = (x_P, y_P)$ such that $y_P \neq 0$, the point doubling $2P = (x_{2P}, y_{2P})$ is calculated as

$$\lambda = \frac{3x_P^2 + 2Ax_P + 1}{2By_P}, \quad x_{2P} = B\lambda^2 - A - 2x_P, \quad y_{2P} = (3x_P + A)\lambda - B\lambda^3 - y_P. \quad (4.4.4)$$

Like for Weierstrass curves, the affine points can be embedded into a projective space to avoid the calculation of inverses in the field. Thus, the affine points are mapped to \mathbb{P}^2 using $(x, y) \mapsto (x : y : 1) \in \mathbb{P}^2$ and $\mathcal{O} \mapsto (0 : 1 : 0)$. The inverse map is $(X : Y : Z) \mapsto (X/Z, Y/Z)$ if $Z \neq 0$, and $(0 : 1 : 0) \mapsto \mathcal{O}$. By applying this map to the Montgomery curve equation, it results in the homogeneous polynomial $f(X, Y, Z) = BY^2Z - X^3 - AX^2Z - XZ^2$, which is used to define $E(\mathbb{F}_q)$, the \mathbb{F}_q -rational points of a Montgomery curve. Therefore $(E(\mathbb{F}_q), \mathcal{O})$ is an elliptic curve.

A distinctive property of Montgomery curves and thus the group $E(\mathbb{F}_q)$ is that its order is always a multiple of four, i.e., $4 \mid \#E(\mathbb{F}_q)$. Although Montgomery curves cannot generate groups of order prime, it is not difficult to find Montgomery curves for which $\#E(\mathbb{F}_q)/4$ is a prime number.

Until now, the arithmetic of points is similar to the arithmetic of Weierstrass curves. However, a striking property observed by Montgomery on implementing the ECM for integer factorization helped to devise faster algorithms for point operations. In ECM,

operations over points of an elliptic curve are used to detect factors of an integer. More specifically, it is enough to test certain property on the x -coordinate of points. Montgomery found a set of curves for which the calculation of point operations can be performed using only the x -coordinate of points. Thus, the y -coordinate is not used. In practical terms, performing point operations using only the x -coordinate of points reduces both the number of field operations for adding points and the storage requirements.

A first observation of Montgomery is the relation between the coordinates of $P + Q$ and $P - Q$. Let x_P and x_Q be the x -coordinate of P and Q ; then the following relation holds whenever $x_P \neq x_Q$:

$$x_{P+Q} \times x_{P-Q} = \left(\frac{x_P x_Q - 1}{x_P - x_Q} \right)^2. \quad (4.4.5)$$

From this relation, Montgomery derived a new point operation called differential addition. Hence, given the x -coordinate of the points P , Q , and $P - Q$ the differential addition of P and Q , which is denoted as $P +_{(P-Q)} Q$, calculates the x -coordinate of $P + Q$ as

$$x_{P+Q} = \frac{1}{x_{P-Q}} \times \left(\frac{x_P x_Q - 1}{x_P - x_Q} \right)^2. \quad (4.4.6)$$

This equation is defined whenever $x_{P-Q} \neq 0$ and $x_P \neq x_Q$. In other words, the differential addition formula has two exceptional cases: when the points to be added differ by \mathcal{T} and when $P = \pm Q$.

Following the same reasoning, Montgomery showed formulas for point doubling that only depend on the x -coordinate of the input point. Let x_P be the x -coordinate of P , the x -coordinate of $2P$ is calculated as

$$x_{2P} = \frac{(x_P^2 - 1)^2}{4(x_P^3 + Ax_P^2 + x_P)}. \quad (4.4.7)$$

This formula is not defined on the points $(\alpha, 0)$, where α is a root of $x^3 + Ax^2 + x = 0$. These exceptional points are \mathcal{T} and $((-A \pm \sqrt{A^2 - 4})/2, 0)$. These points are in $E[2]$, the two-torsion subgroup of the curve, and consequently they have order two. If $A^2 - 4$ is a quadratic residue in \mathbb{F}_q , the points will be \mathbb{F}_q -rational points, and the curve is said to have full rational two-torsion and it holds that $E[2](\mathbb{F}_q) \cong \mathbb{Z}_2 \times \mathbb{Z}_2$. On the other hand, if $A^2 - 4$ is a quadratic non-residue, \mathcal{T} is the only \mathbb{F}_q -rational point of order two.

Formulas using only the x -coordinate of points also work with projective coordinates avoiding the calculation of inverses. Before going into the details of their calculation, we describe the Montgomery's approach using a well-known algebraic variety.

4.4.2 The x -Line Variety

The Kummer variety associated to a curve C is defined as C / \sim , where \sim is an equivalence relation on the points on the curve. Thus, for all $P, Q \in C$, we have $P \sim Q$ iff $Q = -P$. That is, P and $-P$ represent the same point in the Kummer variety. The Kummer variety associated to an elliptic curve is commonly known as *the x -line of the curve*.

The \mathbb{F}_q -rational points of a Montgomery curve E are embedded into its Kummer variety, which is isomorphic to \mathbb{P}^1 , using the following map

$$\begin{aligned} \mathbf{x}: E(\mathbb{F}_q) &\rightarrow E(\mathbb{F}_q)/(-1) \cong \mathbb{P}^1 \\ (X, Y, Z) &\mapsto (X : Z), \text{ if } Z \neq 0. \\ (0, 1, 0) &\mapsto (1 : 0) \end{aligned} \tag{4.4.8}$$

This projection must not be confused with trivially dropping the y -coordinate of points because doing so for $(0 : 1 : 0)$ leads to $(0 : 0)$, which does not belong to \mathbb{P}^1 . The main reason for defining $(0 : 1 : 0) \mapsto (0 : 1)$ is to make the doubling formula complete (as proven by Bernstein and Lange [37]).

Although the x -line variety does not form an abelian group, it inherits some properties of the elliptic curve group. The most relevant of them is the hardness of solving ECDLP. Therefore, working on the x -line of the curve improves the efficiency of point arithmetic without reducing the complexity of the hard problem.

Point Operations on the x -Line

Given $\mathbf{x}(P) = (X_P : Z_P) \in \mathbb{P}^1$ such that $P \in E(\mathbb{F}_q)$, the point $\mathbf{x}(2P) = (X_{2P} : Z_{2P})$ is

$$\begin{aligned} t_0 &\leftarrow X_P + Z_P, & t_1 &\leftarrow t_0^2, \\ t_2 &\leftarrow X_P - Z_P, & t_3 &\leftarrow t_2^2, \\ t_4 &\leftarrow t_1 - t_3, & X_{2P} &\leftarrow t_1 \times t_3, \\ Z_{2P} &\leftarrow t_4 \times \left[t_3 + \left(\frac{A+2}{4} \right) \times t_4 \right]. \end{aligned} \tag{4.4.9}$$

Like in the affine formula, the exceptional cases are the points of order two. However, note that evaluating this projective formula on points of order two gives as a result $(1 : 0) \in \mathbb{P}^1$. For this reason, it makes sense to fix $(0 : 1 : 0) \mapsto (1 : 0)$ in the \mathbf{x} mapping. Therefore, the projective doubling formula works for any projective point and no exceptional cases can occur. This formula calculates a point doubling taking $2\mathbf{M} + 2\mathbf{S} + 1\mathbf{C} + 4\mathbf{A}$ field operations, where \mathbf{C} is the cost of multiplying by $(A+2)/4$, which can be faster than a generic multiplication if A is chosen as a small number.

In the x -line, the point addition is not well defined, since adding $\mathbf{x}(P)$ and $\mathbf{x}(Q)$ could result on either $\mathbf{x}(P+Q)$ or $\mathbf{x}(P-Q)$. However, Montgomery showed that these points are related by Equation (4.4.5). Thus, the differential addition operation takes $\mathbf{x}(P) = (X_P : Z_P)$, $\mathbf{x}(Q) = (X_Q : Z_Q)$, and $\mathbf{x}(P-Q) = (X_{P-Q} : Z_{P-Q})$ and calculates $\mathbf{x}(P+Q) = (X_{P+Q} : Z_{P+Q})$ as

$$\begin{aligned} t_0 &\leftarrow X_P + Z_P, & t_1 &\leftarrow X_P - Z_P, \\ t_2 &\leftarrow X_Q + Z_Q, & t_3 &\leftarrow X_Q - Z_Q, \\ t_4 &\leftarrow t_0 \times t_3, & t_5 &\leftarrow t_1 \times t_2, \\ X_{P+Q} &\leftarrow Z_{P-Q} \times [t_4 + t_5]^2, & Z_{P+Q} &\leftarrow X_{P-Q} \times [t_4 - t_5]^2. \end{aligned} \tag{4.4.10}$$

The exceptional cases of this formula are inherited from the affine formula, i.e., this formula is defined when $\mathbf{x}(P) \neq \mathbf{x}(Q)$ (or equivalently when $X_P/Z_P \neq X_Q/Z_Q$) and $X_{P-Q} \neq 0$. Using this formula, differential additions take $4\mathbf{M} + 2\mathbf{S} + 6\mathbf{A}$ field operations.

Scalar Multiplication on the x -Line

The left-to-right ladder method for scalar multiplication (described in Section 4.2.3) was also introduced by Montgomery [196]. Although Montgomery's ladder method applies generally to any group, when it is instantiated in the x -line variety, the scalar multiplication is calculated with a fewer number of operations.

The ladder step is the core operation of Montgomery ladder since it is performed for every bit of the scalar. In the general case, this step updates two accumulator points $R_0, R_1 \in E(\mathbb{F}_q)$ as $(R_0, R_1) \leftarrow (2R_0, R_0 + R_1)$. However, for the case of the x -line variety, $R_0, R_1 \in \mathbb{P}^1$ such that $R_0 = \mathbf{x}(P)$ and $R_1 = \mathbf{x}(Q)$ for some $P, Q \in E(\mathbb{F}_q)$; thus the ladder step is $(R_0, R_1) \leftarrow (2R_0, R_0 +_{(R_2)} R_1)$ for some point $R_2 \in \mathbb{P}^1$ such that $R_2 = \mathbf{x}(P - Q)$. Algorithm 4.4.11³ shows Montgomery ladder multiplication method applied to the x -line of a Montgomery curve. This method takes $\mathbf{x}(P)$ such that $P \in E(\mathbb{F}_q) \setminus \{\mathcal{O}, \mathcal{T}\}$ to calculate $\mathbf{x}(kP)$ for a positive integer k . Note that $P \notin \{\mathcal{O}, \mathcal{T}\}$ due to the exceptional cases of the differential addition formula.

Algorithm 4.4.11 Montgomery Ladder Algorithm for Scalar Multiplication on the x -Line.

Input: (n, k) are integers such that $n \geq 1$ and $0 \leq k < 2^n$; and $\mathbf{x}(P) \in \mathbb{P}^1$, where $P \in E(\mathbb{F}_q) \setminus \{\mathcal{O}, \mathcal{T}\}$.

Output: $\mathbf{x}(kP) \in \mathbb{P}^1$.

- 1: Let $(k_{n-1}, \dots, k_0)_2$ be the n -bit representation of k and define $k_n = 0$.
 - 2: $R_0 \leftarrow \mathbf{x}(\mathcal{O})$, $R_1 \leftarrow \mathbf{x}(P)$, $R_2 \leftarrow \mathbf{x}(P)$
 - 3: **for** $i \leftarrow n - 1$ **to** 0 **do**
 - 4: $\beta \leftarrow k_i \oplus k_{i+1}$
 - 5: $R_0, R_1 \leftarrow \mathbf{CSWAP}(R_0, R_1, \beta)$
 - 6: $R_0, R_1 \leftarrow 2R_0, R_0 +_{(R_2)} R_1$ //Ladder Step
 - 7: **end for**
 - 8: $R_0, R_1 \leftarrow \mathbf{CSWAP}(R_0, R_1, k_0)$
 - 9: **return** R_0
-

The ladder step can be calculated more efficiently by observing that the formulas for differential addition and point doubling share some intermediate terms. Hence, the ladder step formula takes $\mathbf{x}(P) = (X_P : Z_P)$, $\mathbf{x}(Q) = (X_Q : Z_Q)$, and $\mathbf{x}(P - Q) = (X_{P-Q} : Z_{P-Q})$

³Despite Algorithm 4.4.11 shows the CSWAP variant, the CMOV-based method applies as well.

to calculate the points $\mathbf{x}(2P) = (X_{2P} : Z_{2P})$ and $\mathbf{x}(P + Q) = (X_{P+Q} : Z_{P+Q})$ as follows

$$\begin{aligned}
t_0 &\leftarrow X_P + Z_P, & t_1 &\leftarrow X_P - Z_P, \\
t_2 &\leftarrow X_Q + Z_Q, & t_3 &\leftarrow X_Q - Z_Q, \\
t_4 &\leftarrow t_0 \times t_3, & t_5 &\leftarrow t_1 \times t_2, \\
t_6 &\leftarrow t_0^2, & t_7 &\leftarrow t_1^2, \\
t_8 &\leftarrow t_6 - t_7, & & \\
X_{2P} &\leftarrow t_6 \times t_7, & Z_{2P} &\leftarrow t_8 \times \left[t_7 + \left(\frac{A+2}{4} \right) \times t_8 \right], \\
X_{P+Q} &\leftarrow Z_{P-Q} \times (t_4 + t_5)^2, & Z_{P+Q} &\leftarrow X_{P-Q} \times (t_4 - t_5)^2.
\end{aligned} \tag{4.4.12}$$

whenever $\mathbf{x}(P) \neq \mathbf{x}(Q)$ and $X_{P-Q} \neq 0$. This formula takes $6\mathbf{M} + 4\mathbf{S} + 1\mathbf{C} + 8\mathbf{A}$ field operations, where \mathbf{C} is the cost of a multiplication by $(A+2)/4$.

Montgomery ladder algorithm calculates scalar multiplications on the x -line following a regular execution pattern, since it does not depend on the value of the scalar. This fact is a relevant feature since scalars often represents secret values. In conjunction to its efficiency, Montgomery ladder is a suitable method for its use in cryptographic algorithms.

Scalar Multiplication on Montgomery Curves

Multiplying a point $P \in E(\mathbb{F}_q) \subset \mathbb{P}^2$ by a scalar k uses the scalar multiplication on the x -line as a subroutine. Note that Algorithm 4.4.11 calculates $\mathbf{x}(kP)$ from $\mathbf{x}(P)$ and k . However, $\pm kP \in E(\mathbb{F}_q)$ are two pre-images of $\mathbf{x}(kP)$; for this reason, it is required a method that recovers the right pre-image using the information already processed.

López and Dahab [186] gave a solution to this problem in the context of binary curves. The central idea relies on an equation that recovers the y -coordinate of kP from the knowledge of $P = (x_P, y_P)$, $\mathbf{x}(kP)$ and $\mathbf{x}(kP + P)$. They observed that the latter two points are processed by the Montgomery ladder, and correspond to the final values stored in the accumulator points R_0 and R_1 , respectively. Thus, if Algorithm 4.4.11 is modified to return both accumulator points, one can use this information to univocally determine kP . Lopez-Dahab's method was further adapted to Montgomery curves by Okeya and Sakurai [210] and to Weierstrass curves by Brier and Joye [56].

Recovering the y -coordinate of a point is performed as follows. Let P, Q be two affine points on a Montgomery curve, the y -coordinate of P can be recovered from the x -coordinate of P and $P + Q$ for any point $Q = (x_Q, y_Q) \notin E[2] \cup \{P, -P\}$ as

$$y_P = \frac{(x_P x_Q + 1)(x_P + x_Q + 2A) - 2A - (x_P - x_Q)^2 x_{P+Q}}{2B y_Q}. \tag{4.4.13}$$

This formula is not defined when the y -coordinate of Q is 0, which happens when Q is a two-torsion point. The formula also excludes $Q = \pm P$ as they represent trivial cases of the recovering task. This equation can be easily extended to work with projective points on $E(\mathbb{F}_q)$ by constructing a point $(X : Y : Z) \sim P$. Algorithm 4.4.14 shows the explicit projective formulas given by Okeya and Sakurai [210].

Algorithm 4.4.14 Recovering the y -Coordinate of a Point on a Montgomery Curve.

Input: $\mathbf{x}(P) = (X_P : Z_P)$ and $\mathbf{x}(P + Q) = (X_{P+Q} : Z_{P+Q})$ are two points on the x -line;

$Q = (x_Q, y_Q)$ is an affine point such that $Q \notin E[2] \cup \{P, -P\}$ for some $P, Q \in E(\mathbb{F}_q)$.

Output: $(X : Y : Z) \in \mathbb{P}^2$ such that $(X : Y : Z) \sim P$.

- 1: $X \leftarrow 2By_Q Z_P Z_{P+Q} X_P$
 - 2: $Y \leftarrow Z_{P+Q} [(X_P + x_Q Z_P + 2AZ_P)(X_P x_Q + Z_P) - 2AZ_P^2] - (X_P - x_Q Z_P)^2 X_{P+Q}$
 - 3: $Z \leftarrow 2By_Q Z_P^2 Z_{P+Q}$
 - 4: **return** $(X : Y : Z)$
-

To obtain the y -coordinate of kP , one must instantiate the previous recovering method with $R_0 = \mathbf{x}(kP)$, $R_1 = \mathbf{x}(kP + P)$, and P where R_0 and R_1 are the accumulator points at the end of the Montgomery ladder algorithm for scalar multiplication on the x -line.

Putting all the pieces together, Algorithm 4.4.15 shows how to calculate scalar multiplications on the \mathbb{F}_q -rational points of a Montgomery curve. If P is given in affine coordinates, one multiplication can be saved in the ladder step since $\mathbf{x}(P) = (x_P : 1)$. Therefore, let n be the size in bits of $\#E(\mathbb{F}_q)$, the cost of a scalar multiplication kP on a Montgomery curve is $n(5\mathbf{M} + 4\mathbf{S} + 8\mathbf{A} + 1\mathbf{C})$ (due to scalar multiplication on the x -line) plus $10\mathbf{M} + 1\mathbf{S} + 6\mathbf{A} + 2\mathbf{C}$ (due to y -coordinate recovery) plus $1\mathbf{I} + 2\mathbf{M}$ (due to conversion of kP to affine coordinates).

Algorithm 4.4.15 Scalar Multiplication of Points on a Montgomery Curve.

Input: (n, k) are integers such that $n \geq 1$ and $0 \leq k < 2^n$; and $P \in E(\mathbb{F}_q)$.

Output: $kP \in E(\mathbb{F}_q)$.

- 1: **if** $P = \mathcal{O}$ **then**
 - 2: $Q \leftarrow \mathcal{O}$
 - 3: **else if** $P \in E[2](\mathbb{F}_q)$ **then**
 - 4: $Q \leftarrow (k \bmod 2)P$
 - 5: **else**
 - 6: $\mathbf{x}(kP), \mathbf{x}(kP + P) \leftarrow \text{ScalarMult}(n, k, \mathbf{x}(P))$ //Algorithm 4.4.11
 - 7: $Q \leftarrow y\text{-Recover}(\mathbf{x}(kP), \mathbf{x}(kP + P), P)$ //Algorithm 4.4.14
 - 8: **end if**
 - 9: **return** Q
-

4.4.3 Parallel Montgomery Ladder Step

In this section, we propose implementation techniques for accelerating the execution time of Montgomery ladder. In particular, we look for opportunities to run in parallel some operations of the Montgomery ladder step formula, given in Eq (4.4.12). Thus, we enable the use of parallel prime field operations (described in Section 3.3) and their further implementation using vector units.

Some insights on the parallel execution of ladder step were pointed out. Bernstein's diagram of the ladder step, which appears in [23, App. B], gives a hint of several symmetries of the field operations required for its calculation. Hisil et al. [148] proposed several scheduling of operations when two and four parallel units are available; however, it does

not present actual implementations of them. Building on top of these works, our aim is closing this gap by proposing parallel algorithms for the ladder step formula tailored for their implementation in vector units.

Parallel Scheduling in Two Units

We first tried to execute the operations of the ladder step in two sets of independent operations. For example, one possible way is to perform point doubling in one unit and the differential addition in other unit. However, the execution time of these operations is not even, which is not desirable in parallel environment. Because either it could derive on a sub-par use of resources or it could require to synchronize the units, which is, in general, an expensive task.

We arrived to a better strategy by grouping operations of the same complexity without dependencies between them. In Algorithm 4.4.16, we show a parallel scheduling of operations tailored for two parallel units.

Algorithm 4.4.16 Two-way Parallel Algorithm for Montgomery Ladder Step.

Input: $\mathbf{x}(P) = (X_P : Z_P)$, $\mathbf{x}(Q) = (X_Q : Z_Q)$, and $\mathbf{x}(P - Q) = (X_{P-Q} : Z_{P-Q})$ where $P, Q \in E(\mathbb{F}_q)$ such that $P - Q \notin \{\mathcal{O}, \mathcal{T}\}$.

Output: $\mathbf{x}(2P) = (X_{2P} : Z_{2P})$, and $\mathbf{x}(P + Q) = (X_{P+Q} : Z_{P+Q})$.

UNIT 1		UNIT 2
1: $l_1 \leftarrow X_P + Z_P$		$r_1 \leftarrow X_Q + Z_Q$
2: $l_2 \leftarrow X_P - Z_P$		$r_2 \leftarrow X_Q - Z_Q$
3: $l_3 \leftarrow l_1 \times r_2$		$r_3 \leftarrow l_2 \times r_1$
4: $l_4 \leftarrow l_3 + r_3$		$r_4 \leftarrow l_3 - r_3$
5: $l_5 \leftarrow l_4^2$		$r_5 \leftarrow r_4^2$
6: $X_{2P} \leftarrow Z_{P-Q} \times l_5$		$Z_{2P} \leftarrow X_{P-Q} \times r_5$
7: $l_6 \leftarrow l_1^2$		$r_6 \leftarrow l_2^2$
8: $l_7 \leftarrow l_6 \times \frac{A+2}{4}$		$r_7 \leftarrow r_6 \times \frac{A-2}{4}$
9: $l_8 \leftarrow l_6 - r_6$		$r_8 \leftarrow l_7 - r_7$
10: $X_{P+Q} \leftarrow l_6 \times r_6$		$Z_{P+Q} \leftarrow l_8 \times r_8$
11: return $(X_{2P} : Z_{2P})$ and $(X_{P+Q} : Z_{P+Q})$		

Algorithm 4.4.16 requires $3\mathbf{M}_2 + 2\mathbf{S}_2 + 1\mathbf{C}_2 + 4\mathbf{A}_2$ two-way field operations⁴, where \mathbf{C}_2 is the cost of a two-way multiplication by constant. In comparison with the original formulas, the two-way approach requires half of the number of arithmetic operations. Therefore, the cost of the ladder step is reduced as long as processing two-way operations is faster than processing two consecutive operations.

The two-way scheduling proposed makes a uniform use of both units balancing the complexity of operations performed at each step of the algorithm. By assigning operations to units, we also consider minimizing data transfer between units. The main reason behind this criteria is because moving data between AVX2 vector units is performed by expensive permutation instructions.

⁴The cost of a n -way field operation is denoted as $(\cdot)_n$.

Parallel Scheduling in Four Units

The operations of the ladder step can also be distributed among four parallel units. For example, Hisil et al. [148] proposed two approaches for this case. The first one has an effective cost of $2\mathbf{M}_4 + 2\mathbf{S}_4$ four-way operations; however, it is subpar since their second approach reduces the cost to $2\mathbf{M}_4 + 1\mathbf{S}_4$. This later approach distributes the differential addition in two units and the point doubling in the other two units. However, its downside is that some synchronization between units could be required at the end of the calculation. Because of that, we investigate how to improve the parallel scheduling considering the use of vector units.

We observed that the formulas for ladder step have a chain of dependencies that limits the number of operations to be executed in parallel. The critical path in the ladder step is a chain of multiplication \rightarrow square \rightarrow multiplication. Because of that, we designed a parallel scheduling that pairs field operations of the same complexity covering the critical path of the ladder step.

Our proposed four-way scheduling is shown in Algorithm 4.4.17. The symbol \emptyset stands for an idle operation in the unit. This algorithm takes $2\mathbf{M}_4 + 1\mathbf{S}_4 + 1\mathbf{C}_4 + 3\mathbf{A}_4$ four-way operations. An advantage of this algorithm is that multiplications and squares, which are the most expensive operations, are effective operations (except in the calculation of X_{2P}), then there is a low sub-utilization of computing resources.

Algorithm 4.4.17 Four-way Parallel Algorithm for Montgomery Ladder Step.

Input: $\mathbf{x}(P) = (X_P : Z_P)$, $\mathbf{x}(Q) = (X_Q : Z_Q)$, and $\mathbf{x}(P - Q) = (X_{P-Q} : Z_{P-Q})$ where $P, Q \in E(\mathbb{F}_q)$ such that $P - Q \notin \{\mathcal{O}, \mathcal{T}\}$.

Output: $\mathbf{x}(2P) = (X_{2P} : Z_{2P})$, and $\mathbf{x}(P + Q) = (X_{P+Q} : Z_{P+Q})$.

UNIT 1	UNIT 2	UNIT 3	UNIT 4
1: $e_1 \leftarrow X_P + Z_P$	$f_1 \leftarrow X_P + X_P$	$g_1 \leftarrow Z_Q + Z_Q$	$h_1 \leftarrow X_Q + Z_Q$
2: $e_2 \leftarrow X_P - Z_P$	\emptyset	\emptyset	$h_2 \leftarrow X_Q - Z_Q$
3: $e_3 \leftarrow e_1 \times e_2$	$f_3 \leftarrow f_1 \times g_1$	$g_3 \leftarrow e_1 \times h_2$	$h_3 \leftarrow h_1 \times e_2$
4: \emptyset	\emptyset	$g_4 \leftarrow g_3 + h_3$	$h_4 \leftarrow g_3 - h_3$
5: $e_5 \leftarrow e_3^2$	$f_5 \leftarrow e_1^2$	$g_5 \leftarrow g_4^2$	$h_5 \leftarrow h_4^2$
6: \emptyset	$f_6 \leftarrow f_5 + f_3 \times \frac{A+2}{4}$	\emptyset	\emptyset
7: $X_{2P} \leftarrow e_5 \times 1$	$Z_{2P} \leftarrow f_3 \times f_6$	$X_{P+Q} \leftarrow Z_{P-Q} \times g_5$	$Z_{P+Q} \leftarrow X_{P-Q} \times h_5$
8: return $(X_{2P} : Z_{2P})$ and $(X_{P+Q} : Z_{P+Q})$			

Table 4.4.18 summarizes the operation counts of the proposed parallel algorithms. It can be seen that the two-way version reduces by half the number of operations, except by the multiplication by a constant. However, the four-way scheduling shows the minimum number of multiplications and squares. Note that they cannot run in parallel because they are in the critical path of the Montgomery ladder step.

4.4.4 A Review of Right-To-Left Algorithms

A recent line of research suggests the use of right-to-left ladder algorithms for calculating fixed-point multiplications. Oliveira et al. [211] showed that the Joye's right-to-left algo-

Table 4.4.18: Operation counts of parallel algorithms for Montgomery ladder step.

Parallel Units	Field Operations ¹	Formula
Single	$6\mathbf{M} + 4\mathbf{S} + 1\mathbf{C} + 8\mathbf{A}$	Equation (4.4.12) [196]
2-way	$3\mathbf{M}_2 + 2\mathbf{S}_2 + 1\mathbf{C}_2 + 4\mathbf{A}_2$	Algorithm 4.4.16
4-way	$2\mathbf{M}_4 + 1\mathbf{S}_4 + 1\mathbf{C}_4 + 3\mathbf{A}_4$	Algorithm 4.4.17

¹ \mathbf{C} denotes the cost of multiplying by a constant derived from the curve parameter A .

rithm [163, Alg. 4] can be used to calculate fixed-point multiplications in binary curves. A follow up work by Oliveira, López, and Rodríguez [216] showed that a similar approach also applies for calculating fixed-point multiplications on the x -line variety.

In this section, we give details of right-to-left algorithms for scalar multiplication on the x -line and their adaptation for calculating fixed-point multiplications. This section also serves as a prelude of the optimizations that we found for the three-point ladder algorithm (described in Section 4.4.5).

Right-To-Left Ladder for Scalar Multiplication on the x -Line

Recall that the formulas for differential addition on Montgomery curves have exceptional cases. This lack of completeness introduces an implementation issue on instantiating Joye’s algorithm with the x -line arithmetic. Specifically, the issue appears in the first step of the algorithm; because, it requires to calculate a differential addition with \mathcal{O} as the difference point, which is one of the failure cases of the formula. However, this case only happens when the least-significant bit of the scalar is zero, so when scalars are even.

Rather than restricting the algorithm to odd scalars, Oliveira et al. [216] solved this issue by adding a point $S \notin \langle P \rangle$ to the accumulators. Thus, the difference point is not \mathcal{O} avoiding the exceptional case of the formula provided that $S, P - S \notin E[2](\mathbb{F}_q)$. With the introduction of S , the result of the main loop gets perturbed and produces $S + kP$ instead. One way to remove S is multiplying this point by h , where h is the order of S . This latter multiplication can be performed efficiently if h is, for example, a small number. The result of all these modifications made on top of the right-to-left Joye’s method derived on Algorithm 4.4.19 that calculates hkP on the x -line of a Montgomery curve.

The right-to-left algorithm has additional properties that are not present in the Montgomery ladder algorithm. In the differential addition of this algorithm, the difference point is not fixed for all loop iterations, unlike the Montgomery ladder step. This variation is because the CSWAP operation selects, at each iteration, the difference point between R_0 and R_1 . Either choice is a difference point that has $Z \neq 1$ causing that the differential addition requires an additional multiplication, i.e., it takes $4\mathbf{M} + 2\mathbf{S}$ field operations. Therefore, Algorithm 4.4.19 takes $n(6\mathbf{M} + 4\mathbf{S} + 8\mathbf{A} + 1\mathbf{C})$ field operations plus the cost of a scalar multiplication by h . In summary, this algorithm is more expensive than the left-to-right Montgomery ladder algorithm in the general case. Nonetheless, another interesting property is that the right-to-left method enables the calculation of fixed-point multiplications with a significant reduced cost.

Algorithm 4.4.19 Right-to-Left Ladder Algorithm for Scalar Multiplication on the x -Line [216].

Constants: $\mathbf{x}(S)$ and $\mathbf{x}(S - P)$, where $S \in E(\mathbb{F}_q)$ has order h such that $S \notin \langle P \rangle$ and $S, P - S \notin E[2](\mathbb{F}_q)$.

Input: (n, k) are integers such that $n \geq 1$ and $0 \leq k < 2^n$; and $\mathbf{x}(P) \in \mathbb{P}^1$ such that $P \in E(\mathbb{F}_q) \setminus \{\mathcal{O}, \mathcal{T}\}$.

Output: $\mathbf{x}(hkP) \in \mathbb{P}^1$.

```

1: Let  $(k_{n-1}, \dots, k_0)_2$  be the  $n$ -bit representation of  $k$  and define  $k_{-1} = 0$ .
2:  $R_0 \leftarrow \mathbf{x}(P - S)$ ,  $R_1 \leftarrow \mathbf{x}(S)$ ,  $R_2 \leftarrow \mathbf{x}(P)$ 
3: for  $i \leftarrow 0$  to  $n - 1$  do
4:    $b \leftarrow k_i \oplus k_{i-1}$ 
5:    $R_0, R_1 \leftarrow \mathbf{CSWAP}(R_0, R_1, b)$ 
6:    $R_2, R_0 \leftarrow 2R_2, R_0 +_{(R_1)} R_2$  //Using ladder step.
7: end for
8:  $R_0, R_1 \leftarrow \mathbf{CSWAP}(R_0, R_1, k_{n-1})$ 
9:  $R_1 \leftarrow hR_1$ 
10: return  $R_1$ 

```

Right-To-Left Ladder for Fixed-Point Multiplications

Recall that fixed-point multiplication is a particular case of scalar multiplication that occurs when P is known in advance. Fixed-point multiplication algorithms have two phases. In an off-line phase, some multiples of P are calculated and stored in a table; and during the execution phase, these points help on the calculation of the scalar multiplication. It is expected that the algorithm takes a fewer number of operations than the ones required by a generic scalar multiplication algorithm.

In this setting, Oliveira et al. [216] showed optimizations on top of Algorithm 4.4.19 that allowed calculating fixed-point multiplications. Note that all point doublings of the algorithm operate over P only, and since P is known, they can be performed off-line and stored in a table. Hence, the calculation of the ladder step reduces to calculate one differential addition.

An updated version of Oliveira et al.'s paper [214] showed that the differential addition can be performed faster when one of its operands is known in advance. Let $P, Q \in \mathbb{P}^1$ be points on the x -line, where P is the fixed point; the differential addition $P +_{(P-Q)} Q$, also denoted as $\mathbf{DiffAddFixed}(\mu_P, Q, P - Q)$, is calculated as

$$\begin{aligned} \mu_P &= \frac{X_P + Z_P}{X_P - Z_P}, \\ X_{P+Q} &= Z_{P-Q} \times [(X_Q + Z_Q) + \mu_P(X_Q - Z_Q)]^2, \\ Z_{P+Q} &= X_{P-Q} \times [(X_Q + Z_Q) - \mu_P(X_Q - Z_Q)]^2. \end{aligned} \tag{4.4.20}$$

This formula is defined whenever $X_P/Z_P \neq X_Q/Z_Q$ and $X_{P-Q} \neq 0$. However, μ_P is not defined when $X_P/Z_P = 1$ and is zero when $X_P/Z_P = -1$; these two exceptional cases appear if P is a point of order four. Then, $P \notin \left\{ \left(1, \pm \sqrt{(A+2)/B} \right), \left(-1, \pm \sqrt{(A-2)/B} \right) \right\}$. If μ_P is calculated in advance, Equation (4.4.20) takes $3\mathbf{M}+2\mathbf{S}+4\mathbf{A}$ field operations.

Using these observations, one can adapt the right-to-left ladder, as shown in Algorithm 4.4.21, to perform fixed-point multiplications on the x -line of a Montgomery curve.

Algorithm 4.4.21 Right-to-Left Ladder Algorithm for Fixed-Point Multiplication on the x -Line [214].

Constants: $P, S \in E(\mathbb{F}_q)$, where P is the fixed-point such that $P \notin E[4](\mathbb{F}_q)$; S is a point of order h such that $S \notin \langle P \rangle$, and $S, P - S \notin E[2](\mathbb{F}_q)$.

Precompute: $(\mu_{n-1}, \dots, \mu_0)$, where $\mu_i = (X_i + Z_i)/(X_i - Z_i) \in \mathbb{F}_q$, and $\mathbf{x}(2^i P) = (X_i : Z_i)$ for all $0 \leq i < n$.

Input: (n, k) are integers such that $n \geq 1$ and $0 \leq k < 2^n$.

Output: $\mathbf{x}(hkP) \in \mathbb{P}^1$.

- 1: Let $(k_{n-1}, \dots, k_0)_2$ be the n -bit representation of k and define $k_{-1} = 0$.
- 2: $R_0 \leftarrow \mathbf{x}(P)$, $R_1 \leftarrow \mathbf{x}(S)$, $R_2 \leftarrow \mathbf{x}(P - S)$
- 3: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
- 4: $b \leftarrow k_i \oplus k_{i-1}$
- 5: $R_1, R_2 \leftarrow \text{CSWAP}(R_1, R_2, b)$
- 6: $R_2 \leftarrow \text{DiffAddFixed}(\mu_i, R_2, R_1)$ //Equation (4.4.20)
- 7: **end for**
- 8: $R_1, R_2 \leftarrow \text{CSWAP}(R_1, R_2, k_{n-1})$
- 9: $R_1 \leftarrow hR_1$
- 10: **return** R_1

Algorithm 4.4.21 takes a constant number of operations. In total, this algorithm calculates $n(3\mathbf{M}+2\mathbf{S}+4\mathbf{A})$ field operations plus a scalar multiplication by h . Regarding memory footprint, the precomputed table stores exactly n field elements. A downside of this algorithm is that one can not increase the table size for accelerating even more the calculation as it usually happens on other fixed-point multiplication algorithms. Nonetheless, a big advantage of this algorithm is its regular-execution pattern that could prevent of some side-channel attacks. Moreover, fetching points from the table requires non-secret indexes; i.e., during execution, points can be taken from the table directly. The variability inherent to the bits of the scalar is handled by the CSWAP function, which must interchange the accumulator points in constant-time. Therefore, this method is suitable for calculating fixed-point multiplications using secret scalars.

Table 4.4.22 compares ladder algorithms for scalar multiplication on the x -line. As can be seen, the right-to-left method is slightly slower than Montgomery ladder for performing generic scalar multiplications. But in the fixed-point scenario, the right-to-left method is faster taking $0.56\times$ the cost of Montgomery ladder.

Instantiating the right-to-left ladder algorithm on the x -line brings significant improvements on the calculation of fixed-point multiplications. In the following section, we show the use of right-to-left algorithms for accelerating another point operation on the x -line.

4.4.5 A New Three-Point Ladder Algorithm

The implementation of the Supersingular Isogeny Diffie-Hellman (SIDH) protocol, introduced by Jao et al. [160], uses Montgomery curves as the elliptic curve model due to the

Table 4.4.22: Operation counts of scalar multiplication on the x -line.

Scanning Direction	Algorithm	Fixed-Point	Step Ladder	M-per-bit ¹	Ratio
Left to right	Montgomery ladder [196]	No	5M + 4S	8.2M	1.00 ×
Right to left	Algorithm 4.4.19 [216]	No	6M + 4S	9.2M	1.12 ×
	Algorithm 4.4.21 [214]	Yes	3M + 2S	4.6M	0.56 ×

¹ Assuming 1S = 0.8M.

efficiency of the operations on the x -line. Given a secret scalar k and points $P, Q \in E(\mathbb{F}_q)$, part of this protocol requires calculating $\mathbf{x}(P + kQ)$ efficiently.

