Efficient Algorithms, Protocols and Hardware Architectures for Next-Generation Cryptography in Embedded Systems

by

Utsav Banerjee

B. Tech. (Hons.), Indian Institute of Technology Kharagpur (2013)
S. M., Massachusetts Institute of Technology (2017)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

© Massachusetts Institute of Technology 2021. All rights reserved.
Efficient Algorithms, Protocols and Hardware
Architectures for Next-Generation Cryptography
in Embedded Systems

by

Utsav Banerjee

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

The Internet of Things (IoT) consists of an ever-growing network of wireless-connected
electronic devices which are always collecting, processing and communicating data.
While the IoT has inspired many new applications, these embedded devices have unique
security challenges, thus making IoT security a major concern. Security architectures
for IoT devices, both software and hardware, must be low-power and have low energy
consumption, while still providing strong cryptographic guarantees and side-channel
resilience. Network security protocols use a variety of cryptographic algorithms
to achieve these goals. However, the associated computational complexity makes it
extremely important to have low-power and energy-efficient embedded implementations
of cryptography, especially public key algorithms.

The research presented in this thesis demonstrates the design, implementation and
experimental validation of efficient next-generation cryptography for embedded systems
using software optimization, hardware acceleration and software-hardware co-design,
along with side-channel countermeasures. Using circuit, architecture and algorithm
techniques, efficient hardware-accelerated implementations of elliptic curve cryptography,
pairing-based cryptography, lattice-based cryptography and other post-quantum
cryptography algorithms are demonstrated with up to two orders of magnitude energy
savings compared to state-of-the-art software and hardware. These configurable hard-
ware accelerators are further coupled with a low-power micro-processor to provide the
flexibility to implement a wide variety of security protocols, thus enabling strong and
affordable security for energy-limited IoT nodes.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Vannevar Bush Professor of Electrical Engineering and Computer Science
Dean, MIT School of Engineering
To my dear parents ...
Acknowledgments

My Ph.D. journey at MIT has been an exciting and enriching experience over the past six years. I am extremely fortunate to have so many wonderful people on my side. I will be forever grateful for their support and encouragement.

First and foremost, I would like to thank my advisor Prof. Anantha Chandrakasan for giving me the opportunity to be a part of his research group. I would like to thank him for introducing me to the exciting research area of cryptography and hardware security, guiding me how to choose the right problems to work on, how to take these ideas all the way to real implementations, and also how to effectively present the research results. I thank him for always being a strong advocate of my work, and for always finding time for technical discussions, career advice and detailed feedback on manuscripts and presentations. I would like to thank Prof. Chandrakasan for being such a wonderful teacher and mentor, and for being so kind. I am very fortunate to have been a part of the caring and collaborative environment he has fostered in his group. It has been an honor and a privilege to be his student.

I am very grateful to Prof. Ron Rivest and Prof. Vinod Vaikuntanathan for being part of my thesis committee. I thank them for their time and for sharing their valuable feedback which has greatly improved the quality of this thesis. I would also like to thank Prof. Vaikuntanathan for being a part of my RQE committee and for sharing his thoughts about lattices, post-quantum and emerging directions in cryptography.

I would like to thank Texas Instruments for sponsoring the research presented in this thesis. Thanks to Dr. Xiaolin Lu, Dr. Mahesh Mehendale, Jim Wieser and Thomas Tsai for their support. I am grateful to late Dr. Dennis Buss for initiating this connection. I am very fortunate to have interacted with him and received his advice on research and life in general.

I acknowledge the Irwin Mark Jacobs and Joan Klein Jacobs MIT Presidential Fellowship for financial support during my first year at MIT and the Qualcomm Innovation Fellowship during my second year. I also thank the TSMC University Shuttle Program for generously supporting our chip fabrication.
I would also like to thank Prof. Vivienne Sze for her support. I have been very fortunate to be a student and later a teaching assistant as well as guest lecturer in her digital circuits course. I thank her for giving me the opportunity to teach a new lecture on hardware security and also design a new homework assignment, which was a learning experience for me. I thank her for being a part of my RQE committee and sharing her valuable comments, and also for her advice on teaching and research.

I would like to thank Dr. Sam Fuller for all the insightful discussions we have had over the past years. I also thank him for sharing his detailed and constructive feedback on my research presentations and practice talks during our group meetings.

I thank my graduate counselor Prof. David Perreault for his support in navigating through course registration and staying on track with degree requirements. I thank Prof. Harry Lee and Prof. Jesus del Alamo for their advice at various MTL and CICS events. I would also like to thank Prof. Ingrid Verbauwhede for her suggestions on side-channel analysis during our discussion at ISSCC.

It was a pleasure working with Chiraag Juvekar on my first project. I thank him for his help on the DTLS chip design, especially interfacing the hardware accelerator with the RISC-V, understanding the chip tape-out flow and also the design validation setup. I am grateful to him for patiently answering so many questions. I am thankful to Andrew Wright and Prof. Arvind for sharing their open-source RISC-V design and helping with integrating the accelerator on the DTLS chip. It was a great experience to learn from them about hardware-software co-design, and it eventually became one of the underlying themes of my thesis.

I am also very fortunate to have mentored Madeleine Waller, Abhishek Pathak, Tenzin Ukyab, Siddharth Das, Kaustav Brahma and Gloria Fang. Thanks to Madeleine for meticulously designing the DTLS system demo. Thanks to Tenzin, Abhishek and Siddharth for their help with the software profiling of post-quantum cryptography.

I am grateful to all past and present members of Anantha Group for creating such an amazing work environment. It has been very comforting to have such a wonderful group of colleagues. I am fortunate to have learnt so much from my senior mentors Chiraag Juvekar, Priyanka Raina, Mehul Tikekar, Rabia Tugce Yazıcıgil,
Phillip Nadeau, Avishek Biswas, Nachiket Desai, Michael Price, Frank Yaul and Arun Paidimarri. I thank them for helping me understand the CAD tool flows as well as practical system-level considerations, and for helping me during my first chip tape-out. I would like to thank Mohamed Radwan Abdelhamid, Taehoon Jeong, Preetinder Kaur Garcha and Sirma Orguc for their friendship. Thanks to Lisa Ho and Skanda Koppula for being teammates on my first course project, where we got hands-on experience with power side-channel attacks. Also, thanks to Harneet Singh Khurana, Miaorong Wang, Alex Ji, Di-Chia Chueh, Saurav Maji, Aya Amer, Rishabh Mittal, Kyungmi Lee, Vipasha Mittal, Jongchan Woo, Ray Chen, Maitreyi Ashok, Eunseok Lee and Yeseul Jeon for the many interactions and lively discussions.

I would like to thank Margaret Flaherty for her help with logistics and reimbursements, and for being a wonderful person. Thanks to Jessie-Leigh Thomas for her support. Thanks to Yuvie Cjapi for her help with scheduling meetings, setting up my thesis defense and other administrative matters. I thank Prof. Leslie Kolodziejski, Janet Fischer and Alicia Duarte from the EECS graduate office for their tremendous support throughout these years. I also thank Sylvia Hiestand from the ISO for her support. I would like to thank Michael McIlrath and the MTL Compute team for their support with CAD tools and tape-out logistics. Also, many thanks to the Ashdown House community for providing such a wonderful place to live on campus.

I sincerely thank Saurav Maji and Tathagata Srimani for their friendship and for all the shared experiences. I enjoyed our technical and non-technical discussions, free-food excursions, restaurant hoppings, movie nights and explorations of the MIT campus and Boston-Cambridge area. I will cherish these memories forever.

I would like to thank all my teachers at Vivekananda Mission School and professors at IIT Kharagpur for teaching me important life skills and technical knowledge.

Finally, I am so grateful to my parents, Ramakrishna Banerjee and Mita Banerjee, for their unconditional love and support. I thank them for always being there for me and for believing in me through thick and thin. This work would not have been possible without their inspiration and motivation. I dedicate this thesis to them.
## Contents

List of Figures .......................... xv

List of Tables .......................... xxi

1 Introduction .......................... 1
   1.1 Motivation .......................... 1
   1.2 Cryptographic Primitives .......... 4
   1.3 Implementation Aspects .......... 6
   1.4 Thesis Overview ................. 8

2 Energy-Efficient DTLS Engine with Elliptic Curve Cryptography 13
   2.1 Background ......................... 15
      2.1.1 Elliptic Curve Cryptography ... 15
      2.1.2 Transport Layer Security .... 16
   2.2 Cryptographic Primitives ........ 19
      2.2.1 AES in Galois/Counter Mode (AES-128-GCM) ... 20
      2.2.2 Secure Hash Algorithm (SHA2-256) ........ 23
      2.2.3 Reconfigurable Prime Field ECC .... 24
   2.3 DTLS Engine ....................... 27
      2.3.1 DTLS RAM ...................... 28
      2.3.2 DTLS Controller ............. 29
   2.4 Implementation Results ........... 31
      2.4.1 System Architecture .......... 31
4 Post-Quantum Cryptography using DTLS Engine and RISC-V

4.1 Implementation of SIKE

4.2 Implementation of Other PQC Schemes

4.3 Summary and Contributions

5 Low-Power Elliptic Curve Pairing Crypto-Processor

5.1 Background

5.1.1 Elliptic Curves and Pairings

5.1.2 BLS12-381 Pairing-Friendly Curve

5.2 Hardware Implementation of Pairing

5.2.1 Prime Field Modular Arithmetic

5.2.2 Elliptic Curve and Pairing Computations

5.2.3 Multi-Pairing

5.2.4 Hashing to Points on $G_1$

5.2.5 Point Arithmetic on Jubjub

5.3 Pairing Crypto-Processor

5.4 Implementation Results

5.4.1 System Architecture

5.4.2 Pairing-Based Protocol Implementations

5.4.3 Implementation of Blind Polynomial Evaluation

5.4.4 Comparison with Previous Work

5.4.5 Side-Channel Analysis

5.5 Summary and Contributions

6 Efficient Privacy-Preserving Computation from Pairings

6.1 Background

6.1.1 Functional Encryption

6.1.2 Inner Product Encryption (IPE)

6.1.3 Function-Hiding Inner Product Encryption (FHIPE)

6.2 Optimized FHIPE Encryption and Decryption

6.2.1 Analysis of Computation Cost
List of Figures

1-1 Growth in the number of Internet-connected devices. .................. 2
1-2 Abstraction levels in secure embedded system design. ............... 4
1-3 Design metrics and goals in embedded implementation of cryptography. 7
1-4 Typical energy consumption of IoT building blocks compared with software and hardware implementations of cryptographic primitives. 8
1-5 Implementation of efficient cryptography for securing embedded systems using software optimization, hardware acceleration and software-hardware co-design. ................................................... 9
1-6 Micrographs of the three test chips designed in this thesis. ...... 9

2-1 Overview of DTLS handshake protocol with digital certificate-based mutual authentication and key exchange (dashed arrows indicate encrypted messages). ......................................................... 17
2-2 DTLS computation energy breakdown and percentage of total compute energy spent in handshake, for $N = 32$ bytes of application payload, session duration $t_{\text{session}} = 1$ day and varying application data period $t_{\text{appdata}}$. ......................................................... 19
2-3 Contour plots showing the percentage of total compute energy spent in handshake, for varying application payload size $N$ and varying application data period $t_{\text{appdata}}$, for session duration of (a) 1 day and (b) 1 week. ................................................................. 19
2-4 (a) Implementation of GHASH Galois multiplier in hardware and (b) effect of number of multiplier stages ($n_h$) on area and energy. ........ 22
2-5 Implementation of SHA2-256 round function in hardware.

2-6 Block diagram of the reconfigurable prime-field elliptic curve cryptography accelerator, along with detailed architecture of the modular multiplier implementing interleaved modular reduction.

2-7 Architecture of DTLS engine along with contents of DTLS RAM.

2-8 System block diagram with DTLS engine and RISC-V processor.

2-9 Chip micrograph and test chip specifications.

2-10 (a) Test board with FPGA and (b) power measurement setup.

2-11 Benchmarks for security protocols implemented in SW and SW+HW – (a) ECMQV, (b) Schnorr Prover and (c) Merkle Hashing. Improvements over software are indicated above the bars.

2-12 System demonstration of IoT node with our test chip collecting and transmitting sensor data to a server application over a DTLS-encrypted channel.

2-13 Photographs of (left) power side-channel measurement setup and (right) close-up of test board and differential amplifier.

2-14 Measured power trace demonstrating SPA attack on the simple double-and-add ECSM algorithm implemented in software on RISC-V processor. The double (D) and add (A) steps are marked, along with their key constituent modular arithmetic operations - multiplication (MUL) and inversion (INV). Also shown are bits of the secret scalar successfully recovered from this trace.

2-15 Measured power traces of the SPA-secure hardware ECSM, for 10 random scalars, overlaid together for comparison. The sets of point doubling (DBL) and point addition (ADD) operations are shown in boxes, indicating that the double-and-add patterns are constant irrespective of the secret scalar.

2-16 Measured power trace of DPA-secure hardware ECSM, showing scalar-independent patterns of point doubling and addition.
2-17 Leakage test results for ECSM computation (a) with and (b) without
DPA countermeasure; red dotted line indicates \(|t| = 4.5\) threshold.

2-18 Variation of DPA-secure ECSM leakage \(t\)-value with time.

3-1 Design of our modular adder and subtractor with configurable modulus \(q\).

3-2 Two different single-cycle modular multiplier architectures with (a) fully
configurable and (b) pseudo-configurable modulus for Barrett reduction.

3-3 Comparison of simulated modular multiplication energy for the two
reduction architectures – configurable and pseudo-configurable.

3-4 Unified butterfly in Cooley-Tukey and Gentleman-Sande configurations.

3-5 (a) Memory bank construction using single-port SRAMs and (b) pro-
posed area-efficient NTT architecture using two such memory banks.

3-6 Data-flow of our NTT memory architecture in the first two cycles
(butterfly inputs are in yellow and outputs are in green).

3-7 Memory access patterns for 8-point DIT and DIF NTT using our single-
port SRAM-based memory architecture (R and W denote read and
write respectively).

3-8 Analysis of SHAKE-128, SHAKE-256, AES-128-CTR, AES-256-CTR
and ChaCha20 in terms of energy per bit, bits per cycle and area-energy
product.

3-9 Architecture of discrete distribution sampler with Keccak-based PRNG.

3-10 Sapphire lattice crypto-processor top-level architecture.

3-11 Chip architecture with Sapphire crypto core and RISC-V micro-processor.

3-12 Chip micrograph and test chip specifications.

3-13 Effects of supply voltage scaling as measured from our test chip - (a)
leakage current (b) average active current and maximum frequency.

3-14 Measurement setup with our test chip.

3-15 Configurations of the Sapphire polynomial memory for different Ring-
LWE and Module-LWE schemes.

3-16 Tiling of \(n \times n\) square matrices for Frodo-640, Frodo-976 and Frodo-1344.
3-17 Computation of the matrices \( \mathbf{B} = \mathbf{A} \mathbf{S} + \mathbf{E} \) and \( \mathbf{B}' = \mathbf{S}' \mathbf{A} + \mathbf{E}' \) in Frodo KEM, where the matrices \( \mathbf{S}, \mathbf{E} \) are generated two columns at a time and \( \mathbf{S}', \mathbf{E}' \) are generated two rows at a time.

3-18 Power side-channel measurement setup.

3-19 Measured power waveforms for different polynomial sampling, transform and arithmetic operations along with histograms of energy consumption for 10,000 measurements per operation.

3-20 Difference-of-means test for polynomial number theoretic transform (NTT) with representative power traces from set \( S_0 \) (top left) and \( S_1 \) (top right), difference waveform (bottom left) and difference of means versus number of traces with 99.99\% confidence interval (bottom right).

3-21 Difference-of-means test for polynomial coefficient-wise multiplication with representative power traces from set \( S_0 \) (top left) and \( S_1 \) (top right), difference waveform (bottom left) and difference of means versus number of traces with 99.99\% confidence interval (bottom right).

3-22 Difference-of-means test for polynomial coefficient-wise addition with representative power traces from set \( S_0 \) (top left) and \( S_1 \) (top right), difference waveform (bottom left) and difference of means versus number of traces with 99.99\% confidence interval (bottom right).

3-23 Leakage test results for (a) unmasked and (b) masked NewHope-1024-CPA-PKE.Decrypt, with red dotted line indicating the \( |t| = 4.5 \) threshold.