In this section, we propose a new algorithm that performs this operation faster than two well-known methods. Our algorithm builds on top of the right-to-left ladder described in the previous section. We want to remark that although this calculation is relevant for SIDH, the results of this section apply generally to any set with a differential addition operation defined. The results of this section were published in the TC 2017 journal paper [103] [\(P\)](#), which was co-authored with CINVESTAV IPN's researchers.

Previous Methods for Calculating $\mathbf{x}(P + kQ)$

A direct method is to first calculate $kQ \in E(\mathbb{F}_q)$, a scalar multiplication on the Montgomery curve (using Algorithm 4.4.15), and then add P , to finally obtain $\mathbf{x}(P + kQ)$. This method has a regular execution pattern; however, it requires knowledge of the y -coordinate of P and Q . This method is dominated by the cost of the Montgomery ladder, which takes $n(6\mathbf{M} + 4\mathbf{S} + 8\mathbf{A})$, plus a constant number of field multiplications ($< 30\mathbf{M}$) due to the y -coordinate recovery and the projective point addition.

An alternative method was introduced by Jao et al. [160], who proposed a regular-execution algorithm called three-point ladder. This ladder, shown in Algorithm 4.4.23, performs operations on the x -line taking as input $\mathbf{x}(P)$, $\mathbf{x}(Q)$, and $\mathbf{x}(Q - P)$ to calculate $\mathbf{x}(P + kQ)$. For every bit of the scalar, this algorithm performs a constant number of operations: two differential additions and one point doubling. In the SIDH protocol, the parameter A of the Montgomery curve is not a constant; thus, multiplying times $(A + 2)/4$ must be counted as a generic multiplication. In total, Algorithm 4.4.23 takes $n(9\mathbf{M} + 6\mathbf{S} + 14\mathbf{A})$ field operations. The three-point ladder improves over the direct method because it does not require the y -coordinate of points.

By analyzing the operations of the three-point ladder algorithm, it is clear that the algorithm is, in essence, a Montgomery ladder plugged with a differential addition, for which its difference point is either $\mathbf{x}(P)$ or $\mathbf{x}(Q - P)$ depending on the bits of the scalar. Another similarity is that both algorithms scan the bits of the scalar from the most- to the least-significant bit, i.e., they are left-to-right multiplication algorithms. When looking for alternative methods, we turned our efforts on investigating the use of right-to-left multiplication algorithms.

Algorithm 4.4.23 Left-to-Right Three-Point Ladder Algorithm for $\mathbf{x}(P + kQ)$ [160].

Input: (n, k) are integers such that $n \geq 1$ and $0 \leq k < 2^n$; and $\mathbf{x}(P)$, $\mathbf{x}(Q)$, and $\mathbf{x}(Q - P)$ where $P, Q \in E(\mathbb{F}_q)$ such that $P, Q, Q - P \notin \{\mathcal{O}, \mathcal{T}\}$.

Output: $\mathbf{x}(P + kQ) \in \mathbb{P}^1$.

```

1: Let  $(k_{n-1}, \dots, k_0)_2$  be the  $n$ -bit representation of  $k$ .
2:  $R_0 \leftarrow \mathbf{x}(\mathcal{O})$ ,  $R_1 \leftarrow \mathbf{x}(Q)$ ,  $R_2 \leftarrow \mathbf{x}(P)$ 
3:  $T_0 \leftarrow \mathbf{x}(P)$ ,  $T_1 \leftarrow \mathbf{x}(Q - P)$ ,  $T_2 \leftarrow \mathbf{x}(Q)$ 
4: for  $i \leftarrow n - 1$  to  $0$  do
5:   if  $k_i = 0$  then
6:      $(R_2, R_1, R_0) \leftarrow (R_2 +_{(T_0)} R_0, R_0 +_{(T_2)} R_1, 2R_0)$ 
7:   else
8:      $(R_2, R_0, R_1) \leftarrow (R_2 +_{(T_1)} R_1, R_0 +_{(T_2)} R_1, 2R_1)$ 
9:   end if
10: end for
11: return  $R_2$ 

```

A Faster Three-Point Ladder Algorithm

We now introduce a faster algorithm to calculate $\mathbf{x}(P + kQ)$. By revisiting the right-to-left Joye's algorithm, we found suitable conditions for its use in the SIDH setting. Recall that the right-to-left Joye's algorithm instantiated with operations in the x -line results in a method for calculating $\mathbf{x}(hkP)$, as shown in Algorithm 4.4.19. The additional factor h appears as a side effect produced by the introduction of an auxiliary order- h point $S \notin \langle P \rangle$ that avoids exceptional cases of the differential point addition. However, if the multiplication by h is omitted, the right-to-left ladder produces $\mathbf{x}(S + kP)$ instead. Therefore, we propose Algorithm 4.4.24 that calculates $\mathbf{x}(P + kQ)$ given a scalar k and $\mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(Q - P) \in \mathbb{P}^1$, where $P, Q \in E(\mathbb{F}_q)$ such that $P \notin \langle Q \rangle$.

Algorithm 4.4.24 Right-to-Left Three-Point Ladder Algorithm for $\mathbf{x}(P + kQ)$.

Input: (n, k) are integers such that $n \geq 1$ and $0 \leq k < 2^n$; $\mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(Q - P) \in \mathbb{P}^1$, where $P, Q \in E(\mathbb{F}_q)$ such that $P, Q, Q - P \notin \{\mathcal{O}, \mathcal{T}\}$.

Output: $\mathbf{x}(P + kQ) \in \mathbb{P}^1$.

```

1: Let  $(k_{n-1}, \dots, k_0)_2$  be the  $n$ -bit representation of  $k$  and define  $k_{-1} = 0$ .
2:  $R_0 \leftarrow \mathbf{x}(Q - P)$ ,  $R_1 \leftarrow \mathbf{x}(P)$ ,  $R_2 \leftarrow \mathbf{x}(Q)$ 
3: for  $i \leftarrow 0$  to  $n - 1$  do
4:    $b \leftarrow k_i \oplus k_{i-1}$ 
5:    $R_0, R_1 \leftarrow \text{CSWAP}(R_0, R_1, b)$ 
6:    $R_2, R_0 \leftarrow 2R_2, R_2 +_{(R_1)} R_0$ 
7: end for
8:  $R_0, R_1 \leftarrow \text{CSWAP}(R_0, R_1, k_{n-1})$ 
9: return  $R_1$ 

```

Algorithm 4.4.24 uses three accumulator points, namely $R_0, R_1, R_2 \in \mathbb{P}^1$. For every bit of the scalar, the bit value determines whether R_2 must be accumulated in R_0 or R_1 . After that, R_2 is doubled unconditionally. Its loop-invariant is $R_0 = R_2 - R_1$, which is similar to the one of Montgomery ladder. At every iteration, it calculates one differential

addition and one point doubling; so it takes $n(7\mathbf{M}+4\mathbf{S}+8\mathbf{A})$ field operations in total (counting the multiplication by $(A+2)/4$ as a generic multiplication).

Table 4.4.25 shows operation counts for calculating $\mathbf{x}(P+kQ)$. On the one hand, the direct method is faster than Algorithm 4.4.24; however, it requires knowledge of the y -coordinate of points when calculating the addition of P . On the other hand, Algorithm 4.4.24 is $1.34\times$ faster than the three-point ladder by Jao et al. Moreover, both algorithms have the same interface, so it can be used as a drop-in replacement.

Table 4.4.25: Operation counts of $\mathbf{x}(P+kQ)$ on the x -line.

Scanning	Algorithm	Fixed-Point	Requirements	\mathbf{M} -per-bit ¹
Left to Right	Direct Method ²	No	The y -coord. of P and Q	8.6
	Three-point Ladder [160]	No	$\mathbf{x}(Q-P)$	13.0
Right to Left	Algorithm 4.4.24	No	$\mathbf{x}(Q-P)$	9.6
	Algorithm 4.4.26	Yes	$\mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(Q-P)$.	4.3

¹ Assumes that $1\mathbf{S} = 0.66\mathbf{M}$.

² Cost of Montgomery ladder for $\mathbf{x}(kQ)$. The additional multiplications needed for recovering the y -coordinate and adding P are not counted because they do not depend on k .

A Faster Three-Point Ladder Algorithm: The Fixed-point Case

The right-to-left three-point ladder can be accelerated when P or Q are known in advance. Unlike Jao et al.'s three-point ladder, Algorithm 4.4.26 precomputes some points for calculating $\mathbf{x}(P+kQ)$ faster. This algorithm calculates one differential addition per bit of the scalar k . In total, it takes $n(3\mathbf{M}+2\mathbf{S})$ field operations. In comparison with the original three-point ladder, the right-to-left three-point ladder with precomputation shows a $3\times$ speedup factor for fixed-point multiplications, as shown in Table 4.4.25.

Recovering the y -Coordinate of $P+kQ$

For binary curves, López and Dahab [186] showed a method that recovers the y -coordinate of kP from the coordinates of P , $\mathbf{x}(kP)$ and $\mathbf{x}(P+kP)$. The latter points are obtained as part of the execution of the Montgomery ladder. Okeya and Sakurai [210] showed a similar method for Montgomery curves.

We show a y -recovery method for the three-point ladder algorithm. More specifically, let $T_0 = (x_0, y_0) \in E(\mathbb{F}_q)$ be an affine point with $y_0 \neq 0$, and $\mathbf{x}(T_i) = (X_i : Z_i) \in \mathbb{P}^1$ for $i = 1, 2, 3$ are points on the x -line such that $T_2 = T_1 + T_0$ and $T_3 = T_1 - T_0$. Then, following [210, Corollary 2], there is a point $(X'_1 : Y'_1 : Z'_1) \sim T_1 \in E(\mathbb{F}_q)$, where

$$\begin{aligned} X'_1 &= 4By_0Z_1Z_2Z_3X_1 \\ Y'_1 &= (X_3Z_2 - Z_3X_2)(X_1 - Z_1x_0)^2 \\ Z'_1 &= 4By_0Z_1^2Z_2Z_3. \end{aligned} \tag{4.4.27}$$

Algorithm 4.4.26 Right-to-Left Three-Point Ladder Algorithm for $\mathbf{x}(P + kQ)$ with Fixed-Points.

Constants: $\mathbf{x}(P)$, $\mathbf{x}(Q)$, and $\mathbf{x}(Q - P)$, where $P, Q \in E(\mathbb{F}_q)$ such that $P, Q - P \notin E[2](\mathbb{F}_q)$ and $Q \notin E[4](\mathbb{F}_q)$.

Precompute: $(\mu_{n-1}, \dots, \mu_0)$, where $\mu_i = (X_i + Z_i)/(X_i - Z_i) \in \mathbb{F}_q$, and $\mathbf{x}(2^i Q) = (X_i : Z_i)$ for all $0 \leq i < n$.

Input: (n, k) are integers such that $n \geq 1$ and $0 \leq k < 2^n$.

Output: $\mathbf{x}(P + kQ) \in \mathbb{P}^1$.

- 1: Let $(k_{n-1}, \dots, k_0)_2$ be the n -bit representation of k and define $k_{-1} = 0$.
 - 2: $R_0 \leftarrow \mathbf{x}(Q - P)$, $R_1 \leftarrow \mathbf{x}(P)$, $R_2 \leftarrow \mathbf{x}(Q)$
 - 3: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
 - 4: $b \leftarrow k_i \oplus k_{i-1}$
 - 5: $R_0, R_1 \leftarrow \text{CSWAP}(R_0, R_1, b)$
 - 6: $R_0 \leftarrow \text{DiffAddFixed}(\mu_i, R_2, R_1)$ //Equation (4.4.20).
 - 7: **end for**
 - 8: $R_0, R_1 \leftarrow \text{CSWAP}(R_0, R_1, k_{n-1})$
 - 9: **return** R_1
-

Now, we show that the accumulator points of the right-to-left three-point ladder allows recovering the y -coordinate of $P + kQ$. Since the loop-invariant of the right-to-left ladder is $R_0 = R_2 - R_1$; after n iterations, the accumulators are equal to

$$\begin{aligned} R_0 &= \mathbf{x}((2^n - (k \bmod 2^n))Q - P) \\ R_1 &= \mathbf{x}(P + (k \bmod 2^n)Q) \\ R_2 &= \mathbf{x}(2^n Q). \end{aligned} \tag{4.4.28}$$

With these values, we calculate $R_3 = R_2 +_{(R_0)} R_1$ as follows. Using these values in the previous formula, one can recover the y -coordinate of $P + kQ$ by setting $(T_0, T_1, T_2, T_3) \leftarrow (R_2, R_1, R_3, R_0)$ whenever there is previous knowledge of the y -coordinate of the point $2^n Q$. For instance, in the fixed-point scenario, the point $2^n Q$ can be precomputed. This recovering method increases by one differential addition to the cost of Equation (4.4.27).

4.4.6 An Optimized Point Tripling Formula

Point tripling refers to calculate $3P$ given a point P . This operation is usually performed by first calculating $2P$ and then adding P to it. However, some field operations can be saved by composing the formulas for point doubling and addition.

We look for optimizations for tripling points on the x -line of a Montgomery curve. Hence, we restrict to the case of calculating the x -coordinate of $3^i P$ given the x -coordinate of P and an integer $i > 0$, where the points are in projective coordinates.

As a starting point, note that calculating $\mathbf{x}(3P)$ using a point doubling followed by a differential point addition takes $7\mathbf{M}+4\mathbf{S}+8\mathbf{A}$ field operations. Subramanya Rao [258] showed an efficient formula for point tripling. Let A be the Montgomery curve parameter

and given $\mathbf{x}(P) = (X_P : Z_P)$, such a formula calculates $\mathbf{x}(3P) = (X_{3P} : Z_{3P})$ as

$$\begin{aligned} \lambda &= (X_P^2 - Z_P^2)^2, & \gamma &= 4(X_P^2 + Z_P^2 + AX_PZ_P), \\ X_{3P} &= X_P(\lambda - \gamma Z_P^2)^2, & Z_{3P} &= Z_P(\lambda - \gamma X_P^2)^2. \end{aligned} \quad (4.4.29)$$

This formula was derived by composing the point doubling and differential addition and its computational cost is $6\mathbf{M}+5\mathbf{S}+9\mathbf{A}$ field operations.

In our optimization, we consider the elliptic curve parameter A as an arbitrary value. More generally, we represent A as a quotient $A = A_0/A_1$. We optimize the point tripling formula observing that $2X_PZ_P$ can be calculated from X_P^2 , Z_P^2 , and $(X_P + Z_P)^2$ relying on the following equation

$$2X_PZ_P = (X_P + Z_P)^2 - (X_P^2 + Z_P^2). \quad (4.4.30)$$

Thus, λ from Equation (4.4.29) is alternatively calculated as

$$\begin{aligned} \lambda &= (X_P^2 - Z_P^2)^2 \\ &= (X_P + Z_P)^2(X_P - Z_P)^2 \\ &= (X_P + Z_P)^2[(X_P^2 + Z_P^2) - 2X_PZ_P]. \end{aligned} \quad (4.4.31)$$

Likewise, γ from Equation (4.4.29) is given as

$$\begin{aligned} \gamma &= 4(X_P^2 + Z_P^2 + AX_PZ_P) \\ &= 2[2(X_P^2 + Z_P^2 + AX_PZ_P)] \\ &= 2[2(X_P + Z_P)^2 + (A - 2)(2X_PZ_P)]. \end{aligned} \quad (4.4.32)$$

Applying these three equations and considering that $A = A_0/A_1$, we calculate the point $\mathbf{x}(3P) = (X_{3P} : Z_{3P})$ as

$$\begin{aligned} \lambda &= (2A_1)(X_1 + Z_1)^2[(X_1^2 + Z_1^2) - 2X_1Z_1] \\ \gamma &= 4[(2A_1)(X_1 + Z_1)^2 + (A_0 - 2A_1)(2X_1Z_1)] \\ X_{3P} &= X_P(\lambda - \gamma Z_P^2)^2 \\ Z_{3P} &= Z_P(\lambda - \gamma X_P^2)^2. \end{aligned} \quad (4.4.33)$$

Algorithm 4.4.34 shows this point tripling formula. This algorithm takes $7\mathbf{M}+5\mathbf{S}+11\mathbf{A}$ field operations. However, to calculate $\mathbf{x}(3^iP)$ one can precompute $A'_0 = A_0 - 2A_1$ and $A'_1 = 2A_1$ reducing its cost to $7\mathbf{M} + 5\mathbf{S} + 9\mathbf{A}$ field operations.

Table 4.4.35 shows the cost of several tripling formulas reported in the literature. It can be seen that our formula improves point tripling computation by $1\mathbf{M}-1\mathbf{S}-1\mathbf{A}$ with respect to the formula used by Costello et al. [78]. Independent work of Costello and Hisil [76] showed formulas for point tripling of similar performance.

Algorithm 4.4.34 Point Tripling on the x -Line.

Precompute: $A'_0 \leftarrow A_0 - 2A_1$, and $A'_1 \leftarrow 2A_1$, where $A = A_0/A_1$.

Input: $\mathbf{x}(P) = (X_P : Z_P)$, where $P \in E(\mathbb{F}_q) \notin \{\mathcal{O}, \mathcal{T}\}$.

Output: $\mathbf{x}(3P) = (X_{3P} : Z_{3P})$.

1: $t_0 \leftarrow X_P^2$	8: $t_2 \leftarrow A'_1 \times t_2$	15: $t_2 \leftarrow t_2 \times t_4$
2: $t_1 \leftarrow Z_P^2$	9: $t_5 \leftarrow t_2 + t_5$	16: $t_0 \leftarrow t_2 - t_0$
3: $t_2 \leftarrow X_P + Z_P$	10: $t_5 \leftarrow t_5 + t_5$	17: $t_1 \leftarrow t_2 - t_1$
4: $t_2 \leftarrow t_2^2$	11: $t_5 \leftarrow t_5 + t_5$	18: $t_0 \leftarrow t_0^2$
5: $t_3 \leftarrow t_0 + t_1$	12: $t_0 \leftarrow t_0 \times t_5$	19: $t_1 \leftarrow t_1^2$
6: $t_4 \leftarrow t_2 - t_3$	13: $t_1 \leftarrow t_1 \times t_5$	20: $X_{3P} \leftarrow X_P \times t_1$
7: $t_5 \leftarrow A'_0 \times t_4$	14: $t_4 \leftarrow t_3 - t_4$	21: $Z_{3P} \leftarrow Z_P \times t_0$
22: return $(X_{3P} : Z_{3P})$		

Table 4.4.35: Operation counts of point tripling on the x -line.

$A = A_0/A_1$	Field Operations	Precomputation	Reference
$A_1 = 1$	$7\mathbf{M} + 4\mathbf{S} + 8\mathbf{A}$	$\{(A + 2)/4\}$	Montgomery [196]
	$6\mathbf{M} + 5\mathbf{S} + 9\mathbf{A}$	\emptyset	Subramanya [258]
	$5\mathbf{M} + 6\mathbf{S} + 7\mathbf{A}$	$\{2A\}$	Zanon et al. [273]
A_1 arbitrary	$8\mathbf{M} + 4\mathbf{S} + 8\mathbf{A}$	$\{A_0 + 2A_1, 4A_1\}$	Costello et al. [78]
	$7\mathbf{M} + 5\mathbf{S} + 10\mathbf{A}$	$\{A_0 \pm 2A_1\}$	Costello et al. [76]
	$7\mathbf{M} + 5\mathbf{S} + 9\mathbf{A}$	$\{A_0 - 2A_1, 2A_1\}$	Algorithm 4.4.34 (this work)

4.5 Arithmetic of Twisted Edwards Curves

Edwards [90] showed a form of elliptic curves where the addition law is unified, i.e., the same formula works for both point doubling and additions. Moreover, it is also \mathbb{F}_q -complete under some restrictions. Having complete formulas helps to protect implementation of elliptic curve operations used in cryptographic algorithms. For instance, adding points is performed free of exceptions, unlike other elliptic curve models, where addition formulas have exceptional cases. Building on top of Edwards work, Bernstein et al. [34] augmented the study of these curves and introduced several generalizations that lead to a family of curves known as twisted Edwards curves [26], which inherit the completeness of point addition formulas from Edwards curves.

In this section we describe the arithmetic of twisted Edwards curves; then, we propose algorithms for evaluating point addition formula in parallel, and finally, we present efficient methods for fixed-point multiplications.

4.5.1 Twisted Edwards Curves

Let \mathbb{F}_q be a field of odd characteristic, a *twisted Edwards curve* over \mathbb{F}_q is defined as

$$E/\mathbb{F}_q: ax^2 + y^2 = 1 + dx^2y^2, \quad (4.5.1)$$

such that $a, d \in \mathbb{F}_q$ and $ad(a - d) \neq 0$. The group law of twisted Edwards curves is \mathbb{F}_q -complete provided that a is a square and d is a non-square in \mathbb{F}_q . Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be two points on the curve, the coordinates of $P + Q = (x_{P+Q}, y_{P+Q})$ are calculated as

$$x_{P+Q} = \frac{x_P y_Q + y_P x_Q}{1 + dx_P x_Q y_P y_Q}, \quad y_{P+Q} = \frac{y_P y_Q - ax_P x_Q}{1 - dx_P x_Q y_P y_Q}. \quad (4.5.2)$$

This formula works for any point on the curve including the identity element, which unlike the Weierstrass form, is an affine point $\mathcal{O} = (0, 1)$.

The affine points can also be embedded into a projective space. The projective embedding of the curve in \mathbb{P}^3 (and after resolving singularities) is given by the equations

$$aX^2 + Y^2 = Z^2 + dT^2 \text{ and } XY = TZ. \quad (4.5.3)$$

An affine point $(x, y) \mapsto (x : y : xy : 1) \in \mathbb{P}^3$ with inverse $(X : Y : T : Z) \mapsto (X/Z, Y/Z)$ with $Z \neq 0$. There exist four points at infinity (with $Z = 0$) that satisfy these projective equations. However, these points have coordinates involving the square root of d , this is one of the reasons why choosing d as non-square in \mathbb{F}_q causes the points be defined in an extension of \mathbb{F}_q . Hence, the set of \mathbb{F}_q -rational points, denoted as

$$E(\mathbb{F}_q) = \{(X : Y : T : Z) \in \mathbb{P}^3 : aX^2 + Y^2 = Z^2 + dT^2 \text{ and } XY = TZ\}, \quad (4.5.4)$$

forms an additive group having $\mathcal{O} = (0 : 1 : 0 : 1)$ as its identity element.

Twisted Edwards curves with $a = -1$ have the more efficient formulas for point addition [148]. Let $P = (X_P : Y_P : T_P : Z_P)$ and $Q = (X_Q : Y_Q : T_Q : Z_Q)$ be two projective points on $E(\mathbb{F}_q)$ with $a = -1$, the coordinates of $P + Q = (X_{P+Q} : Y_{P+Q} : T_{P+Q} : Z_{P+Q})$ as calculated as

$$\begin{aligned} A &\leftarrow (Y_P - X_P) \times (Y_Q - X_Q), & B &\leftarrow (Y_P + X_P) \times (Y_Q + X_Q), \\ C &\leftarrow 2d \times T_P \times T_Q, & D &\leftarrow 2Z_P \times Z_Q, \\ E &\leftarrow B - A, & F &\leftarrow D - C, \\ G &\leftarrow D + C, & H &\leftarrow B + A, \\ X_{P+Q} &\leftarrow E \times F, & Y_{P+Q} &\leftarrow G \times H, \\ Z_{P+Q} &\leftarrow F \times G, & T_{P+Q} &\leftarrow E \times H. \end{aligned} \quad (4.5.5)$$

This formula requires $8\mathbf{M} + 1\mathbf{C} + 9\mathbf{A}$ field operations, where \mathbf{C} denotes the cost of multiplying by $2d$. Some operations are saved if one of the points is known in advance. Let $Q = (x_Q : y_Q : x_Q y_Q : 1)$ be the known-point, if the values $y_Q - x_Q$, $y_Q + x_Q$, and $2dx_Q y_Q$ are precomputed, the cost of point addition reduces to $7\mathbf{M} + 7\mathbf{A}$.

Equivalence with Montgomery Curves

Bernstein et al. [26] showed that Montgomery and twisted Edwards curves are related through rational maps.

Given a twisted Edwards curve $ax^2 + y^2 = 1 + dx^2y^2$ where $a, d \in \mathbb{F}_q$ such that $a, d \neq 0$, one can obtain a point on a Montgomery curve $Bv^2 = u^3 + Au^2 + u$ where $A = 2\left(\frac{a+d}{a-d}\right)$ and $B = \frac{4}{a-d}$ using the following map

$$(x, y) \mapsto (u, v) = \left(\frac{1+y}{1-y}, \frac{1+y}{(1-y)x} \right). \quad (4.5.6)$$

Conversely, given a Montgomery curve $Bv^2 = u^3 + Au^2 + u$ where $A, B \in \mathbb{F}_q$ such that $A \notin \{2, -2\}$ and $B \neq 0$, one can obtain a point on a twisted Edwards curve $ax^2 + y^2 = 1 + dx^2y^2$ where $a = \frac{A+2}{B}$ and $d = \frac{A-2}{B}$ using the following map

$$(u, v) \mapsto (x, y) = \left(\frac{u}{v}, \frac{u-1}{(u+1)} \right). \quad (4.5.7)$$

There are points for which the maps are not defined. These exceptional points are handled as special cases during the mapping.

4.5.2 Parallel Point Addition

We describe implementation techniques for performing point addition in parallel and its execution in SIMD units. It is known that the internal field operations of the addition formula can run in parallel. For example, the operations can be distributed in two parallel units as noted by Hisil et al. [148]. They presented a distribution of operations (shown in Figure 4.5.8) that evenly assigns field operations of the same complexity to each unit. However, we noticed that there remain some opportunities for optimizations with respect to its actual implementation using vector units.

Unit 1	Unit 2
$A_1 \leftarrow Y_P - X_P$	$B_1 \leftarrow Y_Q - X_Q$
$A_2 \leftarrow Y_P + X_P$	$B_2 \leftarrow Y_Q + X_Q$
$A_3 \leftarrow A_1 \times \underline{B_1}$	$B_3 \leftarrow \underline{A_2} \times B_2$
$A_4 \leftarrow T_P \times T_Q$	$B_4 \leftarrow Z_P \times Z_Q$
$A_5 \leftarrow 2d \times A_4$	$B_5 \leftarrow 2 \times B_4$
$A_6 \leftarrow \underline{B_3} - A_3$	$B_6 \leftarrow B_5 - \underline{A_5}$
$A_7 \leftarrow \underline{B_5} + A_5$	$B_7 \leftarrow \underline{A_3} + B_3$
$X_{P+Q} \leftarrow A_6 \times \underline{B_6}$	$Y_{P+Q} \leftarrow \underline{A_7} \times B_7$
$T_{P+Q} \leftarrow A_6 \times \underline{B_7}$	$Z_{P+Q} \leftarrow B_6 \times A_7$

Figure 4.5.8: Two-way scheduling of point addition for twisted Edwards curves with $a = -1$ from [148]. Underlined values represent dependencies between units.

In particular, we observed that such a partitioning involves many data dependencies between units. A data dependency exists whenever a unit produces a value that is an input of an operation performed in the other unit. In practice, this kind of dependencies introduces an extra computational cost associated to communication between units. Since we instantiate parallel units through the AVX2 vector unit, sharing values between units translates to moving words within vector registers. As a consequence, it requires using permutation instructions, which are high-latency instructions that cause overheads during execution time. Therefore, for the sake of efficiency, the design of a parallel scheduling of operations must also consider to minimize data dependencies between units.

Taking these aspects into consideration, we propose a parallel scheduling with fewer data dependencies between units, as shown in Figure 4.5.9. In our scheduling, every unit operates as much as possible on values that were previously produced in the same unit, and we pair field operations of similar complexity to balance the amount of work processed by each unit. In comparison with the previous scheduling, ours reduces from nine to four the number of dependencies between units, and during its implementation, we observed a reduction of a half of the number of vector permutations for point addition.

Unit 1	Unit 2
$A_1 \leftarrow Y_P - X_P$	$B_1 \leftarrow Y_P + X_P$
$A_2 \leftarrow Y_Q - X_Q$	$B_2 \leftarrow Y_Q + X_Q$
$A_3 \leftarrow A_1 \times A_2$	$B_3 \leftarrow B_1 \times B_2$
$A_4 \leftarrow T_P \times T_Q$	$B_4 \leftarrow Z_P \times Z_Q$
$A_5 \leftarrow 2d \times A_4$	$B_5 \leftarrow 2 \times B_4$
$A_6 \leftarrow \underline{B_3} - A_3$	$B_6 \leftarrow \underline{A_5} - B_5$
$A_7 \leftarrow B_3 + A_3$	$B_7 \leftarrow A_5 + B_5$
$X_{P+Q} \leftarrow A_6 \times \underline{B_6}$	$Y_{P+Q} \leftarrow B_7 \times \underline{A_7}$
$T_{P+Q} \leftarrow A_6 \times A_7$	$Z_{P+Q} \leftarrow B_7 \times B_6$

Figure 4.5.9: Our proposed two-way scheduling of point addition for twisted Edwards curves with $a = -1$. Underlined values represent dependencies between units.

We use the concept of n -way operations from Section 3.3 to describe the implementation of point additions. Algorithm 4.5.10 shows a sequence of two-way operations for calculating $P + Q$, where $P = (X_P : Y_P : T_P : Z_P)$ and Q is known in advance.

Analogously to the case of two units, point addition formula can also be performed using four execution units [148]. We now extend this previous scheduling to its execution using four units. Algorithm 4.5.11 shows the scheduling of operations to calculate point additions for the family of curves with $a = -1$.

We also define a *four-way point addition* ($1PA_4$) as the set of operations $P_i + Q_i$ for $0 \leq i < 4$. To implement them, we applied four-way field operations from Section 3.3 to calculate the point addition formula. Hence, we store each coordinate of the four points in vector registers, for example, $\langle X_{P_0}, X_{P_1}, X_{P_2}, X_{P_3} \rangle$ is a set of vector registers containing the X -coordinate of each point.

Algorithm 4.5.10 Two-way Point Addition for Twisted Edwards Curves with $a = -1$.

Input: $\langle X_P, Y_P \rangle, \langle T_P, Z_P \rangle, \langle Y_Q - X_Q, Y_Q + X_Q \rangle$ and $\langle 2dX_QY_Q, 2 \rangle$, projective coordinates of $P, Q \in E(\mathbb{F}_p)$.

Output: $\langle X_{P+Q}, Y_{P+Q} \rangle$ and $\langle T_{P+Q}, Z_{P+Q} \rangle$, projective coordinates of $P + Q$.

- 1: $\langle Y_P, X_P \rangle \leftarrow \text{PERM}(\langle X_P, Y_P \rangle, 0\mathbf{x}4\mathbf{E})$
 - 2: $\langle X_P - Y_P, Y_P + X_P \rangle \leftarrow \langle X_P, Y_P \rangle \pm \langle Y_P, X_P \rangle$
 - 3: $\langle A, B \rangle \leftarrow \langle X_P - Y_P, Y_P + X_P \rangle \times \langle Y_Q - X_Q, Y_Q + X_Q \rangle$
 - 4: $\langle C, D \rangle \leftarrow \langle T_P, Z_P \rangle \times \langle 2dX_QY_Q, 2 \rangle$
 - 5: $\langle A, C \rangle \leftarrow \text{PERM}(\langle A, B \rangle, \langle C, D \rangle, 0\mathbf{x}20)$
 - 6: $\langle B, D \rangle \leftarrow \text{PERM}(\langle A, B \rangle, \langle C, D \rangle, 0\mathbf{x}31)$
 - 7: $\langle E, F \rangle \leftarrow \langle B, D \rangle - \langle A, C \rangle$
 - 8: $\langle H, G \rangle \leftarrow \langle B, D \rangle + \langle A, C \rangle$
 - 9: $\langle E, G \rangle \leftarrow \text{BLEND}(\langle E, F \rangle, \langle H, G \rangle, 0\mathbf{x}\mathbf{F0})$
 - 10: $\langle H, F \rangle \leftarrow \text{BLEND}(\langle H, G \rangle, \langle E, F \rangle, 0\mathbf{x}\mathbf{F0})$
 - 11: $\langle F, H \rangle \leftarrow \text{PERM}(\langle H, F \rangle, 0\mathbf{x}4\mathbf{E})$
 - 12: $\langle X_{P+Q}, Y_{P+Q} \rangle \leftarrow \langle F, H \rangle \times \langle E, G \rangle$
 - 13: $\langle T_{P+Q}, Z_{P+Q} \rangle \leftarrow \langle H, F \rangle \times \langle E, G \rangle$
 - 14: **return** $\langle X_{P+Q}, Y_{P+Q} \rangle$, and $\langle T_{P+Q}, Z_{P+Q} \rangle$
-

Performance Benchmark

We implemented the parallel strategies for point additions using the two- and four-way prime field operations from Sections 3.4 and 3.6. We want to determine which of the parallel variants exposed above offers a better performance.

Table 4.5.12 shows the timings obtained by our benchmark. The first row of the table is the baseline of our comparison, since it represents a single point addition implemented with native 64-bit instructions.

Table 4.5.12: Time in clock cycles of point addition on a twisted Edwards curve.

Operation	Field Operations	Parallel Strategy	edwards25519		edwards448	
			Latency ¹	Speedup	Latency ¹	Speedup
1 PA	7 M + 7 A	Single	324	1.00×	820	1.00×
	4 M ₂ + 3 A ₂	Two-way	264	1.22×	571	1.43×
	2 M ₄ + 2 A ₄	Four-way	242	1.33×	504	1.63×
1 PA ₄	7 M ₄ + 7 A ₄	Four-way	833	1.55×	1,520	2.15×

¹ Entries are clock cycles measured on a Skylake processor.

In our implementation, we obtained 1.33× and 1.63× speedup factors, respectively, for edwards25519 and edwards448 curves by using four-way operations. As it can be seen, the acceleration obtained for point additions is smaller than the acceleration of the

Algorithm 4.5.11 Four-way Point Addition for Twisted Edwards Curves with $a = -1$.

Input: $\langle X_P, Y_P, T_P, Z_P \rangle$, and $\langle Y_Q - X_Q, Y_Q + X_Q, 2dX_QY_Q, 2 \rangle$, projective coordinates of $P, Q \in E(\mathbb{F}_p)$.

Output: $\langle X_{P+Q}, Y_{P+Q}, T_{P+Q}, Z_{P+Q} \rangle$, projective coordinates of $P + Q$.

- 1: $\langle Y_P, X_P, 0, 0 \rangle \leftarrow \text{PERM}(\langle X_P, Y_P, T_P, Z_P \rangle, 0\mathbf{x}44)$
 - 2: $\langle Y_P - X_P, Y_P + X_P, T_P, Z_P \rangle \leftarrow \langle Y_P, X_P, 0, 0 \rangle \pm \langle X_P, Y_P, T_P, Z_P \rangle$
 - 3: $\langle A, B, C, D \rangle \leftarrow \langle Y_P - X_P, Y_P + X_P, T_P, Z_P \rangle \times \langle Y_Q - X_Q, Y_Q + X_Q, 2dX_QY_Q, 2 \rangle$
 - 4: $\langle B, D, B, D \rangle \leftarrow \text{PERM}(\langle A, B, C, D \rangle, 0\mathbf{x}DD)$
 - 5: $\langle A, C, A, C \rangle \leftarrow \text{PERM}(\langle A, B, C, D \rangle, 0\mathbf{x}88)$
 - 6: $\langle B - A, D + C, B + A, D - C \rangle \leftarrow \langle B, D, B, D \rangle \pm \langle A, C, A, C \rangle$
 - 7: $\langle E, G, H, F \rangle \leftarrow \langle B - A, D + C, B + A, D - C \rangle$
 - 8: $\langle F, H, E, G \rangle \leftarrow \text{PERM}(\langle E, G, H, F \rangle, 0\mathbf{x}4B)$
 - 9: $\langle X_{P+Q}, Y_{P+Q}, T_{P+Q}, Z_{P+Q} \rangle \leftarrow \langle E, G, H, F \rangle \times \langle F, H, E, G \rangle$
 - 10: **return** $\langle X_{P+Q}, Y_{P+Q}, T_{P+Q}, Z_{P+Q} \rangle$
-

parallel field multiplication (cf. Tables 3.4.20 and 3.6.17). One reason that explains this difference is due to the use of high-latency permutation instructions.

Our implementation attained larger improvements on the calculation of four-way point additions. The timings listed in the last row of Table 4.5.12 show better acceleration factors than calculating one point addition using, for example, four-way operations. In addition, these factors are closer to the acceleration factors obtained for parallel prime field multiplications. We attribute some overheads to the execution of larger codes, which do not fit into the instruction decoding cache of the processor.

4.5.3 Fixed-Point Multiplication

We describe implementation techniques for accelerating the execution of fixed-point multiplications. To do that, we revisited a technique of Bernstein et al. [31] and based on it, we derived a pair of optimizations targeting the use of vector SIMD units.