4-1 Hardware-accelerated addition and subtraction modulo \( 2p \) using our configurable 256-bit modular arithmetic unit.

4-2 Hardware-accelerated 224-bit \( \times \) 224-bit multiplication.

4-3 Hardware-accelerated Montgomery reduction, where \( c \) is the 870-bit input, \( d \) is the 435-bit reduced output and \( \hat{p} = p + 1 \).

4-4 Energy consumption of PQC algorithms with hardware-accelerated AES and SHA2: (a) Kyber, (b) Frodo, (c) ThreeBears and (d) SPHINCS\(^+\).

5-1 Design of modular adder-subtractor for \( \mathbb{F}_p \) and \( \mathbb{F}_q \).
5-2 Synthesized area and simulated energy consumption profiling of CIOS Montgomery product in hardware with different word sizes. 

5-3 Architecture of our CIOS Montgomery product in hardware. 

5-4 Computation stack for extension field and elliptic curve arithmetic. 

5-5 Computation cost of BLS12-381 multi-pairing for different number of pairings in the product \( n \) and with various optimizations. 

5-6 Pairing crypto-processor top-level architecture. 

5-7 Simulated waveforms showing hierarchical memory clock gating in pairing crypto-processor during a snapshot of the final exponentiation computation. 

5-8 Chip architecture with pairing crypto core and RISC-V micro-processor. 

5-9 Chip micrograph and test chip specifications. 

5-10 Measurement setup with our test chip. 

5-11 Pairing-based protocol implementation benchmarks. 

5-12 Comparison of energy consumption of BLS12-381 \( \mathbb{G}_1 \) ECSM and pairing with SPA and DPA side-channel countermeasures. 

5-13 Power side-channel measurement setup. 

5-14 Measured power trace of constant-time SPA-secure hardware-accelerated \( \mathbb{G}_1 \) ECSM, showing double-and-add-always loop and inversion. 

5-15 Measured power trace of constant-time SPA-secure hardware-accelerated pairing, showing Miller loop and final exponentiation 

5-16 Difference-of-means test with 99.99% confidence interval for SPA-secure implementations of (a) \( \mathbb{G}_1 \) ECSM and (b) pairing. 

5-17 Leakage test results for DPA-secure implementations of (a) \( \mathbb{G}_1 \) ECSM and (b) pairing, with red dotted line indicating the \( |t| = 4.5 \) threshold. 

6-1 Computation cost of FHIPE Encrypt for different vector sizes \( n \).
6-2 Computation of $d_1^2, d_1^3, \cdots, d_1^\alpha$ for $d_1 \in \mathbb{G}_T$ and $\alpha = 8$ using (left) repeated multiplications and (right) power tree with squarings and multiplications. Red arrows indicate $\mathbb{G}_T$ multiplications and green arrows indicate cheaper $\mathbb{G}_T$ cyclotomic squarings. .................. 144

6-3 Computation cost of FHIPE Decrypt for different vector sizes ($n$) . . 148

6-4 System diagram showing FHIPE encryption and decryption. .......... 153

6-5 Quantization of normal and abnormal ECG samples .................. 155

6-6 Downsampling of normal and abnormal EEG samples ................. 156

6-7 Example indoor localization scenario and simulated WiFi heat map with $N = 4$ access points $\{AP_1, \cdots, AP_4\}$ and $M = 9$ database locations $\{L_1, \cdots, L_9\}$ .. .................................................. 160
List of Tables

1.1 Standard cryptographic primitives ........................................... 6

2.1 Comparison of our AES-128 with state of the art ......................... 21
2.2 Comparison of our reconfigurable ECC design with state of the art ... 27
2.3 Comparison of our DTLS engine with integrated cryptographic accelerators ................................................................. 36

3.1 Comparison of our NTT with state-of-the-art .............................. 60
3.2 Comparison of CS-PRNG designs ............................................. 61
3.3 Rejection probabilities for different primes with and without fast sampling ......................................................... 63
3.4 Comparison of rejection sampling with software ....................... 64
3.5 Comparison of binomial sampling with state-of-the-art ................. 64
3.6 Comparison of discrete Gaussian sampling with software ............. 65
3.7 Measured energy and performance of key encapsulation schemes ... 75
3.8 Measured energy and performance of digital signature schemes .... 76
3.9 Security of IBE scheme with different error distributions ............ 84
3.10 Performance and Energy Consumption of IBE Implementation ...... 84
3.11 Comparison of our design with state-of-the-art hardware ............ 85

4.1 Profiling of $\mathbb{F}_p$ arithmetic in SIKEp434 .......................... 94
4.2 Performance of SIKEp434 ..................................................... 99

5.1 Descriptions of pairing-based protocols and their applications ...... 119
5.2 Computational requirements of pairing-based protocol implementations 120
5.3 Comparison of our pairing crypto-processor with previous work . . . 122

6.1 BLS12-381 FHIPE Encrypt software evaluation results . . . . . . . . 141
6.2 BLS12-381 FHIPE Encrypt hardware-software co-design results . . . 141
6.3 BLS12-381 FHIPE Decrypt software evaluation results without hash table 149
6.4 BLS12-381 FHIPE Decrypt software evaluation results with hash table 150
6.5 BLS12-381 FHIPE Decrypt hardware-software co-design results . . . 150
6.6 Minimum memory requirement of BLS12-381 FHIPE implementation 151
6.7 Ciphertext sizes for BLS12-381 FHIPE . . . . . . . . . . . . . . . . . 152
Chapter 1

Introduction

1.1 Motivation

The Internet is considered to be one of the most important technological innovations that has transformed our lives in recent times. In its early stages (late 1970s to 1980s), the Internet consisted of a few thousand computers spread across a handful of academic and research organizations in the world. This continued till the late 1980s until the emergence of commercial Internet service providers. In the 1990s, the number of Internet-connected devices grew to several millions spread across several countries. It was also around the same time that the world-wide web was first created [1]. With advances in semiconductor and communication technologies, the volume of Internet traffic started growing rapidly and so did the demand for new software and hardware innovations (also known as Edholm’s law, similar to Moore’s law for semiconductor technology growth [2]). In the 2000s, there was an order of magnitude growth in the number of Internet-connected devices owing to the development of portable computers. We also witnessed the emergence of new Internet applications such as emails, instant messaging, audio and video calls, social networking and online shopping. The next big leaps in the 2010s and 2015s have all been fueled by advancements in computer hardware, integrated circuit technology and software engineering – this includes, smartphones, tablet computers, smart televisions, connected vehicles, etc. The most recent contribution to this unprecedented growth has been from the Internet.
Figure 1-1: Growth in the number of Internet-connected devices (based on [4]).

of Things (IoT). The IoT envisions a scenario where almost all electronic devices we use in our daily activities, including healthcare, education, finance, transportation and infrastructure, are always connected to the Internet. These IoT devices monitor their environment, collect data and then either send them to a remote server or process by themselves in order to decide upon specific actions. A large proportion of these devices, also known as IoT nodes, are embedded systems having limited computing resources and powered by batteries. We are already seeing such devices around us, such as smart watches, health monitors, connected home appliances, surveillance equipment, smart meters, etc. The total number of Internet-connected devices in 2020 was around 20 billion, with almost 50% attributed to the IoT [3]. This number is growing steadily and projected to be around 40 billion by 2025, with almost 75% due to IoT devices [3]. Fig. 1-1 summarizes this tremendous growth of the Internet in the past years, approximated to the order of magnitude, based on data from [4].

Although the Internet of Things has inspired several novel applications, most of these devices are riddled with severe security concerns. Some of the major IoT security flaws identified in the recent years include remote hacking of cars [5, 6], pacemakers [7, 8], insulin pumps [9, 10] and medical devices [11]. We also recently witnessed one of the largest distributed denial of service (DDoS) attacks in history, known as the Mirai botnet attack [12, 13]. Denial of service (DoS) attacks involve cyber-attackers disrupting network services and making them inaccessible to their intended users. Distributed DoS attacks achieve this by overwhelming the network
resources with communication traffic from a large number of connected devices. Security vulnerabilities in millions of IoT devices were exploited to launch the Mirai DDoS attack, and its effects on the global Internet framework are noticeable till this date. According to [14], there are almost 5,000 new mobile malware and ransomware reported per month, and almost 60,000 attacks on IoT devices reported per year. The situation is expected to deteriorate as the proliferation of new wireless networking technologies, such as 5G, further accelerates the growth of connected devices [15].

While there exist standard protocols to secure Internet communications, a similar approach cannot be directly applied to the IoT due to the unique nature of these devices and associated challenges [16]. Firstly, most IoT devices are severely resource-constrained, which makes it extremely difficult to implement computationally expensive cryptographic algorithms on such devices using a traditional software-based approach [17,18]. Next, the massive scale of IoT networks makes it challenging to enable efficient and secure key management, especially with storage limitations on embedded devices [18,19]. Finally, a particularly unique concern with securing embedded systems is physical attacks. Since these devices are easily accessible to attackers, secret information can be extracted by carefully observing execution times, power consumption, electromagnetic emanations and even probing internal circuitry [19–27]. Therefore, securing the IoT requires algorithms, protocols and implementations that are not only efficient but also have reduced computational and storage requirements and are protected against physical attacks.

The focus of this dissertation is on addressing these challenges through efficient IoT security solutions. The three key questions that we answer in this work are:

- Can we build compact and low-power hardware accelerators for next-generation cryptography to secure IoT applications?
- Can we make these designs secure against physical attacks without incurring significant overheads?
- Can we ensure these custom hardware accelerators provide sufficient flexibility to support different protocols as well as seamlessly interface with software running on typical embedded micro-processors?
We achieve this through the unification of software, hardware, algorithm and architecture design. Fig. 1-2 shows the abstraction levels in secure embedded system design - circuit, architecture, algorithm, software and network. These abstraction levels are linked very closely and must be optimized together. We need efficient circuits and architectures to realize efficient algorithms. While complexity analysis of algorithms deals with asymptotic behavior, the constant factors are also very important in practice, which is addressed through circuit-level and architectural decisions. We must have efficient interaction between software and hardware for better flexibility. Finally, it is important to reduce communication overheads along with computation cost, which requires optimizing network protocols. In the rest of this thesis, we discuss how these abstraction levels are optimized for different cryptographic paradigms.

1.2 Cryptographic Primitives

Cryptography is defined as “the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication” [28]. The main objectives of network security protocols are to guarantee confidentiality, integrity, authenticity, availability and non-repudiation of data, services and systems, and they use cryptographic tools, also called cryptographic primitives, to achieve these goals.

Data exchanged between two communicating devices is encrypted and integrity-protected using shared keys stored at both endpoints. Since the same key is used for both encryption and decryption, this is called secret key cryptography or symmetric key cryptography. These shared keys can be computed, while communicating over an
un-encrypted channel, using a key establishment mechanism or key exchange scheme, e.g., the Diffie-Hellman key exchange [28]. The two endpoints can also authenticate their digital identities using a digital signature scheme. Both key establishment and digital signature schemes require pairs of keys – public keys (known to everyone) and private keys (known only to the owner of the key; must never be revealed to others). This is called public key cryptography or asymmetric key cryptography. Public key cryptography is based on the intractability of hard mathematical problems, e.g., integer factorization and discrete logarithms [28]. Examples of symmetric key cryptography include the standard encryption algorithm AES [29] and the standard cryptographic hash functions SHA-2 and SHA-3 [30,31]. Examples of public key cryptography include RSA and ECC [32], both current standards for key establishment, digital signature and authenticated key exchange protocols.

A security protocol (or cryptographic protocol) performs a pre-defined sequence of steps using cryptographic primitives, along with communication with other parties involved in the process, if required, to achieve the desired security goals. Security protocols may use either or both of symmetric key and public key cryptography depending on their objectives. Many end-to-end security protocols, e.g., transport layer security (TLS) [33], use a hybrid approach with public key cryptography for relatively infrequent authentication and key establishment to set up a confidential channel followed by symmetric key cryptography for exchange of encrypted data over this channel. This is primarily because public key cryptography is orders of magnitude more computationally expensive compared to symmetric key cryptography [34,35].

The security level of a cryptographic primitive is conventionally represented in terms of “bits”, where $n$-bit security implies that an adversary must perform $O(2^n)$ computations to successfully attack or break its security. The security of encryption algorithms is directly related to size of the key, e.g., AES with 128-bit, 192-bit and 256-bit keys has 128-bit, 192-bit and 256-bit security respectively. The security of cryptographic hash functions is related to their output digest size, e.g., SHA-2 and SHA-3 with 256-bit, 384-bit and 512-bit digests has 128-bit, 192-bit and 256-bit security respectively. The security of public key cryptographic primitives is determined by the
Table 1.1: Standard cryptographic primitives

<table>
<thead>
<tr>
<th>Security Level</th>
<th>Symmetric Key Crypto</th>
<th>Public Key Crypto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AES</td>
<td>RSA</td>
</tr>
<tr>
<td>128-bit</td>
<td>AES-128</td>
<td>3072b RSA</td>
</tr>
<tr>
<td></td>
<td>SHA-256</td>
<td>256b ECC</td>
</tr>
<tr>
<td>192-bit</td>
<td>AES-192</td>
<td>7680b RSA</td>
</tr>
<tr>
<td></td>
<td>SHA-384</td>
<td>384b ECC</td>
</tr>
<tr>
<td>256-bit</td>
<td>AES-256</td>
<td>15360b RSA</td>
</tr>
<tr>
<td></td>
<td>SHA-512</td>
<td>512b ECC</td>
</tr>
</tbody>
</table>

underlying computational hardness assumptions, e.g., RSA with 3072-bit, 7680-bit and 15360-bit keys has 128-bit, 192-bit and 256-bit security respectively, and ECC based on 256-bit, 384-bit and 512-bit prime fields has 128-bit, 192-bit and 256-bit security respectively. Table 1.1 shows standard cryptographic primitives at different security levels (under classical computational assumptions; cryptographic primitives secure in the quantum computation paradigm will be discussed in Chapters 3 and 4). Further details about the construction of cryptographic primitives and protocols and analysis of their security levels are available in [28,36–39]. Please refer to Appendix B for mathematical background on algebraic structures and computational hardness assumptions used to construct cryptographic primitives.

1.3 Implementation Aspects

As shown in Fig. 1-3, there are six main design goals in embedded implementation of cryptographic primitives – area, energy / power, performance, side-channel resilience, flexibility and security. Design area must not be too high, since it directly affects design cost. Both power and energy consumption must be low. While low energy consumption is critical for battery-operated devices, low power consumption is important for systems powered by energy-harvesting. Performance requirements are moderate as we can usually tolerate some latency overheads in IoT applications, unlike traditional Internet communications. It is desirable to have some flexibility in the design so that new cryptographic algorithms can be supported. Design flexibility also allows easily patching the implementation if security flaws, e.g., side channels, are identified later. Finally, and most importantly, the implementation must be secure, which includes not
only the theoretical security level of the underlying primitive but also side-channel resilience. Secure embedded system designers must trade-off between all these metrics to achieve the required design goals. Different IoT applications may have different design objectives, thus providing ample scope for design space exploration in terms of circuits, algorithms and architectures [18,34,40,41].

Fig. 1-4 shows typical energy consumption of IoT building blocks, such as radio transceiver and embedded micro-processor, compared with software and hardware implementations of cryptography. The transceiver energy represents communication cost while the remaining are indicative of computation cost. Clearly, public key cryptography is significantly more computationally expensive than symmetric key cryptography. Also, hardware acceleration can provide orders of magnitude improvement in energy-efficiency, as will be demonstrated in the rest of this thesis. This is because the arithmetic and logic units in general-purpose micro-processors are not suitable for cryptographic computations, while custom hardware can be designed to handle them more efficiently. Software implementation of cryptography, although not very efficient, provides complete flexibility to update programs whenever required. On the other hand, hardware accelerators are designed to perform only a single task very efficiently, thus allowing little or no flexibility. In this thesis, we also address this issue by introducing more configurability into the custom hardware design.
1.4 Thesis Overview

In this thesis, we describe efficient and side-channel-secure implementations of cryptographic primitives using the following three approaches:

- efficient software implementation of optimized algorithms on embedded micro-processor for faster execution and lower energy consumption.

- design of compact and low-power hardware with specialized circuitry for cryptographic primitives to enable energy-efficiency through data-path design, fast logic-memory interaction and algorithm-architecture co-optimization.

- integration of cryptographic hardware with embedded micro-processor for hardware-software co-design, where computationally expensive algorithms are accelerated in hardware and remaining operations are performed in software.

Our cryptographic hardware designs can be programmed for flexibility while maintaining efficiency. These cryptographic co-processors are then integrated with an open-source RISC-V micro-processor [42] for system-level demonstration, as shown in Fig. 1-5. The micro-processor used for all software-hardware co-design in this research is a low-power 3-stage-pipeline RISC-V core supporting the RV32I(M) instruction set, with 16/32 KB instruction memory and 64 KB data memory. The RISC-V design is
Figure 1-5: Implementation of efficient cryptography for securing embedded systems using software optimization, hardware acceleration and software-hardware co-design.

derived from the MIT Riscy family of open-source processors [43]. All hardware designs are implemented in Bluespec System Verilog and fabricated in TSMC 40nm/65nm low-power technology nodes. The test chips are shown in Fig. 1-6. The following cryptographic primitives are supported:

- Symmetric primitives: AES, SHA-2, SHA-3 (Keccak), ChaCha20, Trivium
- Elliptic curve cryptography using prime field Weierstrass and Montgomery curves
- Lattice-based cryptography using LWE, Ring-LWE and Module-LWE
- Pairing-based cryptography using the pairing-friendly BLS12-381 elliptic curve

Other post-quantum cryptography algorithms like supersingular isogeny, hash-based cryptography and Integer-Module-LWE are also implemented using software-hardware co-design. Next, we summarize the main contributions of this thesis.

Figure 1-6: Micrographs of the three test chips designed in this thesis.
Energy-Efficient Accelerator for Elliptic Curve Cryptography and Datagram Transport Layer Security: In Chapter 2, we describe our implementation of an energy-efficient TLS cryptographic engine. It consists of a configurable accelerator for prime field elliptic curve cryptography (ECC) supporting arbitrary primes up to 256 bits, along with accelerators for AES and SHA-2, together integrated with a state machine for the TLS protocol. Our design provides two orders of magnitude improvement in energy-efficiency and performance and order of magnitude reduction in memory usage compared to software. The TLS engine is coupled with a low-power RISC-V micro-processor to demonstrate a variety of security protocols using hardware-software co-design.

Energy-Efficient Configurable Crypto-Processor for Post-Quantum Lattice-Based Cryptography: In Chapter 3, we describe our design of an energy-efficient and configurable lattice crypto-processor supporting public key encryption, key encapsulation and digital signature schemes based on LWE and its variants Ring-LWE and Module-LWE. All protocol parameters can be configured at run-time and cryptographic algorithms can be programmed using a custom instruction set. The cryptographic core is further integrated with a RISC-V micro-processor to demonstrate NIST post-quantum cryptography candidates NewHope, Kyber, Frodo, qTesla and Dilithium. Compared to state-of-the-art embedded implementations, our design achieves order of magnitude improvement in energy-efficiency and performance.

Acceleration of Post-Quantum Cryptography using Pre-Quantum Cryptographic Accelerator: In Chapter 4, we discuss our efficient hardware-software co-design of NIST post-quantum cryptography candidates SIKE, Kyber, Frodo, Three-Bears and SPHINCS+ using the test chip from Chapter 2. Using our energy-efficient modular arithmetic unit and accelerators for AES and SHA-2, we achieve up to an order of magnitude energy savings compared to embedded software implementations.

Low-Power Crypto-Processor for Pairing-Based Cryptography: In Chapter 5, we describe our design of a low-power crypto-processor supporting elliptic curve cryptography (ECC) and pairing-based cryptography (PBC) using the recently proposed
BLS12-381 curve. We provide detailed discussion on the construction of extension field arithmetic and elliptic curve point and line arithmetic specific to BLS12-381. Our efficient implementation of word-serial modular arithmetic provides two orders of magnitude improvement in energy-efficiency and performance compared to software. The crypto-processor can be programmed with a custom instruction set to accelerate several ECC and PBC protocols. It is also coupled with a low-power RISC-V micro-processor and accelerators for AES and SHA-2 in order to provide further flexibility.

**Efficient Privacy-Preserving Computation using Pairing-Based Inner Product Functional Encryption:** In Chapter 6, we discuss efficient algorithms for encryption and decryption in the pairing-based function-hiding inner product functional encryption scheme. We provide software implementation results on three different platforms along with hardware-software co-design with the test chip from Chapter 5. We also discuss two applications in privacy-preserving classification of biomedical sensor data and privacy-preserving wireless fingerprint-based indoor localization.
Chapter 2

Energy-Efficient DTLS Engine with Elliptic Curve Cryptography

One of the most important requirements in Internet of Things (IoT) networks is to guarantee that the communication channel between each sensor node and the cloud server is secure, even in the presence of untrusted and potentially malicious network infrastructure [16]. This is called end-to-end security, and protocols such as Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) [33, 44] enable the establishment of mutually authenticated confidential channels between IoT sensor nodes and the cloud. DTLS employs elliptic curve-based public key cryptography techniques to authenticate the two end points and establish shared secret keys, which are then used to encrypt application data. TLS version 1.3 has recently been standardized by the Internet Engineering Task Force (IETF), and is considered to be one of the most suited protocols for securing the IoT [16]. While this makes DTLS an ideal solution for IoT, the associated computational cost makes software-only implementations prohibitively expensive for resource-constrained embedded devices [35, 45–47]. IoT devices are usually powered by batteries, which are expected to last several years, or through energy harvesting. Moreover, commercially available IoT platforms use micro-controllers with limited instruction and data memory. Therefore, it is essential to have a DTLS implementation which not only has low energy consumption but also comes with a small memory footprint.
To address these challenges, we present the first hardware implementation of DTLS 1.3, based on version 18 of the protocol draft [33,44]. Our key contributions (described in detail in Sections 2.2, 2.3 and 2.4) are summarized as follows:

- Majority of the computation cost in DTLS is due to expensive authenticated key exchange and digital signature generation/verification using elliptic curve cryptography (ECC). We design an energy-efficient prime-field ECC accelerator which enables two orders of magnitude energy savings in the DTLS handshake.

- Elliptic curves with different parameters have been recommended by standards organizations around the world for cryptographic use. Security protocols may use any of these curves and software libraries can be easily re-programmed to support multiple curves. To provide similar flexibility, our ECC accelerator can also be configured with different primes and curve parameters.

- We design energy-efficient hardware accelerators for AES and SHA2 for authenticated encryption/decryption of application data as well as to perform the remaining cryptographic computations required in the DTLS protocol.

- Apart from cryptographic accelerators, we also design a dedicated DTLS state machine which offloads protocol control flow to hardware, thus reducing program code and data memory usage by an order of magnitude.

- We implement and experimentally validate algorithm-level countermeasures to protect our ECC hardware from common timing and power side-channel attacks.

The cryptographic accelerators are also integrated with a low-power RISC-V microprocessor [42, 43] to demonstrate other security applications using hardware-software co-design. The integration with RISC-V and system-level demonstration is joint work with Chiraag Juvekar, Andrew Wright, Madeleine Waller and Prof. Arvind. The RISC-V processor core is based on an open-source design by Andrew Wright and Prof. Arvind [43]. Design of software and hardware interface between the RISC-V core and the accelerators is done in collaboration with Chiraag Juvekar. The IoT system prototype has been designed in collaboration with Madeleine Waller. Further details and experimental results are provided in Section 2.4.
2.1 Background

2.1.1 Elliptic Curve Cryptography

An elliptic curve $E$ over a finite field $\mathbb{K}$ is defined as:

$$E : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6$$

where $a_1, a_2, a_3, a_4, a_5, a_6 \in \mathbb{K}$. In this work, we consider elliptic curves over finite fields with characteristic $\text{char}(\mathbb{K}) \neq 2, 3$. In particular, we are interested in fields where the characteristic is a very large prime $p$, the corresponding field henceforth denoted as $\mathbb{F}_p$, and two types of elliptic curves over such prime fields:

- Short Weierstrass curves consisting of the set of points $E(\mathbb{F}_p) = \{(x, y) \mid y^2 = x^3 + ax + b \pmod{p}\} \cup \mathcal{O}$

- Montgomery curves consisting of the set of points $E(\mathbb{F}_p) = \{(x, y) \mid by^2 = x^3 + ax^2 + x \pmod{p}\} \cup \mathcal{O}$

where $a, b \in \mathbb{F}_p$ are the curve parameters and $\mathcal{O}$ is the distinguished point at infinity.

For further details on the theory of elliptic curves, please refer to [36,48–50].

The fundamental operations in ECC are point addition ($R = P + Q$) and point doubling ($R = P + P$), where $P, Q, R \in E(\mathbb{F}_p)$. With these operations, the points on the curve $E(\mathbb{F}_p)$ form an abelian group, with $\mathcal{O}$ serving as the identity element, that is, $P + \mathcal{O} = \mathcal{O} + P = P$ for all $P \in E(\mathbb{F}_p)$. The order of this group (number of points in $E(\mathbb{F}_p)$) is denoted by $\#E(\mathbb{F}_p) = q$, and $qP = \mathcal{O}$ for all $P \in E(\mathbb{F}_p)$.

Repeated additions of a point $P$ with itself is called elliptic curve scalar multiplication (ECSM). For any scalar $k$, the scalar multiple $kP$ is computed as

$$kP = P + P + \cdots + P \quad (k-1) \text{ point additions}$$

This computation forms the basis of the elliptic curve discrete logarithm problem (ECDLP) – determine scalar $k$ given the elliptic curve $E(\mathbb{F}_p)$ of order $q$, and points
$P, Q \in E(\mathbb{F}_p)$ such that $Q = kP$. For a $t$-bit prime $p$, the fastest known classical algorithm that can solve ECDLP has time complexity $O(2^{t/2})$ [36]. For sufficiently large primes and appropriate curve parameters, it is considered infeasible for a computationally bounded (non-quantum) adversary to solve ECDLP, and this guarantees the security of ECC and associated public key protocols, e.g., elliptic curve-based key establishment, authentication and digital signatures used in TLS and DTLS.

2.1.2 Transport Layer Security

The DTLS protocol can be divided into two major phases - handshake and application data, as shown in Fig. 2-1. The handshake starts with the client (sensor node) and the server agreeing upon protocol parameters such as the cryptographic algorithms to be used. Next, a Diffie-Hellman key exchange [28] is performed to establish a shared secret over the untrusted channel. The subsequent handshake messages are completely encrypted using keys derived from this shared secret. Following this, the client and the server authenticate each other through digital certificate verification. Finally, the two parties verify the integrity of the information exchanged in the above step, to prevent man-in-the-middle attacks. At this point, a mutually authenticated confidential channel has been established between the client and the server. This channel can then be used, in the application data phase, to exchange data encrypted under a new set of keys derived from the handshake parameters.

The DTLS specification lists a set of recommended cryptographic algorithms, also known as cipher suites, to be used for performing the handshake and encrypting data. In this work, we consider DTLS connections implementing the TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 cipher suite, where elliptic curve cryptography [36] is used for endpoint authentication and key exchange, AES-128-GCM (Advanced Encryption Standard in Galois/Counter Mode) [29,51] is used for authenticated encryption, and SHA2-256 (Secure Hash Algorithm) [30] is used for message hashing, key derivation and pseudo-random number generation. The handshake phase involves $\approx 100$ invocations each of the AES-GCM and SHA primitives, which operate in blocks of 128 bits and 512 bits respectively; one ECDHE
Figure 2-1: Overview of DTLS handshake protocol with digital certificate-based mutual authentication and key exchange (dashed arrows indicate encrypted messages).

(Elliptic Curve Diffie-Hellman Ephemeral), and at least two ECDSA (Elliptic Curve Digital Signature Algorithm) operations (one ECDSA-Sign and at least one ECDSA-Verify). Once the handshake is complete, encryption or decryption of application data requires one invocation of AES-GCM per 128-bit block of data.

While the computation energy spent during each DTLS handshake is constant for a given cipher suite, the energy required during the application data phase is a direct function of the application payload size. Let us denote the handshake energy and the encrypted application data energy per byte of payload as $E_{\text{handshake}}$ and $E_{\text{appdata}}$ respectively (computation cost only), the session duration (time interval between two consecutive handshakes) as $t_{session}$ and the application data period (time interval
between two consecutive application data transmissions) as $t_{\text{appdata}}$. Then, for $N$ bytes of application payload, the total computation energy during a session is given by:

$$E_{\text{total}} = E_{\text{handshake}} + (N \times \frac{t_{\text{session}}}{t_{\text{appdata}}} \times E_{\text{appdata}})$$

since the total number of data transmissions during a session is $t_{\text{session}}/t_{\text{appdata}}$. The fraction of energy spent in handshake computations is $E_{\text{handshake}}/E_{\text{total}}$. The session duration $t_{\text{session}}$ is dictated by security requirements of the application – more frequent handshakes (to establish new session keys), that is, smaller $t_{\text{session}}$, imply stronger security guarantees, e.g., medical devices authenticate more often than industrial sensors. The application data rate is calculated as $N/t_{\text{appdata}}$, which also depends on the application, e.g., industrial sensors typically send small packets of data every hour while medical devices send large amounts of data every minute or every second.

To understand the effect of application data rate on compute energy, we consider $E_{\text{handshake}} = 150$ mJ and $E_{\text{appdata}} = 125$ nJ (computation cost only) as measured from an embedded software implementation of DTLS on an ARM Cortex-M0+ microprocessor operating at 3 V and 48 MHz [35]. For devices handshaking once every day and payload size of $N = 32$, the breakdown of computation energy is shown in Fig. 2-2. We observe that the percentage of energy spent in DTLS handshake is around 30% when data is transmitted every second, and more than 99% when data is transmitted every hour. To further analyze the effects of these parameters, contour plots are shown in Fig. 2-3 for $t_{\text{session}} = 1$ day and $t_{\text{session}} = 1$ week. As expected, the handshake energy becomes a larger fraction of total energy for smaller $N$, larger $t_{\text{appdata}}$ and smaller $t_{\text{session}}$. We observe that the total computation energy for a software implementation of DTLS is in the range 0.1-0.5 J, which is dominated by either handshake computations or application data encryption depending on the application parameters. Therefore, it is essential to design energy-efficient hardware to accelerate both handshake and application data computations for low-power IoT devices secured by DTLS.
Figure 2-2: DTLS computation energy breakdown and percentage of total compute energy spent in handshake, for $N = 32$ bytes of application payload, session duration $t_{\text{session}} = 1$ day and varying application data period $t_{\text{appdata}}$.

Figure 2-3: Contour plots showing the percentage of total compute energy spent in handshake, for varying application payload size $N$ and varying application data period $t_{\text{appdata}}$, for session duration of (a) 1 day and (b) 1 week.

### 2.2 Cryptographic Primitives

As discussed earlier, DTLS requires both symmetric and public key cryptography primitives using AES, SHA and ECC. In this section, we provide details of the energy-efficient implementations of these primitives, including architectural optimizations, design space exploration and on-chip characterization results.
2.2.1 AES in Galois/Counter Mode (AES-128-GCM)

The DTLS protocol uses AES-128 in GCM mode for authenticated encryption with associated data (AEAD), that is, it simultaneously guarantees confidentiality, integrity, and authenticity of the data. The AES-128 cipher uses 128-bit keys to encrypt 128-bit plain-text blocks over 10 iteration rounds, with each round performing a set of linear and non-linear transformations on the cipher’s internal state. The S-Box is the most important non-linear component of AES, used both in encryption and key expansion. It accounts for about 40-50% of the area and power consumption in hardware implementations of AES [45]. In this work, we have used the low-power low-area S-Box from [52].

To explore the effects of AES data-path size on area and energy-efficiency, we implemented two different AES architectures:

- $A_1$, with 8-bit data-path and one S-Box, processes the state and the round key on separate cycles 8 bits at a time, and takes 336 cycles to encrypt a block.
- $A_2$, with 128-bit data-path and 20 S-Boxes, processes the state and the round key together in a single cycle, and takes 11 cycles to encrypt a block.

The 8-bit architecture $A_1$ replicates the optimizations proposed in [53] and [54] to reduce the number of temporary registers. We compared the area, performance and energy-efficiency of the two designs, as determined from post-synthesis simulations in 65 nm LP process at 1.2 V. Design $A_1$ has 2.8k-gate area and requires 336 cycles per block while consuming 14.2 pJ/bit energy. Design $A_2$ is larger, with 8.6 k-gate area, but requires only 11 cycles per block with energy consumption 3.1 pJ/bit. Clearly, the 128-bit parallel design $A_2$ is $4.6 \times$ more energy-efficient and $30 \times$ faster, at the cost of $3 \times$ increase in logic area.