Before describing our optimizations, we show the original technique from [31] for calculating kP . Let P be a fixed point of order r , then define $n = |r| + 1$ and $t = \lceil n/d \rceil$ for some integer $d > 0$. These parameters are used to precompute a set of t look-up tables, each one containing 2^{d-1} points, defined as

$$T_u = \{T_u(v) = 2^{du}vP \mid \text{for } 1 < v \leq 2^{d-1}\} \text{ for } 0 \leq u < t. \quad (4.5.13)$$

Given a pair (u, v) , such that $0 \leq u < t$ and $-2^{d-1} \leq v \leq 2^{d-1}$, the point obtained by querying a look-up table is defined as

$$\phi(T_u, v) = \begin{cases} T_u(v), & \text{if } v > 0, \\ -T_u(-v), & \text{if } v < 0, \\ \mathcal{O}, & \text{otherwise.} \end{cases} \quad (4.5.14)$$

We remark that this query must be performed in constant-time and avoiding access to memory using secret indexes.

The calculation of kP relies on these parameters and proceeds as follows. First, the scalar k must be converted to an expansion of signed digits (k_0, \dots, k_{t-1}) , such that $k = \sum_{i=0}^{t-1} 2^{di} k_i$ and $-2^{d-1} \leq k_i < 2^{d-1}$. Note that to prevent against timing attacks, this conversion must be processed in constant time; for example, following Algorithm 4.5.16. Once the scalar has been converted, kP is calculated as follows

$$kP = \sum_{i=0}^{t-1} \phi(T_i, k_i). \quad (4.5.15)$$

This operation requires the evaluation of $t - 1$ point additions and t queries to the table. Note that the choice of d introduces a trade-off between the memory footprint for storing look-up tables and the computational cost of the scalar multiplication.

Algorithm 4.5.16 Conversion of Integers to Signed Digits.

Input: k is a positive integer, and d is an integer $d > 0$.

Output: (k_0, \dots, k_{t-1}) , such that $k = \sum_{i=0}^{t-1} 2^{di} k_i$ and $-2^{d-1} \leq k_i < 2^{d-1}$.

```

1:  $t \leftarrow \lceil \frac{n}{d} \rceil$ 
2: for  $i \leftarrow 0$  to  $t - 1$  do
3:    $s \leftarrow k \bmod 2^d$ 
4:    $c \leftarrow \lfloor s/2^{d-1} \rfloor$ 
5:    $k_i \leftarrow (s \wedge \neg 2^{d-1}) - (s \wedge 2^{d-1})$             $//k_i$  is the two's complement of  $s$ .
6:    $k \leftarrow \lfloor k/2^d \rfloor + c$ 
7: end for
8: return  $(k_0, \dots, k_{t-1})$ 

```

In [31, 64], it was shown how to reduce by half the number of look-up tables. This is achieved by fixing the same digit size d and precomputing only the even-indexed look-up tables, i.e. all T_u for u even. Then, kP is calculated as

$$kP = \sum_{i=0}^{\lceil t/2 \rceil - 1} \phi(T_{2i}, k_{2i}) + 2^d \sum_{i=0}^{\lfloor t/2 \rfloor - 1} \phi(T_{2i}, k_{2i+1}). \quad (4.5.17)$$

This calculation requires $t - 1$ point additions and d point doublings. Using this strategy, both queries $\phi(T_{2i}, k_{2i})$ and $\phi(T_{2i}, k_{2i+1})$ look up the same table T_{2i} .

Now, we will describe two modifications applied to the previous technique for computing four point additions simultaneously. Our description uses the parameters of the edwards25519 and edwards448 curves.

Fixed-Point Multiplication for Edwards25519

First, we modified the previous technique in such a way that a series of four point additions can be computed simultaneously leading to a direct application of four-way point additions. Second, by extending the previous technique, we were able to reduce by half the number of look-up tables; thus, our implementation has smaller memory footprint.

Our first optimization is for speeding up the calculation of kP . For `edwards25519`, $n = 254$; thus, we selected the digit size to be $d = 4$ and $t = 64$. Taking advantage of the associative property of point addition, we split the workload of Equation (4.5.17) in four independent sums that can be processed in parallel as

$$kP = \sum_{i=0}^{15} \phi(T_{2i}, k_{2i}) + 2^4 \sum_{i=0}^{15} \phi(T_{2i}, k_{2i+1}) + \sum_{j=16}^{31} \phi(T_{2j}, k_{2j}) + 2^4 \sum_{j=16}^{31} \phi(T_{2j}, k_{2j+1}). \quad (4.5.18)$$

Hence, this algorithm can be computed using fifteen four-way point additions and sixteen queries; after that, the partial sums are combined using four point doublings and three point additions. An advantage of this strategy is that no permutation instructions are required for calculating point additions since values in a lane of a vector register are independent of values from other lanes.

In Table 4.5.19, we show the operation counts and memory requirements of the strategies presented above. Unlike the original strategy in [31], our approach allows the calculation of a series of four independent additions, incurring in an overhead of three point additions (cf. the second and third row of Table 4.5.19). Our strategy for fixed-point multiplication, given in Equation (4.5.18), is faster whenever the calculation of four-way point additions is faster than a two-way parallel execution.

Table 4.5.19: Operation counts of fixed-point multiplication on the `edwards25519` curve.

Method	Operations	Points	Storage	Reference
Sequential	63 PA	512	48 KB	Equation (4.5.15)
Two-way	31 PA ₂ + 4 PD	256	24 KB	Equation (4.5.17) ([31, 64])
Four-way	15 PA ₄ + 3 PA + 4 PD	256	24 KB	Equation (4.5.18) (this work)
Four-way	15 PA ₄ + 3 PA + 12 PD	128	12 KB	Equation (4.5.20) (this work)

Our second optimization improves on the memory footprint. We note that the number of tables can be reduced by half and use 16 rather than 32 tables. To this end, we fix $d = 4$ and $t = 64$ as before, but we store only the tables T_u such that $u \equiv 0 \pmod{4}$ for $0 \leq u < 64$. Thus, we calculate kP using also four independent sums as follows

$$kP = \sum_{i=0}^{15} \phi(T_{4i}, k_{4i}) + 2^4 \sum_{i=0}^{15} \phi(T_{4i}, k_{4i+1}) + 2^8 \sum_{i=0}^{15} \phi(T_{4i}, k_{4i+2}) + 2^{12} \sum_{i=0}^{15} \phi(T_{4i}, k_{4i+3}). \quad (4.5.20)$$

In this formulation, one query to a look-up table T_{4i} can feed four points according to the digits k_{4i} , k_{4i+1} , k_{4i+2} , and k_{4i+3} . This is particularly important since memory access tends to be slow when fetching several vector registers.

As Table 4.5.19 shows, Equation (4.5.20) requires fifteen four-way point additions, twelve point doublings, and three point additions. Although this method requires more operations than the previous optimization given in Equation (4.5.18), we achieve a significant reduction on memory footprint halving the size of tables used by previous methods.

Fixed-Point Multiplication for Edwards448

For this curve, $n = 447$; thus, setting the digit size $d = 4$ leads to $t = 112$ look-up tables of eight points. Using similar observations as for edwards25519 curve, we reduce to 28 the number of tables for edwards448. Thus, we store only T_u such that $u \equiv 0 \pmod{4}$ for $0 \leq u < 112$. Analogously to Equation (4.5.20), kP is performed as

$$kP = \sum_{i=0}^{27} \phi(T_{4i}, k_{4i}) + 2^4 \sum_{i=0}^{27} \phi(T_{4i}, k_{4i+1}) + 2^8 \sum_{i=0}^{27} \phi(T_{4i}, k_{4i+2}) + 2^{12} \sum_{i=0}^{27} \phi(T_{4i}, k_{4i+3}). \quad (4.5.21)$$

This equation requires 27 four-way point additions, 12 point doublings, and 3 point additions. Moreover, we could go further and use only 14 tables by using the tables T_u such that $u \equiv 0 \pmod{8}$; this calculation also requires 27 four-way point additions. However, adding eight partial sums takes 28 point doublings and 7 point additions, which introduces an overhead that increases the execution time of fixed-point multiplication.

4.6 Chapter Summary

The properties of elliptic curves make them an efficient choice for instantiating algorithms based on groups with hard discrete logarithm problem. Motivated by the use of alternative elliptic curve models and some recent advances, we investigated efficient ways to calculate elliptic curve operations.

Regarding algorithmic optimizations, we found better algorithms for performing arithmetic over Montgomery curves. In particular, we showed a new three-point ladder algorithm, some improvements on fixed-point multiplications, and an optimized formula for point tripling.

Under the notion of n -way prime field operations, we designed parallel algorithms for point addition formulas of Weierstrass, Montgomery, and twisted Edwards curves. These parallel algorithms can be implemented either in hardware or software. In our study, we use SIMD instructions for their implementation. The design of the parallel algorithms considered the capabilities and limitations of the vector unit studied.

All of these optimizations are of general interest. However, they become relevant when they are applied to cryptographic algorithms, which is the subject of the next chapter.

Chapter 5

Cryptographic Algorithms and Protocols

We have shown how to implement operations over prime fields and elliptic curves. Now, it is time to use these operations as building blocks for implementing cryptographic algorithms and protocols. At this stage, the need for secure software development becomes more evident since algorithms process secret data.

In this chapter, we describe implementation details of ECDH and ECDSA using the P-384 curve; the X25519, X448, and SIDH-751 Diffie-Hellman protocols; and the Ed25519, Ed448, qDSA, XMSS, and XMSS^{MT} digital signature schemes. Also, we show implementation details of the SHA-256 hash function using SHA-NI instructions.

For each of them, we propose a set of implementation techniques that improve their execution time. We demonstrate these techniques through optimized software implementations and report the results of performance benchmarks. Finally, we compare our implementations with other state-of-the-art implementations.

5.1 Implementation of ECDH and ECDSA with P-384

Transitioning to higher security levels is commonly accompanied with overheads. Although many cryptographic libraries support prime curves, most of them do not have optimized code for the P-384 curve, which is a standardized elliptic curve providing a security level of 192 bits.

In this section, we present implementation techniques for accelerating elliptic curve operations on P-384. We apply the vectorized implementation of prime field arithmetic to formulate a parallel scheduling of the complete formulas for point addition. The results described in this section were published in the SBSeg 2016 paper [100] [P](#).

5.1.1 Review of Standard Elliptic Curves

In 1999, the National Institute of Standards and Technology (NIST) [200] recommended a set of fifteen elliptic curves, five of them are defined over prime fields and are known as P-192, P-224, P-256, P-384, and P-521 covering, respectively, security levels of 80, 112, 128, 192, and 256 bits.

The Relevance of P-384

In 2016, the National Security Agency [207] recommended the use of stronger cryptographic algorithms before transitioning to quantum-resistant algorithms. The Commercial National Security Algorithms (CNSA) Suite includes algorithms classified in the TOP SECRET level of Suite B as shown in Table 5.1.1.

Table 5.1.1: Suite B of cryptographic algorithms.

Primitive	Algorithm	SECRET Level	TOP SECRET Level
Data Encryption	AES	AES-128	AES-256
Hash Function	SHA2	SHA-256	SHA-384
Key Agreement	ECDH	P-256	P-384
	DH ¹	2,048-bit modulus	3,072-bit modulus
Digital Signature	ECDSA	P-256	P-384
	DSA ¹	2,048-bit modulus	3,072-bit modulus
	RSA ¹	2,048-bit modulus	3,072-bit modulus

¹ Algorithms considered for legacy support.

In computational systems depending on a cryptographic hardware infrastructure, upgrading to the TOP SECRET level could incur into a serious investment for updating the whole infrastructure in cases when a higher security level is not supported. On the other hand, in systems using a cryptographic software infrastructure, this upgrade can be performed by tuning the appropriate parameters in the software library. Unfortunately, a loss of performance can occur because some cryptographic libraries have generic, non-optimized implementations for the higher security levels. Consequently, transitioning to the TOP SECRET level enhances security but downgrades on performance.

Parameters of P-384

Let $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$ be a prime number, the Weierstrass curve P-384 is defined over $\mathbb{F}_{p_{384}}$ as

$$\text{P-384: } y^2 = x^3 - 3x + B, \quad (5.1.2)$$

where $B = 27580193559959705877849011840389048093056905856361568521428707301988689241309860865136260764883745107765439761230575$. This curve has prime order $r = 2^{384} - 1388124618062372383947042015309946732620727252194336364173$.

This curve can be used to instantiate the Elliptic Curve Diffie-Hellman protocol (ECDH) [9] and the Elliptic Curve Digital Signature Algorithm (ECDSA) [8, 265], which we briefly describe below.

Protocol 5.1.3 (ECDH).

- **Setup.** Let λ be the security parameter. Select a prime p such that $\log_2(p)/2 \approx \lambda$. Select a Weierstrass curve E such that $E(\mathbb{F}_p)$ is a group of prime order r . Fix a generator $G \in (\mathbb{F}_p)$.

- **Key Generation.** Alice selects $k_A \leftarrow \mathbb{Z}/r\mathbb{Z}$ uniformly at random, calculates $P_A \leftarrow k_A G$ and sends this value to Bob. Similarly to Alice, Bob generates k_B and sends P_B to Alice.
- **Shared Secret.** Once Bob receives P_A from Alice, he calculates $k_B P_A$ and this point is the shared secret. Analogously, Alice calculates $k_A P_B$, which is an equivalent point to the one obtained by Bob.

Signature Scheme 5.1.4 (ECDSA).

- **Setup.** Similar setup as in ECDH, and choose a cryptographic hash function H producing $2|p|$ bits.
- **Key Generation.** Select $k \leftarrow \mathbb{Z}/r\mathbb{Z}$ uniformly at random, and calculate $P \leftarrow kG$. Set k as the private key and P as the public key.
- **Signing.** Given a message $M \in \{0, 1\}^*$, and a secret key k . Choose $k_1 \leftarrow \mathbb{Z}/r\mathbb{Z}$ uniformly at random and calculate $(x_R, y_R) \leftarrow k_1 G$ and $x \leftarrow x_R \bmod r$. If $x = 0$ then sample another k_1 . Set $s \leftarrow k_1^{-1}(H(M) + k \cdot x) \bmod r$. If $s = 0$ then sample another k_1 . Declare (x, s) as the signature of M .
- **Verification.** Given a message $M \in \{0, 1\}^*$, a signature (x, s) , and a public key P ; set $u_1 \leftarrow s^{-1} \cdot H(M) \bmod r$ and $u_2 \leftarrow s^{-1} \cdot x \bmod r$. Calculate $(x_Q, y_Q) \leftarrow u_1 G + u_2 P$. Accept the signature if $x = x_Q \bmod r$; otherwise, reject it.

Related Works

Some works in the literature have implemented optimizations targeting specific NIST curves. For instance, Käsper [169] presented a fast implementation of the P-224 curve using a redundant representation for implementing the prime field arithmetic. Gueron and Krasnov [132] showed optimizations for P-256 through a more efficient calculation of prime field multiplication leveraging the use of Montgomery-friendly primes. Granger and Scott [121] described a novel technique for accelerating multiplications on $\mathbb{F}_{2^{521-1}}$, which they used for the implementation of P-521 curve operations; this latter technique can also be extended to pseudo-Mersenne primes.

5.1.2 Implementation Details

We focus on a high-performance software implementation of P-384 curve operations. Our optimizations target three levels. In a lower level, we showed in Section 3.5 how to perform prime field arithmetic in parallel using the AVX2 vector unit. In an upper level, we devised in Section 4.3.2 a scheduling of prime field operations to execute complete formulas for point addition in parallel. Finally, at a higher level, we implemented algorithms suitable for three special cases of scalar multiplication: the variable-point, fixed-point, and double-point multiplications, which are the core operations of ECDH and ECDSA.

Variable-Point Multiplication

For variable-point multiplication, we implement a regular execution algorithm described by Bos et al. [47, Alg. 1]. For P-384, we found $\omega = 6$ is the optimal value for the window value, which implies to calculate a series of five point doublings followed by one point addition. The scalar must be converted to a signed digit expansion, however, the conversion method works only for odd numbers. We convert either k (when is odd) or $r - k$ (when is even), and at the end of the algorithm the point must be inverted according to the parity of the scalar.

Note that conditionally choosing between two values must be performed in constant time to avoid timing attacks. In order to prevent against cache attacks, we read the entire precomputed table and conditionally select the appropriate entry. We implement conditional selection using the CMOV instruction.

Fixed-Point Multiplication

Calculating fixed-point multiplications admits precomputing tables of points. Large tables accelerate calculations, but also increase the memory used. Since the multiplication algorithm must read the entire table, it is desirable that a large part of the table fits on the L1-Data memory cache, which has 32 KB of capacity on the Core i7 processors.

We calculate a fix-point multiplication kP as follows. First, we precompute a table with entries

$$T_i = \{j2^{4i}P : \text{even } i \in [0, 96) \text{ and } 0 \leq j \leq 8\}. \quad (5.1.5)$$

A constant-time query to this table is denoted as $\phi(T_i, j) = \text{sgn}(j)|j|2^{4i}P$ for $0 \leq j \leq 8$. At running time, we split k into 96 signed digits of four bits such that $k = \sum_{i=0}^{95} k_i 2^{4i}$. Then, we calculate $kP = Q_0 + 2^4 Q_1$, where $Q_0 = \sum_{i=0}^{95} \phi(T_i, k_i)$ for $i \equiv 0 \pmod{2}$, and $Q_1 = \sum_{i=0}^{95} \phi(T_{i-1}, k_i)$ for $i \equiv 1 \pmod{2}$.

In total, this method requires 96 point additions and 4 point doublings. Regarding memory footprint, the table stores 384 points in affine coordinates accounting for 36 KB of read-only memory, which is close to the size of L1-Data memory cache.

Double-Point Multiplication

The operation $k_0P + k_1Q$ is used in the verification of ECDSA signatures. It is calculated using faster algorithms and does not require protection against side-channel attacks because no private data is computed. To do so, we implement the interleaved algorithm with ω -NAF representation as described in Algorithm 4.2.16. The values for ω are chosen independently per scalar. In our implementation, we experimentally found that $\omega_P = 7$ and $\omega_Q = 5$ are the values that minimize the running time of double-point multiplication.

5.1.3 Performance Benchmark and Comparison

To evaluate the impact on the performance of our optimizations, we measure the running time of ECDH shared secrets and the ECDSA signature operations using P-384. To enable

comparisons, we also measure the performance of some publicly-available cryptographic libraries. We choose libraries written in C/C++ supporting the P-384 curve.

- OpenSSL (v1.0.2h) is an open-source library that is in continuous development containing a vast number of optimizations [263].
- Nettle (v3.2) is a low-level cryptographic library that provides a back-end functionality to the GnuTLS library [208].
- Previously known as PolarSSL, mbed TLS (v2.2.1) is a multi-platform library with the aim to ease the development of cryptographic algorithms [20].
- Relic toolkit (v0.4.1) is a modern library that supports a broad range of cryptographic algorithms. Measurements were performed using the default configurations, and setting GMP as the arithmetic backend [14].
- BoringSSL (commit `f447ba2f`) is a fork of the OpenSSL library modified to support strict security requirements [116].

A comparison of the performance of these libraries is shown in Table 5.1.6.

Table 5.1.6: Timings of ECDH and ECDSA using P-384.

Software Library	Haswell			Skylake		
	ECDH Shared Secret	ECDSA Signing	ECDSA Verify	ECDH Shared Secret	ECDSA Signing	ECDSA Verify
mbed TLS	18.70	7.15	26.49	17.90	6.72	25.25
BoringSSL	3.60	3.78	4.47	3.43	3.67	4.33
Relic	1.79	0.89	2.40	1.52	0.77	2.06
Nettle	–	0.77	2.07	–	0.62	1.57
OpenSSL	2.12	0.65	2.60	2.03	0.62	2.49
This work	1.25	0.56	1.31	1.11	0.53	1.11

¹ Entries are 10^6 clock cycles.

² All libraries were compiled using the GNU C Compiler v5.3.1.

From the measurements obtained, one can see the cryptographic libraries tested, with the exception of mbed TLS, offer a similar performance for P-384. Note that none of these libraries uses vector instructions nor implements the complete formulas. This means that they use the faster incomplete formulas point addition. In all cases, our implementation using the complete formulas outperforms to the other libraries.

Renes et al. [230] measured the effects of using complete formulas for point additions in OpenSSL. As a result, they observed a slow-down factor of $1.41\times$ in ECDH. In comparison to incomplete formulas, the complete formulas have a significant increase on the number of field additions. Our implementation calculates field additions $3\times$ faster than the OpenSSL library; however, additions are not the bottleneck of the formulas, field multiplications still dominate the execution time. The total speedup we obtained comes from a combination

of using a redundant representation for prime field elements and implementing operations using AVX2 vector instructions. So we reduce the overheads of using complete formulas.

Our implementation benefits more of running in Skylake than the other libraries. For example, the vectorized implementation running on Skylake calculates shared secrets 10 % faster than running in Haswell. Other libraries only get a 5 % of improvement. An explanation for that is due to Skylake has more execution ports for vector instructions than Haswell. If this trend continues in the upcoming micro-architectures, vectorized code will render better performance.

5.2 Implementation of X25519 and X448

The cornerstone Miller’s paper [194] shows how to instantiate a Diffie-Hellman protocol using the group of points of an elliptic curve. Miller observes that the protocol is still functional using only the x -coordinate of points. Montgomery [196] presented a parametrization of curves that allows implementing elliptic curve operations faster than using operations over Weierstrass curves. Later, Bernstein [23] introduced an efficient realization of the Diffie-Hellman protocol using a Montgomery curve called Curve25519.

In this section, we give details on the implementation of X25519 and X448, two instances of the Diffie-Hellman protocol using Montgomery curves called Curve25519 and Curve448, respectively. We leverage the use of the most recent instruction sets for accelerating the running time of these protocols.

The contributions related to the AVX2 parallel implementation were published in the LATINCRYPT paper [99] [\(P\)](#). Building on top of that, we show more optimizations on the usage of AVX2 and these results were published in the TOMS 2019 journal paper [101] [\(P\)](#), which was co-authored with Ricardo Dahab at the University of Campinas.

We also present implementation techniques targeting a 64-bit implementation. These results and a performance benchmark were published in the SAC 2017 paper [213] [\(P\)](#), which was co-authored with Thomaz Oliveira and Francisco Rodríguez-Henríquez at the CINVESTAV-IPN, and Hüseyin Hişil at the Yasar University.

5.2.1 Review of Diffie-Hellman Protocol on the x -Line

The following description shows the parameters and operations required by the Diffie-Hellman protocol.

Protocol 5.2.1 (Diffie-Hellman on the x -Line of a Montgomery Curve).

- **Setup.** Given an integer λ denoting the security parameter, Alice and Bob must agree on some public parameters. Fix a Montgomery curve $E/\mathbb{F}_p: By^2 = x^3 + Ax^2 + x$, where p is a prime number such that $\log_2(p)/2 \approx \lambda$ and $E(\mathbb{F}_p)$ has order hr , where r is a large prime close to p , and the cofactor is $h = 2^c$ for small $c \geq 2$. Fix a point $G \in E(\mathbb{F}_p)$ of order r . A complete setup must consider a secure instance of an elliptic curve. We refer to SafeCurves project [36] for a comprehensive list of security criteria.

- **Key Generation.** Alice chooses $a \in \mathbb{Z}/r\mathbb{Z}$ uniformly at random, calculates $K_a = \mathbf{x}(aG)$, and transmits an encoding of K_a to Bob. Similarly, Bob selects $b \in \mathbb{Z}/r\mathbb{Z}$ uniformly at random, calculates $K_b = \mathbf{x}(bG)$, and transmits an encoding of K_b to Alice.
- **Shared Secret.** Once the points are received, Alice calculates aK_b and Bob calculates bK_a . After that, both entities have $bK_a = aK_b = \mathbf{x}(abG)$ as a shared secret.

In this protocol, only the x -coordinate of points is sent through the communication channel. It is possible to use point arithmetic on the x -line of a Montgomery curve without recovering the other coordinate. The dominant operation is the scalar multiplication, which must be processed in a secure way because the scalar is a secret value. One approach is to use the Montgomery ladder algorithm for multiplying points on the x -line, and thus, to achieve a regular execution of the multiplication.

Instances

The high-level description given above does not consider several criteria for choosing a secure instance of an elliptic curve, and together with its security, efficiency plays also important role. In 2006, Bernstein [23] introduced Curve25519, a Montgomery curve that is defined following a rigid security criteria. The design choices for this instance take into account also the efficiency of operations.

The Curve25519 is a Montgomery curve over $\mathbb{F}_{p_{25519}}$, where $p_{25519} = 2^{255} - 19$ is a prime number, defined by

$$\text{Curve25519: } y^2 = x^3 + 486662x^2 + x, \quad (5.2.2)$$

and its order is $8r$, where $r = 2^{252} + 27742317777372353535851937790883648493$. The point $(9, 14781619447589544791020593568409986887264606134616475288964881837755586237401)$ generates a subgroup of order r .

Due to the main purpose of this curve was to instantiate an efficient Diffie-Hellman protocol, the protocol instance was also referred as Curve25519. However, this clash of names caused ambiguities between the curve and the protocol. For this reason, the Diffie-Hellman protocol using Curve25519 is called as X25519.

Hamburg [139] presented instances of Montgomery curves for accelerating the execution of elliptic curve operations at a 128-bit security level. Hamburg proposes the use of special primes that aids the calculation of reductions modulo p . A follow up work derived on the proposal of elliptic curve parameters for higher security levels. In particular, using the prime $p_{448} = 2^{448} - 2^{224} - 1$ allows to instantiate elliptic curves at 224-bit security level. Hamburg's parametrization derived on an elliptic curve called Goldilocks [140, 141]. This curve is an Edwards curve over $\mathbb{F}_{p_{448}}$ defined by

$$\text{Goldilocks: } x^2 + y^2 = 1 - 39081x^2y^2. \quad (5.2.3)$$

Its order is $4r$, where $r = 2^{446} - 13818066809895115352007386748515426880336692474882178609894547503885$ is a prime number. In other contexts, this curve is also known

as the edwards448 curve, and is isogeneous to a Montgomery curve over the same field defined by

$$\text{Curve448: } y^2 = x^3 + 156326x^2 + x. \quad (5.2.4)$$

This Montgomery curve is the one used for instantiating the X448 Diffie-Hellman protocol.

Related Works

Several works have showed optimized implementations targeting different techniques and instruction sets. We now briefly overview milestone implementations of X25519.

In addition to the introduction of Curve25519, Bernstein [23] gives details of its implementation recommending a reduced-radix arithmetic and the use of the SSE vector unit. This vector unit extends the instruction set architecture with parallel execution of floating-point instructions, an efficient resource available on processors at that time.

Langley [177] followed a similar strategy for prime field arithmetic. His implementation uses a redundant representation with $\rho = 51$ and performs operations with native 64-bit integer instructions. Later, Moon [197] published implementations of X25519 using two machine representations of prime field elements. One of them is based on Langley’s code, and the other one sets the digit size to $\rho = 25.5$, but employs the SSE2 vector unit for performing integer arithmetic operations. At the time of their release, Moon’s implementations had a similar performance in former micro-architectures (more precisely, the vectorized code was slightly slower, according to the timings reported in [197]). However, at the present time, we observed a significant performance gap between these implementations. For the platforms we used in our benchmarks, the SSE2-based code is 30 % slower than the 64-bit implementation. This gap makes evident the enhancements made to the native 64-bit multiplication instructions on the latest processors.

The release of the AVX instruction set [71], firstly introduced in the Sandy Bridge micro-architecture, modified the execution of 128-bit vector instructions. Specifically, the latency of some instructions was reduced, and all vector instructions were promoted to use three-operand codes. These micro-architectural improvements were leveraged by Chou [64], who present an implementation that processes prime field arithmetic using 128-bit AVX vector instructions. As an immediate consequence, his implementation outperformed native 64-bit implementations on Sandy Bridge.

5.2.2 Implementation Details

Micro-architectures superseding the capabilities of Sandy Bridge include more powerful instruction sets. For example, the succeeding processors have a AVX2 vector unit and also support the new 64-bit integer arithmetic instructions [217]. On the presence of these computational resources, we developed two implementations of X25519 and X448.

Vectorized Arithmetic for Montgomery Ladder

In a first stage, we look for implementation techniques that leverage the use of AVX2. In an early exploration paper [98] (P), we showed how to store field elements into 256-bit vector registers and proposed the use AVX2 instructions for performing arithmetic

operations. As part of our first experiments, we noticed that some simple tasks, such as moving words within a vector register, have a higher latency in the AVX2 unit.

Later, we focused on designing a vector implementation of prime field arithmetic that be useful for executing X25519. As a result, we employ the AVX2 vector unit to deliver SIMD parallel processing at two levels. At the higher level, we showed how to schedule the operations of the Montgomery ladder step to be executed in two parallel units (as shown in Section 4.4.3). Hence, we use the 256-bit AVX2 vector unit as two independent 128-bit units dedicated for executing the ladder step in parallel. At the lower level, the internal operations of prime field arithmetic are also executed in parallel by each 128-bit vector unit. Following this approach, we developed a vectorized implementation of X25519.

As part of our continuous research, we identified design choices that led to a better utilization of AVX2. Departing from our previous report [99], we introduced two modifications. First, we modified the way the digits are stored in vector registers. The new distribution of digits allows that independent propagation of carry bits be performed in parallel, which reduces the execution time of digit size reduction.

The second modification was motivated by the micro-architectural improvements of Skylake. Unlike Haswell, Skylake has two units for calculating vector multiplications, which reduces the inverse throughput of vector multiplications. This means that more multiplications can be executed by unit of time provided that they are independent. With this in mind, we adapted the Mastrovito multiplier to the case of prime field multiplications in order to schedule more vector multiplications independently. These two modifications improved our previous AVX2 vector implementation and allowed to speed up the performance of both X25519 and X448.

Implementation using 64-bit Instructions

In a second stage, we focused on a succinct implementation of X25519 and X448 using non-vector 64-bit instructions. To do that, we employed the new instructions for integer arithmetic [217]. It is well-known that the combination of `MULX`, from the BMI2 instruction set, and `ADX` instructions improves the execution of operand-scanning technique for multiplying integers. Sections 3.4 and 3.6 describe the implementation of field arithmetic for X25519 and X448, respectively.

We implement the ladder step in a similar way than the two-way version of the ladder algorithm, but without the use of vector instructions. Instead, we blend the scheduling of two field multiplications (or squares). This approach helps to save some cycles due to improves the register assignment and stack usage.

Methods for Improving Key Generation

Speeding up the key generation phase is as relevant as the shared secret phase, since it is more frequent the enforcement of ephemeral key agreements with the aim to provide perfect forward secrecy. For this reason, we developed two approaches for performing key generation faster.

One approach relies on using the right-to-left algorithms for fixed-point scalar multiplication. As was described in Section 4.4.4, the use of precomputation in the right-to-left

ladder was firstly suggested by Oliveira et al. [211], who improved the timings of fixed-point multiplication on binary curves. Furthermore, a similar technique also works for scalar multiplications on the x -line of a Montgomery curve as described in Section 4.4.4.

A better approach is to offload calculations leveraging the fast point arithmetic of twisted Edwards curves. Note that unlike Montgomery curves, the twisted Edwards curves have complete formulas, which allows designing better algorithms for fixed-point multiplications. For this reason, we can use the rational maps that relate Montgomery and twisted Edwards curves for moving calculations to a more efficient curve.

Let $\Phi: E_M \rightarrow E_{TE}$ and $\Psi: E_{TE} \rightarrow E_M$ be such rational mappings, and G_M be the generator point on a Montgomery curve. To calculate $\mathbf{x}(kG_M)$, the point G_M is mapped to a point on a twisted Edwards curve, say $G_{TE} = \Phi(G_M)$. Then, we perform the fixed-point multiplication kG_{TE} and map back the resulting point to obtain $\Psi(kG_{TE})$, which is on the Montgomery curve; finally, we take its x -coordinate, which holds the following relation $\mathbf{x}(\Psi(kG_{TE})) = \mathbf{x}(kG_M)$.

Although this technique is well-known, we show that the implementation of parallel algorithms for twisted Edwards arithmetic, described in Section 4.5.3, resulted also in improvements for calculating the key generation phase of the Diffie-Hellman protocol.

5.2.3 Performance Benchmark and Comparison

The timings reported in this section are product of a performance benchmarking on Haswell and Skylake micro-architectures. We present measurements of the latency of the X25519 and X448 protocols.

Timings of X25519

Table 5.2.5 shows the timings of several implementations of X25519. The second column refers to the instruction set architecture (ISA) used by each implementation.

Table 5.2.5: Timings of X25519 shared secret.

Implementation	ISA	Haswell ¹	Skylake ¹
Moon [197]	SSE2	237.6	196.2
Moon [197]	x64	166.3	140.0
This work [99]	AVX2	156.5	137.8
Chou [64]	AVX	155.9	137.2
This work [213]	x64, BMI2, ADX	144.3	111.2
This work [101]	AVX2	121.0	99.4
This work [101]	AVX2, AVX-512	—	87.3 ²
This work [101]	AVX-512	—	81.6 ²

¹ Entries are 10^3 clock cycles.

² Measured in a Core i7-7820X SkylakeX processor with support for AVX-512.

The timings listed in the table show a strong relation between the improvements on the execution time of X25519 and the use of a more powerful instruction set. For

example, Moon’s implementations maintained the speed record for long time, but when AVX become available Chou’s implementation broke the record. We closely reached such a record in our first AVX2 implementation (see the third row), since the timings of that implementation were competitive with state-of-the-art implementations.

Our continued efforts on improving field arithmetic resulted in a faster vector implementations using AVX2 and AVX-512 (see the rows at the bottom). Some cycles can be saved by compiling our AVX2 implementation enabling the use of AVX-512 instructions. Our pure AVX-512 implementation offers extra savings on the calculation of X25519.

We set a speed record by achieving the execution of X25519 shared secrets below the 100,000 clock cycles barrier running on a Skylake processor. Other works have accomplished similar speed records in former architectures using other models of elliptic curves; for example, using GLV/GLS prime curves [97], FourQ curve [77], and Koblitz curves [13,215]. The timings of our vector implementations show that the data structures and the instructions used are important factors that impact on performance.

Our 64-bit implementation of X25519 does not rely on parallel computing, but also has good performance. This is explained, in large part, because recent processors have improved the execution of multi-precision arithmetic through the use of dedicated instructions such as the BMI2 and ADX sets. Also, it must be noted that the 64-bit instructions have shorter binary codes, and the decoding engine of the processor can decode them faster. On the other hand, the binary codes of vector instructions are larger and they can only be decoded by a more complex decoding unit.

Timings of X448

A fewer number of works have reported efficient software implementations of X448. The main reference is Hamburg’s software library [140], which employs the new 64-bit integer instructions for performing reduced-radix multiplications over $\mathbb{F}_{p_{448}}$. Table 5.2.6 lists the timings of some implementations of X448.

Table 5.2.6: Timings of X448 shared secret.

Implementation	ISA	Haswell ¹	Skylake ¹
This work [213]	x64, BMI2, ADX	693.0	513.4
Hamburg [140]	x64	626.6	532.0
eBACS ² [35]	x64	532.2	459.7
This work [101]	AVX2	428.1	364.2

¹ Entries are 10^3 clock cycles.

² The eBACS’ timings correspond to the *titan0* and *skylake* machines.

The first row of the table shows that our implementation using 64-bit instructions is faster in Skylake than Hamburg’s code. However, it does not have good performance on Haswell. As noted before, the difference between these implementations relies on the use of ADX instructions. The operand scanning method needs two steps of carry propagation. Without ADX instructions, these two propagations must be performed sequentially, and since the field elements are large, the latency of carry propagation gets affected negatively.

On the other hand, with the presence of ADX instructions, the two carry propagations are blended making the calculation of multiplications faster. This subtle difference explains the performance of our 64-bit implementation.

Regarding our vectorized implementation, it can be observed that it gets a better margin of improvement. Our software is around 19 % faster than implementations published in eBACS [35]. We attribute these savings to the efficiency of our prime field multiplier and the parallel execution of Montgomery ladder step. As the field elements require exactly sixteen words, we were able to implement Karatsuba multiplication using vector instructions. As a result, it is more evident the positive impact on the execution time given by a vectorized implementation with respect to a native 64-bit implementation.

Timings of Key Generation

As described above, we implemented two methods for key generation. In the first approach, we use the right-to-left ladder applied to Montgomery curves; and in the second approach, we performed fixed-point multiplications on the equivalent twisted Edwards curve and mapped back the result to a point on a Montgomery curve.

Table 5.2.7 lists the timings of our implementations of key generation measured on Haswell and Skylake processors. To calculate the percentage of savings, we take as a baseline the fastest timing of shared secrets. We use this baseline because, in practice, it is common that developers use the shared secret function (with the generator point as input) to generate keys.

Table 5.2.7: Timings of the key generation phase of X25519 and X448.

Protocol	Operation	Reference	Storage ¹	Haswell ²	Skylake ²	Savings ³
X25519	Shared Secret	Algorithm 4.4.16	0	121.0	99.4	–
	Key Generation	Algorithm 4.4.21	8	90.7	72.5	27.1 %
		Equation (4.5.20)	12	46.0	37.1	62.7 %
		Equation (4.5.18)	24	43.7	34.5	65.3 %
X448	Shared Secret	Algorithm 4.4.16	0	428.1	364.2	–
	Key Generation	Algorithm 4.4.21	25	401.2	320.7	11.9 %
		Equation (4.5.21)	36.5	129.0	107.7	70.4 %

¹ Entries are kilobytes.

² Entries are 10^3 clock cycles.

³ Savings are calculated with respect to the timing of shared secrets in Skylake.

Using the right-to-left method (Algorithm 4.4.21), we experimentally observed that the key generation phase is 27 % faster than the calculation of shared secrets for X25519. In the case of X448, this method is 11 % faster, which is explained because the execution time of the shared secret function was also reduced by the two-way parallel implementation. Regarding memory footprint, the precomputed tables consists of a field element per bit of the key, resulting in tables of 8 KB and 25 KB for X25519 and X448, respectively. This

optimization saves only the calculation of point doublings in the ladder step, unfortunately the table’s size can not be extended to save more operations.