Table 2.1 compares our AES-128 design with state of the art, both in terms of area and energy. Our design is smaller than the 128-bit data-path 2-stage pipelined AES design in [55], while having comparable energy consumption, after accounting for voltage and technology scaling. In comparison to [53,54,56], which are all 8-bit data-path serial implementations, our design is more energy-efficient, when accounting
Table 2.1: Comparison of our AES-128 with state of the art

<table>
<thead>
<tr>
<th>Design</th>
<th>Tech (nm)</th>
<th>Area (mm²)</th>
<th>Cycles / Block</th>
<th>Voltage (V)</th>
<th>Energy (pJ/bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamalainen et al. [56] a</td>
<td>130</td>
<td>-</td>
<td>3.2</td>
<td>1.2</td>
<td>37.5</td>
</tr>
<tr>
<td>Mathew et al. [55]</td>
<td>45</td>
<td>0.15</td>
<td>-</td>
<td>1.1</td>
<td>2.3</td>
</tr>
<tr>
<td>Mathew et al. [53]</td>
<td>22</td>
<td>0.0022</td>
<td>1.9</td>
<td>0.9</td>
<td>30.1</td>
</tr>
<tr>
<td>Zhang et al. [54]</td>
<td>40</td>
<td>0.0043</td>
<td>2.3</td>
<td>0.9</td>
<td>8.9</td>
</tr>
<tr>
<td>This work</td>
<td>65</td>
<td>0.015 b</td>
<td>10.6 b</td>
<td>1.2</td>
<td>9.59 c</td>
</tr>
</tbody>
</table>

a Post-synthesis area and power reported in [56]
b Area of final placed-and-routed design
c Measured energy
d Simulated energy

for voltage and technology scaling, but at the cost of larger area. Our AES could not be experimentally characterized at voltages smaller than 0.8 V because all logic and SRAMs on our test chip are powered by a single supply rail. We provide the simulated post-layout energy consumption of our AES design at 0.45 V in Table 2.1.

AES-GCM uses the AES forward cipher for both encryption and decryption, and a Galois multiplication-based special hash function called $GHASH$ for authentication [51]. AES-GCM employs the counter mode of operation, which concatenates a counter value with the initialization vector $IV$, and encrypts it with the secret key using AES. The result of this encryption is then XOR-ed with the plain-text to generate the cipher-text. Like all counter modes, this essentially acts as a stream cipher, therefore it is important to ensure that a different $IV$ is used for each stream that is encrypted.

The Galois multiplier in $GHASH$ can be implemented in hardware using one or more copies of the basic function which we denote as $h$: $Z_{i+1} = Z_i \oplus x_i \cdot V_i$ and $V_{i+1} = (V_i >> 1) \oplus \{LSB(V_i)\} \cdot (1110001||0^{120})$, as shown in Fig. 2-4a. A Galois multiplier with $n_h$ stages requires $128/n_h$ cycles per multiplication, and the number of $h$-stages directly affects area, cycles per operation and energy consumption. Multiple Galois multipliers were synthesized to determine a suitable architecture, and their area-energy...
products were plotted as a function of the number of $h$-stages, as shown in Fig. 2-4b. We observed that a 32-cycle design, with $n_h = 4$, has the lowest area-energy product, hence this version was used in our AES-GCM implementation. Since AES-GCM involves computing GHASH on the cipher-text, our design performs encryption and Galois multiplications in parallel, at 32 cycles per 128-bit data block. For $m$ blocks of associated data and $n$ blocks of plain-text (cipher-text), it takes $54 + 32 \cdot (m+n)$ cycles to encrypt (decrypt) and generate (verify) the GCM tag, where the fixed 54-cycle overhead accounts for computing the hash key, hashing the data length, computing the tag as well as configuring the key, IV and other encryption parameters. The final placed-and-routed design occupies 29.9 kGE area, including the 10.6 kGE AES, of which about 25% is attributed to registers used to store input/output data, keys, intermediate states and configuration values. Energy consumption of our AES-GCM design, as measured from our test chip, is 11.88 pJ/bit at 0.8 V.

Figure 2-4: (a) Implementation of GHASH Galois multiplier in hardware and (b) effect of number of multiplier stages ($n_h$) on area and energy.
2.2.2 Secure Hash Algorithm (SHA2-256)

The SHA2-256 hash algorithm compresses messages of arbitrary lengths (< $2^{64}$ bits) and generates a unique 256-bit message digest. Since SHA2-256 operates on 512-bit blocks, the input message is padded to a multiple of 512 bits. The internal state of the hash function is initialized according to the SHA2 specification [30]. The Message Schedule takes 512-bit blocks of the padded message and sends 32-bit words $W_i$ to the main SHA2-256 Round function, along with a round constant $K_i$. Each 512-bit block is digested over 64 iterations of the round function, and the state is updated. This continues till the entire message has been processed, and the final value of the state is the message digest.

Fig. 2-5 shows details of the round function. The internal state consists of 16 32-bit registers $H_0 - H_7$ and $a - h$. The $\Sigma_0$, $\Sigma_1$, Maj and Ch functions are specified in [30], while $\oplus$ denotes 32-bit addition modulo $2^{32}$, that is, the final carry is ignored. $H'_0 - H'_7$ and $a' - h'$ denote the updated state values after one iteration. Although the state of the hash function is defined by $H_0 - H_7$, $a - h$ and the message schedule, we note that $H_0 - H_7$ completely define the SHA2-256 state after every 64 iterations of the round, that is, after every 512-bit block has been processed. We exploit this property to implement efficient running hashes, as discussed in Section 2.3.

Figure 2-5: Implementation of SHA2-256 round function in hardware.
The critical paths in the round function were implemented using a combination of carry-save and ripple-carry adders to reduce latency. Messages are sent to the SHA2 core one byte at a time, and a counter is used to track the input data length, which is used by the SHA2 core to perform message padding. The SHA2-256 core computes $a' - h'$ in parallel to achieve increased energy-efficiency. Our final design occupies 18.2 kGE, and takes 65 cycles to process a 512-bit input block, with measured energy consumption of 4.43 pJ/bit at 0.8 V.

### 2.2.3 Reconfigurable Prime Field ECC

Elliptic curve cryptography (ECC) is used in DTLS for both key exchange and digital signature protocols. We consider two types of elliptic curves over finite fields $\mathbb{F}_p$ of large prime characteristic $p$ – short Weierstrass curves ($y^2 = x^3 + ax + b$) and Montgomery curves ($by^2 = x^3 + ax^2 + x$). All other prime curves (for $p \neq 2, 3$) can be transformed into the short Weierstrass form with a simple change of variables [36]. ECC-based protocols can choose from a large set of standard curves, e.g., NIST, Curve25519, Brainpool, SEC and ANSSI. While existing literature in ECC hardware mostly focus on implementing a single family of curves [57–60], a similar approach is not suitable for DTLS because the standard allows a much wider choice of curves. This provides the motivation for our reconfigurable prime field ECC design, and we support curves over any prime up to 256 bits, which correspond to at most 128 bits of security.

Prime field elliptic curve operations such as point addition, point doubling and elliptic curve scalar multiplication (ECSM) can be decomposed into arithmetic in the finite field $\mathbb{F}_p$. This makes efficient modular arithmetic integral to both software and hardware implementations of ECC. Fig. 2-6 describes our energy-efficient ECSM hardware, which can be configured with prime $p$ of variable length $t$ (up to 256 bits) and curve parameters $a$ and $b$. Given scalar $k$ and point $P(x, y)$, it generates $Q = kP$.

One of the key components of our design is an efficient modular multiplier, shown in Fig. 2-6. In order to support arbitrary prime fields, it performs multiplication with interleaved modular reduction [61]. Three adders are used for this computation, one for addition and two for reduction. The reduction uses conditional subtractions, all
performed in the same cycle so that the modular multiplication is constant time and there is no potential timing side-channel. The same circuitry is re-used for modular addition and modular subtraction as well. While previous ECC designs [57–60] have chosen 16-bit or 32-bit data-paths for modular arithmetic, we have used full 256-bit adders for energy-efficiency [45], with higher bits of the data-path gated when working with smaller primes.

Prior work on hardware implementations of ECC [57–60] re-used the modular multiplier to perform modular inversion using Fermat’s theorem: $x^{-1} = x^{p-2} \mod p$ [36]. This method uses repeated modular multiplications (384 on average for 256-bit primes) for exponentiation. Therefore, inversion using Fermat’s theorem ($I_{Fermat}$) is slow, but doesn’t require any additional logic area. In this design, we make an energy-
area trade-off and implement dedicated hardware to perform modular inversion [45] using the extended Euclidean algorithm \(I_{Euclid}\) [36], which involves modular additions, subtractions and bit-shifts. Similar to the multiplier, our inverter also consists of 256-bit adders for energy-efficiency. Energy consumption of the two types of inversions related to multiplication \(M\) as: \(I_{Fermat} \approx 384M\) and \(I_{Euclid} \approx 3M\), as obtained from simulation and verified through experimental measurement. This indicates that \(I_{Euclid}\) is 128× more efficient, albeit at the cost of increased logic area.

Having optimized the modular arithmetic implementations, the next step is to select an efficient ECSM algorithm. Traditional window-based ECSM [36] requires 256 DBL and 64 ADD operations for window-size \(w = 4\). Instead, a pre-computation-based comb algorithm [36,62,63] is implemented, which involves 64 DBL and 64 ADD operations, thus reducing ECSM energy by 2.5×. A 4 KB cache stores pre-computed comb data for up to six points, including generator points and public keys, which is specifically used to speed up the DTLS handshake, as explained in Section 2.3.

The final optimization step in our design is the appropriate choice of coordinates for elliptic curve points. Resource-constrained ECC implementations [57–60] typically use projective coordinates to avoid modular inversions in the ECSM inner loop, at the cost of extra multiplications and a final expensive Fermat inversion. In projective coordinates, the costs of point operations are ADD = 8\(M\) and DBL = 11\(M\). Since we have an efficient dedicated modular inverter, we use affine coordinates where ADD = 2\(M\) + \(I\) and DBL = 3\(M\) + \(I\). The total ECSM costs for projective and affine coordinates are calculated as \(E_{proj} = 64 \times (8M + 11M) + 4M + I_{Fermat} = 1604M\) and \(E_{aff} = 64 \times (5M + 2I_{Euclid}) = 704M\). Therefore, using affine coordinates saves \(\approx 2\times\) in energy by trading off the extra multiplications for cheaper Euclid inversions.

For a 256-bit short Weierstrass curve, our design takes \(\approx 320k\) cycles for comb pre-computations and \(\approx 180k\) cycles for SPA-secure ECSM. Table 2.2 compares our design with previous work in ECC hardware. Our design is the most energy-efficient and flexible, but has larger area owing to the dedicated modular inverter and full data-path modular multiplier. Reconfigurability of our ECC core is also responsible for some of the area overheads, since fixed prime field arithmetic (such as NIST
Table 2.2: Comparison of our reconfigurable ECC design with state of the art

<table>
<thead>
<tr>
<th>Design</th>
<th>Tech (nm)</th>
<th>Voltage (V)</th>
<th>Logic Area (kGE)</th>
<th>Supported Curve(s) / ECSM</th>
<th>Cycles</th>
<th>Energy a (µJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hutter et al. [57] b</td>
<td>350</td>
<td>3.3</td>
<td>9.5</td>
<td>NIST P-192</td>
<td>753k</td>
<td>1423.6</td>
</tr>
<tr>
<td>Roy et al. [58] b</td>
<td>32</td>
<td>1.0</td>
<td>26</td>
<td>SEC P-160</td>
<td>250k</td>
<td>2.25</td>
</tr>
<tr>
<td>Roy et al. [58] b</td>
<td>32</td>
<td>1.0</td>
<td>26</td>
<td>NIST P-192</td>
<td>350k</td>
<td>3.15</td>
</tr>
<tr>
<td>Pessl et al. [59] b</td>
<td>130</td>
<td>1.2</td>
<td>8.6</td>
<td>NIST P-160</td>
<td>100k</td>
<td>4.4</td>
</tr>
<tr>
<td>Hutter et al. [60] b</td>
<td>130</td>
<td>1.2</td>
<td>32.6</td>
<td>Curve25519</td>
<td>811k</td>
<td>56.8</td>
</tr>
<tr>
<td>This work</td>
<td>65</td>
<td>0.8</td>
<td>65.5 c (49.1 d)</td>
<td>All prime curves up to 256 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>160-bit</td>
<td>74k</td>
<td>2.22 e</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>192-bit</td>
<td>102k</td>
<td>3.11 e</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>256-bit</td>
<td>180k</td>
<td>6.47 e</td>
</tr>
</tbody>
</table>

a Base point ECSM energy  b Post-synthesis area and power reported in [57–60]  
c Area of final placed-and-routed design  d Synthesized area for comparison  e Measured energy

primes) can be implemented with smaller logic with hard-wired parameters. For applications requiring higher security levels (beyond 128-bit security), our configurable ECC architecture can be easily scaled to larger prime fields by using wider data-path adders and small changes to control logic. Of course, this will also require re-design and re-fabrication of the circuitry.

2.3 DTLS Engine

At the core of DTLS is its state machine, which controls all handshaking protocols and related computations. Since the DTLS state machine supports a variety of configurations [33, 44], software implementations can be error-prone and has lead to attacks in the past [64]. To avoid such issues, we enable only a carefully chosen secure subset of all the configurations supported by DTLS. In this work, we have implemented the cipher suite with ECDHE, ECDSA, AES-128-GCM and SHA2-256, requiring mandatory server/client authentication. The Client Certificate URL extension is used, that is, client certificates are not transmitted. The Cached Information extension is made optional, and the server decides whether to use it during the handshake. Certificate Authority (CA) public keys are cached by both parties, and CA certificates are never exchanged (still maintaining compliance with the TLS specification).
Fig. 2-7 shows the architecture of our DTLS engine (DE), with its key components – (1) energy-efficient cryptographic accelerators, (2) DTLS controller and (3) DTLS RAM. The cryptographic primitives, described in Section 2.2, not only accelerate DTLS computations but can also be accessed individually to implement standalone protocols. Details of the DTLS RAM and DTLS controller are discussed next.

### 2.3.1 DTLS RAM

The 2 KB DTLS RAM can be divided into three sections - DTLS micro stack, DTLS Config memory and Accelerator Config memory. The 1.25 KB DTLS micro stack acts as scratch-pad for temporary variables computed during the DTLS handshake, including DRBG states and DTLS session keys. The DTLS stack is not accessible through the memory-mapped interface so that secret session information, including encryption...
keys, cannot be read by software. The 0.45 KB DTLS Config memory is used to store public keys, secret keys and certificate details, which can be programmed through the RISC-V processor, while the remaining 0.3 KB Accelerator Config memory stores accelerator configuration values for standalone cryptographic operations. Contents of the Config memory and the micro stack are detailed in Fig. 2-7.

While our prototype chip uses only SRAMs and flip-flops as on-chip storage elements, a commercial product replicating this design would replace the DTLS Config memory and micro stack with non-volatile storage in order to enable power gating while still retaining the configuration values, DRBG state and session keys.

### 2.3.2 DTLS Controller

The DTLS controller implements a micro-coded DTLS 1.3 state machine for pseudo-random number generation, key schedule, session transcripts, encrypted packet framing, parsing and validation of X.509 digital certificates and re-transmission timeouts.