Moving the calculations to the twisted Edwards curve accelerates key generation significantly. Its execution time reduces 65-70% at the price of larger precomputed tables. It is clear that our efforts on optimizing fixed-point multiplications for twisted Edwards curves have a positive effect beyond the scope of digital signatures.

Timings of Diffie-Hellman Protocol based on Elliptic Curves

In Table 5.2.8 we summarize the timings of several implementations of the Diffie-Hellman protocol using different models of elliptic curves. The total time of the Diffie-Hellman protocol is sometimes estimated as twice the time of shared secret calculation. However, from the timings in the table, it can be seen that accelerating the key generation phase is also a valuable optimization.

Table 5.2.8: Timings of the Diffie-Hellman protocol at the 128-bit security level.

Elliptic Curve	Field	Key Gen. ¹	Shared Secret ¹	Total ¹
FourQ [77]	$\mathbb{F}_{(2^{127}-1)^2}$	33.8	67.4	101.2
Koblitz (3τ -NAF) [215]	$\mathbb{F}_{4^{149}}$	69.0	69.0	138.0
Curve25519 (This work)	$\mathbb{F}_{2^{255}-19}$	43.7	121.0	164.7
KL25519(82, 77) [167]	$\mathbb{F}_{2^{255}-19}$	101.3	137.9	239.2
NIST P-256 [132]	$\mathbb{F}_{2^{256}-2^{224}+2^{192}+2^{96}-1}$	67.0	312.0	379.0

¹ Entries represent 10^3 clock cycles measured on Haswell.

In comparison to standardized elliptic curves, our implementation of X25519 is $2.2\times$ faster than the ECDH algorithm using the P-256 curve on a Haswell micro-architecture. Also, our implementation of X448 is $4.8\times$ and $3.53\times$ faster than the ECDH algorithm using the P-384 and P-521 curves, respectively.

Some works have reported efficient implementations using different curve models. For example, both the FourQ [77] curve and a Koblitz curve defined over $\mathbb{F}_{4^{149}}$ [215] render faster performance than X25519. Besides their optimized code, these developments have in common the use of elliptic curves with efficiently-computable endomorphisms that enable further optimizations. More recently, Karati and Sarkar [167] implemented the Kummer curve KL25519(82, 77) using AVX2 and their implementation takes 137,900 clock cycles. It is interesting to know if there are other curve models that improve the state of the art.

5.3 Implementation of Ed25519 and Ed448

We developed optimized implementations of two instances of the Edwards Digital Signature Algorithm (EdDSA). We show how to apply the parallel execution of operations over prime fields and elliptic curves to the signature algorithms. The implementation techniques for EdDSA were published in the TOMS 2019 journal paper [101] [\(P\)](#), which was co-authored with Ricardo Dahab at the University of Campinas.

5.3.1 Review of Edwards Digital Signature Algorithm

The Edwards Digital Signature Algorithm (EdDSA) is a Schnorr-based signature scheme that uses twisted Edwards curves for performing operations. EdDSA was proposed by Bernstein et al. [30,31] in 2011 as an efficient method to generate digital signatures.

Signature Scheme 5.3.1 (Edwards Digital Signature Algorithm).

- **Setup.** Given an integer λ denoting the security parameter. Fix a twisted Edwards curve $E/\mathbb{F}_p: ax^2 + y^2 = 1 + dx^2y^2$ such that $\log_2(p)/2 \approx \lambda$ and $E(\mathbb{F}_p)$ has order hr , where r is a prime close to p and $h = 2^c$ for small $c \geq 2$. Fix a point $B \in E(\mathbb{F}_p)$ of order r . Define $b = 8\lceil(|p| + 1)/8\rceil$, and select a cryptographic hash function H than outputs $2b$ bits. Define two conversion functions as follows.
 - **Encode** takes as input a point $(x, y) \in E(\mathbb{F}_p)$ and returns a b -bit string u that is calculated as $u = (x \bmod 2) \parallel y$, where \parallel denotes bit concatenation.
 - **Decode** takes as input a b -bit string u from which it parses $y = (u_{b-2}, \dots, u_0)_2$, verifies that $y \in \mathbb{F}_p$, calculates $x = \sqrt{(y^2 - 1)/(dy^2 - a)} \in \mathbb{F}_p$, and verifies that the root exists. If $u_{b-1} \neq x \bmod 2$, it sets $x \leftarrow -x$. Finally, it returns (x, y) if none of the verification steps failed; otherwise, it fails in constant time.

A complete setup must consider a secure instance of an elliptic curve. We refer to SafeCurves project [36] for a comprehensive list of security criteria.

- **Key generation.** The secret key (**sk**) and public key (**pk**) are obtained as follows.
 1. Select $\mathbf{sk} \leftarrow \{0, 1\}^b$ uniformly at random.
 2. Obtain $H(\mathbf{sk}) = (h_{2b-1}, \dots, h_0)_2$.
 3. Construct $s = (2^n + \sum_{i=c}^{n-1} 2^i h_i) \bmod r$, where $n = \lfloor \log_2(r) \rfloor + 1$.
 4. Obtain $\mathbf{pk} = \text{Encode}(sB)$.
 5. Return (**sk**, **pk**)
- **Sign.** Given the pair of keys (**sk**, **pk**) owned by the signer, the signature of a message $M \in \{0, 1\}^*$ is calculated following these steps.
 1. Obtain $H(\mathbf{sk}) = (h_{2b-1}, \dots, h_b, h_{b-1}, \dots, h_0)_2$.
 2. Calculate $k = H((h_{2b-1}, \dots, h_b)_2 \parallel M) \bmod r$.
 3. Obtain $R = \text{Encode}(kB)$.
 4. Calculate $S = k + H(R \parallel \mathbf{pk} \parallel M) \cdot s \bmod r$, where s is defined as in Step (3) of key generation.
 5. Return $(R \parallel S)$ as the signature of M .
- **Verification.** Given the public key (**pk**) of the signer, the following steps must be completed to verify whether an alleged signature $(R \parallel S)$ of the message M is valid.
 1. Obtain $R' = \text{Decode}(R)$ and $P = \text{Decode}(\mathbf{pk})$.

2. Check whether $R', P \in E(\mathbb{F}_p)$ and $0 \leq S < r$.
3. Calculate $h = H(R \parallel \mathbf{pk} \parallel M) \bmod r$.
4. Check that equality $2^c R' = 2^c S B - 2^c h P$ holds.
5. If none of the check steps failed, then accept the signature; otherwise, reject it.

Instances

The initial specification of EdDSA [30] recommends the use of a twisted Edwards curve targeting the 128-bit security level. The curve used is derived from Curve25519 and is defined over $\mathbb{F}_{p_{25519}}$ as

$$\text{edwards25519: } -x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2. \quad (5.3.2)$$

Later in 2015, Bernstein et al. [33] extended the use of EdDSA to other prime fields in order to achieve higher security levels. RFC 8032 [162] describes two instances of EdDSA.

- **Ed25519.** This signature instance uses the parameters of edwards25519 curve, setting $b = 256$ and $H = \text{SHA512}$ [203].
- **Ed448.** This signature instance uses the parameters of edwards448 curve (defined in Equation (5.2.3)) setting $b = 456$ and $H = \text{SHAKE256}$ [205] outputting 912 bits.

Related Works

The introduction of EdDSA presented the first implementations of Ed25519. One of them uses a 64-bit polynomial representation, and the other one uses redundant arithmetic setting $\rho = 51$. The signing and key generation are accelerated by using precomputed tables of 24 and 30 KB, respectively.

Moon [198] developed an optimized library targeting the use of 64-bit instructions and SSE2 vector instructions (128-bit registers). In 2015, Chou [64] reported timings of a vector implementation targeting the Sandy Bridge micro-architecture. Both prime field arithmetic and elliptic curve were parallelized using SSE2 and AVX instructions.

For Ed448, there are fewer points of comparison. The main reference is the Hamburg's library [140], an optimized library that calculates fixed-point multiplications using a signed multi-comb algorithm [139] and uses precomputed tables of 15 KB.

5.3.2 Implementation Details

For Ed25519 and Ed448, we use the AVX2 vector implementation of prime field arithmetic described in Sections 3.4 and 3.6, respectively. For elliptic curve arithmetic, we reviewed aspects related to the parallel execution of the point addition formula for twisted Edwards curves, and we also proposed parallel implementations of fixed-point multiplication algorithms, as shown in Section 4.5.2. The remainder of this section describes how to ensemble all these parts together for implementing Ed25519 and Ed448.

Implementing Key Generation and Signing

The execution time of key generation and signing is dominated by the calculation of fixed-point multiplications. Benefiting from the fact that the point addition formula is complete, designing a multiplication algorithm admits a more flexible way to distribute point operations without taking care of special cases of point addition.

We analyzed several ways of getting a better utilization of AVX2. We implemented prime field arithmetic using both two-way and four-way operations, and with them, we constructed parallel point arithmetic. Thus, we execute one point addition per lane of the AVX2 register, resulting in a four-way point addition operation. This operation helps for distributing the workload of fixed-point multiplication in four additions running in parallel. Proceeding this way, one might assume that the fixed-point multiplication using 256-bit registers takes, for example, half of the time than using 128-bit AVX registers. However, although the algorithm is evenly parallelized, some overheads appear. Most of them are related to fetching more data from memory. Despite the presence of these overheads, the fixed-point multiplication runs faster injecting a significant acceleration.

Alternatively, we looked for optimizations on memory footprint. Like previous implementations, we use 24 KB read-only memory for precomputed tables. We noticed that splitting the workload in eight independent sums then half of the table can be saved, as shown in Section 4.5.3. Using AVX-512, an eight-way SIMD calculation is straightforwardly achieved. However with AVX2, these eight sums are executed by sequentially running two batches of four-way point additions. Therefore, we can calculate fixed-point additions slightly slower but using only 12 KB of read-only memory.

We replicate these techniques for Ed448 and we observed significant overheads. We attribute this performance degradation to the size of operands. For example, any four-way operation requires more than the sixteen vector registers available in the AVX2 vector unit. We observed a better register usage in the AVX-512 unit, which has double capacity. Choosing the size of the look-up tables for fixed-point multiplications presents a compromise between time and space. In our implementation, we favored performance and used a table with 36.75 KB. On the other hand, Hamburg’s library uses a signed multi-comb algorithm that allows using smaller look-up tables with 15 KB. We acknowledge that there could exist better ways of scheduling of operations in parallel that alleviate the issues we found when calculating Ed448 operations.

Implementing the Verification Procedure

The execution time of the verification procedure is dominated by the calculation of a double-point multiplication. We implemented the interleaving algorithm [112] in conjunction with the Non-Adjacent Form (ω -NAF) representation [252], as it was described in Algorithm 4.2.16. Unlike the algorithms for fixed-point multiplication, this algorithm exhibits an inherently sequential pattern of operations. In this case, a simultaneous calculation of point additions is difficult to be performed since point additions are arbitrarily processed whenever a non-zero digit is found.

As an alternative method, we take advantage of the degree of parallelism present inside the point addition formulas. In Section 4.5.2, we show that the point addition

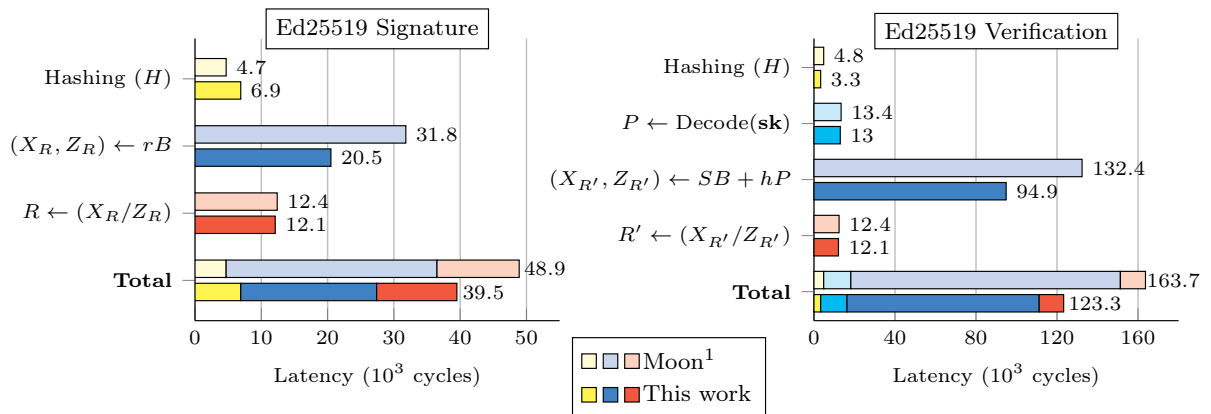
can be distributed among four processing units. Hence, we use four-way prime field operations to implement Algorithm 4.5.11 using the AVX2 instructions. As a result, the point additions required by the double-point multiplication algorithm are performed in parallel internally. We experimentally searched optimal values for ω_P and ω_Q arriving at $\omega_P = 5$ for the arbitrary point, and $\omega_Q = 7$ for the known point.

5.3.3 Performance Benchmark and Comparison

To determine the impact of our implementation techniques, we measured the running time of key generation, signing, and verification; and compared our implementation with the fastest publicly-available implementations of EdDSA.

Timings of Ed25519

Figure 5.3.3 shows a breakdown of the internal operations of Ed25519 comparing to Moon’s implementation. As can be seen, our implementation achieved a noticeable acceleration factor of $1.55\times$ for fixed-point multiplications. This factor matches with the speedup factor of the parallel four-way addition from Table 4.5.12. In this case, computing four point additions in parallel was crucial for accelerating fixed-point multiplications, and consequently, the running time of signature generation was reduced as well.



¹ Moon’s implementation [198].

Figure 5.3.3: Performance profiling of Ed25519 signature operations.

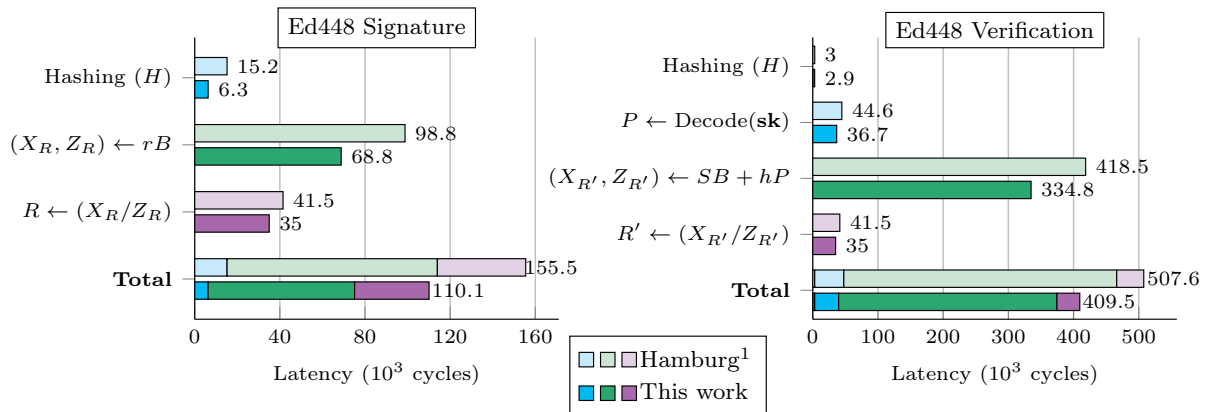
At a first glance, one may state that the latency of Ed25519 signatures is highly dominated by the cost of the scalar multiplication. However, Figure 5.3.3 shows that the calculation of a multiplicative inverse takes a significant portion of the time for generating signatures. Indeed, the cost of this operation gets more significant as the time of calculating a scalar multiplication is reduced. For example, in Moon’s implementation, the multiplicative inverse represents a 25.3% of the calculation of signatures, whereas in our library, the multiplicative inverse takes 30.6% of the total time. Therefore, the cost of this operation must not be disregarded.

On the other hand, although the signature verification procedure requires two costly operations such as an inverse and a square root, here the double-point multiplication widely dominates the cost of verification, as shown in Figure 5.3.3. For this operation, we

leveraged the parallel execution of the internal prime field operations required for point addition. Our implementation is $1.39\times$ faster for double-point multiplications. So, this optimization introduces a 24% reduction of the time of signature verification.

Timings of Ed448

Figure 5.3.4 shows a breakdown of the internal operations of Ed448 comparing to Hamburg’s implementation. The impact of using four-way point additions results in a $1.43\times$ speedup factor for fixed-point multiplications.



¹ Hamburg’s implementation [140].

Figure 5.3.4: Performance profiling of Ed448 signature operations.

We want to emphasize the improvements achieved in the calculation of multiplicative inverses and square roots over $\mathbb{F}_{p_{448}}$. This optimization acquires more relevance at this level, since Ed448 signature operations are 4-5% faster when using the AVX2 implementation of the multiplicative inverses and square roots.

The timings of double-point multiplication reduce by implementing point additions with four-way operations. As Figure 5.3.4 shows, our code is $1.25\times$ faster for calculating double-point multiplications providing a 17% of savings in signature verification.

Timings in Haswell and Skylake

Table 5.3.5 summarizes the timings of several implementation found in eBACS, a website that contains a number of software implementations contributed for public scrutiny.

In Haswell and Skylake, the performance of our implementation outperforms the timings of state-of-the-art implementations. Note that Moon’s 128-bit implementation may serve as a baseline for comparing the performance achieved by using 256-bit instructions. Our 256-bit implementation is almost twice as fast as Moon’s 128-bit implementation, which is the acceleration expected by moving from a 128- to a 256-bit vector unit.

Particularly on Skylake, the running time of the signing procedure is 19% faster than the time achieved by the 64-bit Moon’s implementation; note that although both implementations use a look-up table of 24 KB, ours is faster due to the use of the AVX2 vector unit. Moreover, using a look-up table of half the size (12 KB), our implementation is still 13% faster.

As it can be seen in Table 5.3.5, our library is 29% faster for Ed448 signing when is compared to Hamburg’s implementation running on Skylake. We estimate that a parallel implementation of a signed multi-comb algorithm will lead to similar timings as the ones we reported using a table of 21 KB. The parallel implementation of comb-based algorithms is left as future work.

Table 5.3.5: Timings of Ed25519 and Ed448.

S.	M.	Implementation	ISA	Mult. Instr.	ROM ³	Keygen ¹	Sign ¹	Verify ¹
Ed25519	Haswell	Moon [198]	SSE2	PMULUDQ	24	88.2	91.4	293.9
		slide [35] ²	x64	MULQ	30	66.2	61.4	185.3
		titan0 [35] ²	x64	MULQ	30	57.7	61.2	182.8
		Moon [198]	x64	MULQ	24	56.3	59.2	193.9
		This work	AVX2	VPMULUDQ	12	45.9	51.5	
						<i>18.47%</i>	<i>13.00%</i>	156.0
		This work	AVX2	VPMULUDQ	24	42.8	48.6	<i>14.66%</i>
						<i>23.97%</i>	<i>17.90%</i>	
	Skylake	Moon [198]	SSE2	PMULUDQ	24	72.1	75.3	233.8
		skylake [35] ²	x64	MULQ	24	54.7	49.8	163.2
Moon [198]		x64	MULQ	24	46.1	48.9	163.7	
This work		AVX2	VPMULUDQ	12	36.9	42.3		
						<i>19.95%</i>	<i>13.42%</i>	123.3
	This work	AVX2	VPMULUDQ	24	34.8	39.5	<i>24.44%</i>	
					<i>24.51%</i>	<i>19.22%</i>		
Ed448	Haswell	slide [35] ²	x64	MULX	15	193.8	203.4	673.9
		titan0 [35] ²	x64	MULX	15	176.6	185.0	586.3
		Hamburg [140]	x64	MULX	15	159.7	169.8	596.2
		This work	AVX2	VPMULUDQ	36	126.7	132.7	465.8
						<i>20.66%</i>	<i>21.84%</i>	<i>20.55%</i>
	Skylake	skylake [35] ²	x64	MULX	15	153.0	160.7	498.6
		Hamburg [140]	x64	MULX	15	145.6	155.5	507.6
This work		AVX2	VPMULUDQ	36	104.9	110.1	409.5	
					<i>27.95%</i>	<i>29.19%</i>	<i>17.87%</i>	

¹ Entries are 10^3 clock cycles and numbers in italics are the percentage of savings with respect to the fastest previous implementation.

² The entries correspond to slide/20160806, titan0/20171218, and skylake/20161026 implementations reported in eBACS.

³ Denotes the number of kilobytes of read-only memory for precomputed tables.

5.4 Implementation of qDSA using Curve25519

The Quotient Digital Signature Algorithm (qDSA) was recently introduced aiming to have key compatibility with the X25519 Diffie-Hellman protocol. Due to the novelty of qDSA, there remains a need for an optimized implementation that allows identifying the real impact of this new algorithm. We present a secure and efficient implementation of

qDSA using Curve25519. The results presented in this section were published in the SPACE paper [95] (P), which was co-authored with Hayato Fujii and Diego Aranha at the University of Campinas.

5.4.1 Review of the Quotient Digital Signature Algorithm

Renes and Smith [231] proposed a signature scheme called Quotient Digital Signature Algorithm (qDSA) which is based on the Schnorr [238] and EdDSA signature schemes. Unlike other algorithms, qDSA performs operations over a Kummer variety. Given an elliptic curve group $E(\mathbb{F}_q)$, the Kummer variety associated to it is the quotient group $E(\mathbb{F}_q)/\sim$ where $P \sim Q$ iff $Q = -P$ for all $P, Q \in E(\mathbb{F}_q)$. For example, by using Montgomery curves, the Kummer variety resultant is the x -line, which is isomorphic to a one-dimensional projective space $\mathbb{P}^1(\mathbb{F}_q)$. Although the Kummer variety does not preserve the group law of $E(\mathbb{F}_q)$, it is still possible to compute multiplications by integers inheriting the hardness of the elliptic curve discrete logarithm problem.

Signature Scheme 5.4.1 (Quotient Digital Signature Algorithm with Montgomery Curves).

- **Setup.** Given an integer λ denoting the security parameter. Fix a Montgomery curve $E/\mathbb{F}_p: By^2 = x^3 + Ax^2 + x$, where p is a prime number such that $\log_2(p)/2 \approx \lambda$ and $E(\mathbb{F}_p)$ has order hr , where r is a large prime close to p , and the cofactor is $h = 2^c$ for small $c \geq 2$. Fix a point $G \in E(\mathbb{F}_p)$ of order r . Select a cryptographic hash function H that outputs $2N$ bits, where $N = 8\lceil |p|/8 \rceil$. A complete setup must consider a secure instance of an elliptic curve. We refer to SafeCurves project [36] for a comprehensive list of security criteria.
- **Key Generation.** Select a bit string $d \in \{0, 1\}^N$ uniformly at random and obtain $H(d) = d_1 \parallel d_0$, where d_0 and d_1 are interpreted as N -bit integers. Calculate $Q = \mathbf{x}(d_0G)$ and normalize this point to obtain $Q = (x_Q : 1)$. Finally, set (d_0, d_1) as the secret key, and x_Q as the public key.
- **Signing.** Let (d_0, d_1) and x_Q be the signer's keys, the signature of a message $M \in \{0, 1\}^*$ is calculated as follows. Obtain $k_0 = H(d_1 \parallel M) \bmod r$, and calculate $(x_R : 1) = \mathbf{x}(k_0G)$ using this value to obtain $k_1 = H(x_R \parallel x_Q \parallel M)$. Finally, calculate $s = k_0 - k_1 \cdot d_0 \bmod r$, and declare (x_R, s) as the signature of M .
- **Verification.** Let x_Q be a public key, and $(x_R \parallel s)$ is an alleged signature of a message $M \in \{0, 1\}^*$. First, calculate $k_1 = H(x_R \parallel x_Q \parallel M) \bmod r$, and $R_0 = \mathbf{x}(sG)$. Then, set $Q = (x_Q : 1)$ and obtain $R_1 = \mathbf{x}(k_1Q)$. Finally, if $\text{Check}(x_R, R_0, R_1)$ returns True, accept the signature; otherwise, reject it.
- **Check.** Given $x_R \in \mathbb{F}_p$ and $\mathbf{x}(P), \mathbf{x}(Q) \in \mathbb{P}^1$ such that $P, Q \in E(\mathbb{F}_p)$, determines whether $(x_R : 1) \in \{\mathbf{x}(P + Q), \mathbf{x}(P - Q)\}$. To do that, assume $\mathbf{x}(P) = (X_P : Z_P)$

and $\mathbf{x}(Q) = (X_Q : Z_Q)$, define $f(x) = f_2x^2 + f_1x + f_0$ where

$$\begin{aligned} f_0 &= (X_P X_Q - Z_P Z_Q)^2, \\ f_1 &= -2(X_P X_Q + Z_P Z_Q)(X_P Z_Q + X_Q Z_P) - 4A X_P X_Q Z_P Z_Q, \\ f_2 &= (X_P Z_Q - X_Q Z_P)^2. \end{aligned} \tag{5.4.2}$$

Then, if $f(x_R) = 0$, return True; otherwise, return False.

Instances

The authors of qDSA instantiated a signature algorithm using Curve25519. Thus, in addition to the parameters of the curve, they set $N = 256$ and choose H as the SHAKE128 function outputting 512 bits [205].

Another way to instantiate qDSA is using Kummer surfaces. For example, a recent work by Renes and Hisil [147] propose an implementation of qDSA using a Kummer line derived from the Jacobian of a genus two curve.

5.4.2 Implementation Details

The implementation of qDSA has several parts in common with X25519; mainly, the prime field arithmetic. We use our 64-bit implementation optimized with BMI2 and ADX instruction sets, which was shown in Section 3.4. Moreover, we present some other contributions that are particular to the signature algorithm.

For elliptic curve arithmetic, we apply right-to-left algorithms for accelerating fixed-point multiplications showing how to avoid the use of low-order points. In addition, we show a method that verifies signatures unequivocally. Finally, we present a performance benchmark of our implementation. All these modifications help to improve the performance and security of all qDSA operations.

Integrating Right-to-Left Fixed-Point Ladder

The running time of qDSA is dominated by the calculation of four scalar multiplications. Particularly, three of them are fixed-point multiplications, which makes this scenario suitable for using algorithms with precomputation.

We adapted the right-to-left ladder that is used in the implementation of the Diffie-Hellman protocol, shown in section 4.4.4, to the case of qDSA. Recall that the goal of Algorithm 4.4.21 is to calculate $\mathbf{x}(hkG)$ for some scalar k and curve cofactor $h \geq 4$. This algorithm first calculates $\mathbf{x}(S + kG)$ for some $S \notin \langle G \rangle$. By multiplying this point by h , the point S will vanish assuming S has order four, and it results in $\mathbf{x}(hkG)$.

Due to qDSA works on the x -line of a Montgomery curve, we implement Algorithm 4.4.21 for calculating fixed-point multiplications. This algorithm has several advantages such as it follows a regular execution pattern and uses non-secret indexes for accessing to points stored in the precomputed tables. However, the algorithm internally uses low-order points, which could bring undesired properties to the signature algorithms.

Circumventing the Use of Low-Order Points

We introduce a set of modifications to avoid using low-order points in the right-to-left ladder. The misuse of low-order points could introduce some vulnerabilities [94] that could be even more devastating in the case of digital signatures [253]. Therefore, it is imperative to protect the implementation against this potential threat.

We show how to avoid the use of the low-order point S . First, note that replacing S by \mathcal{O} in Algorithm 4.4.21 causes a failure only when the least-significant bit of k is zero; nonetheless, it always computes the correct point multiplication whenever k is odd. This observation indicates that by setting $S = \mathcal{O}$, the algorithm calculates scalar multiplications for odd scalars only.

Clearly, if G has odd order r (which is the case as r is prime), the parity of an element in $\{1, \dots, r-1\}$ determines a bijection between the disjoint sets of even and odd elements.

Proposition 5.4.3. Let r be an odd number. For any value a such that $0 < a < r$ define $b = r - a$; we have that if a is even, then b is odd.

Proof. Note that b is bounded as $0 < b < r$. Since $a < r$, then $b = r - a > 0$. Suppose that $b \geq r$, then by the definition of b , we have that $r - a \geq r$, i.e., $a \leq 0$, which is a contradiction, since $a > 0$; thus, $0 < b < r$. Since r is odd and a is even, then there exist some $i, j \in \mathbb{Z}$ such that $b = r - a = 2i + 1 - 2j = 2(i - j) + 1$; showing that b is odd. \square

We show modifications in the multiplication algorithm for supporting even scalars. Using this proposition, one can calculate $\mathbf{x}(kG)$ as $\mathbf{x}(k'G)$, for $k' = r - k$, whenever the scalar k is even. If this operation were performed using affine points, the point $k'G$ should be inverted to obtain kG . Fortunately, this is not needed because we are operating on the x -line, which maps the points kG and $k'G$ to the same element. Among the changes made on the algorithm, now it starts calculating $k' = r - k$, and selecting the appropriated scalar between k and k' . This selection introduces a time variability in its execution; hence, it must be processed using a regular execution pattern. This task can be achieved using the CSWAP function. Thus, after computing the conditional swap, the scalar selected will always be odd allowing to start the main-loop of the algorithm from the second iteration. Therefore, the multiplication algorithm can now support any positive scalar without employing low-order points. All these modifications do not interfere significantly with the performance of the algorithm. Thus, the implementation of qDSA leverages the speed of the fixed-point right-to-left ladder.

5.4.3 Unequivocal Verification Methods

The central operation of the verification procedure is that given an alleged signature (x_R, s) , it must determine whether x_R is the x -coordinate of $R_0 + R_1$, where $R_0 = sG$ and $R_1 = k_1Q$ for a scalar k_1 that depends on the message and the public key.

The authors of qDSA provided a function, called Check, that verifies a weaker relation. In their method, the signature is accepted whenever $f(x_R) = 0$, where f is the quadratic polynomial $f(x) = f_2x^2 + f_1x + f_0$ as defined in Equation (5.4.2). This method works since one of the roots of f is x_R . Unfortunately, one disadvantage of this approach is that

there exists another value, say x' , that also passes this verification procedure. Specifically, x' is the other root of f , which corresponds to the x -coordinate of $R_0 - R_1$. Therefore, (x_R, s) and (x', s) are valid signatures.

Although a low adversarial advantage can be exploited from this relaxed verification method, it has a high risk to introduce a misuse of the cryptographic scheme, such as the ones reported in [91, 93, 158]. To avoid potential issues in future implementations, we looked for an alternative method that verifies signatures unequivocally.

Our Proposed Unequivocal Verification Method

Let x_S and x_D be, respectively, the x -coordinate of $R_0 + R_1$ and $R_0 - R_1$ for two points $R_0, R_1 \in E(\mathbb{F}_p)$. Given a signature (x_R, s) , we look for a relation that allows us to determine whether x_R is equal to x_S from the x -coordinate of R_0 and R_1 . This is a stronger relation than the one Check asserts, i.e., whether $x_R \in \{x_S, x_D\}$.

Inspired by Montgomery's insights [196], we started with the following relations that are valid for Montgomery curves:

$$x_S + x_D = \beta/\alpha, \quad (5.4.4)$$

$$x_S \times x_D = \gamma/\alpha, \quad (5.4.5)$$

$$x_S - x_D = \delta/\alpha, \quad (5.4.6)$$

such that α , β , γ and δ are defined as

$$\begin{aligned} \alpha &= (x_{R_0} - x_{R_1})^2, & \beta &= 2(x_{R_0} x_{R_1} + 1)(x_{R_0} + x_{R_1}) + 4A x_{R_0} x_{R_1}, \\ \gamma &= (x_{R_0} x_{R_1} - 1)^2, & \delta &= -4B y_{R_0} y_{R_1}. \end{aligned} \quad (5.4.7)$$

First, note that the coefficients of f are derived by solving Equation (5.4.4) for x_D , and plugging this into Equation (5.4.5) resulting in a second-degree polynomial function of x_S . Thus, f can be written as $f(x) = \alpha x^2 - \beta x + \gamma$. On the other hand, solving Equation (5.4.4) for x_S and substituting this into Equation (5.4.5) yields in a second-degree polynomial function of x_D that has the same coefficients as f . This means that both x_S and x_D are the roots of f . Unfortunately, f does not help to distinguish between x_S and x_D .

Our key idea is to obtain a (linear) polynomial that has a zero in x_S . To that end, we start by solving Equation (5.4.6) for x_S and substituting this into Equation (5.4.5); thus we obtain $g_0(x) = \alpha x^2 - \delta x - \gamma$. Analogously, we apply the same procedure, but this time solving for x_D , and we obtain $g_1(x) = \alpha x^2 + \delta x - \gamma$. So far, we have that $g_0 \neq g_1$, which means that by using g_0 , we are now able to distinguish between x_S and x_D , since $g_0(x_S) = 0$ and $g_0(x_D) \neq 0$. However, g_0 has zeros in x_S and in $-x_D$.

We unequivocally identify x_S using $f(x) = (x - x_S)(x - x_D)$ and $g_0(x) = (x - x_S)(x + x_D)$ as follows. Assuming $x \neq 0$ and since $f(x_S) = 0$ and $g_0(x_S) = 0$, we define

$$\begin{aligned} h_0(x) &= (f + g_0)/x = 2\alpha x - \delta - \beta, \\ h_1(x) &= f - g_0 = (\delta - \beta)x + 2\gamma, \end{aligned} \quad (5.4.8)$$

where x_S is a zero of both polynomials.

Listing 5.4.9 shows a SageMath [229] computer script that validates the formulas presented in this section. The derivation of these formulas can be easily extended to use projective points.

Our signature verification method proceeds as follows. Given a signature (x_R, s) , calculate α , β , and δ from the coordinates of R_0 and R_1 . Then, the signature is valid if $h_0(x_R) = 0$. Alternatively, calculate γ instead of α and accept the signature if $h_1(x_R) = 0$. We have shown two relations that allow to verify a signature unequivocally.

Listing 5.4.9 SageMath script for the validation of formulas in \mathbb{Q} .

```

1 R.<x1,y1,x2,y2,A,B> = PolynomialRing(Rationals(),6,"x1,y1,x2,y2,A,B")
2 I = R.ideal([
3   B*y1**2-x1**3-A*x1**2-x1,
4   B*y2**2-x2**3-A*x2**2-x2 ])
5 FQuo = Frac(R.quotient(I))
6 evaluate = lambda F,X: FQuo(F.subs(x=X).rational_simplify())
7
8 def addMontgomery(X1,Y1,X2,Y2):
9   global A, B
10  Xs = B*((Y1-Y2)/(X1-X2))**2-A*X1-X2
11  Ys = (2*X1+X2+A)*(Y2-Y1)/(X2-X1)-B*(Y2-Y1)**3/(X2-X1)**3-Y1
12  return Xs,Ys
13
14 xs,ys = addMontgomery(x1,y1,x2,y2)
15 xd,yd = addMontgomery(x1,y1,x2,-y2)
16
17 alpha = (x1-x2)**2
18 beta = 2*(x1*x2+1)*(x1+x2)+4*A*x1*x2
19 gamma = (x1*x2-1)**2
20 delta = -4*B*y1*y2
21
22 relAdd = FQuo(xs+xd)
23 relPro = FQuo(xs*xd)
24 relDif = FQuo(xs-xd)
25 # Verifying Relations
26 assert( relAdd == beta/alpha )
27 assert( relPro == gamma/alpha )
28 assert( relDif == delta/alpha )
29 # Renes&Smith's f polynomial and testing its zeros
30 f = alpha*x**2-beta*x+gamma
31 assert( evaluate(f,xs) == evaluate(f,xd) == 0 )
32 # Defining g0 and g1 and testing their zeros
33 g0 = alpha*x**2-delta*x-gamma
34 g1 = alpha*x**2+delta*x-gamma
35 assert( evaluate(g0, xs) == evaluate(g0,-xd) == 0 )
36 assert( evaluate(g1, -xs) == evaluate(g1, xd) == 0 )
37 # Defining h0 and h1 and testing their zeros
38 h0 = 2*alpha*x-delta-beta
39 h1 = (delta-beta)*x+2*gamma
40 assert( evaluate(h0,xs) == evaluate(h1,xs) == 0 )

```

Trade-off Analysis of Our Method

In contrast to the original procedure, our verification method requires calculating δ , which assumes knowledge of the y -coordinate of $R_0 = sG$ and $R_1 = hQ$. Recovering the y -coordinate of R_0 and R_1 requires some auxiliary points namely $R_2 = (s + 1)G$ and $R_3 = (h + 1)Q$, which are also calculated by the Montgomery ladder algorithm. Thus, following [210, Theorem 2], we have

$$\begin{aligned} y_{R_0} &= [(x_{R_0}x_G + 1)(x_{R_0} + x_G + 2A) - 2A - (x_{R_0} - x_G)^2x_{R_2}] / (2By_G) \\ y_{R_1} &= [(x_{R_1}x_Q + 1)(x_{R_1} + x_Q + 2A) - 2A - (x_{R_1} - x_Q)^2x_{R_3}] / (2By_Q). \end{aligned} \quad (5.4.10)$$

Then, δ can be written as $\delta = -4By_{R_0}y_{R_1} = T/(By_Gy_Q)$, where T is

$$\begin{aligned} T &= - [(x_{R_0}x_G + 1)(x_{R_0} + x_G + 2A) - 2A - (x_{R_0} - x_G)^2x_{R_2}] \\ &\quad \times [(x_{R_1}x_Q + 1)(x_{R_1} + x_Q + 2A) - 2A - (x_{R_1} - x_Q)^2x_{R_3}]. \end{aligned} \quad (5.4.11)$$

A relevant observation is that the verifier must know y_Gy_Q . There are several alternatives to obtain this value. The simplest solution is to append y_Gy_Q (or $(By_Gy_Q)^{-1}$) to the public key for calculating δ straightforwardly. The downside of this approach is that the public key's size doubles.