**Pseudo-Random Number Generation:** An HMAC-based Deterministic Random Bit Generator (HMAC-DRBG) [65] is used to generate cryptographically secure pseudo-random numbers, while an HMAC-based Key Derivation Function (HKDF) [66] is used to compute DTLS handshake and session keys. Both HMAC-DRBG and HKDF use the SHA2-256 cryptographic accelerator to efficiently compute HMACs (keyed-hash message authentication codes) [67]. The DRBG is first initialized using the input seed material (as obtained from the DTLS Config memory), also known as the *Instantiate* phase. During the *Generate* phase, pseudo-random numbers are generated 256 bits at a time. The DRBG is initialized (seeded) only once at the time of device setup, and it can be used for up to $2^{48}$ invocations of the *Generate* step, with up to $2^{19}$ bits generated in each invocation, as per the NIST DRBG specification [65]. Since the DRBG *Generate* function is used 3 times (for generating client random and scalars for ECDHE and ECDSA-Sign) during a DTLS handshake (DRBG not required in the application data phase), it need not be re-seeded for $\approx 9.4 \times 10^{13}$ handshakes, which exceeds the life of the IoT device.
Session Transcript Computation: The DTLS handshake involves 6 session hash (transcript) computations, that is, hash of the concatenation of all messages exchanged till that point in the handshake. Software implementations of DTLS typically save all handshake messages, and compute the hash over all of them every time a transcript is required. Handshakes can be as large as 2-3 KB and repeatedly reading them from SRAMs can be very expensive. To eliminate the need to store the entire handshake, we implement a running hash by exploiting the property of SHA2-256 that the internal registers $H_0 - H_7$ completely define the hash state every time a 512-bit block has been digested, as discussed in Section 2.2. Handshake bytes are pushed into a 64-byte FIFO, and a 512-bit block is sent to the SHA2-256 core whenever the FIFO is full. This ensures that session hash computations always digest data in blocks of 64 bytes, except for the last block, and when computing the hash $H(m)$ of an $N$-byte message $m$, the intermediate hash of $\lfloor N/64 \rfloor$ blocks of $m$ is stored in $H_0 - H_7$. After every session hash, the FIFO state (containing any un-hashed bytes) and the registers $H_0 - H_7$ are copied to the DTLS stack, so that the SHA2 core can be used for other computations. This is particularly useful for later phases of the handshake which involve hashing large digital certificates. Our proposed approach reduces the total session transcript memory usage from several kilobytes down to only 96 bytes (64 bytes for the SHA2 state and up to 32 bytes for the un-hashed portions of the messages).

ECC Computations in DTLS: The reconfigurable ECC core is used to perform both ECDH and ECDSA-Sign/Verify computations, where the deterministic ECDSA scheme [68] is used to securely generate signatures. The DTLS handshake involves up to 7 ECSM computations, and we have seen in Section 2.2 that ECSM energy can be reduced by $2.5 \times$ if pre-computed comb points are available. Our ECC comb point cache supports up to 6 pre-computed base points, which are used to minimize the energy consumption of the ECDHE_ECDSA handshake. Comb points are computed and stored for the curve generator point $G$, the CA public key $Q_{CA}$ and the server public key $Q_{SRV}$. This one-time pre-computation requires around 33 $\mu$J of energy, which gets amortized over all the subsequent handshakes, but provides up to $2.2 \times$ reduction in the energy consumption of each DTLS handshake. The pre-computation...
for $G$ is essential for both ECDH and ECDSA, while pre-computations for $Q_{CA}$ and $Q_{SRV}$ are used to verify signatures from the CA and the server respectively. The rest of the point cache is used for ECDH and ECDSA with random points without corrupting the stored points required by DTLS.

**DTLS State Machine:** The DTLS state machine is used to generate and process messages at different steps of the handshake as well as exchange encrypted data after the handshake. A 64-bit counter is used to implement the DTLS Retransmission Timer [44] which handles dropped packets. The time-out value can be configured externally, and the state machine re-transmits the previous flight whenever the timer expires. When the DTLS state machine waits for the next flight, all cryptographic accelerators are clock-gated in order to reduce power consumption. Three 256-byte FIFOs are used to fetch input messages (IN FIFO), send output messages (OUT FIFO) and read application data packets (DATA FIFO). The IN FIFO ensures that the DTLS controller starts parsing input messages only when a fully formed packet is available, and sends out complete output messages to the OUT FIFO. For encrypted application data, the state machine also implements the packet optimizations proposed in [35], with the option to enable AES-GCM tag truncation.

### 2.4 Implementation Results

#### 2.4.1 System Architecture

Fig. 2-8 shows the system block diagram. Along with the DTLS engine (DE), it consists of a 3-stage (IF: instruction fetch, EX: execute, WB: write back) RISC-V processor [43] supporting the RV32I instruction set [42], with 16 KB instruction cache and 64 KB data memory, and an SD (Secure Digital) card used as backing store for larger programs. Sleep mode is implemented on the RISC-V, to save power, by gating its clock when cryptographic tasks are delegated to the DE. The DE uses a dedicated hardware interrupt to wake the processor on completion of these tasks. The DE is clocked by a software-controlled divider to decouple the processor operating frequency
from the long critical paths in the ECC accelerator. A memory-mapped interface provides access to the DTLS engine, through the DTLS RAM, not only for executing DTLS protocol workloads but also for standalone computations in the cryptographic accelerators. The same interface is also used to communicate with peripherals such as GPIO (General Purpose Input / Output), UART (Universal Asynchronous Receiver / Transmitter) and SPI (Serial Peripheral Interface) through RISC-V software.

The test chip, shown in Fig. 2-9, was fabricated in the TSMC 65nm low-power CMOS process. The RISC-V processor occupies 0.0489 mm$^2$ (34 kGE) area and interfaces with 16 KB instruction cache and 64 KB data cache. The DTLS engine requires 0.214 mm$^2$ (149k GE) logic area, and uses total 6.75 KB of SRAM. The chip supports voltage scaling from 1.2 V down to 0.8 V. The RISC-V core achieves a maximum frequency of 78 MHz at 1.2 V and 20 MHz at 0.8 V. The DTLS engine

![Figure 2-8: System block diagram with DTLS engine and RISC-V processor.](image)

32
can operate in a frequency range of 16 MHz (at 0.8 V) to 20 MHz (at 1.2 V). All measurement results for the RISC-V processor and the DTLS engine are reported at 16 MHz and 0.8 V.

Fig. 2-10 shows our test board and measurement setup. The test chip is housed in a QFN64 socket soldered to the board, and an Opal Kelly XEM7001 FPGA [69] is used to interface with the chip. A Keithley 2602A source meter [70] is used to
supply power to the chip. Both the FPGA and the source meter are controlled from a host computer through USB and GPIB interfaces respectively. While our chip has an SD interface which can communicate with standard SD cards, we use the FPGA to emulate the SD card program memory so that we can eliminate the overhead otherwise imposed by real SD card access times and thus allow fair software benchmarking.

2.4.2 Protocol Benchmarks

The DTLS engine supports handshake in two modes:

- Full – with verification of server certificate
- Cached – with caching of server certificate information

The cached mode requires one less ECDSA-Verify operation, with 36% reduction in handshake time and energy. Energy consumption of the hardware-accelerated DTLS handshake is 68.94 $\mu$J and 44.08 $\mu$J in the full and cached modes respectively. In the application data phase, the chip consumes 0.89 nJ per byte of encrypted data.

In order to analyze the efficiency of our DTLS hardware accelerator, we compared resource utilization in three scenarios: DTLS fully implemented as RISC-V software (SW), the cryptographic kernels accelerated in hardware and only the DTLS controller implemented in software (SW+HW), and DTLS fully implemented in hardware (HW). Test software was implemented using the cryptographic libraries provided by ARM mbedTLS [71]. Since mbedTLS does not support cached server certificates, all analyses were performed with the DE in non-cached mode. The use of cryptographic accelerators alone results in over 2 orders of magnitude improvement in run time and energy efficiency (SW vs. SW+HW). The hardware DTLS controller reduces code size by 60 KB, while the DTLS micro stack results in 13 KB reduction in data memory usage (SW+HW vs. HW). When DTLS is accelerated in hardware, code size goes down to only 8 KB, including system functions. We also note that the area occupied by the DTLS state machine and control logic is 5$\times$ smaller than the area of SRAM otherwise required to accommodate the DTLS program in software. Please refer to [45,72,73] for further details on the protocol implementation and evaluation.
Security applications beyond DTLS can also be implemented using software-hardware co-design with RISC-V and the cryptographic accelerators. We illustrate this flexibility using three benchmark applications – (a) ECMQV, an alternative to ECDHE/ECDSA-based authenticated key exchange, (b) Schnorr Prover, an interactive zero-knowledge prover of identity, and (c) Merkle Hashing, used to ensure data integrity in peer-to-peer network protocols. The reduction in resource utilization for all three applications is shown in Fig. 2-11. The ECC-based applications achieve over 200× increase in energy-efficiency, while Merkle hashing sees 6× energy savings.
2.4.3 System Demonstration

To demonstrate the functionality of our chip in a complete system, an IoT node was designed with the test chip collecting data from a temperature sensor and an accelerometer, encrypting it and then transmitting it through a Bluetooth Low Energy (BLE) transceiver, where all data communications with our test chip are through SPI. A Raspberry Pi module [74] acts as a gateway forwarding these encrypted packets to the application software running on a PC. The system setup is shown in Fig. 2-12, along with a screenshot of the server application displaying decrypted packet details.

2.4.4 Comparison with Previous Work

Table 2.3 compares this work with previous designs which integrate multiple cryptographic accelerators. This work implements a flexible ECC accelerator which supports arbitrary primes up to 256 bits, in contrast with [57] and [60] which only support fixed 192 and 255-bit curves respectively. [75] only supports binary field modular arithmetic in hardware. Our ECC accelerator is $458 \times$ and $9 \times$ more energy-efficient than [57] and [60] respectively at comparable security levels.

<table>
<thead>
<tr>
<th>Hutter et al. [57]</th>
<th>Hutter et al. [60]</th>
<th>Zhang et al. [75]</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tech (nm)</td>
<td>350</td>
<td>130</td>
<td>40</td>
</tr>
<tr>
<td>Voltage (V)</td>
<td>3.3</td>
<td>1.2</td>
<td>0.7</td>
</tr>
<tr>
<td>Freq (MHz)</td>
<td>0.847</td>
<td>1</td>
<td>28.8</td>
</tr>
</tbody>
</table>

**Table 2.3: Comparison of our DTLS engine with integrated cryptographic accelerators**

<table>
<thead>
<tr>
<th></th>
<th>Hutter et al. [57]</th>
<th>Hutter et al. [60]</th>
<th>Zhang et al. [75]</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total Area</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.21 mm$^2$</td>
</tr>
<tr>
<td><strong>Logic Gates</strong></td>
<td>12.8 kGE</td>
<td>32.6 kGE</td>
<td>-</td>
<td>148 kGE</td>
</tr>
<tr>
<td><strong>SRAM</strong></td>
<td>0.25 KB</td>
<td>0.28 KB</td>
<td>8 KB$^b$</td>
<td>6.75 KB</td>
</tr>
<tr>
<td><strong>Hardware ECC</strong></td>
<td>Only NIST P-192</td>
<td>Only Curve25519</td>
<td>-$^c$</td>
<td>All prime curves up to 256 bits</td>
</tr>
<tr>
<td><strong>Base point</strong></td>
<td>1423.6 $\mu$J (192b)</td>
<td>-</td>
<td>-$^c$</td>
<td>3.11 $\mu$J (192b)</td>
</tr>
<tr>
<td><strong>ECSM energy</strong></td>
<td>-</td>
<td>56.8 $\mu$J (255b)</td>
<td>-$^c$</td>
<td>6.34 $\mu$J (256b)</td>
</tr>
<tr>
<td><strong>AES energy</strong></td>
<td>8558.04 nJ</td>
<td>521.01 nJ$^d$</td>
<td>7.05 nJ</td>
<td>6.21 nJ</td>
</tr>
<tr>
<td><strong>SHA energy</strong></td>
<td>6876.3 nJ$^e$</td>
<td>-</td>
<td>48.7 nJ$^e$</td>
<td>24.3 nJ$^e$</td>
</tr>
<tr>
<td><strong>DTLS in H/W</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

$a$ Post-synthesis results reported by [57] and [60]  
$b$ [75] implements in-memory computing for crypto  
$c$ [75] supports only binary field arithmetic in hardware  
$d$ [60] uses Salsa20 instead of AES  
$e$ [57] uses SHA-1, [75] uses SHA-3, and this work uses SHA-2
2.4.5 Side-Channel Analysis

Public-key cryptography algorithms, such as ECC, are prone to side-channel attacks due to their expensive computations and long execution times [76]. We experimentally analyzed the side-channel security of our hardware-accelerated ECC implementation. Fig. 2-13 shows our power side-channel measurement setup. A 20 $\Omega$ resistor was placed in series between the chip’s $V_{DD}$ pin and power supply. The voltage across this resistor, proportional to the chip’s instantaneous current consumption, was magnified using a differential amplifier (AD8001 op-amp [77], with 6 dB flat gain up to 100 MHz, in the non-inverting configuration with resistors of appropriate sizes) and then observed using a 2.5 GS/s Tektronix MDO3024 mixed domain oscilloscope [78].

The first attack we analyze is simple power analysis (SPA). Simple double-and-add ECSM algorithms perform conditional point additions in the outer loop depending on whether the corresponding bit in the secret scalar is a 1. Since DBL and ADD involve distinct arithmetic, the power consumption of the chip can leak this information. For reference, we demonstrate an SPA attack on a software implementation of this algorithm, as shown in Fig. 2-14. The slower operations – multiplication and inversion, can be clearly inferred from a single power trace, and the bits of the secret scalar can
be successfully determined. In order to prevent SPA attacks, we use a zero-less signed digit (ZSD) representation of the scalar [63] in conjunction with the comb technique, which transforms the scalar to have no zero bits, thus avoiding conditional point additions. This also reduces the number of pre-computed comb points per ECSM from 16 to 8. Fig. 2-15 shows power traces of our SPA-secure implementation for 10 random scalars overlaid together, where both DBL and ADD are computed at each iteration of the outer loop, irrespective of the bits of the scalar.

The binary scalar \( k = (k_{t-1}, k_{t-2}, \ldots, k_1, k_0) \) needs to be odd to have a valid ZSD form, that is, the least significant bit \( k_0 = 1 \) [63]. To prevent leaking any information about whether \( k \) is even or odd, we initially compute \( k' = k + 1 \) if \( k \) is even, and \( k' = k + 2 \) if \( k \) is odd. Then, \( Q' = k'P \) is computed, and finally, we obtain \( Q = kP \) as \( Q' - P \) if \( k \) is even, and \( Q' - 2P \) if \( k \) is odd. We use a compact scalar encoding, which we denote as ZSD*, of the ZSD scalar where the 1-bit represents ‘1’ and the 0-bit represents ‘-1’, inspired by [79]. This compact form of scalar \( k \) can be computed “on-the-fly” as \( \text{ZSD}^*(k) = (1, k_{t-1}, \ldots, k_2, k_1) \) since the following equation holds:

\[
(1, k_{t-1}, \ldots, k_2, k_1) = 2^{t-1} + \underbrace{k - 1}_2 - \underbrace{(2^{t-1} - 1 - \frac{k - 1}{2})}_2 = k
\]

+1 bits of \((k_{t-1}, \ldots, k_1)\) -1 bits of \((k_{t-1}, \ldots, k_1)\)

Therefore, no additional circuitry is required to convert \( k \) to the ZSD* form. The SPA countermeasure introduces 5 extra point additions, on average, for 256-bit scalars [36], which translates to \approx 4\% energy and performance overhead.

More sophisticated side-channel attacks on ECC involve statistical metrics, e.g., differential power analysis (DPA) [76], and they require several power traces for a single scalar. Since the same scalar is never used twice for any of the ECSM computations during the DTLS handshake, these attacks are not relevant to DTLS (with ephemeral key exchange using ECDHE). For other ECC-based protocols, countermeasures, require some form of input randomization, as discussed next.
Figure 2-14: Measured power trace demonstrating SPA attack on the simple double-and-add ECSM algorithm implemented in software on RISC-V processor. The double (D) and add (A) stpng are marked, along with their key constituent modular arithmetic operations - multiplication (MUL) and inversion (INV). Also shown are bits of the secret scalar successfully recovered from this trace.

Figure 2-15: Measured power traces of the SPA-secure hardware ECSM, for 10 random scalars, overlaid together for comparison. The sets of point doubling (DBL) and point addition (ADD) operations are shown in boxes, indicating that the double-and-add patterns are constant irrespective of the secret scalar.
For applications requiring higher security, we implement DPA-secure ECSM using the scalar randomization technique [76]. The secret scalar $k$ is split into two parts $r$ and $k - r$, where $r$ is a random scalar with the same bit-length as $k$. A new $r$ is generated for every ECSM computation $kP$, which now works as $rP + (k - r)P$. The energy consumption and execution time of this DPA-secure ECSM are both approximately double that of SPA-secure ECSM. Fig. 2-16 shows a measured power trace of our hardware-accelerated DPA-secure ECSM. Despite the point doubling (DBL) and addition (ADD) operations having distinct power signatures, the number of DBL and ADD operations is always constant, irrespective of the secret scalar, due to the zero-less signed digit scalar representation discussed earlier. Therefore, the DPA-secure ECSM is also SPA-secure, as desired.

Typically, a non-specific fixed vs. random $t$-test [80] is performed to statistically quantify the information leakage through side channels. For non-specific $t$-test, the power traces are divided into two sets $Q_0$ (with fixed input) and $Q_1$ (with random input) of sizes $N_0$ and $N_1$ respectively. Let $\mu_0$, $\mu_1$ and $\sigma_0^2$, $\sigma_1^2$ be the means and standard variances of sets $Q_0$, $Q_1$ respectively. Then, the $t$-test statistic is given by:

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{\sigma_0^2}{N_0} + \frac{\sigma_1^2}{N_1}}}$$
Figure 2-17: Leakage test results for ECSM computation (a) with and (b) without DPA countermeasure; red dotted line indicates $|t| = 4.5$ threshold.

Figure 2-18: Variation of DPA-secure ECSM leakage $t$-value with time.

The absolute values of $t$ are then plotted as function of $N = N_0 + N_1$, with $|t| > 4.5$ indicating information leakage [80]. For testing ECC implementations, the $t$-test involves scalar multiplication with fixed scalar $k_0$ to generate set $Q_0$ and scalar multiplication with random scalars $k_r$ to generate set $Q_1$. We performed this test both with and without the DPA countermeasure, and measurement results are shown in Fig. 2-17. We observed that $|t| < 4.5$ consistently for 100,000 measurements with scalar randomization, while it crosses the $|t| = 4.5$ threshold around 170 measurements without the countermeasure, thus validating the side-channel resistance of our implementation. Fig. 2-18 shows how the $t$-value varies with time (within the threshold) for a DPA-secure ECSM execution, as measured during the leakage test.
2.5 Summary and Contributions

Datagram Transport Layer Security (TLS) is a widely used Internet security protocol which is also being adopted for securing the IoT. However, software implementations of DTLS on embedded micro-controllers are prohibitively expensive due to the computation cost of elliptic curve-based public key cryptography and the memory cost of protocol control functions. In this chapter, we have presented energy-efficient configurable cryptographic hardware which makes DTLS a practical solution for implementing end-to-end security on resource-constrained IoT devices.

Energy-efficient accelerators for ECC, AES and SHA provide more than two orders of magnitude improvement in performance and energy-efficiency compared to software implementations of DTLS. This allows IoT sensor nodes to re-authenticate more frequently for applications that demand stronger security guarantees. Our hardware-accelerated DTLS has energy consumption of 68.94 $\mu$J per authentication handshake and 0.89 nJ per byte of encrypted application data, when operating at 0.8 V. Several circuit, architecture and algorithm techniques are leveraged to achieve this energy-efficient design. A 128-bit parallel AES architecture gives $5\times$ energy savings compared to a conventional 8-bit serial architecture. Similarly, our ECC modular arithmetic unit consists of a 256-bit wide parallel data-path, which leads to $3\times$ energy savings compared to a traditional 32-bit serialized design. A dedicated modular inverter is combined with elliptic curve point operations in affine coordinates to further reduce ECC energy by $2.5\times$. Pre-computation techniques are utilized for ECC generator points and public keys to reduce DTLS handshake energy by $\approx 2\times$.

Our ECC hardware performs modular multiplication with interleaved reduction in order to support arbitrary prime fields. Different curve parameters, for short Weierstrass and Montgomery curves, can also be configured for further flexibility. The 256-bit modular arithmetic data-path is gated appropriately when working with smaller primes to provide energy scalability.

We have also implemented several countermeasures in our ECC hardware to prevent common timing and power side-channel attacks. The secret scalar is converted
into a compact zero-less signed digit form, without using any additional circuitry, to prevent simple power analysis. We also support randomized scalar splitting to prevent differential power analysis. Experimental validation results are provided.

A dedicated DTLS 1.3 protocol controller further enables reduction in memory consumption, with code size only 8 KB and data memory usage only 3 KB. The DTLS controller consists of a micro-coded state machine which performs protocol functions such as key schedule, session transcripts, encrypted packet framing, parsing and validation of digital certificates and re-transmission timeouts in hardware. This allows IoT platforms to implement application programs without worrying about the overheads otherwise imposed by the security protocol.

To further demonstrate the suitability of our design for practical applications, we have also integrated the DTLS cryptographic engine with an open-source RISC-V micro-processor. Protocols beyond DTLS can be implemented using software-hardware co-design in conjunction with the cryptographic accelerators, while still getting the benefits of energy-efficiency and performance.
Chapter 3

Energy-Efficient Post-Quantum Lattice Crypto-Processor

Modern public key cryptography relies on hard mathematical problems such as integer factorization, discrete logarithms over finite fields and discrete logarithms over elliptic curve groups. However, these problems can be solved by a large-scale quantum computer in polynomial time using Shor’s algorithm [81], thus making today’s public key protocols like RSA and ECC vulnerable to quantum attacks. Given the rapid advancement in quantum computing technology over the past few years, cryptographers are developing quantum-secure public key algorithms to protect today’s data from tomorrow’s threats. Lattice-based cryptography is considered one of the most promising candidates for post-quantum cryptographic protocols because of its extensive security analysis as well as small public key and signature sizes. In the recent past, lattice-based constructions of public key cryptography primitives such as encryption [82–85], key exchange [86], key encapsulation [87, 88], digital signatures [89, 90], functional encryption [91] and homomorphic encryption [92, 93] have gained a lot of interest.

The National Institute of Standards and Technology (NIST) formally initiated the process of standardizing post-quantum cryptography (PQC) in 2016 [94], which includes public key encryption (PKE), key encapsulation mechanism (KEM) (a type of key establishment) and digital signature schemes. The first round of candidates were announced in late 2017, with lattice-based cryptography accounting for 48%
of the PKE/KEM schemes and 25% of the signature schemes. In early 2019, the candidates moving on to the second round were announced [95], and lattice-based cryptography accounts for 53% (9 out of 17) and 33% (3 out of 9) of the candidates for PKE/KEM and signature schemes respectively. In 2020, NIST announced the third round finalists and alternate candidates [96], among which 55% (5 out of 9) of the PKE/KEM and 33% (2 out of 6) of the signature schemes are lattice-based. The theoretical foundation of several of these lattice-based protocols lies in the learning with errors (LWE) problem [82] and its variants such as Ring-LWE [83] and Module-LWE [97], and the hardness of LWE has been well-studied in the presence of both classical and quantum adversaries [98,99]. This has been accompanied by several software and hardware implementations [100–111] of LWE and Ring-LWE-based public key encryption and key encapsulation protocols, each supporting specific lattice parameters chosen for increased performance and efficiency. Existing lattice-based cryptography implementations, both in software and hardware, have been thoroughly surveyed in [112]. Most of the hardware implementations focus on FPGA demonstration in order to support reconfigurability of lattice parameters, which is especially important for a fast evolving field like lattice-based cryptography, while existing ASIC implementations either lack configurability or have power and area overheads. Some of the key challenges of implementing lattice-based cryptography in ASICs have been discussed in [113], and this work presents a solution using a combination of architectural and algorithmic techniques.

In this work, we present Sapphire – a configurable lattice cryptography processor – which combines low-power modular arithmetic, area-efficient memory architecture and fast sampling to achieve high energy-efficiency, suitable for securing low-power embedded systems. Our key contributions (described in detail in Sections 3.2, 3.3, 3.4 and 3.5) are summarized as follows:

- Lattice-based cryptography requires polynomial arithmetic operations over different prime fields. To accelerate these computations, we use a low-power modular arithmetic core with configurable prime modulus; a pseudo-configurable modular multiplier is also implemented, which provides up to 3× energy savings.
• The computation cost is dominated by the sampling of polynomials from different discrete probability distributions. We combine an efficient Keccak core with fast post-processing to speed up polynomial sampling by an order of magnitude, while supporting a variety of discrete distribution parameters.

• The memory required for storage and manipulation of polynomials accounts for majority of the hardware area cost. We propose a single-port-memory number theoretic transform (NTT) architecture which provides 30% area savings without any impact on performance or energy-efficiency.

• We integrate these efficient hardware building blocks together with an instruction memory and decoder to build our crypto-processor, which can be programmed with custom instructions for polynomial sampling and arithmetic.

• The Sapphire crypto-processor is coupled with a low-power RISC-V microprocessor (based on an open-source design [43]) to demonstrate several NIST Round 2 lattice-based key encapsulation and signature protocols such as Frodo [114], NewHope [115], qTESLA [116], CRYS-TALS-Kyber [117] and CRYS-TALS-Dilithium [118] using software-hardware co-design. We achieve more than an order of magnitude improvement in performance and energy-efficiency compared to state-of-the-art assembly-optimized software and hardware implementations.

• All the key building blocks, such as NTT, polynomial arithmetic and binomial sampling, are constant-time and we implement countermeasures to prevent timing and simple power analysis attacks. To protect against stronger differential power analysis attacks, our crypto-processor can also be programmed to implement masking-based countermeasures. Experimental validation results are provided.

The software implementation and evaluation is joint work with Abhishek Pathak and Tenzin Ukyab. The software profiling of different PQC schemes on ARM Cortex-M4 is in collaboration with Abhishek Pathak. The interfacing of our crypto-processor with RISC-V software is in collaboration with Tenzin Ukyab.
3.1 Background

In this section, we provide a brief introduction to LWE, Ring-LWE and Module-LWE along with the associated computations. We use bold lower-case symbols to denote vectors and bold upper-case symbols to denote matrices. The symbol \( \log \) is used to denote all logarithms with base 2. The set of all integers is denoted as \( \mathbb{Z} \) and the quotient ring of integers modulo \( q \) is denoted as \( \mathbb{Z}_q \). For two \( n \)-dimensional vectors \( a \) and \( b \), their inner product is written as \( \langle a, b \rangle = \sum_{i=0}^{n-1} a_i \cdot b_i \). The concatenation of two vectors \( a \) and \( b \) is written as \( a \| b \).

3.1.1 LWE and Related Lattice Problems

The Learning with Errors (LWE) problem [82] acts as the foundation for several modern lattice-based cryptography schemes. The LWE problem states that given a polynomial number of samples of the form \( (a, \langle a, s \rangle + e) \), it is difficult to determine secret vector \( s \in \mathbb{Z}_q^n \), where vector \( a \in \mathbb{Z}_q^n \) is sampled uniformly at random and error \( e \) is sampled from the appropriate error distribution \( \chi \). Examples of secure LWE parameters are \((n, q) = (640, 2^{15})\), \((n, q) = (976, 2^{16})\) and \((n, q) = (1344, 2^{16})\) for Frodo-640, Frodo-976 and Frodo-1344 respectively [114].

LWE-based cryptosystems involve large matrix operations which are computationally expensive and also result in large key sizes. To solve this problem, the Ring-LWE problem [83] was proposed, which uses ideal lattices. Let \( \mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1) \) be the ring of polynomials where \( n \) is power of 2. The Ring-LWE problem states that given samples of the form \( (a, a \cdot s + e) \), it is difficult to determine the secret polynomial \( s \in \mathcal{R}_q \), where the polynomial \( a \in \mathcal{R}_q \) is sampled uniformly at random and the coefficients of the error polynomial \( e \) are small samples from the error distribution \( \chi \). Examples of secure Ring-LWE parameters are \((n, q) = (512, 12289)\) and \((n, q) = (1024, 12289)\) for NewHope-512 and NewHope-1024 respectively [115].

Module-LWE [97] provides a middle ground between LWE and Ring-LWE. By using module lattices, it reduces the algebraic structure present in Ring-LWE and increases security while not compromising too much on the computational efficiency. The
Module-LWE problem states that given samples of the form \((a, a^T s + e)\), it is difficult to determine the secret vector \(s \in \mathbb{R}^k\), where the vector \(a \in \mathbb{R}^k\) is sampled uniformly at random and the coefficients of the error polynomial \(e\) are small samples from the error distribution \(\chi\). Examples of secure Module-LWE parameters are \((n, k, q) = (256, 2, 7681), (n, k, q) = (256, 3, 7681)\) and \((n, k, q) = (256, 4, 7681)\) for Kyber-v1-512, Kyber-v1-768 and Kyber-v1-1024 respectively, and \((n, k, q) = (256, 2, 3329), (n, k, q) = (256, 3, 3329)\) and \((n, k, q) = (256, 4, 3329)\) for Kyber-v2-512, Kyber-v2-768 and Kyber-v2-1024 respectively [117].

### 3.1.2 Number Theoretic Transform

While the protocols based on standard lattices (LWE) involve matrix-vector operations modulo \(q\), all the arithmetic is performed in the ring of polynomials \(\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)\) when working with ideal and module lattices. There are several efficient algorithms for polynomial multiplication [119], and the Number Theoretic Transform (NTT) [120,121] is one such technique widely used in lattice-based cryptography.

The NTT is a generalization of the well-known Fast Fourier Transform (FFT) where all the arithmetic is performed in a finite field instead of complex numbers. Instead of working with powers of the \(n\)-th complex root of unity \(\exp(-2\pi j/n)\), NTT uses the \(n\)-th primitive root of unity \(\omega_n\) in the ring \(\mathbb{Z}_q\), that is, \(\omega_n\) is an element in \(\mathbb{Z}_q\) such that \(\omega_n^n = 1 \mod q\) and \(\omega_n^i \neq 1 \mod q\) for \(i \neq n\). In order to have elements of order \(n\), the modulus \(q\) is chosen to be a prime such that \(q \equiv 1 \mod n\). A polynomial \(a(x) \in \mathcal{R}_q\) with coefficients \(a(x) = (a_0, a_1, \ldots, a_{n-1})\) has the NTT representation \(\hat{a}(x) = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_n)\), where

\[
\hat{a}_i = \sum_{j=0}^{n-1} a_j \omega_n^{ij} \mod q \forall i \in [0, n - 1]
\]

The inverse NTT (INTT) operation converts \(\hat{a}(x) = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_n)\) back to \(a(x)\) as

\[
a_i = \frac{1}{n} \sum_{j=0}^{n-1} \hat{a}_j \omega_n^{-ij} \mod q \forall i \in [0, n - 1]
\]
Note that the INTT operation is similar to NTT, except that \( \omega_n \) is replaced by \( \omega_n^{-1} \mod q \) and the final results is divided by \( n \). The PolyBitRev function performs a permutation on the input polynomial \( a \) such that \( \hat{a}[i] = \text{PolyBitRev}(a)[i] = a[\text{BitRev}(i)] \), where BitRev is formally defined as \( \text{BitRev}(i) = \sum_{j=0}^{\log_2 n - 1} (((i \gg j) & 1) \ll (\log_2 n - 1 - i)) \) (for positive integer \( i \) and power-of-two \( n \)), that is, bit-wise reversal of the binary representation of the index \( i \). Since there are \( \log_2 n \) stages in the NTT outer loop, with \( O(n) \) operations in each stage, its time complexity is \( O(n \log_2 n) \). The factors \( \omega \) are called the \textit{twiddle factors}, similar to FFT.

The NTT provides a fast multiplication algorithm in \( \mathcal{R}_q \) with time complexity \( O(n \log n) \) instead of \( O(n^2) \) for schoolbook multiplication. Given two polynomials \( a, b \in \mathcal{R}_q \), their product \( c = a \cdot b \in \mathcal{R}_q \) can be computed as

\[
c = \text{INTT}( \text{NTT}(a) \odot \text{NTT}(b) )
\]

where \( \odot \) denotes coefficient-wise multiplication of the polynomials. Since the product of \( a \) and \( b \), before reduction modulo \( f(x) = x^n + 1 \), has \( 2n \) coefficients, using the above equation directly to compute \( a \cdot b \) will require padding both \( a \) and \( b \) with \( n \) zeros. To eliminate this overhead, the \textit{negative-wrapped convolution} [122] is used, with the additional requirement \( q \equiv 1 \mod 2n \) so that both the \( n \)-th and \( 2n \)-th primitive roots of unity modulo \( q \) exist, respectively denoted as \( \omega_n \) and \( \psi = \sqrt{\omega_n} \mod q \). By multiplying \( a \) and \( b \) coefficient-wise by powers of \( \psi \) before the NTT computation, and by multiplying \( \text{INTT}( \text{NTT}(a) \odot \text{NTT}(b) ) \) coefficient-wise by powers of \( \psi^{-1} \mod q \), no zero padding is required and the \( n \)-point NTT can be used directly.