Another solution is appending an extra bit to the public key. The verification procedure calculates $\{y', y''\} = \pm\sqrt{B^{-1}(x_Q^3 + Ax_Q^2 + x_Q)}$ and uses $y_{Q(0)}$, the least-significant bit of y_Q , for selecting one of the roots. Hence, if $y' \equiv y_{Q(0)} \pmod{2}$, it sets $y_Q \leftarrow y'$; otherwise it assigns $y_Q \leftarrow y''$. After that, it calculates y_Gy_Q . Note that y_G must be also known, fortunately, this is a fixed parameter of the scheme. This method has the advantage that the public key size is not increased significantly; for example using Curve25519, $(x_Q, y_{Q(0)})$ fits in $N = 256$ bits. The cost of verification increases by one square-root and a few multiplications. This approach is summarized in Algorithm 5.4.12. We want to remark that verifying qDSA signatures unambiguously requires that the verifier knows both the y -coordinate of G (which is a fixed parameter) and the y -coordinate of Q .

Algorithm 5.4.12 Unequivocal Verification Procedure for qDSA.

Constants: (x_G, y_G) are the affine coordinates of $G \in E(\mathbb{F}_p)$.

Input: (x_R, s) is a signature, $M \in \{0, 1\}^*$ is a message, and $(x_Q, y_{Q(0)})$ is the signer's public key.

Output: True, if the signature is valid; otherwise, False.

- 1: $k_1 \leftarrow H(x_R \parallel x_Q \parallel M) \bmod r$
 - 2: $Q \leftarrow (x_Q : 1)$, $R_0 \leftarrow \mathbf{x}(sG)$, $R_1 \leftarrow \mathbf{x}(k_1Q)$
 - 3: $\{y', y''\} \leftarrow \pm\sqrt{B^{-1}(x_Q^3 + Ax_Q^2 + x_Q)}$.
 - 4: Set $y_Q \leftarrow y'$, if $y' \equiv y_{Q(0)} \pmod{2}$; otherwise, $y_Q \leftarrow y''$.
 - 5: Calculate α , β , and δ . //Equation (5.4.7)
 - 6: **if** $h_0(x_R) = 0$ **then** //Equation (5.4.8)
 - 7: **return** True
 - 8: **else**
 - 9: **return** False
 - 10: **end if**
-

5.4.4 Performance Benchmark and Comparison

We focused on the development a software library that supports qDSA using Curve25519. First of all, we want to highlight the acceleration introduced by the right-to-left fixed-point multiplication. We measured the percentage of improvement in the execution time of the qDSA operations. Table 5.4.13 shows the timings on Haswell and Skylake processors.

Table 5.4.13: Timings of qDSA using the right-to-left fixed-point ladder.

Operation	Montgomery ladder	Right-to-left ladder	Savings
Key Generation ¹	171.5	103.8	39.5 %
Signing ¹	197.3	130.1	34.1 %
Verification ¹	347.3	279.5	19.5 %
Code size ²	30,037	41,000	-36.4 %

¹ Entries in the row are 10^3 clock cycles measured in Haswell.

² Entries in the row are bytes.

The inclusion of optimized prime field arithmetic and the fixed-point multiplication algorithm reduces the execution time of qDSA considerably. The unique pointer to another implementation of qDSA is the reference implementation [231], which is not optimized for 64-bit architectures. The impact of the right-to-left ladder is more evident on key generation and signing achieving, respectively, a 39 % and 34 % reduction in their execution time. Likewise, verification of signatures is also accelerated by 19 %.

Regarding memory footprint, the last row of Table 5.4.13 shows the overhead introduced by integrating the use of precomputation. Thus, by including the 8 KB table stored as read-only memory, the code's size of our implementation increases in around 36 %. We recall using precomputation always incur on trade-offs between space and time; the best approach depends on several engineering aspects.

In Table 5.4.14, we summarize the timings of our qDSA implementation measured in Haswell and Skylake processors, and compares the timings of the new verification method.

Table 5.4.14: Timings of qDSA operations.

Operation	Haswell ¹	Skylake ¹
Key Generation	103.8	86.8
Signing	130.1	114.6
Original Verification	279.5	231.1
Unequivocal Verification	309.6	253.5

¹ Entries are 10^3 clock cycles.

Our method calculates one square-root and a few multiplications to recover the y -coordinate of the public key. These extra operation amounts an overhead of 9.7 % of the execution time. This timing penalty is compensated by the security benefits that our verification method provides, and it prevents issues that could appear in the future.

Comparison of Digital Signature Schemes

Table 5.4.15 shows a performance comparison of several digital signature algorithms.

Table 5.4.15: Timings of qDSA and other digital signature schemes.

Algorithm	Instance	Implementation	Sign/sec	Verify/sec
RSA	2048	OpenSSL v1.0.2	1,618	36,576
DSA	2048	OpenSSL v1.0.2	2,071	1,883
ECDSA	P-256	OpenSSL v1.0.2	25,344	10,198
EdDSA	Ed25519	Moon [198]	48,701	17,167
qDSA	Curve25519	This work	25,109	12,109

As expected, the qDSA's signing procedure has a better performance than RSA and DSA signature schemes. In addition, qDSA generates signatures as fast as ECDSA does; however, the qDSA's verification procedure is faster. This positions qDSA as a more efficient alternative for deploying digital signatures than standardized signature algorithms.

The calculation of Ed25519 signatures is approximately twice as fast as the calculation of qDSA signatures. One of the reasons that explains this performance gap relies on the properties of the elliptic curve model used by each scheme. On twisted Edwards curves, the point addition formula is complete allowing to associate point additions in many different ways. This benefits Comb-based fixed-point algorithms, which have more degrees of freedom on their operation and it allows the use of larger precomputed tables. For example, Moon's [198] implementation uses a table of 24 KB, whereas Chou's [64] implementation increases table's size to 30 KB for further speed. Contrary, our implementation of qDSA using Curve25519 uses a table of 8 KB, which is a third of the table size used in Ed25519's implementations.

Future Work

If more speed is needed, two optimizations can be incorporated to our implementation. First, one can replace the 64-bit implementation with the AVX2 vector implementation, which performs Montgomery ladder faster as it runs in parallel. Second, qDSA can be also implemented using elliptic curve operations using twisted Edwards arithmetic. Thus, qDSA can obtain a performance closer to the Ed25519's one.

5.5 Implementation of SIDH-751

Isogeny-based cryptography is a recent line of research of quantum-resistant cryptography. This approach relies on the difficulty of finding isogenies between elliptic curves, rather than on the hardness of the discrete logarithm problem. Inherited from elliptic curves, one salient feature of isogeny-based cryptography is the use of short key sizes, and are actually the shortest in comparison to other quantum-resistant algorithms.

One of the first algorithms of isogeny-based cryptography is a Diffie-Hellman protocol proposed by Stolbunov and Rostovtsev [236, 257] in 2006. Its security is based on the

difficulty of finding isogenies between (ordinary) elliptic curves. Later in 2011, Jao and De Feo [160] introduced the Supersingular Isogeny Diffie-Hellman (SIDH) protocol, which works with supersingular curves rather than ordinary curves. Significant progress in this area has been made. CSIDH [62] is an alternative method to SIDH that retakes the initial ideas of Stolbunov to work with ordinary curves. Also, hash functions [63] and digital signature schemes, such as CSI-FiSh [82] and SQISign [39], have been proposed.

In 2015, NIST initiated a Post-Quantum Cryptography project [206] looking for standardization of quantum-resistant algorithms. The only isogeny-based algorithm in this contest is the Supersingular Isogeny Key Encapsulation (SIKE) [159] mechanism. SIKE uses SIDH as a subroutine. The most recent breakthrough in the area was by Castryck and Decru [61], who showed a key recovery attack on SIKE breaking its security.

One issue isogeny-based algorithms have in common is that they are time consuming. In this section, we present several algorithmic optimizations targeting both elliptic-curve and field arithmetic operations for SIDH-751, an instance of the SIDH protocol. These contributions were published in the TC 2017 journal paper [103] [\(P\)](#), which was co-authored with Francisco Rodríguez-Henríquez and Eduardo Ochoa at the CINVESTAV IPN.

We remark that the contributions presented in this section remain valid despite the recent attack on SIKE mentioned above. Our algorithmic optimizations and implementation techniques are of general interest since they target the arithmetic of finite fields and elliptic curve operations.

5.5.1 Review of Supersingular Isogeny Diffie-Hellman

The Supersingular Isogeny-based Diffie-Hellman (SIDH) protocol was proposed by Jao, De Feo and Plût [160, 161]. The core operation is the calculation of smooth-degree isogenies between supersingular curves.

Protocol 5.5.1 (Supersingular Isogeny Diffie-Hellman Protocol).

- **Setup.** Given an integer λ denoting the security parameter, fix a supersingular elliptic curve E over \mathbb{F}_{p^2} , where p is a number of the form

$$p = l_A^{e_A} l_B^{e_B} f - 1, \quad (5.5.2)$$

where l_A and l_B are small prime numbers, e_A and e_B are positive integers, and f is chosen such that p is prime. Choose p such that $\lambda \approx \log_2(p)/6$. Fix a pair of points that generate the r -torsion $E[r]$ for r equal to $r_A = l_A^{e_A}$ and $r_B = l_B^{e_B}$. Thus, $\langle P_A, Q_A \rangle = E[r_A]$ and $\langle P_B, Q_B \rangle = E[r_B]$.

- **Key Generation.** Alice selects $m_A, n_A \in \mathbb{Z}/r_A\mathbb{Z}$ uniformly at random and calculates an isogeny $\phi_A: E \rightarrow E_A$ whose kernel is generated by $m_A P_A + n_A Q_A$. Alice uses this isogeny to calculate $\phi_A(P_B)$ and $\phi_A(Q_B)$. Then, she sends these points to Bob together with a description of E_A . Bob acts analogously interchanging the subindexes A and B .
- **Shared Secret.** Once Alice received $\{E_B, \phi_B(P_A), \phi_B(Q_A)\}$ from Bob, she calculates an isogeny $\phi_{AB}: E_B \rightarrow E_{AB}$ whose kernel is generated by $m_A \phi_B(P_A) +$

$n_A \phi_B(Q_A)$. Then, Alice calculates the j -invariant of E_{AB} which is the resultant shared secret. If Bob performs analogous operations (interchanging the subindexes A and B), he will obtain the same j -invariant value.

The main parameters of SIDH are the prime¹ that defines the field and the elliptic curve model. The authors of SIDH considered convenient to chose $l_A = 2$, $l_B = 3$, and f as small as possible. To instantiate an elliptic curve efficiently, both Montgomery and twisted Edwards curves provide better performance than using the arithmetic of the short Weierstrass model. However, since isogeny formulas can also be expressed as a function of the x -coordinate of points, the x -line of a Montgomery curve is the model chosen for SIDH. De Feo [81] presented several sets of parameters to instantiate SIDH.

Sampling Torsion Points

For generating the kernel of an isogeny, both Alice and Bob must sample an r -torsion point uniformly at random from $E[r]$ for $r \in \{r_A, r_B\}$. Since $E[r] \cong \mathbb{Z}/r\mathbb{Z} \times \mathbb{Z}/r\mathbb{Z}$, any r -torsion point can be expressed as $k_0P + k_1Q$ where $\{P, Q\}$ is a basis of $E[r]$, and $k_0, k_1 \in \mathbb{Z}/r\mathbb{Z}$ are chosen at random. Compounded to the problem of obtaining a torsion point, it must be noted that the point is considered secret as well as the operations for calculating it. Hence, the scalars are considered as secret values and the torsion point must be calculated without leaking any information.

This problem can be simplified under certain assumptions. First, if k_0 is a unit (i.e., it has multiplicative inverse), the point $P + k_0^{-1}k_1Q = P + kQ$ where $k \in \mathbb{Z}/r\mathbb{Z}$ is also an r -torsion point suitable for generating the kernel of an isogeny. Second, since these operations are performed over points on the x -line of a Montgomery curve, sampling an r -torsion point reduces to the problem of securely calculating $\mathbf{x}(P + kQ)$ given a secret scalar k and points $P, Q \in E[r]$.

Jaio et al. [160] addressed this issue with an algorithm called *three-point ladder* that calculates $\mathbf{x}(P + kQ)$. This algorithm closely matches the regular execution pattern of Montgomery ladder algorithm, but has a higher cost by taking two differential additions and one point doubling per bit of k . Note that this method does not improve the operation counts of the two-dimensional ladder algorithm, which could be used in principle to calculate an r -torsion point as $\mathbf{x}(k_0P + k_1Q)$ using the algorithm shown by Bernstein [24, 37].

The three-point ladder increases the size of the keys interchanged. The algorithm requires that both participants send to each other an isogeny evaluation of the difference of the base points. For example, Alice must also send the point $\phi_A(P_B - Q_B)$ to Bob.

Instances

SIDH-751: Costello et al. [78] instantiated SIDH with $p_{751} = 2^{372}3^{239} - 1$ targeting the 128-bit security level. The initial curve is $E/\mathbb{F}_q: y^2 = x^3 + x$, where $\mathbb{F}_q \cong \mathbb{F}_{p_{751}}[i]/(i^2 + 1)$. The base points are chosen such that $P_A, P_B \in E(\mathbb{F}_{p_{751}})$, and $Q_A = \tau(P_A)$ and $Q_B = \tau(P_B)$, where $\tau: (x, y) \mapsto (-x, yi)$ is the distortion map.

¹Some primes used in the SIDH protocol are Pierpont primes (a prime number of the form $p = 2^i3^j - 1$, for $i, j > 0$, is known as a Pierpont prime [59, 224]) By extending this definition, a *generalized Pierpont (GP)* prime has the form $p = \pm 1 + \prod_{i=1}^k (p_i)^{e_i}$, where p_i are distinct primes.

5.5.2 Implementation Details

We describe optimizations and implementation techniques for the SIDH protocol. Some of them apply generally, and others could require some modifications for using in instances different than SIDH-751. We propose a new three-point ladder algorithm for calculating $\mathbf{x}(P + kQ)$ and is used sampling torsion points. This algorithm also admits the use of precomputed values. We also present an optimized point tripling formula and compare the performance of our implementation.

Field Arithmetic

Techniques for implementing of prime field arithmetic and the quadratic extension field were presented at Section 3.7. In particular, we show that reductions modulo p can be performed faster by avoiding loop-carried dependencies during the REDC algorithm.

Faster Sampling Torsion Points

We found a faster algorithm for calculating $\mathbf{x}(P + kQ)$, which is the core operation of sampling torsion points. In Section 4.4.5, we presented Algorithm 4.4.24 in the context of arithmetic for Montgomery curves. In this setting, this algorithm stands as an efficient method that accelerates the task of sampling torsion points in the SIDH protocol. More specifically, our algorithm reduces the number of point operations with respect to the original three-point ladder by Jao and De Feo, it saves one differential addition per bit of k , which represents a theoretical improvement equivalent to $1.34\times$ speedup factor. Our algorithm has the same interface as three-point ladder, so a few changes are needed to integrate it in existing implementations.

Our algorithm offers more savings during the key generation phase. Recall that in this phase the input points are known and fixed as they form a basis of the initial curve. Because of that, our algorithm can leave out the calculation of point doublings, and calculate only one differential addition per bit of k . Every point doubling, previously required in the general case, is now precomputed and the point can be stored in a table.

As a proof of concept, we integrated our three-point ladder algorithm in the SIDH v2 implementation by Costello et al. [78]. We found a limitation in the key generation phase caused by the choice of SIDH-751 parameters. By construction, all base points were chosen in such a way that their x -coordinate is an element of the prime field, which allows sampling r -torsion points with faster arithmetic. To keep operations on the prime field, our algorithm requires that $\mathbf{x}(Q - P) \in \mathbb{P}^1(\mathbb{F}_p)$; unfortunately, this is not the case for the SIDH-751 parameters because $\mathbf{x}(Q - P) = ((x_P^2 + 1)i : 2x_P) \notin \mathbb{P}^1(\mathbb{F}_p)$. Consequently, it would appear that the point selection method restricts the use of our algorithm.

We show that the direct method for calculating $\mathbf{x}(P + kQ)$ can also benefit from precomputation. The central idea is using the right-to-left ladder with precomputation (Algorithm 4.4.21) to calculate $x(kQ)$, then recovering its y -coordinate, and adding P . This method takes one differential addition per bit of k plus a constant number of multiplications. Note that all of these operations are performed using prime field arithmetic. In comparison to the original three-point ladder, this approach is around $3\times$ faster.

A relevant aspect of the right-to-left ladder is to find a suitable point $S \notin \langle Q \rangle$. We must guarantee that $x(S - Q) \in \mathbb{P}^1(\mathbb{F}_p)$ and that S has low order h , which makes cofactor multiplication cheaper. A natural choice to obtain low-order independent points is that Alice uses Bob's points, and vice versa. For example, one can set

$$S = \begin{cases} S_A = [3^{e_B-1}]Q_B, & \text{for Alice,} \\ S_B = [2^{e_A-2}]Q_A, & \text{for Bob,} \end{cases} \quad (5.5.3)$$

which are, respectively, points of order three and four.

Integrating the Point Tripling Formula

Calculating an isogeny of degree three, which is often needed by Bob during the SIDH protocol, requires of a point of order three. To obtain this point, Bob has available a point $R \in E(\mathbb{F}_q)$ of order 3^e for some positive integer e . Thus, by calculating $3^{e-1}R$, Bob will obtain a point of order three. This latter operation can be performed iterating point tripling over R . For this reason, there is a need for efficient formulas for point tripling.

In Section 4.4.6, we showed improvements on point tripling on the x -line of a Montgomery curve. Our formula considers that the Montgomery curve parameter A is given as an arbitrary rational value, i.e., $A = A_0/A_1$. Once this parameter is known, a pair of field additions can be calculated once and apply our formula at a lower cost.

5.5.3 Performance Benchmark and Comparison

We measured the performance of our implementation integrating the changes proposed for SIDH. Section 3.7.1 presented timings of the reduction modulo p_{751} ; and in this section, we summarize the impact of the remainder contributions.

Impact of Point Tripling Formula

The new point tripling formula saves up to 400 clock cycles corresponding to the cost of **1M-1S-1A** (cf. Table 4.4.35). This reduction in the cost of the point tripling computation yields a slight acceleration of the whole SIDH protocol of around 1-2% of its execution.

Impact of Three-Point Ladder

We determined the effect of including only the new three-point ladder algorithm in the SIDH v2 implementation. The timings measured are shown in Table 5.5.4.

These timings corroborate the theoretical estimations given in Table 4.4.25. In the variable-point scenario, the SIDH v2 library executes a three-point ladder multiplication in 11.2×10^6 Haswell clock cycles. However, using the new three-point ladder, the same calculation is performed $1.38\times$ faster. At protocol level, this optimization saves 6-7% of the time of the shared secret phase. In the fixed-point case, our approach is $1.7\times$ faster than the previous methods implemented in the SIDH v2 library. The precomputed table takes 35 KB of read-only memory. Note that a large part of the table could fit in the Level-1 Data cache memory that has 32 KB of capacity.

Table 5.5.4: Timings of $\mathbf{x}(P + [k]Q)$ for SIDH-751.

Scenario	Field	Haswell ¹	Skylake ¹	Algorithm
Fixed-point	\mathbb{F}_{p^2}	6.7	4.9	Direct method. ²
		3.9	2.9	This work [103, Alg. 3]
	\mathbb{F}_p	2.5	1.7	Direct method. ²
		1.5	1.0	This work [103, Alg. 4]
Variable-point	\mathbb{F}_{p^2}	11.2	8.1	Three-point ladder
		8.0	5.9	This work (Algorithm 4.4.24)

¹ Entries are 10^6 clock cycles.

² The direct method is calculating $\mathbf{x}(kQ)$ using Montgomery ladder, recovering its y -coordinate, and adding P .

Our implementation contains some countermeasures against timing attacks. In particular, the right-to-left three-point ladder is implemented using a regular execution pattern, and precomputed tables are accessed using non-secret values.

Performance of SIDH-751

Table 5.5.5 shows the timings of the SIDH-751 protocol. The speedups show a high correlation with the acceleration of multiplications on the quadratic extension field reported in Section 3.7. For all of the SIDH operations, the performance measured on Skylake was between 1.38 to 1.41 times faster than the one measured on the Haswell processor. We attribute this acceleration as a consequence of using the BMI2 and ADX integer instruction sets supported in Skylake.

Table 5.5.5: Timings of SIDH-751.

Protocol Phase		Haswell			Skylake		
		CLN ²	This work	Speedup	CLN ²	This work	Speedup
Key Generation	Alice	48.3	38.0	1.27×	35.7	26.9	1.33×
	Bob	54.5	42.8	1.27×	39.9	30.5	1.31×
Shared Secret	Alice	45.7	34.3	1.33×	33.6	24.9	1.35×
	Bob	52.8	39.6	1.33×	38.4	28.6	1.34×

¹ Entries are 10^6 clock cycles.

² Timings of the SIDH v2 implementation by CLN [78].

These implementation techniques are not strictly bidden to the SIDH-751 instance, since the same techniques can be applied to other instances with few changes in the parameters. We have shown that the implementation of the quadratic extension relies on the special shape of the prime modulus. This constraint is not difficult to fulfill as SIDH requires working with this primes to instantiate supersingular curves. Some other contributions such as the point tripling and the three-point ladder algorithm are of independent interest for other applications.

5.6 Implementation of SHA-256

The Secure Hash Standard (SHA) [203] defines a family of cryptographically secure hash functions. Belonging to this family, the SHA-256 provides a security level of 128 bits against collision attacks. This function is widely used in practice in cryptographic algorithms such as signature schemes.

In this section, we show how to implement SHA-256 using the SHA New instructions (SHA-NI) [136]. We first review the SHA-256 algorithm and describe details about its implementation. We show results of performance benchmarks of several implementations, which allows us to determine the impact of this hardware extension.

5.6.1 The SHA-256 Algorithm

SHA-256 takes as input an arbitrary-length message $M \in \{0, 1\}^*$. The message must be padded in such a way that its length in bits is a multiple of 512. After performing a padding rule, the message is split into n blocks of 512 bits denoted as M_j , for $j = 1$ to n .

The SHA-256 algorithm uses a 256-bit state that is initialized as

$$S_0 = \begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix} = \begin{bmatrix} 0x6a09e667 & 0xbb67ae85 & 0x3c6ef372 & 0xa54ff53a \\ 0x510e527f & 0x9b05688c & 0x1f83d9ab & 0x5be0cd19 \end{bmatrix}. \quad (5.6.1)$$

Each message block is used to generate a new state according to the following recurrence:

$$S_j = \text{Update}(S_{j-1}, M_j), \quad \text{for } j = 1 \text{ to } n. \quad (5.6.2)$$

Thus, the hash of M is defined as $\text{SHA-256}(M) = S_n$.

Before presenting the definition of the Update function, we review some notation and auxiliary functions. SHA-256 is a word-oriented algorithm because its operations are performed on words of 32-bits. Let x , y , and z denote generic 32-bit words; and \wedge , \oplus , and \neg denote, respectively, the AND, XOR, and NOT Boolean operations; and \ll and \gg denote, respectively, a logical left and right 32-bit shift. The Update function also defines some auxiliary functions and a set of 64 constant values, which are conveniently displayed in matrix form:

$$\text{Rot}(x, n) = (x \gg n) \oplus (x \ll (32 - n)) \quad (5.6.3)$$

$$\sigma_0(x) = \text{Rot}(x, 7) \oplus \text{Rot}(x, 18) \oplus (x \gg 3) \quad (5.6.4)$$

$$\sigma_1(x) = \text{Rot}(x, 17) \oplus \text{Rot}(x, 19) \oplus (x \gg 10) \quad (5.6.5)$$

$$\Sigma_0(x) = \text{Rot}(x, 2) \oplus \text{Rot}(x, 13) \oplus \text{Rot}(x, 22) \quad (5.6.6)$$

$$\Sigma_1(x) = \text{Rot}(x, 6) \oplus \text{Rot}(x, 11) \oplus \text{Rot}(x, 25) \quad (5.6.7)$$

$$\text{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z) \quad (5.6.8)$$

$$\text{Maj}(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \quad (5.6.9)$$

$$\begin{bmatrix} K_0 \\ K_4 \\ \vdots \\ K_{60} \end{bmatrix} = \begin{bmatrix} k_0 & \dots & k_3 \\ \vdots & \ddots & \vdots \\ k_{60} & \dots & k_{63} \end{bmatrix} = \begin{bmatrix} 0x428a2f98 & 0x71374491 & 0xb5c0fbcf & 0xe9b5dba5 \\ 0x3956c25b & 0x59f111f1 & 0x923f82a4 & 0xab1c5ed5 \\ 0xd807aa98 & 0x12835b01 & 0x243185be & 0x550c7dc3 \\ 0x72be5d74 & 0x80deb1fe & 0x9bdc06a7 & 0xc19bf174 \\ 0xe49b69c1 & 0xefbe4786 & 0x0fc19dc6 & 0x240ca1cc \\ 0x2de92c6f & 0x4a7484aa & 0x5cb0a9dc & 0x76f988da \\ 0x983e5152 & 0xa831c66d & 0xb00327c8 & 0xbf597fc7 \\ 0xc6e00bf3 & 0xd5a79147 & 0x06ca6351 & 0x14292967 \\ 0x27b70a85 & 0x2e1b2138 & 0x4d2c6dfc & 0x53380d13 \\ 0x650a7354 & 0x766a0abb & 0x81c2c92e & 0x92722c85 \\ 0xa2bfe8a1 & 0xa81a664b & 0xc24b8b70 & 0xc76c51a3 \\ 0xd192e819 & 0xd6990624 & 0xf40e3585 & 0x106aa070 \\ 0x19a4c116 & 0x1e376c08 & 0x2748774c & 0x34b0bcb5 \\ 0x391c0cb3 & 0x4ed8aa4a & 0x5b9cca4f & 0x682e6ff3 \\ 0x748f82ee & 0x78a5636f & 0x84c87814 & 0x8cc70208 \\ 0x90bfeffa & 0xa4506ceb & 0xbef9a3f7 & 0xc67178f2 \end{bmatrix} \tag{5.6.10}$$

The Update function consists of two phases called message schedule and update state as shown in Algorithm 5.6.11.

Algorithm 5.6.11 Update Function for SHA-256.

Input: $S \in \{0, 1\}^{256}$ is the state, and $M \in \{0, 1\}^{512}$ is a message block.

Output: $\bar{S} = \text{Update}(S, M)$.

Message Schedule Phase

- 1: $(w_0, \dots, w_{15}) \leftarrow M$
- 2: **for** $t \leftarrow 16$ **to** 63 **do**
- 3: $w_t \leftarrow \sigma_0(w_{t-15}) + \sigma_1(w_{t-2}) + w_{t-7} + w_{t-16}$
- 4: **end for**

Update State Phase

- 5: $(a, b, c, d, e, f, g, h) \leftarrow S$ //Split state into eight 32-bit words.
 - 6: **for** $i \leftarrow 0$ **to** 63 **do**
 - 7: $t_1 \leftarrow h + \Sigma_1(e) + \text{Ch}(e, f, g) + k_i + w_i$
 - 8: $t_2 \leftarrow \Sigma_0(a) + \text{Maj}(a, b, c)$
 - 9: $h \leftarrow g, g \leftarrow f, f \leftarrow e, e \leftarrow d + t_1$
 - 10: $d \leftarrow c, c \leftarrow b, b \leftarrow a, a \leftarrow t_1 + t_2$
 - 11: **end for**
 - 12: $(\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{e}, \bar{f}, \bar{g}, \bar{h}) \leftarrow S$ //Split state into eight 32-bit words.
 - 13: **return** $\bar{S} \leftarrow (a + \bar{a}, b + \bar{b}, c + \bar{c}, d + \bar{d}, e + \bar{e}, f + \bar{f}, g + \bar{g}, h + \bar{h})$
-

5.6.2 Implementation Details

We describe how to use SHA-NI instructions for implementing the message schedule phase. First of all, it is required that the message block (the values w_0, \dots, w_{15}) be stored into

four 128-bit vector registers W_0 , W_4 , W_8 , and W_{12} as follows $W_i = [w_i, w_{i+1}, w_{i+2}, w_{i+3}]$. After that, the `SHA256MSG1` and `SHA256MSG2` instructions will help on the message schedule phase for calculating the values w_{16}, \dots, w_{63} . To have a better understanding of the design rationale of SHA-NI, we describe the steps to calculate W_{16} as follows:

$$\begin{array}{c} \begin{bmatrix} w_{16} \\ w_{17} \\ w_{18} \\ w_{19} \end{bmatrix} = \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix} + \sigma_0 \left(\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} \right)}_{\text{SHA256MSG1}} + \underbrace{\begin{bmatrix} w_9 \\ w_{10} \\ w_{11} \\ w_{12} \end{bmatrix}}_{\text{PALIGNR}} + \sigma_1 \left(\begin{bmatrix} w_{12} \\ w_{13} \\ w_{14} \\ w_{15} \end{bmatrix} \right) \\ \underbrace{\hspace{15em}}_{\text{PADD}} \\ \underbrace{\hspace{15em}}_{\text{SHA256MSG2}} \end{array}$$

The `SHA256MSG1` instruction updates the vector register X with four 32-bit words as

$$\begin{aligned} X &\leftarrow \text{SHA256MSG1}(W_0, W_4) \\ &= \text{SHA256MSG1}([w_0, w_1, w_2, w_3], [w_4, w_5, w_6, w_7]) \\ &= [\sigma_0(w_1) + w_0, \sigma_0(w_2) + w_1, \sigma_0(w_3) + w_2, \sigma_0(w_4) + w_3]. \end{aligned} \quad (5.6.12)$$

In the next step, the `PALIGNR` instruction obtains W_9 from a 32-bit shift applied to the concatenation of W_{12} and W_8 as

$$\begin{aligned} W_9 &\leftarrow \text{PALIGNR}(W_{12}, W_8, 4) \\ &= \text{PALIGNR}([w_{12}, w_{13}, w_{14}, w_{15}], [w_8, w_9, w_{10}, w_{11}], 4) \\ &= [w_9, w_{10}, w_{11}, w_{12}]. \end{aligned} \quad (5.6.13)$$

Then, the vector Y will store the word-wise addition of the vector registers X and W_9 as

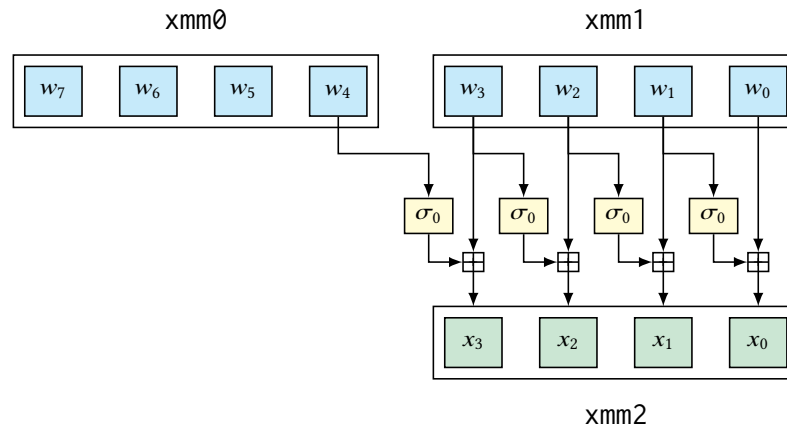
$$\begin{aligned} Y &\leftarrow \text{PADDD}(X, W_9) \\ &= [\sigma_0(w_1) + w_0 + w_9, \sigma_0(w_2) + w_1 + w_{10}, \\ &\quad \sigma_0(w_3) + w_2 + w_{11}, \sigma_0(w_4) + w_3 + w_{12}]. \end{aligned} \quad (5.6.14)$$

Finally, the `SHA256MSG2` instruction produces W_{16} from Y and W_{12} as

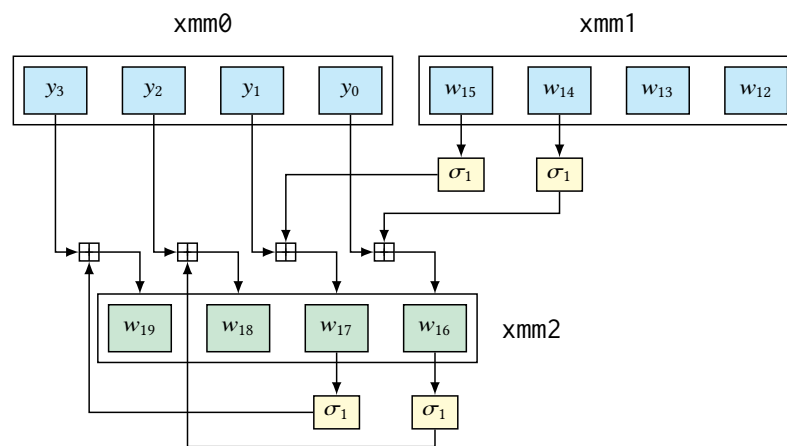
$$\begin{aligned} W_{16} &\leftarrow \text{SHA256MSG2}(Y, W_{12}) = [w_{16}, w_{17}, w_{18}, w_{19}] \\ &= [\sigma_0(w_1) + \sigma_1(w_{14}) + w_0 + w_9, \\ &\quad \sigma_0(w_2) + \sigma_1(w_{15}) + w_1 + w_{10}, \\ &\quad \sigma_0(w_3) + \sigma_1(w_{16}) + w_2 + w_{11}, \\ &\quad \sigma_0(w_4) + \sigma_1(w_{17}) + w_3 + w_{12}]. \end{aligned} \quad (5.6.15)$$

The remainder values W_{20}, \dots, W_{60} are calculated repeating the same strategy. Hence, calculating W_{4i} , for $i = 4, \dots, 15$, depends on $W_{4(i-1)}$, $W_{4(i-2)}$, $W_{4(i-3)}$, and $W_{4(i-4)}$; the latter register can be overwritten to store W_{4i} allowing to reuse vector registers. Figure 5.6.16 illustrates the operation of `SHA256MSG1` and `SHA256MSG2` instructions.

Now we describe the implementation of the update state phase. The `SHA256RND2S` instruction, part of SHA-NI, assumes that S is stored into two 128-bit vector registers as follows $A = [a, b, e, f]$ and $C = [c, d, g, h]$. The reasoning behind this representation



(a) SHA256MSG1 instruction.



(b) SHA256MSG2 instruction.

Figure 5.6.16: SHA-NI instructions for message schedule phase.

relies on the following observation: after processing two iterations of the second for-loop of Algorithm 5.6.22, some words of the state remain unmodified. More generally, let A_i and C_i be the values of the state at the i -th iteration of the for-loop, then it holds that $C_{i+2} = A_i$ for $i \geq 0$. Figure 5.6.21 illustrates this property.

Based on this property, the SHA256RND2 instruction performs two iterations of the state update phase as follows

$$C_{i+2} = A_i, \text{ and } A_{i+2} = \text{SHA256RND2}(C_i, A_i, X) \quad (5.6.17)$$

where X is a vector register containing $[w_i + k_i, w_{i+1} + k_{i+1}, \emptyset, \emptyset]$, where k_i and k_{i+1} are constant values defined in Equation (5.6.10), and \emptyset is an unused value. Four iterations can be performed applying again the same property; thus we have

$$C_{i+4} = A_{i+2}, \text{ and } A_{i+4} = \text{SHA256RND2}(C_{i+2}, A_{i+2}, Y), \quad (5.6.18)$$

which is equivalent to

$$\begin{aligned} C_{i+4} &= \text{SHA256RND2}(C_i, A_i, X), \text{ and} \\ A_{i+4} &= \text{SHA256RND2}(A_i, C_{i+4}, Y) \end{aligned} \quad (5.6.19)$$

such that Y is a register containing $[w_{i+2} + k_{i+2}, w_{i+3} + k_{i+3}, \emptyset, \emptyset]$ and the register X and Y are calculated as

$$\begin{aligned} X &\leftarrow \text{PADDD}(K_{4i}, W_{4i}) = [w_i + k_i, w_{i+1} + k_{i+1}, w_{i+2} + k_{i+2}, w_{i+3} + k_{i+3}] \\ Y &\leftarrow \text{PSRLDQ}(X, 8) = [w_{i+2} + k_{i+2}, w_{i+3} + k_{i+3}, 0, 0], \end{aligned} \tag{5.6.20}$$

where $K_{4i} = [k_{4i}, k_{4i+1}, k_{4i+2}, k_{4i+3}]$. This execution pattern is repeated sixteen times to perform 64 iterations of the update state phase. The complete implementation of SHA-256 using SHA-NI is shown in Algorithm 5.6.22.