Similar to FFT, the NTT inner loop involves butterfly computations. There are two types of butterfly operations – Cooley-Tukey (CT) and Gentleman-Sande (GS) [123]. The CT butterfly-based NTT requires inputs in normal order and generates outputs in bit-reversed order, similar to the \textit{decimation-in-time} FFT. The GS butterfly-based NTT requires inputs to be in bit-reversed order while the outputs are generated in normal order, similar to the \textit{decimation-in-frequency} FFT. Using the same butterfly for both NTT and INTT requires a bit-reversal permutation. However, the bit-reversal can be avoided by using CT for NTT and GS for INTT [123].
3.1.3 Sampling

In lattice-based protocols, the public vectors $a$ are generated from the uniform distribution over $\mathbb{Z}_q$ through rejection sampling. The secret vectors $s$ and error terms $e$ are sampled from the distribution $\chi$ typically with zero mean and appropriate standard deviation $\sigma$. Accurate sampling of $s$ and $e$ is critical to the security of these protocols, and the sampling must be constant-time to prevent side-channel leakage of the secret information. Although the original LWE proof used discrete Gaussian distributions for sampling the error terms, several lattice-based schemes use binomial, uniform and ternary distributions for efficiency. A detailed survey of different sampling techniques is available in [112].

3.2 Modular Arithmetic and NTT

The core arithmetic and logic unit (ALU) of the Sapphire crypto-processor consists of a 24-bit data-path, with modular operations in $\mathbb{F}_q$ for configurable $q$. In this section, we describe the details of our energy-efficient modular arithmetic implementation, the ALU design and our area-efficient NTT memory architecture.

3.2.1 Modular Arithmetic Implementation

The modular arithmetic core consists of a 24-bit adder, a 24-bit subtractor and a 24-bit multiplier along with associated modular reduction logic. Our modular adder and subtractor designs are shown in Fig. 3-1. Both designs use a pair of adder and subtractor, and modular reduction is performed using conditional subtraction and addition, which are computed in the same cycle to avoid timing side-channels.

For modular multiplication, we use a 24-bit multiplier followed by Barrett reduction [124] modulo a prime $q$ of size up to 24 bits. Barrett reduction does not exploit any special property of the modulus $q$, thus making it ideal for supporting configurable moduli. Let $z$ be the 48-bit product to be reduced to $\mathbb{Z}_q$, then Barrett reduction computes $z \mod q$ by estimating the quotient $\lfloor z/q \rceil$ without performing any division.
Barrett reduction involves two multiplications, one subtraction, one bit-shift and one conditional subtraction. The value of $1/q$ is approximated as $m/2^k$, with the error of approximation being $e = 1/q - m/2^k$, therefore the reduction is valid as long as $ze < 1$. Since $z < q^2$, $k$ is set to be the smallest number such that $e = 1/q - ([2^k/q]/2^k) < 1/q^2$. Typically, $k$ is very close to $2 \lceil \lg q \rceil$, the bit-size of $q^2$.

In order to understand the trade-offs between flexibility and efficiency in modular multiplication, we have implemented two different architectures of Barrett reduction logic: (1) with fully configurable modulus ($q$ can be any arbitrary prime within a certain range) and (2) with pseudo-configurable modulus ($q$ belongs to a specific set of primes), as shown in Fig. 3-2.

Apart from the prime $q$ (which can be up to 24 bits), the fully configurable version requires two additional inputs $m$ and $k$ such that $m = [2^k/q]$ ($m$ and $k$ are allowed to be up to 24 bits and 6 bits respectively). It consists of total 3 multipliers, as shown in Fig. 3-2a, the first two being used to compute $z = x \cdot y$ and $z \cdot m$ respectively. For obtaining $t = (z \cdot m) \gg k$, the bit-wise shift is implemented purely using combinational
logic (multiplexers) because shifting bits sequentially in registers can be extremely inefficient in terms of power consumption. We assume that \(16 \leq k \leq 48\) since \(q\) is not larger than 24 bits, \(q\) is typically not smaller than 8 bits and we know that \(k \approx 2^{\lceil \log q \rceil}\).

The third multiplier is used to compute \(t \cdot q\), and a pair of subtractors is used to calculate \(z - (t \cdot q)\) and perform the final reduction step. All the steps are computed in a single cycle to avoid any potential timing side-channels.

The pseudo-configurable modular multiplier implements Barrett reduction logic for the following primes used by NIST Round 1 lattice-based candidates: 7681 (CRYSTALS-Kyber-v1) [117], 12289 (NewHope) [115], 40961 (R.EMBLEM) [125], 65537 (pqNTRUSign) [126], 120833 (Ding Key Exchange) [127], 133121 / 184321 (LIMA) [128], 8380417 (CRYSTALS-Dilithium) [118], 8058881 (qTESLA v1.0) and 4205569 / 4206593 / 8404993 (qTESLA v2.0) [116]. As shown in Fig. 3-2b, there are dedicated reduction blocks for each of these primes, and the \(q_{SEL}\) input is used to select the output of the appropriate block while the inputs to the other blocks are data-gated to save power. Since the reduction blocks have the parameters \(m, k\) and \(q\) coded in digital logic and do not require explicit multipliers, they involve lesser computation than the fully configurable reduction circuit from Fig. 3-2a, albeit at the cost of some additional area and decrease in flexibility. The reduction becomes particularly efficient when at least one of \(m\) and \(q\) or both can be written in the form \(2^{l_1} \pm 2^{l_2} \pm \cdots \pm 1\), where \(l_1, l_2, \cdots\) are not more than four positive integers. For example, let us consider the CRYSTALS primes: for \(q = 7681 = 2^{13} - 2^9 + 1\) we have \(k = 21\) and \(m = 273 = 2^8 + 2^4 + 1\), and for \(q = 8380417 = 2^{23} - 2^{13} + 1\) we have \(k = 46\) and \(m = 8396807 = 2^{23} + 2^{13} + 2^3 - 1\). Therefore, the multiplications by \(q\) and \(m\) can be converted to significantly cheaper bit-shifts and additions / subtractions. Further details about the optimized reduction algorithms are available in [129–131]. With similar optimized reduction for all the supported primes, this design also performs modular multiplication in a single cycle.

In Fig. 3-3, we compare post-synthesis simulated energy (at 1.1 V and 100 MHz) of the two modular multiplier architectures. As expected, the multiplication itself consumes the same energy in both cases, but the modular reduction energy is up
Figure 3-3: Comparison of simulated modular multiplication energy for the two reduction architectures – configurable and pseudo-configurable.

to 6× lower for the pseudo-configurable design. The overall decrease in modular multiplication energy, considering both multiplication and reduction together, is up to 3×, clearly highlighting the benefit of the dedicated modular reduction data-paths when working with prime moduli. For reduction modulo $2^m (m < 24)$, e.g., in the case of Frodo, the output of the 24-bit multiplier is simply bit-wise AND-ed with $2^m - 1$ implying that the modular reduction energy is negligible.

### 3.2.2 Butterfly Unit and ALU

Next, we elaborate how the modular arithmetic units described earlier are integrated together to build the butterfly module. As discussed in Section 3.1, NTT computations involve butterfly operations similar to the Fast Fourier Transform, with the only difference being that all arithmetic is performed modulo $q$ instead of complex numbers. There are two butterfly configurations – Cooley-Tukey (or DIT) and Gentleman-Sande (or DIF). In terms of arithmetic, the DIT butterfly computes $(a + \omega b \mod q, a - \omega b \mod q)$ and the DIF butterfly computes $(a + b \mod q, (a - b)\omega \mod q)$, where $a$ and $b$ are the inputs to the butterfly and $\omega$ is the twiddle factor. The DIT butterfly requires inputs to be in bit-reversed order and the DIF butterfly generates outputs in bit-reversed order, thus making DIF and DIT suitable for NTT and INTT respectively.

While software implementations have the flexibility to program both configurations, hardware designs typically implement either DIT or DIF, thus requiring bit-reversals.
To solve this problem, we have implemented a unified butterfly architecture which can be configured as both DIT and DIF, as shown in Fig. 3-4. It consists of two sets of modular adders and subtractors along with some multiplexing circuitry to select whether the multiplication with $\omega$ is performed before or after the addition and subtraction. Since the critical path of the design is inside the modular multiplier, there is no impact on system performance. The associated area overhead is also negligible.

The modular arithmetic blocks inside the butterfly are re-used for coefficient-wise polynomial arithmetic operations as well as for multiplying polynomials with the appropriate powers of $\psi$ and $\psi^{-1}$ during negative-wrapped convolution. Apart from butterfly and arithmetic modulo $q$, the Sapphire arithmetic and logic unit (ALU) also supports the bit-wise operations AND, OR, XOR, left shift and right shift.

### 3.2.3 NTT Memory Architecture

Hardware architectures for polynomial multiplication using NTT consist of memory banks for storing the polynomials along with the ALU which performs butterfly computations. Since each butterfly needs to read two inputs and write two outputs all in the same cycle, these memory banks are typically implemented using dual-port RAMs [100, 110, 121, 132] or four-port RAMs [108]. Although true dual-port
memory is easily available in state-of-the-art commercial FPGAs in the form of block RAMs (BRAMs), use of dual-port SRAMs in ASIC can pose large area overheads in resource-constrained devices. Compared to a simple single-port SRAM, a dual-port SRAM has double the number of row and column decoders, write drivers and read sense amplifiers. Also, the bit-cells in a low-power dual-port SRAM consist of ten transistors (10T) compared to the usual six transistor (6T) bit-cells in a single-port SRAM [133]. Therefore, area of a dual-port SRAM can be as much as double that of a single-port SRAM with the same number of bits and column muxing. To reduce this area overhead, we implement an area-efficient NTT memory architecture which uses the constant-geometry FFT data-flow [134] and consists of single-port SRAMs only.

In the constant geometry NTT algorithm [132,135], coefficients of the polynomial are accessed in the same order for each stage, thus simplifying the read/write control circuitry. For constant geometry DIT NTT, the butterfly inputs are $a[2j]$ and $a[2j + 1]$ and the outputs are $\hat{a}[j]$ and $\hat{a}[j + n/2]$, while the inputs are $a[j]$ and $a[j + n/2]$ and
the outputs are $\hat{a}[2j]$ and $\hat{a}[2j+1]$ for DIF NTT. However, the constant geometry NTT is inherently out-of-place, therefore requiring storage for both polynomials $a$ and $\hat{a}$. For our hardware implementation, we create two memory banks – left and right – to store these two polynomials while allowing the butterfly inputs and outputs to ping-pong between them during each stage of the transform. Although out-of-place NTT requires storage for both the input and output polynomials, this does not affect the total memory requirements of the crypto-processor because the total number of polynomials required to be stored during the protocol execution is greater than two, e.g., four polynomials are involved in any computation of the form $b = a \cdot s + e$.

Next, we describe how these memory banks are constructed using single-port SRAMs so that each butterfly can be computed in a single cycle without causing read/write hazards. As shown in Fig. 3-5a, each polynomial is split among four single port SRAMs Mem 0-3 on the basis of the least and most significant bits (LSB and MSB) of the coefficient index (or address $addr$). This allows simultaneously accessing coefficient index pairs of the form $(2j, 2j+1)$ and $(j, j+n/2)$. Our NTT memory architecture is shown in Fig. 3-5b, which consists of two such memory banks labelled as LWE Poly Mem. In every cycle, the butterfly inputs are read from two different single-port SRAMs (out of four SRAMs in the input memory bank) and the outputs are also written to two different single-port SRAMs (out of four SRAMs in the output memory bank), thus avoiding hazards. The data flow in the first two cycles of NTT is shown in Fig. 3-6, where the input polynomial $a$ is stored in the left bank and the output polynomial $\hat{a}$ is stored in the right bank.
output polynomials exchange their memory banks from one stage to the next, our NTT control circuitry ensures that the same data-flow is maintained. To illustrate this, the memory access patterns for all three stages of an 8-point NTT are shown in Fig. 3-7 for both decimation-in-time and decimation-in-frequency.

The two memory banks consist of four 1024 × 24-bit single-port SRAMs each (24 KB total). Together they store 8192 entries, which can be split into four 2048-dimension polynomials or eight 1024-dimension polynomials or sixteen 512-dimension polynomials or thirty-two 256-dimension polynomials or sixty-four 128-dimension polynomials or one-hundred-twenty-eight 64-dimension polynomials. By constructing this memory using single-port SRAMs (and some additional read-data multiplexing circuitry), we have achieved area savings equivalent to 124k GE compared to a dual-port SRAM-
based implementation. This is particularly important since SRAMs account for a large portion of the total hardware area in ASIC implementations of lattice-based cryptography [108, 136].

In order to allow configurable parameters, our NTT hardware also requires additional storage (labelled as NTT Constants RAM in Fig. 3-5) for the pre-computed twiddle factors: $\omega_{2i}^j$, $\omega_{2i}^{-j} \mod q$ for $i \in [1, \lg n]$ and $j \in [0, 2^{i-1})$ and $\psi^i$, $n^{-1}\psi^{-i} \mod q$ for $i \in [0, n)$. Since $n \leq 2048$ and $q < 2^{24}$, this would require another 24 KB of memory. To reduce this overhead, we exploit the following properties of $\omega$ and $\psi$: $\omega_{n/2} = \omega_n^2$, $\omega_{n}^{-j} = \omega_{n}^{n-j}$ and $\omega = \psi^2$ [121]. Then, it’s sufficient to store only $\omega_n^j$ for $j \in [0, n/2)$ and $\psi^i$, $n^{-1}\psi^{-i} \mod q$ for $i \in [0, n)$, thus reducing the twiddle factor memory size by 37.5\% down to 15 KB.

Finally, we compare the energy-efficiency and performance of our NTT with state-of-the-art software and ASIC hardware implementations in Table 3.1. For the software implementation, we have used assembly-optimized code for ARM Cortex-M4 from the PQM4 crypto library [137], and measurements were performed using the NUCLEO-F411RE development board [138]. Total cycle count of our NTT is $(\frac{n}{2} + 1) \lg n + (n + 1)$, including the multiplication of polynomial coefficients with powers of $\psi$. Our hardware-accelerated NTT is order of magnitude more energy-efficient than the software implementation, after accounting for voltage scaling. It is also significantly more energy-efficient compared to previous NTT hardware designs [105] and [105] respectively. The energy-efficiency of our NTT implementation is largely due to the careful design of low-power modular arithmetic, as discussed earlier, which decreases overall modular reduction complexity and simplifies the logic circuitry. However, our NTT is still less energy-efficient compared to [108], primarily due to the fact that [108] uses 16 parallel butterfly units along with dedicated four-port scratch-pad buffers to achieve higher parallelism and lower energy consumption at the cost of significantly larger chip area (2.05 mm$^2$) compared to our design (0.28 mm$^2$). As will be discussed in Section 3.5, sampling accounts for majority of the computational cost in Ring-LWE and Module-LWE schemes, therefore justifying our choice of area-efficient NTT architecture at the cost of some energy overhead.
### Table 3.1: Comparison of our NTT with state-of-the-art

<table>
<thead>
<tr>
<th>Design</th>
<th>Platform</th>
<th>Tech (nm)</th>
<th>VDD (V)</th>
<th>Parameters</th>
<th>NTT Cycles</th>
<th>NTT Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>ASIC</td>
<td>40</td>
<td>0.68</td>
<td>$(n = 256, q = 7681)$</td>
<td>1,289</td>
<td>63.43 nJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(n = 512, q = 12289)$</td>
<td>2,826</td>
<td>156.88 nJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(n = 1024, q = 12289)$</td>
<td>6,155</td>
<td>341.75 nJ</td>
</tr>
<tr>
<td>Software</td>
<td>ARM Cortex-M4</td>
<td>-</td>
<td>3.0</td>
<td>$(n = 256, q = 7681)$</td>
<td>22,031</td>
<td>13.55 μJ</td>
</tr>
<tr>
<td>[137]</td>
<td></td>
<td></td>
<td></td>
<td>$(n = 512, q = 12289)$</td>
<td>34,262</td>
<td>21.07 μJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(n = 1024, q = 12289)$</td>
<td>75,006</td>
<td>46.13 μJ</td>
</tr>
<tr>
<td>Song et al. [108]</td>
<td>ASIC</td>
<td>40</td>
<td>0.9</td>
<td>$(n = 256, q = 7681)$</td>
<td>160</td>
<td>31 nJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(n = 512, q = 12289)$</td>
<td>492</td>
<td>96 nJ</td>
</tr>
<tr>
<td>Nejatollahi et al. [105]</td>
<td>ASIC</td>
<td>45</td>
<td>1.0</td>
<td>$(n = 512, q = 12289)$</td>
<td>2,854</td>
<td>1016.02 nJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11,053</td>
<td>596.86 nJ</td>
</tr>
<tr>
<td>Fritzmann et al. [136]</td>
<td>ASIC</td>
<td>65</td>
<td>1.2</td>
<td>$(n = 256, q = 7681)$</td>
<td>2,056</td>
<td>254.52 nJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(n = 512, q = 12289)$</td>
<td>4,616</td>
<td>549.98 nJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(n = 1024, q = 12289)$</td>
<td>10,248</td>
<td>1205.03 nJ</td>
</tr>
<tr>
<td>Roy et al. [100]</td>
<td>FPGA</td>
<td>-</td>
<td>-</td>
<td>$(n = 256, q = 7681)$</td>
<td>1,691</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(n = 512, q = 12289)$</td>
<td>3,443</td>
<td>-</td>
</tr>
<tr>
<td>Du et al. [121]</td>
<td>FPGA</td>
<td>-</td>
<td>-</td>
<td>$(n = 256, q = 7681)$</td>
<td>4,066</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(n = 512, q = 12289)$</td>
<td>8,806</td>
<td>-</td>
</tr>
</tbody>
</table>

### 3.3 Discrete Distribution Sampler

Hardness of the LWE problem and its derivatives, e.g., Ring-LWE and Module-LWE, is directly related to statistical properties of the error samples. Therefore, an accurate and efficient sampler is a critical component of any lattice cryptography implementation. Sampling accounts for a major portion of the computational overhead in software implementations of protocols based on ideal and module lattices [129, 139].