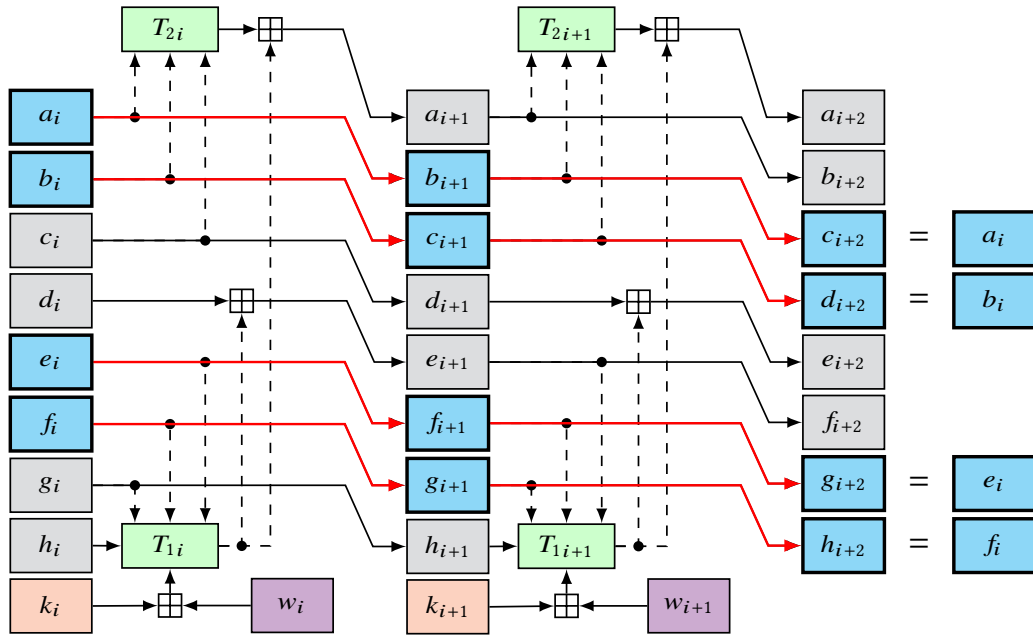


Figure 5.6.21: Every two consecutive iterations of the update state phase, it holds that the values (c, d, g, h) at the $(i + 2)$ -th iteration are exactly the values (a, b, e, f) of the i -th iteration. For this reason, the values (a, b, e, f) of the $(i + 2)$ -th iteration are calculated using the SHA256RND2 instruction.

5.6.3 Performance Benchmark and Comparison

A performance benchmark allows us determine the effects of using SHA-NI. We took two implementations as a baseline for making comparisons. One of the fastest implementations found in SUPERCOP [35] is the sphlib implementation that was written by Thomas Pornin. Also, OpenSSL [263] has a 64-bit implementation based on optimization techniques described in [128, 135]. We measured the number of clock cycles taken for hashing messages. From these measurements, we calculate the cycles-per-byte (cpb) ratio, which is conventionally used as a metric of performance. Figure 5.6.23 shows the performance of these implementations running on Zen.

For messages larger than 256 bytes, the sphlib library takes around 9.6 cpb, whereas, the OpenSSL library offers a better performance taking 7.7 cpb. On the other hand, the SHA-NI implementation takes 1.8 cpb; this is approximately $5.1\times$ faster than sphlib and is $4.2\times$ faster than OpenSSL.

Algorithm 5.6.22 SHA-256 Update Implemented with SHA-NI.**Input:** $S \in \{0, 1\}^{256}$ is the state, and $M \in \{0, 1\}^{512}$ is a message block.**Output:** $\bar{S} = \text{Update}(S, M)$.*Message Schedule Phase*

- 1: Load M into 128-bit vector registers: $W_0, W_4, W_8,$ and W_{12} .
- 2: **for** $i \leftarrow 0$ **to** 11 **do**
- 3: $X \leftarrow \text{SHA256MSG1}(W_{4i}, W_{4i+4})$
- 4: $W_{4i+9} \leftarrow \text{PALIGNR}(W_{4i+12}, W_{4i+8}, 4)$
- 5: $Y \leftarrow \text{PADDD}(X, W_{4i+9})$
- 6: $W_{4i+16} = \text{SHA256MSG2}(Y, W_{4i+12})$
- 7: **end for**

Update State Phase

- 8: $(a, b, c, d, e, f, g, h) \leftarrow S$
- 9: $A \leftarrow [a, b, e, f]$ and $C \leftarrow [c, d, g, h]$.
- 10: **for** $i \leftarrow 0$ **to** 15 **do**
- 11: $X \leftarrow \text{PADDD}(W_{4i}, K_{4i})$
- 12: $Y \leftarrow \text{PSRLDQ}(X, 8)$
- 13: $C \leftarrow \text{SHA256RNDS2}(C, A, X)$
- 14: $A \leftarrow \text{SHA256RNDS2}(A, C, Y)$
- 15: **end for**
- 16: $(\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{e}, \bar{f}, \bar{g}, \bar{h}) \leftarrow S$
- 17: $\bar{A} \leftarrow [\bar{a}, \bar{b}, \bar{e}, \bar{f}]$ and $\bar{C} \leftarrow [\bar{c}, \bar{d}, \bar{g}, \bar{h}]$
- 18: $[a, b, e, f] \leftarrow \text{PADDD}(A, \bar{A})$ and $[c, d, g, h] \leftarrow \text{PADDD}(C, \bar{C})$
- 19: **return** $\bar{S} \leftarrow (a, b, c, d, e, f, g, h)$

The speedup factor observed by using SHA-NI is plotted on the right of Figure 5.6.23. We want to remark that for short-length messages the improvement on the performance is also significant, since it achieves a $3.0\times$ factor of speedup. The improvement on the running time is an evidence of the relevance of including specialized instruction sets that support cryptographic algorithms.

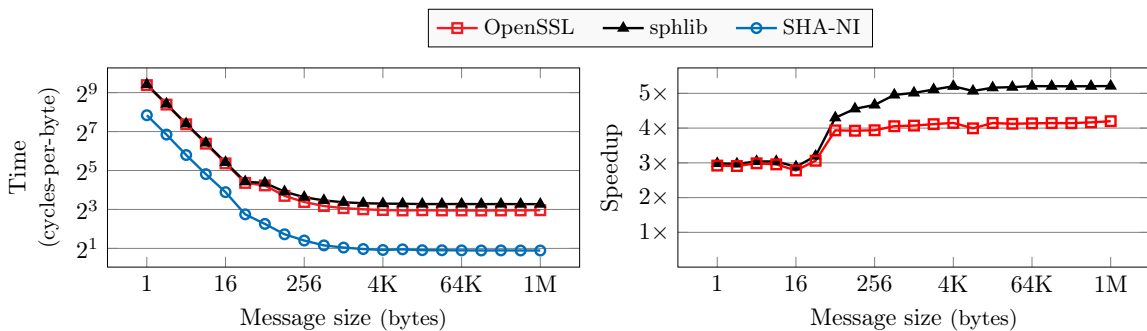


Figure 5.6.23: Performance of SHA-256 measured on Zen. Plots display the cycles-per-byte taken for hashing a message and the speedup factor yielded by using SHA-NI.

5.7 Implementation of XMSS and XMSS^{MT}

In public-key cryptography, the construction of hash-based digital signatures is getting more attention as they are part of a new cryptographic portfolio of quantum-resistant algorithms [151]. We study the implementation of XMSS [58] and XMSS^{MT} [153] hash-based signature schemes.

In this section, we focus on two approaches that improve the performance of SHA-256 in the multiple-message scenario, and evaluate the impact on the performance of these signature schemes. The results presented in this section were published in the APKC 2018 paper [102] (P), which was co-authored with Ana K. D. S. de Oliveira at the Federal University of Mato Grosso do Sul.

5.7.1 Review of Hash-based Signatures

The first scheme is the eXtended Merkle Signature Scheme (XMSS) [58], which uses a Merkle tree (a binary hash-tree) in which the leaves store a public key of a one-time signature scheme, such as WOTS⁺ [152], and the private keys are derived from a pseudo-random number generator fed by a secret seed. In the Merkle tree, every internal node stores a value v that is the hash of its left and right children, i.e., $v = H(v_L, v_R)$. Thus, the XMSS public key is the root of the Merkle tree. Let h be the height of the tree, then XMSS can sign at most 2^h messages using 2^h different key pairs.

The second scheme is the Multi-Tree XMSS (XMSS^{MT}) [153], which is an extension of XMSS that provides a larger number of signatures. Assuming that d divides h , XMSS^{MT} can generate at most 2^h signatures using a d -height hypertree such that each of its nodes is an h/d -height XMSS tree.

In both signature schemes, the calculation of Merkle trees is the performance-critical operation, since a large number of hash calculations is required. Note that the nodes at the same height can calculate their hash values without dependency between them, this means that this workload has a high degree of parallelism. Hence, this is a suitable scenario for applying optimizations based on parallel computing.

A *multiple-message hashing* is the task of hashing several messages of the same length independently. Multiple-message hashing is also known in the literature as multi-buffer hashing, simultaneous hashing, or n -way hashing [2, 118, 128].

Instances

A set of parameters for these schemes are given in RFC-8391 [151]. For example, to achieve a 128-bit post-quantum security level, it is recommended to use a hash function with an output of $n = 32$ bytes, such as SHA-256 or SHAKE [205].

For XMSS, it is suggested to set $h \in \{10, 16, 20\}$; and for the XMSS^{MT} scheme setting $(h, d) \in \{(20, 2), (20, 4), (40, 2), (40, 4), (40, 8), (60, 3), (60, 6), (60, 12)\}$. From these parameters, we selected $h = 20$ for XMSS and $h = 60, d = 6$ for XMSS^{MT} because they allow the largest number of signatures.

5.7.2 Implementation Details

To accelerate the calculation of hash-based signatures, we focus on the efficient implementation of multiple-message hashing. It is clear that the use of SIMD vectorization is suitable in this scenario as several works in the literature have proposed. For this reason, we proceed with an alternative and more efficient approach that relies on the application of SHA-NI extensions. We show how to efficiently use these instructions for multiple-message hashing and compare with SIMD processing.

SIMD Multiple-Message Hashing

In order to hash n messages of the same length using vector instructions, one can modify Algorithm 5.6.11 as follows. First, replace each 32-bit word by a vector register; thus, a set of eight vector registers A, \dots, H will represent the state, where $A = [a_1, \dots, a_n]$ is a vector register containing the word a of each message. The same rationale applies to the rest of the state.

Since now all variables are vectors, every scalar operation must be replaced by its corresponding vector instruction, which simultaneously executes operations in each lane of a vector register. At the end of the algorithm, the digest of the i -th message must be recovered concatenating the i -th lane of each vector register A, \dots, H .

Related Works

In the literature, there exist several works that used SIMD instructions for implementing either single or multiple message hashing.

Gueron [128] showed the n -SMS technique that parallelizes the message schedule phase leading to a faster single message hashing. Following the work of Aci mez [2], Gueron and Krasnov [129] reported an implementation that uses the SSE unit to perform four SHA-256 digests simultaneously. As a result, their implementation runs $2.2\times$ faster than the n -SMS single-message implementation of Gueron [128]. They also extended the parallelization to 256-bit registers, however, although AVX2 was not available at the time their work was published, their estimations accurately match the performance exhibited by processors supporting AVX2.

Intel [117] contributed to the OpenSSL library with optimized code for multiple-message hashing. In v1.0.2, OpenSSL has a function, called `sha256_multi_block`², that calculates either four digests using SSE or eight digests using AVX2 instructions.

Pipelining SHA-NI Instructions

We present optimization strategies that schedule SHA-NI instructions leveraging the capabilities of the processor’s pipeline.

In addition to the latency, there exist other metrics to determine the performance of processor instructions. According to Fog’s definition [109], the throughput of an instruction is “*the maximum number of instructions of the same kind that are executed per*

²Located at: https://github.com/openssl/openssl/blob/OpenSSL_1_0_2-stable/crypto/sha/asm/sha256-mb-x86_64.pl.

clock cycle”, and the reciprocal throughput refers to “the number of cycles to wait until an execution unit starts processing an instruction of the same type”. There is a lack of information about these metrics on the Zen documentation; however, accurate timings can be found using experimental measurements. The following table shows the timings measured by Fog for the SHA-NI instructions on Zen:

	SHA256MSG1	SHA256MSG1	SHA256RND2
Latency	2	3	4
Recip. Throughput	0.5	2	2

The SHA256RND2 instruction takes four clock cycles to be completed; however, its reciprocal throughput is only two cycles, this means that once a SHA256RND2 instruction is issued to the execution unit, a second SHA256RND2 instruction can be issued two clock cycles after the first one. Therefore, by issuing two SHA256RND2 instructions consecutively, the processor will take only six clock cycles to compute them instead of taking eight clock cycles. Figure 5.7.1 shows a pipelined execution of SHA256RND2 instructions.

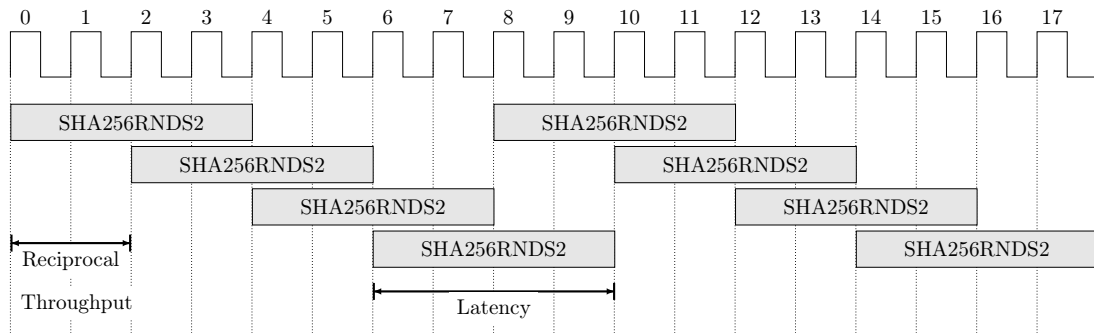


Figure 5.7.1: Pipeline execution of SHA256RND2 instructions.

Observe that to achieve an execution in pipeline; these instructions must have no data dependencies; otherwise, the processor will wait until the data dependency be resolved. After a dependence analysis of Algorithm 5.6.22, it can be noted that in the message schedule phase, the SHA256MSG2 instruction takes as an input the value produced by the SHA256MSG1 instruction. Another data dependency appears in the update state phase, where both SHA256RND2 instructions dependent one to the other. Hence, these data dependencies limit the use of the pipelining techniques for single message hashing. However, hashing multiple messages is a suitable scenario that leverages the capabilities of the processor’s pipeline.

The central idea of our pipelined implementation resembles Gueron’s approach [125] for implementing the AES-CTR encryption algorithm using AES-NI instructions. In the case of SHA-256, at every two clock cycles one SHA256RND2 instruction is issued to the pipeline, such that each instruction operates over a state of a different message. Thus, consecutive instructions do not have data dependencies at all, and as a consequence, their execution can be executed by the processor’s pipeline. Let k be the number of messages to be hashed simultaneously, we developed pipelined SHA-NI implementations of SHA-256 multiple-message hashing for $k \in \{2, 4, 8\}$.

5.7.3 Performance Benchmark and Comparison

Timings of SIMD Hashing

We want to determine the impact on the running time when vectorization is applied to the multiple-message hashing scenario. To do that, we conducted a performance benchmark of the vectorized implementation provided by the OpenSSL library. Figure 5.7.2 shows the result of measuring the performance of the `sha256_multi_block` function.

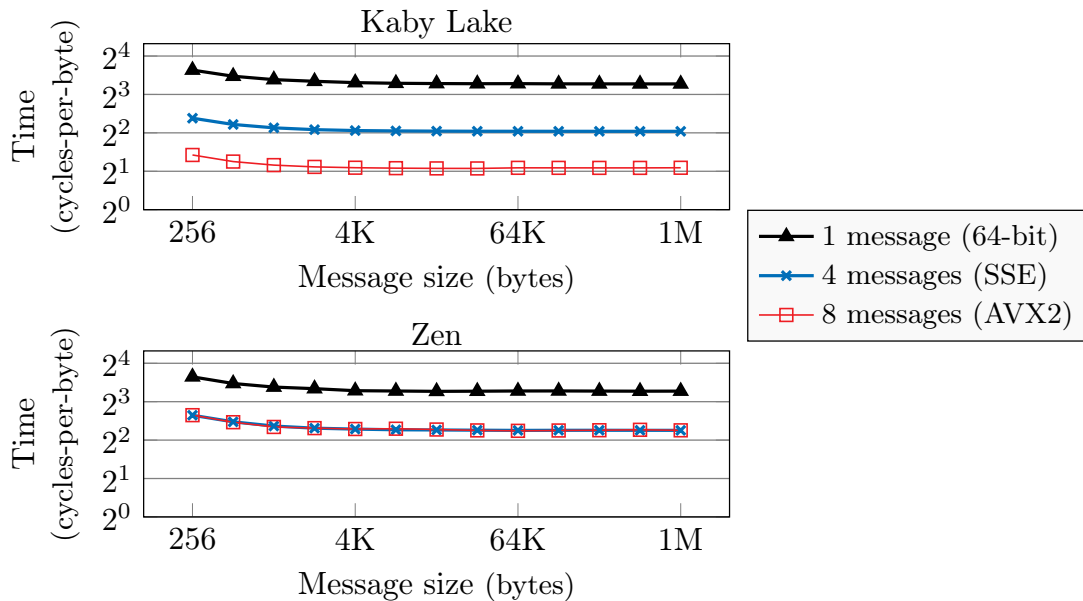


Figure 5.7.2: Timings of multiple-message hashing on Kaby Lake and Zen. The baseline single-message implementation is the sphlib implementation by Pornin (taken from SUPERCOP [35]).

On Kaby Lake, calculating four hashes simultaneously is $2.35\times$ faster than four consecutive invocations to the single-message implementation. Moreover, the AVX2 implementation is $4.5\times$ faster than sphlib for calculating eight hashes in a row. The use of vector instructions on Kaby Lake shows a noticeable improvement in the performance of multiple-message hashing. However, the story is different on Zen.

The graphs in Figure 5.7.2 show that Zen offers a similar performance as Kaby Lake for single-message hashing. The SSE implementation, which computes four hashes simultaneously, renders a better performance on Kaby Lake. However on Zen, the performance of the AVX2 implementation shows the same performance as the one exhibited by the SSE implementation, i.e., no benefits were observed by running AVX2 code on the Zen micro-architecture.

This performance downgrade on Zen is because the latency of AVX2 instructions is twice slower than the latency of the SSE instructions. The micro-architectural design of Zen *emulates* a 256-bit vector instruction splitting the workload in two parts, and then it executes them in a 128-bit vector unit sequentially. Therefore, the expected performance of any AVX2 code is reduced by half on Zen.

Timings of Pipelined SHA-NI Hashing

The results of the performance benchmark are shown in Figure 5.7.3. To hash two messages, it is more convenient to use the pipelined implementation ($k = 2$), which is 18% faster than performing two consecutive hashes using the single-message SHA-NI implementation. Similarly, for $k = 4$ a reduction of a 21% of the running time is obtained. However, note that for $k = 8$ the performance downgrades, which can be explained because hashing eight messages requires a larger space to store all the states causing that vector registers be spilled to memory more often. These results show that a significant improvement for multiple-message hashing is achieved using an efficient instruction scheduling of SHA-NI instructions.

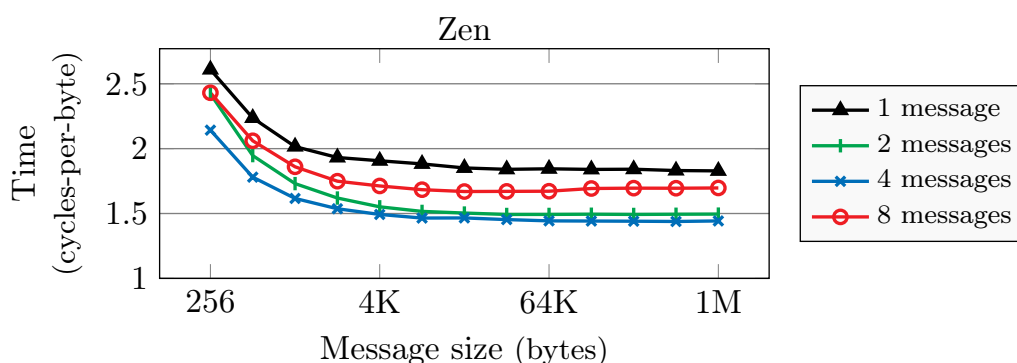


Figure 5.7.3: Timings multiple-message SHA-256 hashing on Zen. The single-message implementation is the SHA-NI implementation of Algorithm 5.6.22.

Timings of Hash-based Digital Signatures

We compare the performance of the signature schemes using as a building block the following implementations of SHA-256. The sphlib implementation that uses 64-bit instructions; the sequential SHA-NI implementation of Algorithm 5.6.22; vectorized implementations using 128- and 256-bit instructions from [83]; the pipelined SHA-NI implementation from this section. Table 5.7.4 list the timings measured on Zen of these implementations.

Taking as a baseline the sphlib implementation, it can be noted that multiple-message hashing has a higher impact on the key generation and the signing operation and a lesser impact on the verification procedure. Note that the running time of the key generation and the signing operation can be reduced by almost half using SSE vector instructions, whereas the AVX2 implementation renders a poor performance on Zen; this was already expected from the measurements presented for multiple-message hashing.

It can be observed that for XMSS the SHA-NI implementation yields a speedup factor between $3.5\times$ to $4\times$ in contrast to the 64-bit sphlib implementation. Moreover, extra savings were achieved by using the pipelined SHA-NI implementation, which increased the speedup factor to $4.3\times$ and $4.6\times$ for XMSS and XMSS^{MT}, respectively, improving a 10% the key generation, and a 7% the signature operation.

Table 5.7.4: Timings of XMSS and XMSS^{MT} measured on Zen.

(a) Timings of XMSS-SHA2_20_256.

Impl.	Parallel	Key Gen. ¹		Signing ²		Verify ²	
sphlib	No	4.50	1.00×	21.56	1.00×	2.16	1.00×
SSE	4-way	2.60	1.72×	12.76	1.68×	1.78	1.21×
AVX2	8-way	3.81	1.18×	19.56	1.10×	3.65	0.59×
SHA-NI	No	1.12	4.01×	5.39	3.99×	0.61	3.51×
SHA-NI	4-Pipelined	1.01	4.46×	5.00	4.30×	0.74	2.89×

¹ Entries are 10¹² clock cycles.² Entries are 10⁶ clock cycles.(b) Timings of XMSS^{MT}-SHA2_60/6_256.

Impl.	Parallel	Key Gen. ¹		Signing ²		Verify ²	
sphlib	No	51.63	1.00×	46.35	1.00×	27.97	1.00×
SSE	4-way	27.44	1.88×	25.27	1.83×	18.94	1.48×
AVX2	8-way	40.11	1.29×	38.31	1.21×	36.81	0.76×
SHA-NI	No	11.87	4.35×	10.69	4.34×	6.45	4.33×
SHA-NI	4-Pipelined	10.78	4.79×	9.99	4.64×	7.64	3.66×

¹ Entries are 10⁹ clock cycles.² Entries are 10⁶ clock cycles.

Timings Running on Kaby Lake

To have a better panorama of the performance of hash-based signatures, we reproduce the experiments using Kaby Lake. Recall that although Kaby Lake does not support the SHA-NI instructions, it contains a faster vector unit. Figure 5.7.5 shows the running time to generate an XMSS signature ($h = 20$) measured on Zen and Kaby Lake.

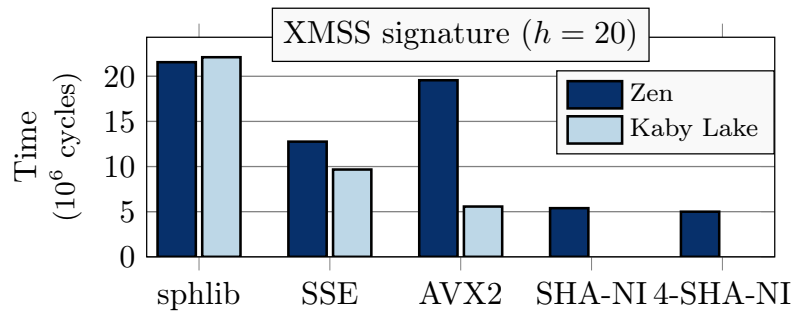


Figure 5.7.5: Performance comparison of XMSS-SHA2_20_256 running on Kaby Lake and Zen.

First of all, it can be observed that Zen delivers better performance for signing using the sequential sphlib implementation. However, the vectorized implementations offer a significant acceleration when running on Kaby Lake, but not in Zen. For example, by using AVX2, an XMSS signature can be computed 4× faster than the sequential implementation. On the other hand, Zen renders a similar performance using the sequen-

tial SHA-NI implementation. However, the pipelined SHA-NI implementation offers the fastest timings. Therefore, we have shown that the performance of hash-based signatures can be accelerated either by using the fast AVX2 unit on Kaby Lake or using a pipelined SHA-NI implementation on Zen.

Future Work

Note that SIMD parallel implementations can also be extended for using AVX-512 instructions [72]; thus, the multiple-message hashing will process sixteen messages simultaneously. Following an analogous analysis that the one performed by Gueron and Krasnov [133]; preliminary evaluation of an implementation of SHA-256 using AVX-512 reveals that the number of instructions does not increase significantly, which could lead to performance improvements on the forthcoming AVX-512 processors.

It would be interesting to reproduce our experiments on ARM processors that support the ARMv8 Cryptographic Extensions [18] and/or the Scalable Vector Extensions [254], however the performance evaluation in this platform turns to be more complex due to the wide variety of processor's implementations.

Finally, the optimizations presented in this work are also applicable to other algorithms, such as the SPHINCS+ [32] hash-based signature scheme, which is participating on the NIST's Post-Quantum Standardization [206].

5.8 Chapter Summary

In this chapter we showed how to optimize implementations of Diffie-Hellman protocols and digital signature schemes based on elliptic curves. We showed some algorithmic optimizations that reduce the number of operations to be computed. In addition, we adapt some algorithms in such a way they benefit from parallel computing. In particular, we showed that scheduling prime field and elliptic curve operations in parallel improves their execution time. This was experimentally demonstrated with our vectorized software and backed with performance benchmarks. The sum of all of these optimizations improves the execution time of the algorithms studied.

Chapter 6

Conclusions

In the past few years, we have accompanied a research trend that proposes the use of modern elliptic curve models in the design of new public-key cryptographic algorithms. Adjacently, we have also witnessed an incessant inclusion of new hardware extensions to the computer architecture supporting mainly the SIMD parallel computing. In these circumstances, our research investigates how to efficiently use these computer architecture extensions, such as SIMD and others, with the aim to accelerate the execution of cryptographic algorithms, and with special interest in those based on elliptic curves.

6.1 Concluding Remarks

Based on the experimentation performed during our investigation, we conclude that the application of SIMD vector instructions does reduce the execution time of both prime field operations and elliptic curve arithmetic resulting in observable improvements in high-level cryptographic algorithms. However, we remark that in order to get better performance several changes in the algorithms are needed. Some of them are naturally motivated by the SIMD parallel computing paradigm, but others arose from the instruction set used in the implementation.

To benefit from the SIMD approach, we proposed some parallel algorithms and adapted several others to increase the degree of parallelism of their internal operations. For instance, the machine representation of integers leverages the large size of operands to perform smaller operations in parallel. For higher level operations, we implemented vectorized code for running prime field operations in parallel, and extended this idea to elliptic curve operations. Our parallel algorithms are applicable to any computing environment that supports SIMD processing.

Our investigation provided explicit optimizations and implementation techniques that resulted in a faster execution of cryptographic algorithms than those existent in the state of the art. Our software implementations render better performance when using AVX2 vector instructions for the X25519 and X448 Diffie-Hellman protocols, and the Ed25519 and Ed448 digital signature schemes. We also identified trade-offs and limitations of these developments, which can provide insights for future improvements.

6.2 Retrospective

We want to provide some observations collected in retrospective about some issues and challenges we faced. We believe these comments are primary input for future developments and research projects.

6.2.1 Specialized Literature

Practitioners on this area have a few resources for the development of secure and efficient software. For instance, several number theory or cryptography books contain descriptions for performing multi-precision arithmetic. Most of them describe the algorithms narrowing to a basic set of computer instructions. However, current processors contain several layers of optimizations for maximizing the number of instructions per clock cycle. Thus, developers lacks of documentation that explains how to take advantage of these hardware extensions since there are almost no resources specialized in their application to cryptography.

In our case, we relied mainly on the official Intel manuals and white papers as well as on the Fog's manuals [109]. All of them are a reliable source of information about SIMD instructions. Another invaluable source of bit tricks we used can be found at [10, 19, 269]. But none of them is actually focused on the security of software. Because of that, we believe our work serves as a guide for developing high-performance cryptographic software covering modern elliptic curve algorithms and the most recent computer architectures.

6.2.2 Programming Languages

On the availability of more computational resources, programming languages should offer support for accessing these resources more easily. For example, the SSE and AVX/AVX2 instructions are available through a library of C functions (also known as intrinsics). However, not all programming languages provide a way for accessing SIMD instructions directly. One solution is given by some advanced compilers that have the ability to issue SIMD instructions; however, although this works well on easily-recognizable workloads, the most common case is resorting to write code in assembly language.

We also found a lack of tools that explicitly leverage the internal parallelism inside the execution engine of processors. Even in the assembler language, the programmer is not able to force the use of certain hardware optimizations such as pipelining or being able to scheduling instructions to different execution units (ports). For example, it would be valuable for the programmer a way for explicitly indicating which instructions can run in pipeline. Armed with these kind of tools, programmers could improve the task of optimizing programs.

Compilation and Measurements

Vector instructions are exposed to the programmer as C functions, which allowed us to write software in C language without recurring to assembler code. We found advantageously to use intrinsics as a clean way for accessing to low-level instructions, and also

because it off-loads to the compiler all the register allocations and the stack management. However, compilers could introduce unexpected effects on the performance of programs with intrinsics.

Compilers have different algorithms for producing assembler code. For example, it is common that compilers reorder instructions and assign registers in different ways. For this reason, vectorized implementations written with intrinsics are susceptible to changes not only on the compiler used, but also to the compiler version. It was common sometimes that a newer compiler version alters (positively or negatively) the performance of a program. Therefore, we recommend always report timings precisely indicating the name of the compiler, its version, and the compiler flags used to enable fair comparisons.

It remains open to provide the programmer with a better way (or tools) for using SIMD instructions. For example, this could be done by unifying the way that compilation of SIMD instructions is performed, in such a way it is reproducible across different compilers.

6.2.3 Usage of AVX2

Limitations of AVX2

During our study we encounter some limitations of the AVX2 vector unit. High-latency instructions were the main obstacle we found for improving performance. The vector integer multiplier and some permutation instructions are notorious cases.

The vector integer multiplication is an expensive instruction. Compounded to that, the lack of a 64-bit vector multiplier imposed a limitation on the representation of field elements using digits shorter than 32 bits. For example, de Valence [85] showed an implementation using a 52-bit multiplier (available in AVX-512-IFMA set) that improves the latency of field operations. To our perspective, a fast, wider integer multiplier is a critical piece for accelerating integer multiplication, and consequently, speeding up high-level operations.

Permutation instructions are mainly used for accommodating words inside vector registers. Moving words between registers can be regarded as negligible or with low impact on the performance; however, it is not the case. Some permutation instructions are three times slower than arithmetic instructions, which produces a negative effect on the performance of operations.

Due to these limitations, a north star that guided our optimizations was to minimize the use of these costly instructions, which limited at a certain degree the way we perform certain operations. If permutation instructions become faster, a better organization of words may lead to faster implementations. For this reason, we reshaped the execution flow and the storage of words in vector registers so more operations can run in parallel.

Prime Field Arithmetic

There are some crucial points that we identified when implementing finite field arithmetic using AVX2 instructions.

The representation of field elements heavily depends on the instruction set. The lack of certain instructions in AVX2 conducted us to design different implementations. For

example, neither the SSE nor the AVX/AVX2 instruction sets include a vector instruction for addition with carry. The absence of this sole functionality avoids (in part) the use of saturated arithmetic. Therefore, we had to find an alternative representation that is more appropriate given the instructions available.

Unsaturated arithmetic is suitable for SIMD processing. A redundant representation has a higher degree of parallelism. In fact, large-integer operations, such as additions, are executed faster when the propagation of carry bits is postponed. Also, the propagation of carry bits is also performed in parallel. We note that it is convenient to maintain operands in this representation, and avoid moving back and forth between representations. One downside of this approach is to manage the growth in size of digits to prevent a loss of precision in the computation. Programmer must keep track of the operands' size and perform a propagation of carry bits from time to time. A systematic method that identifies when to perform propagation is subject of further study.

Applying SIMD Broadly

We showed that SIMD instructions can perform not only finite field operations, but also higher-level operations such as point additions or scalar multiplications. For example, we achieve good performance by processing several point additions simultaneously. Initially, we started with the notion of n -way field operations, and then we extend this idea to construct n -way point additions. We could go further to perform, for example, n -way digital signatures. This parallel approach works well when processing independent batches of signatures (this requires to transpose the inputs to be easily loaded into vector registers). Nonetheless, although batching signatures could be a real workload, the most common case is to process one signature at a time; this is one of the reasons why we focused on reducing the latency of a single signature.

6.2.4 New Instructions

We describe some feature requests for designers of hardware processors that we identify as useful for cryptography.

Dynamic Execution of Vector Instructions

It is common to see that vectorized implementations start by detecting the processor's capabilities through CPUID instruction, and then, branching to different parts of code that are optimized for each instruction set. As a result, a single algorithm has as many implementations as the instructions sets. Having several code paths is more difficult to maintain in the long term.

One proposed solution to this problem is writing code using SIMD vectors of *arbitrary size*. That is, during execution time, the processor will execute these high-order vector instructions using vector (and a combination of scalar) instructions according to the capabilities of the processor. This execution mode is more ambitious since it can be regarded as auto-vectorization at run-time (performed entirely by hardware) rather than performed at compilation time.

Native Support for Fixed-Precision Modular Arithmetic

This feature could be in the form of a set of instructions parametrized by the size (in words) of the integer modulus.

For instance, suppose that AX and BX are pointers to an array of four integers representing field elements, and CX is a pointer to a prime modulus stored in four machine words; then the instruction

$$\text{FP_ADD } AX, BX, CX, 4$$

computes an addition of $AX + BX \bmod CX$.

Analogously, we could include instructions for subtractions and multiplications. An advantage of this approach is that the hardware will be in charge of the correctness and the security (ensuring a constant-time execution). This approach off-loads to the processor a common source of failures and vulnerabilities often found in software implementations of cryptographic algorithms.

Instructions for Handling Keys

Most computer architectures do not provide mechanisms to manipulate secret data. For example, keys are fetched from memory and stored in general-purpose registers, just like any other piece of data. To make a distinction, processors could include secure (restricted-access) registers to store key material rather than using general-purpose registers. The difference relies on the accessing control to the data stored in these registers.

This idea also applies to the instructions accessing key material. For example, in the Montgomery ladder, the bits of the key are used to move elements conditionally. It would be helpful to have an assembler instruction that has exclusive access to the bits of the key. A strong premise is the inclusion of countermeasures against side-channel attacks.

6.3 Summary of Contributions

We give a brief overview of our contributions.

6.3.1 Algorithmic Optimizations

For Montgomery curves, we introduced a new three-point ladder algorithm that calculates the x -coordinate of $P + kQ$. Our algorithm improves in three aspects. First, it requires fewer operations than previously-known algorithms [78, 161]. Second, when P and Q are known in advance, the algorithm allows faster execution employing precomputation. Third, if precomputation is used, fetching data from the pre-computed tables requires non-secret indexes. This algorithm is suitable for the Diffie-Hellman protocol, specially when the points are public and fixed. Also, it improves the sampling of r -torsion points in the SIDH protocol. We showed its immediate application on concrete cryptographic algorithms, such as X25519, X448, qDSA with Montgomery curves, and the SIDH/SIKE protocol. All of these algorithms have better performance when using the three-point ladder algorithm. This improvement is independent of the computer's architecture.

We showed an optimized formula for tripling points on Montgomery curves. This operation is relevant on multi-base scalar multiplication methods and on isogeny-based cryptography. In latter case, SIDH evaluates $3^i P$ for some integer $i > 0$. We proposed an efficient way to compute point tripling, reducing the number of field operations. We acknowledge some trade-offs against formulas independently proposed in [76, 273].

6.3.2 Implementation Techniques

On the availability of extensions to the instruction set architecture, we focus on their efficient application to accelerate the arithmetic of prime fields and elliptic curves.

Implementations using SIMD Extensions

We initially focus on the SIMD parallel processing for field arithmetic. Our study covered four families of prime moduli. For each family, we showed how to perform field operations using scalar and vector instructions. Our benchmark analysis showed that improvements in performance are more significant when operating over larger numbers. For smaller prime fields, the cost of manipulating data inside vector registers resulted in a notorious overhead limiting the amount of improvement.

We show other ways of using SIMD units efficiently by taking the SIMD's approach to higher abstraction levels. We follow the notion of n -way operations using the n words of a vector register for calculating n field operations in parallel. This approach was motivated due to the overheads of using SIMD instructions to perform single field operations. As a direct application of n -way field operations, we turned our attention to investigate parallel algorithms for elliptic curve arithmetic.

For Montgomery curves, we showed how to calculate the Montgomery ladder step using two parallel units. The common implementation strategy for these two models consisted on using the 256-bit AVX2 vector unit for simulating two 128-bit parallel units. Thus, each 128-bit unit can also be seen as two 64-bit parallel units that are dedicated to field arithmetic. By following this approach, we reduce the number of permutation instructions minimizing the overhead observed on the implementation of field arithmetic.

For Edwards curves, we focused on parallel algorithms for point addition, point doubling, and scalar multiplication. Our implementation strategy was to perform 4-way operations using the 256-bit vector unit. Then, we implemented point addition (and doubling) using 4-way field operations. Additionally, we constructed a 4-way point addition that allowed us to perform parts of the scalar multiplication in parallel. The design of all parallel algorithms minimizes the use of costly permutation instructions. We accelerated the calculation of scalar multiplications on Edwards curves. For Weierstrass curves, we split the calculation of the \mathbb{F}_q -complete formulas for point addition and doubling.

With savings in both the prime field arithmetic and the elliptic curve arithmetic, we applied them to some cryptographic algorithms. We developed vectorized implementations of the ECDH and ECDSA with the P-384 curve; the X25519, X448, and SIDH Diffie-Hellman protocols; and the Ed25519 and Ed448 digital signature schemes. In all cases, we observed improvement on performance by using AVX2 vector instructions.

Implementations using other Extensions

In addition to the SIMD extensions, we studied the efficient application of other hardware extensions such as BMI2, ADX, and SHA-NI instruction sets.

We developed efficient implementations of field arithmetic using MULX (from the BMI instruction set) and ADCX/ADOX (from the ADX instruction set) instructions. Using these instructions, our implementations render better performance than using previous basic instructions. Nonetheless, vectorized implementations offer superior improvements in prime fields of larger size.

The availability of SHA-NI allowed us to evaluate the performance of SHA-256. First, we showed a pipelined implementation that performs a 4-way version of the SHA-256 function. We applied this function to the XMSS and XMSS^{MT} hash-based signatures. Using SHA-NI, we observed that signature operations run up to four times faster than using a non-hardware aided implementation. We also show that its performance is slightly better than implementing 4-way SHA-256 with a SIMD vector instructions.

6.4 Future Work

In some parts of the document we have indicated several paths for further research. Now we made additional suggestions that can provide a better understanding of the SIMD instructions and their usage in other applications.

The number of extensions for cryptographic applications is increasing. As shown in Chapter 2, processors will be able to perform vectorized AES and Galois field operations. Although some estimations of performance were provided by Drucker et al. [89] based on the number of instructions, it remains unknown to what extent these new instructions accelerate data encryption.

More SIMD extensions are released in the AVX-512 instruction set. It is interesting to investigate the performance of these new instructions, and also their use in cryptographic algorithms, and in particular, to elliptic curve arithmetic.

Most of the proposals contending at the NIST Post-Quantum competition [206] present implementations accelerated with vector instructions. In this case, the workload of algorithms is quite diverse, which makes that the requirements, implementation techniques, and optimizations be too specific rather than general. It is interesting to see more applications of the AVX-512 instructions to quantum-resistant algorithms. For example, the optimizations proposed for SIDH/SIKE can also be extended to use AVX-512. We developed a pipelined implementation of SHA-256 for XMSS and XMSS^{MT}; however, other algorithms such as SPHINCS [32] can also benefit from this implementation.

We hope our work and the ideas here presented motivate future projects, students, and researchers. We encourage them to extend and improve our results augmenting the knowledge base of this gratifying field of study called cryptography.

Bibliography

- [1] Tolga Acar and Dan Shumow. Modular Reduction without Pre-computation for Special Moduli. <https://www.microsoft.com/en-us/research/publication/modular-reduction-without-pre-computation-for-special-moduli>, January 2010. Cited in page 62.
- [2] Onur Aciğmez. Fast hashing on Pentium SIMD architecture. Master's thesis, Oregon State University, 2005. <http://hdl.handle.net/1957/11799>. Cited in 2 pages: 208 and 209.
- [3] Gora Adj, Daniel Cervantes-Vázquez, Jesús J. Chi-Domínguez, Alfred Menezes, and Francisco Rodríguez-Henríquez. On the Cost of Computing Isogenies Between Supersingular Elliptic Curves. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 322–343, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-10970-7_15. Cited in page 122.
- [4] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540, May 2013. doi:10.1109/SP.2013.42. Cited in page 24.
- [5] Shoukat Ali and Murat Cenk. A New Algorithm for Residue Multiplication Modulo $2^{521} - 1$. In Seokhie Hong and Jong Hwan Park, editors, *Information Security and Cryptology – ICISC 2016*, pages 181–193, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-53177-9_9. Cited in page 63.
- [6] AMD. AMD64 Technology 128-bit SSE5 Instruction Set, August 2007. http://developer.amd.com/wordpress/media/2012/10/AMD64_128_Bit_SSE5_Instrs.pdf. Cited in page 45.
- [7] AMD. 3DNow! Instructions are Being Deprecated. <http://developer.amd.com/community/blog/3dnow-deprecated/>, aug 2010. Cited in page 44.
- [8] American National Standards Institute. Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical Report ANSI X9.62-1998, 1998. <https://webstore.ansi.org/Standards/ASCX9/ANSIX9621998>. Cited in 4 pages: 26, 31, 138, and 171.
- [9] American National Standards Institute. Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve

- Cryptography. Technical Report ANSI X9.63-2001, November 2001. <https://webstore.ansi.org/Standards/ASCX9/ANSIX9632001>. Cited in 3 pages: 26, 31, and 171.
- [10] Sean Eron Anderson. Bit Twiddling Hacks, 2005. <https://graphics.stanford.edu/~seander/bithacks.html>. Cited in page 216.
- [11] Kazumaro Aoki, Fumitaka Hoshino, Tetsutaro Kobayashi, and Hiroaki Oguro. Elliptic Curve Arithmetic Using SIMD. In George I. Davida and Yair Frankel, editors, *Information Security*, pages 235–247, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-45439-X_16. Cited in page 141.
- [12] Diego F. Aranha, Paulo S. L. M. Barreto, Geovandro C. C. F. Pereira, and Jefferson E. Ricardini. A note on high-security general-purpose elliptic curves. Cryptology ePrint Archive, Paper 2013/647, 2013. <https://eprint.iacr.org/2013/647>. Cited in page 26.
- [13] Diego F. Aranha, Armando Faz-Hernández, Julio López, and Francisco Rodríguez-Henríquez. Faster Implementation of Scalar Multiplication on Koblitz Curves. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATIN-CRYPT 2012*, pages 177–193, Berlin, Heidelberg, October 2012. Springer. doi:10.1007/978-3-642-33481-8_10. Cited in 2 pages: 49 and 180.
- [14] Diego F. Aranha and Conrado. P. L. Gouvêa. RELIC is an Efficient Library for Cryptography. <https://github.com/relic-toolkit/relic>. Cited in page 174.
- [15] Diego F. Aranha, Julio López, and Darrel Hankerson. Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *Progress in Cryptology – LATIN-CRYPT 2010*, pages 144–161, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-14712-8_9. Cited in page 27.
- [16] Christophe Arene, David Kohel, and Christophe Ritzenthaler. Complete addition laws on abelian varieties. *LMS Journal of Computation and Mathematics*, 15:308–316, 2012. doi:10.1112/S1461157012001027. Cited in 2 pages: 127 and 140.
- [17] ARM. ARM NEON Intrinsics. <https://developer.arm.com/architectures/instruction-sets/intrinsics>. Cited in page 44.
- [18] ARM. ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile, December 2017. <https://developer.arm.com/documentation/ddi0487/ca>. Cited in page 214.
- [19] Jörg Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. Springer, Berlin Heidelberg, 1 edition, 2011. doi:10.1007/978-3-642-14764-7_1. Cited in page 216.
- [20] Paul Bakker. mbed TLS, 2008. <https://tls.mbed.org/>. Cited in page 174.

- [21] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86: Proceedings*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. doi:10.1007/3-540-47721-7_24. Cited in page 59.
- [22] Daniel J. Bernstein. Cache-timing attacks on AES, November 2004. <http://cr.yp.to/papers.html#cachetiming>. Cited in page 137.
- [23] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. doi:10.1007/11745853_14. Cited in 8 pages: 26, 28, 29, 31, 150, 175, 176, and 177.
- [24] Daniel J. Bernstein. Differential addition chains, February 2006. <https://cr.yp.to/ecdh/diffchain-20060219.pdf>. Cited in page 198.
- [25] Daniel J. Bernstein. Batch Binary Edwards. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009: 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, pages 317–336, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-03356-8_19. Cited in page 73.
- [26] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards Curves. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-68164-9_26. Cited in 3 pages: 125, 161, and 163.
- [27] Daniel J. Bernstein, Chitchanok Chuengsatiansup, David Kohel, and Tanja Lange. Twisted Hessian Curves. In Kristin Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, volume 9230 of *Lecture Notes in Computer Science*, pages 269–294. Springer International Publishing, 2015. doi:10.1007/978-3-319-22174-8_15. Cited in 2 pages: 28 and 125.
- [28] Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba Revisited. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014*, volume 8731, pages 316–334. Springer Berlin Heidelberg, September 2014. doi:10.1007/978-3-662-44709-3_18. Cited in page 26.
- [29] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Peter Schwabe. Kummer Strikes Back: New DH Speed Records. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, volume 8873 of *Lecture Notes in Computer Science*, pages 317–337. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-662-45611-8_17. Cited in 2 pages: 28 and 136.

- [30] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-Speed High-Security Signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-23951-9_9. Cited in 2 pages: 183 and 184.
- [31] 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. doi:10.1007/s13389-012-0027-1. Cited in 7 pages: 28, 31, 138, 166 (2), 167, 168 (2), and 183.
- [32] Daniel J Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O’Hearn. SPHINCS: practical stateless hash-based signatures. In *Advances in Cryptology – EUROCRYPT 2015*, pages 368–397, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46800-5_15. Cited in 2 pages: 214 and 221.
- [33] Daniel J. Bernstein, Simon Josefsson, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. EdDSA for more curves. Cryptology ePrint Archive, Paper 2015/677, 2015. <https://eprint.iacr.org/2015/677>. Cited in page 184.
- [34] Daniel J. Bernstein and Tanja Lange. Faster Addition and Doubling on Elliptic Curves. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-76900-2_3. Cited in page 161.
- [35] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. Accessed on 20 March 2015, March 2015. <http://bench.cr.yp.to/supercop.html>. Cited in 5 pages: 180, 181, 188 (6), 206, and 211.
- [36] Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. <http://safecurves.cr.yp.to> accessed 20 March 2015, 2015. Cited in 5 pages: 26, 128, 175, 183, and 189.
- [37] Daniel J. Bernstein and Tanja Lange. Montgomery Curves and the Montgomery Ladder. In Joppe W. Bos and Arjen K.Editors Lenstra, editors, *Topics in Computational Number Theory Inspired by Peter L. Montgomery*, pages 82–115. Cambridge University Press, 2017. doi:10.1017/9781316271575.005. Cited in 2 pages: 147 and 198.
- [38] Daniel J. Bernstein and Peter Schwabe. NEON Crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-33027-8_19. Cited in page 28.

- [39] Ward Beullens, Thorsten Kleinjung, and Frederik Vercauteren. CSI-FiSh: Efficient Isogeny Based Signatures Through Class Group Computations. In Steven D. Galbraith and Shihō Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019*, pages 227–247, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-34578-5_9. Cited in page 197.
- [40] Olivier Billet and Marc Joye. The Jacobi Model of an Elliptic Curve and Side-Channel Analysis. In Marc Fossorier, Tom Høholdt, and Alain Poli, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 34–42, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/3-540-44828-4_5. Cited in page 125.
- [41] G. R. Blakley. A Computer Algorithm for Calculating the Product AB Modulo M . *IEEE Transactions on Computers*, C-32(5):497–500, May 1983. doi:10.1109/TC.1983.1676262. Cited in page 59.
- [42] Erich Bloch. The Engineering Design of the Stretch Computer. In *Proceedings of the Eastern Joint Computer Conference*, pages 48–58, New York, NY, USA, December 1959. Association for Computing Machinery. doi:10.1145/1460299.1460304. Cited in page 38.
- [43] Manuel Bluhm and Shay Gueron. Fast software implementation of binary elliptic curve cryptography. *Journal of Cryptographic Engineering*, 5(3):215–226, 2015. doi:10.1007/s13389-015-0094-1. Cited in page 28 (2).
- [44] Joppe W. Bos. Constant time modular inversion. *Journal of Cryptographic Engineering*, 4(4):275–281, Nov 2014. doi:10.1007/s13389-014-0084-8. Cited in page 61.
- [45] Joppe W. Bos, Craig Costello, Huseyin Hisil, and Kristin Lauter. Fast Cryptography in Genus 2. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 194–210. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38348-9_12. Cited in 2 pages: 26 and 28.
- [46] Joppe W. Bos, Craig Costello, Huseyin Hisil, and Kristin Lauter. High-Performance Scalar Multiplication Using 8-Dimensional GLV/GLS Decomposition. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 331–348. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-40349-1_19. Cited in page 28 (2).
- [47] 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. doi:10.1007/s13389-015-0097-y. Cited in page 173.

- [48] Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Specification of Curve Selection and Supported Curve Parameters in MSR ECCLib. Technical report, Microsoft Research, June 2015. <https://www.microsoft.com/en-us/research/publication/specification-of-curve-selection-and-supported-curve-parameters-in-msr-ecclib/>. Cited in page 26.
- [49] Joppe W. Bos and Simon J. Friedberger. Fast Arithmetic Modulo $2^x p^y - 1$. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pages 148–155, July 2017. doi:10.1109/ARITH.2017.15. Cited in 4 pages: 62, 113, 114, and 116 (3).
- [50] Joppe W. Bos and Simon J. Friedberger. Faster Modular Arithmetic For Isogeny Based Crypto on Embedded Devices. *Journal of Cryptographic Engineering*, 10, April 2019. doi:10.1007/s13389-019-00214-6. Cited in page 116.
- [51] Joppe W. Bos and Thorsten Kleinjung. ECM at Work. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 467–484. Springer, 2012. doi:10.1007/978-3-642-34961-4_29. Cited in page 42.
- [52] W. Bosma and H. W. Lenstra, Jr. Complete Systems of Two Addition Laws for Elliptic Curves. *Journal of Number Theory*, 53(2):229–240, 1995. doi:10.1006/jnth.1995.1088. Cited in 2 pages: 127 and 140.
- [53] W. J. Bouknight, S.A. Denenberg, D.E. McIntyre, J. M. Randall, A.H. Sameh, and D.L. Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, 1972. doi:10.1109/PROC.1972.8647. Cited in page 41.
- [54] H. C. Brearley. ILLIAC II-A Short Description and Annotated Bibliography. *IEEE Transactions on Electronic Computers*, EC-14(3):399–403, 1965. doi:10.1109/PGEC.1965.264146. Cited in page 38.
- [55] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David B. Wilson. Fast Exponentiation with Precomputation. In Rainer A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT’ 92*, pages 200–207, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. doi:10.1007/3-540-47555-9_18. Cited in page 137.
- [56] Éric Brier and Marc Joye. Weierstraß Elliptic Curves and Side-Channel Attacks. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography*, pages 335–345, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45664-3_24. Cited in page 149.
- [57] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. In *Proceedings of the 12th Conference on USENIX Security Symposium*, SSYM’03, pages 1–13. USENIX Association, August 2003. doi:10.5555/1251353.1251354. Cited in page 24.


- [58] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, pages 117–129, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25405-5_8. Cited in 2 pages: 31 and 208 (2).
- [59] Chris K. Caldwell. The Prime Glossary, December 2016. <http://primes.utm.edu/glossary/xpage/PierpontPrime.html>. Cited in page 198.
- [60] Danilo Câmara, Conrado P. L. Gouvêa, Julio López, and Ricardo Dahab. Fast Software Polynomial Multiplication on ARM Processors Using the NEON Engine. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics*, pages 137–154, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40588-4_10. Cited in page 28.
- [61] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Paper 2022/975, 2022. <https://eprint.iacr.org/2022/975>. Cited in page 197.
- [62] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An Efficient Post-Quantum Commutative Group Action. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 395–427, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-03332-3_15. Cited in 2 pages: 122 and 197.
- [63] Denis X Charles, Kristin E Lauter, and Eyal Z Goren. Cryptographic hash functions from expander graphs. *Journal of Cryptology*, 22(1):93–113, January 2009. doi:10.1007/s00145-007-9002-x. Cited in page 197.
- [64] Tung Chou. Sandy2x: New Curve25519 Speed Records. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography – SAC 2015*, pages 145–160, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-31301-6_8. Cited in 8 pages: 28, 29, 167, 168, 177, 179, 184, and 196.
- [65] Dalin Chu, Johann Großschädl, Zhe Liu, Volker Müller, and Yang Zhang. Twisted Edwards-form Elliptic Curve Cryptography for 8-bit AVR-based Sensor Nodes. In *Proceedings of the First ACM Workshop on Asia Public-key Cryptography*, AsiaPKC ’13, pages 39–44, New York, NY, USA, 2013. ACM. doi:10.1145/2484389.2484398. Cited in page 62.
- [66] D. V. Chudnovsky and G. V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, 7(4):385–434, 1986. doi:10.1016/0196-8858(86)90023-0. Cited in 3 pages: 125, 139, and 140.
- [67] Neill Michael Clift. Calculating optimal addition chains. *Computing*, 91(3):265–284, Mar 2011. doi:10.1007/s00607-010-0118-8. Cited in page 61.

- [68] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient Elliptic Curve Exponentiation Using Mixed Coordinates. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology — ASIACRYPT'98*, pages 51–65, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. doi:10.1007/3-540-49649-1_6. Cited in 2 pages: 139 and 140.
- [69] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990. doi:10.1147/sj.294.0526. Cited in page 69.
- [70] Intel Corporation. Prescott New Instructions Software Developer's Guide, June 2003. Cited in page 38.
- [71] Intel Corporation. Intel® Advanced Vector Extensions Programming Reference. <https://software.intel.com/sites/default/files/m/f/7/c/36945>, June 2011. Cited in page 177.
- [72] Intel Corporation. Intel Instruction Set Architecture Extensions. Available at <https://software.intel.com/en-us/intel-isa-extensions>, July 2013. Cited in page 214.
- [73] Intel Corporation. Intel® Architecture Instruction Set Extensions and Future Features Programming Reference. Technical Report 319433-033, Intel Corporation, March 2018. <https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf>. Cited in page 51.
- [74] Intel Corporation. Intel® C/C++ Compiler. January 2018. <https://software.intel.com/en-us/c-compilers>. Cited in 2 pages: 31 and 42.
- [75] Intel Corporation. Intel® Intrinsics Guide, February 2022. <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>. Cited in 2 pages: 31 and 47.
- [76] Craig Costello and Huseyin Hisil. A Simple and Compact Algorithm for SIDH with Arbitrary Degree Isogenies. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 303–329, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-70697-9_11. Cited in 3 pages: 160, 161, and 220.
- [77] Craig Costello and Patrick Longa. FourQ: Four-Dimensional Decompositions on a \mathbb{Q} -curve over the Mersenne Prime. In Tetsu Iwata and Hee Jung Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 214–235, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-48797-6_10. Cited in 5 pages: 26, 63, 64, 180, and 182 (2).
- [78] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient Algorithms for Supersingular Isogeny Diffie-Hellman. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 572–601, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. doi:10.1007/978-3-662-53018-4_21. Cited in 9 pages: 113, 117, 121 (2), 160, 161, 198, 199, 201, and 219.

- [79] Richard Crandall and Carl Pomerance. *Prime Numbers*. Springer, New York, NY, 2nd edition. doi:10.1007/0-387-28979-8. Cited in 2 pages: 27 and 63.
- [80] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998. doi:10.1109/99.660313. Cited in page 41.
- [81] Luca De Feo. Software for “Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies”, 2011. <http://github.com/defeo/ss-isogeny-software>. Cited in page 198.
- [82] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact Post-quantum Signatures from Quaternions and Isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 64–93, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-64837-4_3. Cited in page 197.
- [83] Ana Karina D. S. de Oliveira and Julio López. An Efficient Software Implementation of the Hash-Based Signature Scheme MSS and Its Variants. In Kristin Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, pages 366–383, Guadalajara, Mexico, August 2015. Springer International Publishing. doi:10.1007/978-3-319-22174-8_20. Cited in page 212.
- [84] Henry de Valence. Accelerating Edwards Curve Arithmetic with Parallel Formulas. Article at A Medium Corporation, August 2018. <https://medium.com/@hdevalence/accelerating-edwards-curve-arithmetic-with-parallel-formulas-ac12cf5015be>. Cited in page 29.
- [85] Henry de Valence. Even faster Edwards curves with IFMA. Article at A Medium Corporation, December 2018. <https://medium.com/@hdevalence/even-faster-edwards-curves-with-ifma-8b1e576a00e9>. Cited in 2 pages: 29 and 217.
- [86] K. Diefendorff and P.K. Dubey. How multimedia workloads will change processor design. *Computer*, 30(9):43–45, 1997. doi:10.1109/2.612247. Cited in page 44.
- [87] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976. doi:10.1109/TIT.1976.1055638. Cited in page 21 (2).
- [88] Jason A. Donenfeld. Wireguard Linux Compat. Patch to Wireguard, February 2018. https://git.zx2c4.com/wireguard-linux-compat/commit/src/crypto/curve25519-x86_64.h?id=186be2742c948351c27bc068102252e10a28959b. Cited in page 34.
- [89] Nir Drucker, Shay Gueron, and Vlad Krasnov. Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 115–119, June 2018. doi:10.1109/ARITH.2018.8464777. Cited in page 221.

- [90] Harold M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, July 2007. Cited in 2 pages: 125 and 161.
- [91] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13*, pages 73–84, New York, NY, USA, 2013. ACM. doi:10.1145/2508859.2516693. Cited in page 192.
- [92] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. pages 10–18, 1985. doi:10.1007/3-540-39568-7_2. Cited in page 22.
- [93] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12*, pages 50–61, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2382196.2382205. Cited in page 192.
- [94] Junfeng Fan, Benedikt Gierlichs, and Frederik Vercauteren. To Infinity and Beyond: Combined Attack on ECC Using Points of Low Order. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 143–159, Berlin, Heidelberg, October 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-23951-9_10. Cited in page 191.
- [95] Armando Faz-Hernández, Hayato Fujii, Diego F. Aranha, and Julio López. A Secure and Efficient Implementation of the Quotient Digital Signature Algorithm (qDSA). In Sk Subidh Ali, Jean-Luc Danger, and Thomas Eisenbarth, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 170–189, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-71501-8_10. Cited in page 189.
- [96] 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 Josh Benaloh, editor, *Topics in Cryptology – CT-RSA 2014*, volume 8366 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2014. doi:10.1007/978-3-319-04852-9_1. Cited in page 137.
- [97] 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 (extended version). *J. Cryptographic Engineering*, 5(1):31–52, 2015. doi:10.1007/s13389-014-0085-7. Cited in 2 pages: 28 and 180.
- [98] Armando Faz-Hernández and Julio López. On Software Implementation of Arithmetic Operations on Prime Fields using AVX2. In Jeroen van de Graaf, José Marcos Nogueira, and Leonardo Barbosa Oliveira, editors, *Anais: XIV Simpósio Brasileiro*

- de Segurança da Informação e de Sistemas Computacionais*, volume 14, pages 338–341, Belo Horizonte, November 2014. Sociedade Brasileira de Computação - SBC. doi:10.5753/sbseg.2014.20148. Cited in page 177.
- [99] Armando Faz-Hernández and Julio López. Fast Implementation of Curve25519 Using AVX2. In Kristin Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, volume 9230 of *Lecture Notes in Computer Science*, pages 329–345. Springer International Publishing, August 2015. doi:10.1007/978-3-319-22174-8_18. Cited in 3 pages: 175, 178, and 179.
- [100] Armando Faz-Hernández and Julio López. Speeding up Elliptic Curve Cryptography on the P-384 Curve. In *XVI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, volume 16, pages 170–183. Sociedade Brasileira de Computação – SBC, Nov 2016. doi:10.5753/sbseg.2016.19306. Cited in page 170.
- [101] Armando Faz-Hernández, Julio López, and Ricardo Dahab. High-performance Implementation of Elliptic Curve Cryptography Using Vector Instructions. *ACM Transactions on Mathematical Software (TOMS)*, 45(3):1–35, July 2019. doi:10.1145/3309759. Cited in 4 pages: 175, 179 (3), 180, and 182.
- [102] Armando Faz-Hernandez, Julio López, and Ana Karina D. S. de Oliveira. SoK: A Performance Evaluation of Cryptographic Instruction Sets on Modern Architectures. In *Proceedings of the 5th ACM on ASIA Public-Key Cryptography Workshop*, APKC '18, pages 9–18, New York, NY, USA, June 2018. ACM. doi:10.1145/3197507.3197511. Cited in page 208.
- [103] Armando Faz-Hernández, Julio López, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A Faster Software Implementation of the Supersingular Isogeny Diffie-Hellman Key Exchange Protocol. *IEEE Transactions on Computers*, 67(11):1622–1636, Nov 2018. doi:10.1109/TC.2017.2771535. Cited in 4 pages: 114, 156, 197, and 201 (2).
- [104] Min Feng, Bin B. Zhu, Cunlai Zhao, and Shipeng Li. Signed MSB-Set Comb Method for Elliptic Curve Point Multiplication. In Kefei Chen, Robert Deng, Xuejia Lai, and Jianying Zhou, editors, *Information Security Practice and Experience: Second International Conference, ISPEC 2006, Hangzhou, China, April 11-14, 2006. Proceedings*, pages 13–24, Berlin, Heidelberg, April 2006. Springer Berlin Heidelberg. doi:10.1007/11689522_2. Cited in page 137.
- [105] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, Inc., Indianapolis, Indiana, 2010. Cited in page 24.
- [106] Achim Flammenkamp. Shortest Addition Chains, November 2016. http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html. Cited in page 61.

- [107] Michael Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec 1966. doi:10.1109/PROC.1966.5273. Cited in 2 pages: 29 and 40.
- [108] Michael Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972. doi:10.1109/TC.1972.5009071. Cited in 2 pages: 29 and 40.
- [109] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, December 2014. Cited in 2 pages: 209 and 216.
- [110] Freescale Semiconductor. *AltiVec Technology Programming Interface Manual*, June 1999. <https://www.nxp.com/docs/en/reference-manual/ALTIVECPIM.pdf>. Cited in page 44.
- [111] Steven D. Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 518–535. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-01001-9_30. Cited in 2 pages: 28 and 64.
- [112] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-44647-8_11. Cited in 4 pages: 28, 137, 138, and 185.
- [113] Pierrick Gaudry. Fast genus 2 arithmetic based on Theta functions. *J. Mathematical Cryptology*, 1(3):243–265, 2007. doi:10.1515/JMC.2007.012. Cited in page 28.
- [114] Pierrick Gaudry and David Lubicz. The arithmetic of characteristic 2 Kummer surfaces and of elliptic Kummer lines. *Finite Fields and Their Applications*, 15(2):246–260, 2009. doi:10.1016/j.ffa.2008.12.006. Cited in page 125.
- [115] GNU Project. GCC, the GNU Compiler Collection. <http://www.gnu.org/software/gcc/gcc.html>. Cited in 2 pages: 31 and 42.
- [116] Google. BoringSSL, October 2015. <https://boringssl.googlesource.com/boringssl/+fe47ba2fc5512436696f745b5756d08c7d8ceb0b>. Cited in page 174.
- [117] Vinodh Gopal, Sean Gulley, Wajdi Feghali, Dan Zimmerman, and Ilya Albrekht. Improving OpenSSL Performance. Technical report, Intel Corporation, May 2015. <https://software.intel.com/en-us/articles/improving-openssl-performance>. Cited in page 209.
- [118] Vinodh Gopal, Jim Gullford, Wajdi Feghali, Erdinc Ozturk, Gil Wolrich, and Martin Dixon. Processing Multiple Buffers in Parallel to Increase Performance on  Intel

- Architecture Processors. Technical Report 324101, Intel Corporation, July 2010. <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-multi-buffer-paper.pdf>. Cited in page 208.
- [119] Raveen R. Goundar, Marc Joye, and Atsuko Miyaji. Co-Z Addition Formulæ and Binary Ladders on Elliptic Curves. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 65–79, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-15031-9_5. Cited in page 133.
- [120] Conrado P. L. Gouvêa and Julio López. Implementing GCM on ARMv8. In Kaisa Nyberg, editor, *Topics in Cryptology — CT-RSA 2015*, pages 167–180, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-16715-2_9. Cited in page 28.
- [121] Robert Granger and Michael Scott. Faster ECC over $\mathbb{F}_{2^{521}-1}$. In Jonathan Katz, editor, *Public-Key Cryptography – PKC 2015*, pages 539–553, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46447-2_24. Cited in 3 pages: 27, 63, and 172.
- [122] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. <http://gmplib.org/>. Cited in page 65.
- [123] Johann Großschädl. TinySA: A Security Architecture for Wireless Sensor Networks. In *Proceedings of the 2006 ACM CoNEXT Conference*, CoNEXT '06, pages 55:1–55:2, New York, NY, USA, 2006. ACM. doi:10.1145/1368436.1368500. Cited in page 62.
- [124] Johann Großschädl, Roberto M. Avanzi, Erkay Savaş, and Stefan Tillich. Energy-Efficient Software Implementation of Long Integer Modular Arithmetic. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005: 7th International Workshop, Edinburgh, UK, August 29 – September 1, 2005. Proceedings*, pages 75–90, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11545262_6. Cited in page 112.
- [125] Shay Gueron. Intel’s New AES Instructions for Enhanced Performance and Security. In Orr Dunkelman, editor, *Fast Software Encryption: 16th International Workshop, FSE 2009 Leuven, Belgium, February 22-25, 2009 Revised Selected Papers*, pages 51–66, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-03317-9_4. Cited in 3 pages: 27, 28 (2), and 210.
- [126] Shay Gueron. Efficient software implementations of modular exponentiation. *Journal of Cryptographic Engineering*, 2(1):31–43, 2012. doi:10.1007/s13389-012-0031-5. Cited in page 29.
- [127] Shay Gueron and Michael Kounavis. Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. *Information*

- Processing Letters*, 110(14):549–553, 2010. doi:10.1016/j.ipl.2010.04.011. Cited in 2 pages: 28 and 49.
- [128] Shay Gueron and Vlad Krasnov. Parallelizing message schedules to accelerate the computations of hash functions. *Journal of Cryptographic Engineering*, 2(4):241–253, Nov 2012. doi:10.1007/s13389-012-0037-z. Cited in 3 pages: 206, 208, and 209 (2).
- [129] Shay Gueron and Vlad Krasnov. Simultaneous Hashing of Multiple Messages. *Journal of Information Security*, 3(4):319–325, October 2012. doi:10.4236/jis.2012.34039. Cited in page 209.
- [130] Shay Gueron and Vlad Krasnov. Software Implementation of Modular Exponentiation, Using Advanced Vector Instructions Architectures. In Ferruh Özbudak and Francisco Rodríguez-Henríquez, editors, *Arithmetic of Finite Fields*, pages 119–135, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-31662-3_9. Cited in page 29.
- [131] Shay Gueron and Vlad Krasnov. Speeding Up Big-Numbers Squaring. In *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*, pages 821–823, April 2012. doi:10.1109/ITNG.2012.61. Cited in page 29.
- [132] Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *Journal of Cryptographic Engineering*, pages 1–11, 2014. doi:10.1007/s13389-014-0090-x. Cited in 6 pages: 27, 29, 62 (2), 113, 172, and 182.
- [133] Shay Gueron and Vlad Krasnov. Accelerating Big Integer Arithmetic Using Intel IFMA Extensions. In *2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH)*, pages 32–38, Santa Clara, CA, USA, July 2016. IEEE. doi:10.1109/ARITH.2016.22. Cited in 2 pages: 52 and 214.
- [134] Shay Gueron and Vlad Krasnov. Speed Records for Multi-prime RSA Using AVX2 Architectures. In Shahram Latifi, editor, *Information Technology: New Generations: 13th International Conference on Information Technology*, pages 237–245, Cham, March 2016. Springer International Publishing. doi:10.1007/978-3-319-32467-8_22. Cited in page 29.
- [135] Jim Guilford, Kirk Yap, and Vinodh Gopal. Fast SHA-256 Implementations on Intel ® Architecture Processors. Technical Report 327457-001, Intel Corporation, May 2012. <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/sha-256-implementations-paper.pdf>. Cited in page 206.
- [136] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Gullford, and Gil Wolrich. Intel ® SHA Extensions New Instructions Supporting the Secure Hash Algorithm on Intel ® Architecture Processors. Technical report, Intel Corporation, July 2013. <https://software.intel.com/sites/default/files/article/402097/intel-sha-extensions-white-paper.pdf>. Cited in 2 pages: 51 and 202.

- [137] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 119–132, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-28632-5_9. Cited in page 60.
- [138] Richard K. Guy. *Prime Numbers*, pages 3–43. Springer New York, New York, NY, 1994. doi:10.1007/978-1-4899-3585-4_2. Cited in page 62.
- [139] Mike Hamburg. Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Paper 2012/309, 2012. <http://eprint.iacr.org/2012/309>. Cited in 4 pages: 28, 137, 176, and 184.
- [140] Mike Hamburg. Ed448-Goldilocks, February 2014. <http://ed448goldilocks.sourceforge.net>. Cited in 8 pages: 26, 28, 29, 176, 180 (2), 184, 187, and 188 (2).
- [141] Mike Hamburg. Ed448-Goldilocks, a new elliptic curve. In *Workshop on Elliptic Curve Cryptography Standards*, Gaithersburg, USA, June 2015. NIST. <http://csrc.nist.gov/groups/ST/ecc-workshop-2015/papers/session7-hamburg-michael.pdf>. Cited in 3 pages: 63, 107, and 176.
- [142] Darrel Hankerson, Koray Karabina, and Alfred Menezes. Analyzing the Galbraith-Lin-Scott Point Multiplication Method for Elliptic Curves over Binary Fields. *IEEE Transactions on Computers*, 58(10):1411–1420, October 2009. doi:10.1109/TC.2009.61. Cited in page 26.
- [143] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. Cited in 2 pages: 91 and 138.
- [144] Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau. A comb method to render ECC resistant against Side Channel Attacks. Cryptology ePrint Archive, Report 2004/342, December 2004. <http://eprint.iacr.org/2004/342>. Cited in page 137.
- [145] Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau. Countermeasures for Preventing Comb Method Against SCA Attacks. In Robert H. Deng, Feng Bao, HweeHwa Pang, and Jianying Zhou, editors, *Information Security Practice and Experience*, pages 85–96, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/978-3-540-31979-5_8. Cited in page 137.
- [146] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. doi:10.5555/1999263. Cited in 4 pages: 37, 38, 39, and 41.
- [147] Huseyin Hisil and Joost Renes. On Kummer Lines with Full Rational 2-Torsion and Their Usage in Cryptography. *ACM Trans. Math. Softw.*, 45(4), December 2019. doi:10.1145/3361680. Cited in page 190.

- [148] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards Curves Revisited. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-89255-7_20. Cited in 5 pages: 150, 152, 162, 163 (2), and 164.
- [149] Jeffrey Hoffstein, Jill Pipher, and Joseph .H. Silverman. *An Introduction to Mathematical Cryptography*. Springer, New York, NY, 2008. doi:10.1007/978-0-387-77993-5. Cited in page 22.
- [150] Zhi Hu, Patrick Longa, and Maozhi Xu. Implementing the 4-dimensional GLV method on GLS elliptic curves with j-invariant 0. *Designs, Codes and Cryptography*, 63(3):331–343, 2012. doi:10.1007/s10623-011-9558-1. Cited in page 28.
- [151] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, May 2018. doi:10.17487/RFC8391. Cited in page 208 (2).
- [152] Andreas Hülsing. W-OTS⁺ – Shorter Signatures for Hash-Based Signature Schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, pages 173–188, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-38553-7_10. Cited in page 208.
- [153] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal Parameters for XMSS^{MT}. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics*, pages 194–208, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40588-4_14. Cited in 2 pages: 31 and 208 (2).
- [154] Michael Hutter and Erich Wenger. Fast Multi-precision Multiplication for Public-Key Cryptography on Embedded Microprocessors. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 459–474, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-23951-9_30. Cited in page 69.
- [155] IEEE Standard Specifications for Public-Key Cryptography, August 2000. doi:10.1109/IEEESTD.2000.92292. Cited in page 26.
- [156] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78(3):171–177, 1988. doi:10.1016/0890-5401(88)90024-7. Cited in 4 pages: 61, 81, 98, and 103.
- [157] Tetsuya Izu and Tsuyoshi Takagi. Fast Elliptic Curve Multiplications with SIMD Operations. In Robert Deng, Feng Bao, Jianying Zhou, and Sihan Qing, editors, *Information and Communications Security*, pages 217–230, Berlin, Heidelberg, 2002. Springer. Cited in page 141.

- [158] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. Practical Invalid Curve Attacks on TLS-ECDH. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*, pages 407–425, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-24174-6_21. Cited in page 192.
- [159] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, and Geovandro Pereira. Supersingular Isogeny Key Encapsulation, November 2017. <https://sike.org>. Cited in 3 pages: 34, 121 (2), and 197.
- [160] David Jao and Luca De Feo. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 – December 2, 2011. Proceedings*, pages 19–34, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-25405-5_2. Cited in 7 pages: 26, 155, 156, 157, 158, 197 (2), and 198.
- [161] David Jao, Luca De Feo, and Jérôme Plût. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. *Journal of Mathematical Cryptology*, 8(3):209–247, September 2014. doi:10.1007/978-3-642-25405-5_2. Cited in 2 pages: 197 and 219.
- [162] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017. doi:10.17487/rfc8032. Cited in 3 pages: 31, 104, and 184.
- [163] Marc Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 135–147, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74735-2_10. Cited in 2 pages: 133 and 153.
- [164] Marc Joye and Jean-Jacques Quisquater. Hessian Elliptic Curves and Side-Channel Attacks. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 402–410, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-44709-1_33. Cited in page 28.
- [165] Marc Joye and Michael Tunstall. Exponent Recoding and Regular Exponentiation Algorithms. In Bart Preneel, editor, *Progress in Cryptology – AFRICACRYPT 2009*, pages 334–349, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-02384-2_21. Cited in page 132.

- [166] Koray Karabina. Point Decomposition Problem in Binary Elliptic Curves. In Soonhak Kwon and Aaram Yun, editors, *Information Security and Cryptology - ICISC 2015*, pages 155–168, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-30840-1_10. Cited in page 26.
- [167] Sabyasachi Karati and Palash Sarkar. Kummer for Genus One over Prime Order Fields. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017*, pages 3–32, Cham, 2017. Springer. doi:10.1007/978-3-319-70697-9_1. Cited in page 182 (2).
- [168] A. Karatsuba and Yu. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics—Doklady*, 7(7):595–596, January 1963. Cited in 3 pages: 58, 72, and 106.
- [169] Emilia Käsper. Fast Elliptic Curve Cryptography in OpenSSL. In George Danezis, Sven Dietrich, and Kazue Sako, editors, *Financial Cryptography and Data Security*, volume 7126 of *Lecture Notes in Computer Science*, pages 27–39. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-29889-9_4. Cited in 2 pages: 27 and 172.
- [170] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Cryptography and Network Security Series. Chapman & Hall/CRC, 1st edition, 2007. Cited in page 20.
- [171] Neal Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987. Cited in page 22.
- [172] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski, Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, June 1996. doi:10.1109/40.502403. Cited in page 60.
- [173] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer Berlin Heidelberg, 1996. doi:10.1007/3-540-68697-5_9. Cited in page 24.
- [174] David Kohel. Addition law structure of elliptic curves. *Journal of Number Theory*, 131(5):894–919, 2011. Elliptic Curve Cryptography. doi:10.1016/j.jnt.2010.12.001. Cited in 2 pages: 127 and 140.
- [175] L. Kohn and N. Margulis. Introducing the Intel i860 64-bit microprocessor. *Micro, IEEE*, 9(4):15–30, 1989. doi:10.1109/40.31485. Cited in page 31.
- [176] Brian Koziel, Amir Jalali, Reza Azarderakhsh, David Jao, and Mehran Mozaffari-Kermani. NEON-SIDH: Efficient Implementation of Supersingular Isogeny Diffie-Hellman Key Exchange Protocol on ARM. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security*, pages 88–103, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-48965-0_6. Cited in page 113.

- [177] Adam Langley. Implementations of a fast Elliptic-curve Diffie-Hellman primitive, August 2008. <https://code.google.com/archive/p/curve25519-donna> and <https://github.com/ag1/curve25519-donna>. Cited in page 177.
- [178] Adam Langley. Lucky Thirteen attack on TLS CBC, February 2013. <https://www.imperialviolet.org/2013/02/04/luckythirteen.html>. Cited in page 24.
- [179] H. W. Lenstra, Jr. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126(3):649–673, 1987. doi:10.2307/1971363. Cited in page 23.
- [180] Chae Hoon Lim and Pil Joong Lee. More Flexible Exponentiation with Precomputation. In Yvo G. Desmedt, editor, *Advances in Cryptology — CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer Berlin Heidelberg, 1994. doi:10.1007/3-540-48658-5_11. Cited in page 137.
- [181] Zhe Liu and Johann Großschädl. New Speed Records for Montgomery Modular Multiplication on 8-Bit AVR Microcontrollers. In David Pointcheval and Damien Vergnaud, editors, *Progress in Cryptology – AFRICACRYPT 2014*, pages 215–234, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-06734-6_14. Cited in page 60.
- [182] Zhe Liu, Hwajeong Seo, Johann Großschädl, and Howon Kim. Reverse Product-Scanning Multiplication and Squaring on 8-Bit AVR Processors. In Lucas C. K. Hui, S. H. Qing, Elaine Shi, and S. M. Yiu, editors, *Information and Communications Security*, pages 158–175, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-21966-0_12. Cited in page 70.
- [183] Zhe Liu, Erich Wenger, and Johann Großschädl. MoTE-ECC: Energy-Scalable Elliptic Curve Cryptography for Wireless Sensor Networks. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, pages 361–379, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07536-5_22. Cited in page 62.
- [184] Patrick Longa and Ali Miri. Fast and Flexible Elliptic Curve Point Arithmetic over Prime Fields. *IEEE Transactions on Computers*, 57(3):289–302, 2008. doi:10.1109/TC.2007.70815. Cited in page 141.
- [185] Patrick Longa and Francesco Sica. Four-Dimensional Gallant–Lambert–Vanstone Scalar Multiplication. *Journal of Cryptology*, 27(2):248–283, 2014. doi:10.1007/s00145-012-9144-3. Cited in page 28.
- [186] Julio López and Ricardo Dahab. Fast Multiplication on Elliptic Curves Over $\text{GF}(2^m)$ without precomputation. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems*, pages 316–327, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-48059-5_27. Cited in 2 pages: 149 and 158.

- [187] Edoardo D. Mastrovito. VLSI designs for multiplication over finite fields $GF(2^m)$. In Teo Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 297–309, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. doi:10.1007/3-540-51083-4_67. Cited in page 84.
- [188] David A. McGrew and John Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In Anne Canteaut and Kapaleeswaran Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004*, pages 343–355, Berlin, Heidelberg, 2005. Springer. doi:10.1007/978-3-540-30556-9_27. Cited in 2 pages: 28 and 49.
- [189] Brendan McMillion. Package p384 is an AMD64-optimized P-384 implementation, May 2017. <https://github.com/Bren2010/p384>. Cited in page 100 (2).
- [190] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. Cited in 3 pages: 22, 54, and 55 (2).
- [191] Johannes Merkle and Manfred Lochter. Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS). RFC 7027, October 2013. doi:10.17487/rfc7027. Cited in page 26.
- [192] Ralph C. Merkle. Secure Communications over Insecure Channels. *Commun. ACM*, 21(4):294–299, April 1978. doi:10.1145/359460.359473. Cited in page 21.
- [193] Andrea Miele, Joppe W. Bos, Thorsten Kleinjung, and Arjen K. Lenstra. Cofactorization on Graphics Processing Units. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 335–352. Springer, 2014. Cited in page 42.
- [194] Victor S. Miller. Use of Elliptic Curves in Cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg. doi:10.1007/3-540-39799-X_31. Cited in 3 pages: 22, 140, and 175.
- [195] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985. doi:10.2307/2007970. Cited in 3 pages: 59 (2), 60, and 111.
- [196] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, 1987. doi:10.1090/S0025-5718-1987-0866113-7. Cited in 10 pages: 23, 125, 133, 144, 148, 153, 156, 161, 175, and 192.
- [197] Andrew Moon. Implementations of a fast Elliptic-curve Diffie-Hellman primitive, 2012. <https://github.com/floodyberry/curve25519-donna/>. Cited in 3 pages: 29, 177 (2), and 179 (2).

- [198] Andrew Moon. Implementations of a fast Elliptic-curve Digital Signature Algorithm, March 2012. <https://github.com/floodyberry/ed25519-donna>. Cited in 5 pages: 29, 184, 186, 188 (4), and 196 (2).
- [199] Erick Nascimento, Łukasz Chmielewski, David Oswald, and Peter Schwabe. Attacking Embedded ECC Implementations Through cmov Side Channels. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 99–119, Cham, 2017. Springer International Publishing. Cited in page 136.
- [200] National Institute of Standards and Technology. Recommended elliptic curves for federal government use, July 1999. <http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf>. Cited in 3 pages: 26, 63, and 170.
- [201] National Institute of Standards and Technology. Advanced Encryption Standard (AES). Technical Report FIPS PUB 197, Gaithersburg, MD, USA, November 2001. doi:10.6028/NIST.FIPS.197. Cited in page 48 (2).
- [202] National Institute of Standards and Technology. Recommendation for Block Cipher Modes of Operation. Technical Report NIST SP 800-38A, Gaithersburg, MD. USA, December 2001. doi:10.6028/NIST.SP.800-38A. Cited in page 51.
- [203] National Institute of Standards and Technology. Secure Hash Standard. Technical Report FIPS PUB 180-2, Gaithersburg, MD. USA, August 2002. <https://csrc.nist.gov/CSRC/media/Publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf>. Cited in 4 pages: 31, 51, 184, and 202.
- [204] National Institute of Standards and Technology. Digital Signature Standard (DSS). Technical Report FIPS PUB 186-4, July 2013. doi:10.6028/NIST.FIPS.186-4. Cited in page 26.
- [205] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report FIPS PUB 202, Gaithersburg, MD. USA, August 2015. doi:10.6028/NIST.FIPS.202. Cited in 4 pages: 51, 184, 190, and 208.
- [206] National Institute of Standards and Technology. Post-Quantum Cryptography Standardization, December 2016. <https://www.nist.gov/pqcrypto>. Cited in 4 pages: 34, 197, 214, and 221.
- [207] National Security Agency. CNSA Suite and Quantum Computing FAQ. Technical report, January 2016. <https://apps.nsa.gov/iaarchive/programs/iad-initiatives/cnsa-suite.cfm>. Cited in page 171.
- [208] Niels Möller. Nettle, 2001. <http://www.lysator.liu.se/~nisse/nettle>. Cited in page 174.
- [209] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: architecture and implementations. *Micro, IEEE*, 19(2):37–48, 1999. doi:10.1109/40.755466. Cited in page 44.

- [210] Katsuyuki Okeya and Kouichi Sakurai. Efficient Elliptic Curve Cryptosystems from a Scalar Multiplication Algorithm with Recovery of the y -Coordinate on a Montgomery-Form Elliptic Curve. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 126–141, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-44709-1_12. Cited in 3 pages: 149 (2), 158 (2), and 194.
- [211] Thomaz Oliveira, Diego Aranha, Julio López, and Francisco Rodríguez-Henríquez. Fast Point Multiplication Algorithms for Binary Elliptic Curves with and without Precomputation. In Antoine Joux and Amr Youssef, editors, *Selected Areas in Cryptography – SAC 2014*, pages 324–344, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-13051-4_20. Cited in 3 pages: 28, 152, and 179.
- [212] Thomaz Oliveira, Julio López, Diego F. Aranha, and Francisco Rodríguez-Henríquez. Two is the fastest prime: lambda coordinates for binary elliptic curves. *Journal of Cryptographic Engineering*, 4(1):3–17, 2014. doi:10.1007/s13389-013-0069-z. Cited in page 28.
- [213] Thomaz Oliveira, Julio López, Hüseyin Hışıl, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. How to (Pre-)Compute a Ladder. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography – SAC 2017*, pages 172–191, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-72565-9_9. Cited in 3 pages: 175, 179, and 180.
- [214] Thomaz Oliveira, Julio López, Hüseyin Hışıl, and Francisco Rodríguez-Henríquez. A note on how to (pre-)compute a ladder. Cryptology ePrint Archive, Report 2017/264, 2017. [v. 2017-05-11] <https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2017/264&version=20170511:062052&file=264.pdf>. Cited in 3 pages: 154, 155, and 156.
- [215] Thomaz Oliveira, Julio López, and Francisco Rodríguez-Henríquez. Software Implementation of Koblitz Curves over Quadratic Fields. In Benedikt Gierlichs and Y. Axel Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 259–279, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. doi:10.1007/978-3-662-53140-2_13. Cited in 2 pages: 180 and 182 (2).
- [216] Thomaz Oliveira, Julio López, and Francisco Rodríguez-Henríquez. A note on how to (pre-)compute a ladder. Cryptology ePrint Archive, Report 2017/264, 2017. [v. 2017-03-25] <https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2017/264&version=20170325:201758&file=264.pdf>. Cited in 3 pages: 153 (2), 154 (2), and 156.
- [217] Erdinc Ozturk, James Guilford, Vinodh Gopal, and Wajdi Feghali. New Instructions Supporting Large Integer Arithmetic on Intel® Architecture Processors. Intel Corporation, White Paper 327831-001, August 2012. <http://www.intel>.

- com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf. Cited in 3 pages: 50 (2), 177, and 178.
- [218] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009. doi:10.1007/978-3-642-04101-3. Cited in page 48.
- [219] Gabriele Paoloni. How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures. Intel Corporation, White Paper 324264-001, September 2010. <http://www.intel.com/content/www/us/en/embedded/training/ia-32-ia-64-benchmark-code-execution-paper.html>. Cited in page 32.
- [220] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997. doi:10.1109/40.592312. Cited in page 41.
- [221] David A. Patterson. Reduced Instruction Set Computers. *Commun. ACM*, 28(1):8–21, January 1985. doi:10.1145/2465.214917. Cited in page 36.
- [222] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, 1996. doi:10.1109/40.526924. Cited in page 44.
- [223] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for Multimedia PCs. *Commun. ACM*, 40(1):24–38, jan 1997. doi:10.1145/242857.242865. Cited in page 44.
- [224] James Pierpont. On an undemonstrated theorem of the Disquisitiones Arithmeticae. *Bull. Amer. Math. Soc.*, 2(3):77–83, 12 1895. <http://projecteuclid.org/euclid.bams/1183414527>. Cited in 2 pages: 62 and 198.
- [225] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance (Corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, January 1978. doi:10.1109/TIT.1978.1055817. Cited in page 128.
- [226] J.M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, September 1975. doi:10.1007/BF01933667. Cited in 2 pages: 23 and 128.
- [227] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 983–1002, 2020. doi:10.1109/SP40000.2020.00114. Cited in page 34.

- [228] Charles J. Purcell. The Control Data STAR-100: Performance Measurements. In *Proceedings of the May 6-10, 1974, National Computer Conference and Exposition, AFIPS '74*, pages 385–387, New York, NY, USA, 1974. Association for Computing Machinery. doi:10.1145/1500175.1500257. Cited in page 41.
- [229] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 7.6)*, 2017. <http://www.sagemath.org>. Cited in page 193.
- [230] Joost Renes, Craig Costello, and Lejla Batina. Complete Addition Formulas for Prime Order Elliptic Curves. In 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*, pages 403–428, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. doi:10.1007/978-3-662-49890-3_16. Cited in 4 pages: 27, 140 (2), 144 (2), and 174.
- [231] Joost Renes and Benjamin Smith. qDSA: Small and Secure Digital Signatures with Curve-Based Diffie–Hellman Key Pairs. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 273–302, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-70697-9_10. Cited in 3 pages: 31, 189, and 195.
- [232] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. doi:10.1145/359340.359342. Cited in page 22.
- [233] Ronald L. Rivest and Burt Kaliski. RSA Problem. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 1065–1069. Springer US, Boston, MA, 2011. doi:10.1007/978-1-4419-5906-5_475. Cited in page 22.
- [234] Andres Rodriguez, Eden Segal, Etay Meiri, Evarist Fomenko, Young Jim Kim, Haihao Shen, and Barukh Ziv. Lower Numerical Precision Deep Learning Inference and Training. Technical report, Intel Corporation, January 2018. Cited in page 52.
- [235] Raúl Rojas. Konrad Zuse’s legacy: the architecture of the Z1 and Z3. *IEEE Annals of the History of Computing*, 19(2):5–16, 1997. doi:10.1109/85.586067. Cited in page 38.
- [236] Alexander Rostovtsev and Anton Stolbunov. Public-key cryptosystem based on isogenies. Cryptology ePrint Archive, Paper 2006/145, 2006. <https://eprint.iacr.org/2006/145>. Cited in page 196.
- [237] Richard M. Russell. The CRAY-1 Computer System. *Commun. ACM*, 21(1):63–72, January 1978. doi:10.1145/359327.359336. Cited in page 41.
- [238] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991. doi:10.1007/BF00196725. Cited in page 189.

- [239] Mike Scott. Ed3363 (HighFive) – An alternative Elliptic Curve. Cryptology ePrint Archive, Report 2015/991, 2015. <http://eprint.iacr.org/2015/991>. Cited in page 26.
- [240] Mike Scott. Missing a trick: Karatsuba variations. Cryptology ePrint Archive, Report 2015/1247, 2015. <http://eprint.iacr.org/2015/1247>. Cited in page 27.
- [241] Igor Semaev. New algorithm for the discrete logarithm problem on elliptic curves. Cryptology ePrint Archive, Report 2015/310, 2015. <http://eprint.iacr.org/>. Cited in page 26.
- [242] Hwajeong Seo and Howon Kim. Optimized Multi-Precision Multiplication for Public-Key Cryptography on Embedded Microprocessors. *International Journal of Computer and Communication Engineering*, 3(2):255–259, May 2013. doi:10.7763/IJCCE.2013.V2.183. Cited in page 69.
- [243] Hwajeong Seo and Howon Kim. Consecutive Operand-Caching Method for Multiprecision Multiplication, Revisited. *Journal of Information and Communication Convergence Engineering*, 13(1):27–35, Mar 2015. doi:10.6109/jicce.2015.13.1.027. Cited in page 69.
- [244] Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. SIDH on ARM: Faster Modular Multiplications for Faster Post-Quantum Supersingular Isogeny Key Exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):1–20, Aug. 2018. <https://tches.iacr.org/index.php/TCHES/article/view/7266>, doi:10.13154/tches.v2018.i3.1-20. Cited in page 116.
- [245] Hwajeong Seo, Zhe Liu, Yasuyuki Nogami, Jongseok Choi, and Howon Kim. Hybrid Montgomery Reduction. *ACM Trans. Embed. Comput. Syst.*, 15(3):58:1–58:13, May 2016. doi:10.1145/2890502. Cited in page 112.
- [246] Hwajeong Seo, Zhe Liu, Yasuyuki Nogami, Taehwan Park, Jongseok Choi, Lu Zhou, and Howon Kim. Faster ECC over $\mathbb{F}_{2^{521}-1}$ (feat. NEON). In Soonhak Kwon and Aaram Yun, editors, *Information Security and Cryptology - ICISC 2015*, pages 169–181, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-30840-1_11. Cited in page 63.
- [247] Dafna Sheinwald, Patricia Thaler, Julian Satran, and Vincente Cavanna. Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations. RFC 3385, September 2002. doi:10.17487/RFC3385. Cited in page 45.
- [248] Anand Lai Shimpi. Intel’s Haswell Architecture Analyzed: Building a New PC and a New Intel. Online resource found at <http://www.anandtech.com/print/6355/intels-haswell-architecture>, October 2013. Cited in page 38.
- [249] Dan Shumow and Niels Ferguson. On the Possibility of a Back Door in the NIST SP800-90 Dual EC PRNG, 2007. <http://rump2007.cr.yp.to/15-shumow.pdf>. Cited in page 26.

- [250] Joseph H. Silverman. The Geometry of Elliptic Curves. In *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*, pages 41–114. Springer New York, New York, NY, 2009. doi:10.1007/978-0-387-09494-6_3. Cited in 3 pages: 124, 125 (2), and 126 (2).
- [251] Jerome A. Solinas. Generalized Mersenne Numbers. Technical Report 39, Center of Applied Cryptographic Research (CACR), 1999. Cited in 2 pages: 63 and 92.
- [252] Jerome A. Solinas. Efficient Arithmetic on Koblitz Curves. *Designs, Codes and Cryptography*, 19(2-3):195–249, 2000. doi:10.1023/A:1008306223194. Cited in 3 pages: 131, 138, and 185.
- [253] Riccardo Spagni. Disclosure of a Major Bug in CryptoNote Based Currencies. Announcement on <https://getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html>, May 2017. Cited in page 191.
- [254] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2):26–39, Mar 2017. doi:10.1109/MM.2017.35. Cited in page 214.
- [255] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The First Collision for Full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 570–596, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-63688-7_19. Cited in page 51.
- [256] Douglas R. Stinson. *Cryptography Theory and Practice*. Chapman & Hall/CRC, 2006. Cited in page 22.
- [257] Anton Stolbunov. *Cryptographic Schemes Based on Isogenies*. PhD thesis, Norwegian University of Science and Technology, January 2012. <http://hdl.handle.net/11250/262577>. Cited in page 196.
- [258] Srinivasa Rao Subramanya Rao. Three Dimensional Montgomery Ladder, Differential Point Tripling on Montgomery Curves and Point Quintupling on Weierstrass’ and Edwards Curves. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2016*, pages 84–106, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-31517-1_5. Cited in 2 pages: 159 and 161.
- [259] Jonathan Taverne, Armando Faz-Hernández, Diego Aranha, Francisco Rodríguez-Henríquez, Darrel Hankerson, and Julio López. Software Implementation of Binary Elliptic Curves: Impact of the Carry-Less Multiplier on Scalar Multiplication. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 108–123, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-23951-9_8. Cited in 2 pages: 28 and 49.

- [260] Jonathan Taverne, Armando Faz-Hernández, Diego F. Aranha, Francisco Rodríguez-Henríquez, Darrel Hankerson, and Julio López. Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. *J. Cryptographic Engineering*, 1(3):187–199, 2011. doi:10.1007/s13389-011-0017-8. Cited in 2 pages: 28 (2) and 49.
- [261] Shreekanth Thakkar and Tom Huff. Internet Streaming SIMD Extensions. *Computer*, 32(12):26–34, 1999. doi:10.1109/2.809248. Cited in page 45.
- [262] The LLVM Project. Clang: a C language family frontend for LLVM, 2007. <https://clang.llvm.org>. Cited in page 31.
- [263] The OpenSSL Project. OpenSSL: The Open Source toolkit for SSL/TLS. www.openssl.org, April 2003. Cited in 4 pages: 27, 100 (2), 174, and 206.
- [264] Sean Turner, Adam Langley, and Mike Hamburg. Elliptic Curves for Security. RFC 7748, January 2016. doi:10.17487/rfc7748. Cited in page 31.
- [265] Scott Vanstone, Ronald L. Rivest, Martin E. Hellman, John C. Anderson, and John W. Lyons. Responses to NIST’s Proposal. *Communications of the ACM*, 35(7):41–54, July 1992. (John Anderson communicated Vanstone’s proposal). doi:10.1145/129902.129905. Cited in page 171.
- [266] Colin D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35:1831–1832(1), October 1999. http://digital-library.theiet.org/content/journals/10.1049/e1_19991230. Cited in page 60.
- [267] Colin D. Walter. The Montgomery and Joye Powering Ladders are Dual. Cryptology ePrint Archive, Report 2017/1081, 2017. <https://eprint.iacr.org/2017/1081>. Cited in page 133.
- [268] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology—CRYPTO 2005*, pages 17–36. Springer, 2005. Cited in page 51.
- [269] Henry S. Warren. *Hacker’s Delight*. Addison-Wesley Professional, 2nd edition, 2012. doi:10.5555/2462741. Cited in page 216.
- [270] W. J. Watson. The TI ASC: a highly modular and flexible super computer architecture. In *Proceedings of the Fall Joint Computer Conference, part I, AFIPS ’72 (Fall, part I)*, pages 221–228, New York, NY, USA, December 1972. ACM. doi:10.1145/1479992.1480022. Cited in page 41.
- [271] André Weimerskirch and Christof Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations. Cryptology ePrint Archive, Report 2006/224, 2006. <http://eprint.iacr.org/2006/224>. Cited in page 27.
- [272] George Woltman. Great Internet Mersenne Prime Search GIMPS, Dec 2018. <https://www.mersenne.org/>. Cited in page 62.

- [273] Gustavo H. M. Zanon, Marcos A. Simplicio, Geovandro C. C. F. Pereira, Javad Doliskani, and Paulo S. L. M. Barreto. Faster Key Compression for Isogeny-Based Cryptosystems. *IEEE Transactions on Computers*, 68(5):688–701, May 2019. doi:10.1109/TC.2018.2878829. Cited in 2 pages: 161 and 220.
- [274] Yang Zhang and J. Großschädl. Efficient prime-field arithmetic for elliptic curve cryptography on wireless sensor nodes. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 459–466, Dec 2011. doi:10.1109/ICCSNT.2011.6181997. Cited in page 62.

Appendix A

Research Production

This chapter provides detailed bibliographical information of the articles published in peer-reviewed venues.

A.1 Journal Articles

A.1.1 A Faster Implementation of SIDH

This article presents theoretical and implementation improvements for the SIDH protocol. Our main contribution is a new three-point ladder algorithm for calculating $P + kQ$ in the SIDH setting. We also show an optimized formula for point tripling on Montgomery curves. This paper was written in collaboration with Eduardo Ochoa-Jiménez and Francisco Rodríguez-Henríquez at the CINVESTAV-IPN.

Faz-Hernández, A., López, J., Ochoa-Jiménez, E., and Rodríguez-Henríquez, F. (2018). A Faster Software Implementation of the Supersingular Isogeny Diffie-Hellman Key Exchange Protocol. *IEEE Transactions on Computers*. 67(11), 1622–1636. <https://doi.org/10.1109/TC.2017.2771535>

Bibtex A.1.1: A faster implementation of SIDH

```
@article{flor_sidh_x64,
  doi      = {10.1109/TC.2017.2771535},
  title    = {{A Faster Software Implementation of the Supersingular
              Isogeny Diffie-Hellman Key Exchange Protocol}},
  author   = {Armando Faz-Hern\`{a}ndez and
              Julio L\`{o}pez and
              Eduardo Ochoa-Jim\`{e}nez and
              Francisco Rodr\`{i}guez-Henr\`{i}quez},
  journal  = {IEEE Transactions on Computers},
  publisher = {Institute of Electrical and Electronics Engineers (IEEE)},
  volume   = {67}, number = {11}, month = {Nov}, year = {2018},
  issn     = {0018-9340}, pages = {1622-1636},
}
```

A.1.2 High-Performance Implementation of ECC

The article shows software implementation techniques for X25519, X448, and EdDSA. We contribute with a detailed description about the use of vector instructions on the implementation of field arithmetic and elliptic curve operations. This paper was written in collaboration with Ricardo Dahab at the University of Campinas.

Faz-Hernández, A., López, J., and Dahab, R. (2019). High-performance Implementation of Elliptic Curve Cryptography Using Vector Instructions. *ACM Transactions on Mathematical Software*, 45(3), 1–35. <https://doi.org/10.1145/3309759>

Bibtex A.1.2: High-Performance Implementation of ECC

```
@article{FazLopDahToms2019,
  doi      = {10.1145/3309759},
  title    = {{High-performance Implementation of Elliptic
              Curve Cryptography Using Vector Instructions}},
  author   = {Armando Faz-Hern\'{a}ndez and
              Julio L\'{o}pez and
              Ricardo Dahab},
  journal  = {{ACM Transactions on Mathematical Software}},
  publisher = {Association for Computing Machinery (ACM)},
  volume  = {45}, number = {3}, month = jul, year = {2019},
  issn    = {0098-3500}, pages = {1-35},
  address = {New York, NY, USA},
}
```

A.2 Publications in Conference Proceedings

A.2.1 Software Implementation of Prime Fields

This extended abstract shows early results on the implementation of prime field arithmetic. We describe a redundant representation for large integer arithmetic, and present preliminary timings.

Faz-Hernández A. and López J. (2014). On Software Implementation of Arithmetic Operations on Prime Fields using AVX2. In *Anais do XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg 2014)*. 14, 338–341. Sociedade Brasileira de Computação. <https://doi.org/10.5753/sbseg.2014.20148>

Bibtex A.2.1: Software Implementation of Prime Fields

```

@inproceedings{Faz-Hernandez2014a,
  doi      = {10.5753/sbseg.2014.20148},
  title    = {{On Software Implementation of Arithmetic
              Operations on Prime Fields using AVX2}},
  author   = {Armando Faz-Hern\`{a}ndez and Julio L\`{o}pez},
  booktitle = {{Anais do XIV Simp\`{o}sio Brasileiro de
              Seguran\c{c}a da Informa\c{c}\~{a}o e
              de Sistemas Computacionais (SBSEg 2014)}},
  publisher = {Sociedade Brasileira de Computa\c{c}\~{a}o - SBC},
  editor    = {Jeroen van de Graaf and Jos\`{e} Marcos Nogueira
              and Leonardo Barbosa Oliveira},
  address  = {Belo Horizonte, MG, Brasil},
  month    = nov, year = {2014},
  issn     = {2176-0063}, pages = {338-341},
}

```

A.2.2 Curve25519 using AVX2

This article presents software implementation techniques that leverage the use of AVX2 instructions to calculate arithmetic operations over $\mathbb{F}_{2^{255}-19}$, and its impact on the X25519 Diffie-Hellman protocol.

Faz-Hernández, A., and López, J. (2015). Fast Implementation of Curve25519 Using AVX2. Lecture notes in computer science. Progress in Cryptology – LATINCRYPT 2015. 329-345. Springer International Publishing. https://doi.org/10.1007/978-3-319-22174-8_18

Bibtex A.2.2: Curve25519 using AVX2

```

@incollection{ecdh_avx2,
  doi      = {10.1007/978-3-319-22174-8_18},
  title    = {Fast implementation of {Curve25519} using {AVX2}},
  author   = {Armando Faz-Hern\`{a}ndez and Julio L\`{o}pez},
  booktitle = {{Progress in Cryptology -- LATINCRYPT 2015}},
  publisher = {Springer International Publishing},
  editor    = {Kristin Lauter and Francisco Rodr\`{i}guez-Henr\`{i}quez},
  volume   = {9230}, month = aug, year = {2015},
  isbn     = {978-3-319-22173-1}, pages = {329-345},
  series   = {{Lecture Notes in Computer Science}},
}

```

A.2.3 How to Precompute a Ladder

This article shows the application right-to-left ladder algorithms for implementing the X25519 and X448 Diffie-Hellman protocols. We contribute with optimized 64-bit implementations of field arithmetic and Montgomery ladder algorithms. This paper was written in collaboration with Thomaz Oliveira and Francisco Rodríguez-Henríquez at the CINVESTAV-IPN, and Hüseyin Hişil at the Yasar University.

Oliveira, T., López, J., Hişil, H., Faz-Hernández, A., and Rodríguez-Henríquez, F. (2017). How to (Pre-)Compute a Ladder. Lecture notes in computer science. Selected Areas in Cryptography – SAC 2017. 172–191. Springer International Publishing. https://doi.org/10.1007/978-3-319-72565-9_9

Bibtex A.2.3: How to Precompute a Ladder

```
@inproceedings{precmp_ladder,
  doi      = {10.1007/978-3-319-72565-9_9},
  title    = {{How to (Pre-)Compute a Ladder}},
  subtitle = {{Improving the Performance of X25519 and X448}},
  author   = {Thomaz Oliveira and
             Julio L{\`o}pez and
             H{"u}seyin Hi{\c s}il and
             Armando Faz-Hern{\`a}ndez and
             Francisco Rodr\`{i}guez-Henr\`{i}quez},
  booktitle = {{Selected Areas in Cryptography -- SAC 2017}},
  publisher = {Springer International Publishing}, address = {Cham},
  editor    = {Carlisle Adams and Jan Camenisch},
  volume    = {10719}, month = {aug}, year = {2018},
  isbn      = {978-3-319-72564-2}, pages = {172-191},
  series    = {{Lecture Notes in Computer Science}},
}
```

A.2.4 Speeding up the P-384 Curve

This article presents the use of vector instructions on the parallel implementation of elliptic curve operations using the P-384 curve. This paper received the “Honorable Mention Award” given by the conference committee.

Faz-Hernández A. and López J. (2016). Speeding up Elliptic Curve Cryptography on the P-384 Curve. Anais do XVI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais. 170-183. Sociedade Brasileira de Computação. <https://doi.org/10.5753/sbseg.2016.19306>

Bibtex A.2.4: Speeding up the P-384 Curve

```

@inproceedings{nistp384_avx2,
  doi          = {10.5753/sbseg.2016.19306},
  title       = {{Speeding up Elliptic Curve Cryptography
                 on the P-384 Curve}},
  author      = {Armando Faz-Hern{\'a}ndez and Julio L{\'o}pez},
  booktitle  = {{Anais do XVI Simp{\'o}sio Brasileiro em Seguran{\c c}a da
                 Informa{\c c}{\~a}o e de Sistemas Computacionais}},
  publisher   = {Sociedade Brasileira de Computa{\c c}{\~a}o - SBC},
  editor      = {Igor Monteiro Moraes and
                 Ant{\'o}nio Augusto de Arag{\'a}o Rocha},
  address    = {Niteroi, RJ, Brasil},
  volume     = {16}, month = nov, year = {2016},
  issn       = {2176-0063}, pages = {170-183},
}

```

A.2.5 Implementation of qDSA

This article presents an optimized software implementation of qDSA, a new digital signature that uses Montgomery curves. We contribute with 64-bit implementations and an alternative method for verifying qDSA signatures. This paper was written in collaboration with Hayato Fujii and Diego Aranha at the University of Campinas.

Faz-Hernández, A., Fujii, H., Aranha, D. F., and López, J. (2017). A Secure and Efficient Implementation of the Quotient Digital Signature Algorithm (qDSA). Lecture notes in computer science. Security, Privacy, and Applied Cryptography Engineering. 170–189. Springer International Publishing. https://doi.org/10.1007/978-3-319-71501-8_10

Bibtex A.2.5: Implementation of qDSA

```

@inproceedings{qdsa_space2017,
  doi      = {10.1007/978-3-319-71501-8_10},
  title    = {{A Secure and Efficient Implementation of
              the Quotient Digital Signature Algorithm (qDSA)}},
  author   = {Armando Faz-Hern\`{a}ndez and
              Hayato Fujii and
              Diego F. Aranha and
              Julio L\`{o}pez},
  booktitle = {{Security, Privacy, and Applied Cryptography Engineering}},
  publisher = {Springer International Publishing}, address = {Cham},
  editor    = {Sk Subidh Ali and Jean-Luc Danger and Thomas Eisenbarth},
  volume    = {10662}, month = nov, year = {2017},
  isbn      = {978-3-319-71500-1}, pages = {170-189},
  series    = {{Lecture Notes in Computer Science}},
}

```

A.2.6 Performance Evaluation of Cryptographic Instructions

This article shows the implementation of symmetric-key encryption algorithms and hash functions using dedicated hardware instructions. We contribute with optimized code using the SHA new instructions. This paper was written in collaboration with Ana D. S. de Oliveira at the Federal University of Mato Grosso do Sul.

Faz-Hernández, A., López, J., and de Oliveira, A. K. D. S. (2018). SoK: A Performance Evaluation of Cryptographic Instruction Sets on Modern Architectures. In APKC '18: Proceedings of the 5th ACM on ASIA Public-Key Cryptography Workshop. 9-18. ASIA CCS '18: ACM Asia Conference on Computer and Communications Security. <https://doi.org/10.1145/3197507.3197511>

Bibtex A.2.6: Performance Evaluation of Cryptographic Instructions

```

@inproceedings{flo_apkc2018,
  doi      = {10.1145/3197507.3197511},
  title    = {{SoK: A Performance Evaluation of Cryptographic
              Instruction Sets on Modern Architectures}},
  author   = {Armando Faz-Hern\`{a}ndez and
              Julio L\`{o}pez and
              Ana Karina D. S. de~Oliveira},
  booktitle = {{APKC '18: Proceedings of the 5th ACM on ASIA
              Public-Key Cryptography Workshop}},
  editor   = {Keita Emura and Jae Hong Seo and Yohei Watanabe},
  publisher = {ACM}, address = {New York, NY, USA},
  location = {Incheon, Republic of Korea},
  month    = jun, year = {2018},
  isbn     = {978-1-4503-5756-2}, pages = {9-18},
}

```

A.2.7 Vectorization of Hash to Curve Functions

This article presents vectorization techniques for a hash to curve function and their implementation using AVX2 vector instructions.

Faz-Hernández A. and López J. (2020). Generation of Elliptic Curve Points in Tandem. Anais do XX Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg 2020). 97-105. Sociedade Brasileira de Computação. <https://doi.org/10.5753/sbseg.2020.19230>

Bibtex A.2.7: Vectorization of Hash to Curve Functions

```

@inproceedings{faz_lopez_intandem,
  doi      = {10.5753/sbseg.2020.19230},
  title    = {{Generation of Elliptic Curve Points in Tandem}},
  author   = {Armando Faz-Hern{\`{a}}ndez and Julio L{\`{o}}pez},
  booktitle = {{Anais do XX Simp{\`{o}}sio Brasileiro em Seguran{\c c}a da
              Informa{\c c}{\`{a}}o e de Sistemas Computacionais}},
  publisher = {Sociedade Brasileira de Computa{\c c}{\`{a}}o - SBC},
  editor   = {Igor Monteiro Moraes and Luis Kowada},
  address  = {Petr\`{o}polis, RJ, Brasil},
  volume   = {20}, month = oct, year = {2020},
  pages    = {97-105},
}

```


A.3 Book Chapters

A.3.1 Implementation of Cryptographic Algorithms

This chapter is an introductory-level course about the implementation of basic cryptographic algorithms. The manuscript was written in Portuguese and in collaboration with Roberto Cabral and Diego Aranha at the University of Campinas.

Faz-Hernández A. and López J. (2015). Implementação Eficiente e Segura de Algoritmos Criptográficos. Minicursos do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais. 93-140. Sociedade Brasileira de Computação. <https://doi.org/10.5753/sbc.9004.8.3>

Bibtex A.3.1: Implementation of Cryptographic Algorithms

```
@incollection{faz2015_minicursos,
  doi      = {10.5753/sbc.9004.8.3},
  title    = {{Implementa{\c c}{\~a}o Eficiente e Segura de
              Algoritmos Criptogr{\'}a}ficos}},
  author   = {Armando Faz-Hern\'}{a}ndez and
              Roberto Cabral and
              Diego F. Aranha and
              Julio L\'}{o}pez},
  booktitle = {{Minicursos do Simp{\'}{o}sio Brasileiro de Seguran{\c c}a
              da Informa{\c c}{\~a}o e de Sistemas Computacionais}},
  publisher = {Sociedade Brasileira de Computa{\c c}{\~a}o},
  editor    = {Eduardo Souto and Michelle Wingham and Joni Fraga},
  address   = {Florian\'}{o}polis, SC, Brasil},
  volume   = {183}, month = nov, year = {2015},
  isbn     = {978-85-7669-304-8}, pages = {93-140},
}
```