A cryptographically secure pseudo-random number generator (CS-PRNG) is used to generate uniformly random numbers, which are then post-processed to convert them into samples from different discrete probability distributions. In this section, we describe our design of energy-efficient CS-PRNG along with fast sampling techniques for configurable distribution parameters. Our design can be used to efficiently sample from various distributions such as uniform, binomial, trinary and discrete Gaussian.
Table 3.2: Comparison of CS-PRNG designs

<table>
<thead>
<tr>
<th>PRNG</th>
<th>Area (kGE) a</th>
<th>Cycles / Round</th>
<th>No. of PRNG Bits</th>
<th>Energy (pJ/bit) b</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAKE-128</td>
<td>34.5 (23.5)</td>
<td>24</td>
<td>1344</td>
<td>0.64</td>
</tr>
<tr>
<td>SHAKE-256</td>
<td></td>
<td></td>
<td></td>
<td>1088</td>
</tr>
<tr>
<td>ChaCha20</td>
<td>21.1 (17.5)</td>
<td>20</td>
<td>512</td>
<td>1.35</td>
</tr>
<tr>
<td>AES-128-CTR</td>
<td>15.0 (11.1)</td>
<td>11</td>
<td>128</td>
<td>1.95</td>
</tr>
<tr>
<td>AES-256-CTR</td>
<td></td>
<td>15</td>
<td>128</td>
<td>2.89</td>
</tr>
</tbody>
</table>

a Area of placed-and-routed design (post-synthesis area in brackets)

b Energy measured from test chip operating at 0.68 V

3.3.1 Energy-Efficient CS-PRNG

Some of the standard choices for CS-PRNG are SHA-3 in the SHAKE mode [31], AES in counter mode [29] and ChaCha20 [140]. In order to identify the most efficient among these, we have compared them in terms of area, pseudo-random bit generation performance and energy consumption, as shown in Table 3.2. Only place-and-route area and measured energy are considered for all analysis, and synthesis area is reported for reference. For fair comparison, all the three primitives – SHA-3, AES and ChaCha20 – were implemented as full data path architectures. From Fig. 3-8, we observe that although all three primitives have comparable area-energy product, SHA-3 is $2 \times$ more energy-efficient than ChaCha20 and $3 \times$ more energy-efficient than AES; and this is mostly because SHA-3 generates the highest number of pseudo-random bits per round.

![Figure 3-8](image)

Figure 3-8: Analysis of SHAKE-128, SHAKE-256, AES-128-CTR, AES-256-CTR and ChaCha20 in terms of energy per bit, bits per cycle and area-energy product.
The basic building block of SHA-3 is the Keccak permutation function [141]. Therefore, our PRNG consists of a 24-cycle Keccak-f[1600] core which can be configured in different SHA-3 modes and consumes 0.89 nJ per round at 0.68 V. Its 1600-bit state is processed in parallel, thus avoiding expensive register shifts and multiplexing required in serial architectures. Fig. 3-9 shows the overall architecture our discrete distribution sampler with the energy-efficient SHA-3 core. Pseudo-random bits generated by SHAKE-128 or SHAKE-256 are stored in the 1600-bit Keccak state register, and shifted out 32 bits at a time as required by the sampler. The sampler then feeds these bits (AND-ed with the appropriate bit mask to truncate them to desired size) to the post-processing logic to perform one of the following five operations – rejection sampling in $[0, q)$, binomial sampling with standard deviation $\sigma$, discrete Gaussian sampling with standard deviation $\sigma$ and desired precision up to 32 bits, uniform sampling in $[-\eta, \eta]$ for $\eta < q$ and trinary sampling in $\{-1, 0, +1\}$ with specified weights for the $+1$ and $-1$ samples.

### 3.3.2 Rejection Sampling

The public polynomial $a$ in Ring-LWE and the public vector $\mathbf{a}$ in Module-LWE have their coefficients uniformly drawn from $\mathbb{Z}_q$ through rejection sampling, where uniformly random numbers of desired bit size are obtained from the PRNG as candidate samples and only numbers smaller than $q$ are accepted. The probability that a random number is not accepted is known as the rejection probability. For prime $q$, the rejection probability is calculated as $(1 - q/2^{\lfloor \lg q \rfloor})$. In Table 3.3, we list the rejection probabilities for primes mentioned earlier in Section 3.2. Clearly, different primes have
Table 3.3: Rejection probabilities for different primes with and without fast sampling

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7681</td>
<td>13</td>
<td>0.06</td>
<td>1</td>
<td>0.06</td>
<td>-</td>
</tr>
<tr>
<td>12289</td>
<td>14</td>
<td>0.25</td>
<td>5</td>
<td>0.06</td>
<td>0.19</td>
</tr>
<tr>
<td>40961</td>
<td>16</td>
<td>0.37</td>
<td>3</td>
<td>0.06</td>
<td>0.31</td>
</tr>
<tr>
<td>65537</td>
<td>17</td>
<td>0.50</td>
<td>7</td>
<td>0.12</td>
<td>0.38</td>
</tr>
<tr>
<td>120833</td>
<td>17</td>
<td>0.08</td>
<td>1</td>
<td>0.08</td>
<td>-</td>
</tr>
<tr>
<td>133121</td>
<td>18</td>
<td>0.49</td>
<td>7</td>
<td>0.11</td>
<td>0.38</td>
</tr>
<tr>
<td>184321</td>
<td>18</td>
<td>0.30</td>
<td>11</td>
<td>0.03</td>
<td>0.27</td>
</tr>
<tr>
<td>8380417</td>
<td>23</td>
<td>≈ 0</td>
<td>1</td>
<td>≈ 0</td>
<td>-</td>
</tr>
<tr>
<td>8058881</td>
<td>23</td>
<td>0.04</td>
<td>1</td>
<td>0.04</td>
<td>-</td>
</tr>
<tr>
<td>4205569</td>
<td>23</td>
<td>0.50</td>
<td>7</td>
<td>0.12</td>
<td>0.38</td>
</tr>
<tr>
<td>4206593</td>
<td>23</td>
<td>0.50</td>
<td>7</td>
<td>0.12</td>
<td>0.38</td>
</tr>
<tr>
<td>8404993</td>
<td>24</td>
<td>0.50</td>
<td>7</td>
<td>0.12</td>
<td>0.38</td>
</tr>
</tbody>
</table>

very different rejection probabilities, often as high as 50%, which can be a bottleneck in lattice-based protocols. To solve this problem, we refer to [142] where pseudo-random numbers smaller than 5q are accepted for q = 12289, thus reducing the rejection probability from 25% to 6%. We extend this technique for any prime q by scaling the rejection bound from q to kq, for appropriate small integer k, so that the rejection probability is now \((1 - kq/2^{\lceil \lg kq \rceil})\). We list these scaling factors for the primes in Table 3.3 along with the corresponding decrease in rejection probability.

Although this method reduces rejection rates, the output samples now lie in \([0, kq]\) instead of \([0, q]\). In [142], for q = 12289 and k = 5, the accepted samples are reduced to \(\mathbb{Z}_q\) by subtracting q from them up to four times. Since k is not fixed for our rejection sampler, we employ Barrett reduction [124] for this purpose. Unlike modular multiplication, where the inputs lie in \([0, q^2]\), the inputs here are much smaller; so the Barrett reduction parameters are also quite small, therefore requiring little additional logic. In Table 3.4, we compare our rejection sampler performance (SHAKE-128 used as PRNG) with software implementation on ARM Cortex-M4 using assembly-optimized Keccak [137].
### Table 3.4: Comparison of rejection sampling with software

<table>
<thead>
<tr>
<th>Design</th>
<th>Platform</th>
<th>Tech (nm)</th>
<th>VDD (V)</th>
<th>Parameters</th>
<th>Samp. Cycles</th>
<th>Samp. Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>ASIC</td>
<td>40</td>
<td>0.68</td>
<td>$(n = 256, q = 7681)$ $(n = 512, q = 12289)$ $(n = 1024, q = 12289)$</td>
<td>461</td>
<td>19.45 nJ</td>
</tr>
<tr>
<td>Software [137]</td>
<td>ARM Cortex-M4</td>
<td>-</td>
<td>3.0</td>
<td>$(n = 256, q = 7681)$ $(n = 512, q = 12289)$ $(n = 1024, q = 12289)$</td>
<td>60,433</td>
<td>37.17 μJ</td>
</tr>
</tbody>
</table>

#### 3.3.3 Binomial Sampling

For binomial sampling, we take two $k$-bit chunks from the PRNG and computes the difference of their Hamming weights, as proposed in [115]. The resulting samples follow a binomial distribution with standard deviation $\sigma = \sqrt{k/2}$. We allow configuring $k$ to any value up to 32, thus providing the flexibility to support different standard deviations. We compare our binomial sampling performance (SHAKE-256 used as PRNG) with state-of-the-art software and hardware implementations in Table 3.5. Our sampler is more than two orders of magnitude more energy-efficient compared to the software implementation on ARM Cortex-M4 which uses assembly-optimized Keccak [137]. It is also an order of magnitude more efficient than [108] which uses Knuth-Yao sampling [143] for binomial distributions with ChaCha20 as PRNG.

### Table 3.5: Comparison of binomial sampling with state-of-the-art

<table>
<thead>
<tr>
<th>Design</th>
<th>Platform</th>
<th>Tech (nm)</th>
<th>VDD (V)</th>
<th>Parameters</th>
<th>Samp. Cycles</th>
<th>Samp. Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>ASIC</td>
<td>40</td>
<td>0.68</td>
<td>$(n = 256, k = 4)$ $(n = 512, k = 8)$ $(n = 1024, k = 8)$</td>
<td>505</td>
<td>22.24 nJ</td>
</tr>
<tr>
<td>Software [137]</td>
<td>ARM Cortex-M4</td>
<td>-</td>
<td>3.0</td>
<td>$(n = 256, k = 4)$ $(n = 512, k = 8)$ $(n = 1024, k = 8)$</td>
<td>52,603</td>
<td>32.35 μJ</td>
</tr>
<tr>
<td>Song et al. [108]</td>
<td>ASIC</td>
<td>40</td>
<td>0.9</td>
<td>$(n = 512, k = 16)$</td>
<td>3,704</td>
<td>1.25 μJ</td>
</tr>
<tr>
<td>Oder et al. [104]</td>
<td>FPGA</td>
<td>-</td>
<td>-</td>
<td>$(n = 1024, k = 16)$</td>
<td>33,792</td>
<td>-</td>
</tr>
</tbody>
</table>
3.3.4 Discrete Gaussian Sampling

Our discrete Gaussian sampler implements the inversion method of sampling [144] from a discrete symmetric zero-mean distribution \( \chi \) on \( \mathbb{Z} \) with small support which approximates a rounded continuous Gaussian distribution, e.g., in Frodo [114] and R.EMBLEM [125]. For a distribution with support \( S_\chi = \{-s, \cdots, -1, 0, 1, \cdots, s\} \), where \( s \) is a small positive integer, the probabilities \( \Pr(z) \) for \( z \in S_\chi \), such that \( \Pr(z) = \Pr(-z) \) can be derived from the cumulative distribution table (CDT) \( T_\chi = (T_\chi[0], T_\chi[1], \cdots, T_\chi[s]) \), where \( 2^{-r} \cdot T_\chi[0] = \Pr(0)/2 - 1 \) and \( 2^{-r} \cdot T_\chi[z] = \Pr(0)/2 - 1 + \sum_{i=1}^{z-1} \Pr(i) \) for \( z \in [1, s] \) for a given precision \( r \).

The sampling must be constant-time in order to eliminate timing side-channels, therefore the algorithm does a complete loop through the entire table \( T_\chi \). The comparison of each \( T_\chi[z] \) with the random input must also be constant-time. Our design adheres to these requirements and uses a \( 64 \times 32 \) RAM to store the CDT, allowing the parameters \( s \leq 64 \) and \( r \leq 32 \) to be configured according to the choice of the distribution. In Table 3.6, we have compared our discrete Gaussian sampler performance (SHAKE-256 used as PRNG) with software implementation on ARM Cortex-M4 using assembly-optimized Keccak [137], and we observe more than an order of magnitude improvement in energy-efficiency after accounting for voltage scaling. Hardware architectures for Knuth-Yao sampling have been proposed by [100] and [108], but they are for discrete Gaussian distributions with larger standard deviation and higher precision, which are not required for the post-quantum cryptography protocols supported by our design.

Table 3.6: Comparison of discrete Gaussian sampling with software

<table>
<thead>
<tr>
<th>Design</th>
<th>Platform</th>
<th>Tech (nm)</th>
<th>VDD (V)</th>
<th>Parameters</th>
<th>Samp. Cycles</th>
<th>Samp. Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>ASIC</td>
<td>40</td>
<td>0.68</td>
<td>((n = 512, \sigma = 25.0, s = 54))</td>
<td>29,169</td>
<td>471.08 nJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>((n = 1024, \sigma = 2.75, s = 11))</td>
<td>15,330</td>
<td>247.58 nJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>((n = 1024, \sigma = 2.30, s = 10))</td>
<td>14,306</td>
<td>231.04 nJ</td>
</tr>
<tr>
<td>Software</td>
<td>ARM Cortex-M4</td>
<td>-</td>
<td>3.0</td>
<td>((n = 512, \sigma = 25.0, s = 54))</td>
<td>397,921</td>
<td>244.72 (\mu)J</td>
</tr>
<tr>
<td>[137]</td>
<td></td>
<td></td>
<td></td>
<td>((n = 1024, \sigma = 2.75, s = 11))</td>
<td>325,735</td>
<td>200.33 (\mu)J</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>((n = 1024, \sigma = 2.30, s = 10))</td>
<td>317,541</td>
<td>195.29 (\mu)J</td>
</tr>
</tbody>
</table>
3.3.5 Other Distributions

Several lattice-based protocols, such as CRYSTALS-Dilithium [118] and qTESLA [116], require polynomials to be sampled with coefficients uniformly distributed in the range \([-\eta, \eta]\) for a specified bound \(\eta < q\). For this, we again use rejection sampling. Unlike rejection sampling from \(\mathbb{Z}_q\), we do not require any special techniques since \(\eta\) is typically small or an integer close to a power of two. We have also implemented a trinary sampler for polynomials with coefficients from \([-1, 0, +1]\). We classify these polynomials into three categories: (1) with \(m\) non-zero coefficients, (2) with \(m_0 +1\)'s and \(m_1 -1\)'s, and (3) with coefficients distributed as \(\Pr(x = 1) = \Pr(x = -1) = \rho/2\) and \(\Pr(x = 0) = 1 - \rho\) for \(\rho \in \{1/2, 1/4, 1/8, \cdots, 1/128\}\). For (1) and (2), we start with a zero-polynomial of size \(n\). Then, uniformly random coefficient indices \(\in [0, n)\) are generated, and the corresponding coefficients are replaced with \(-1\) or \(+1\) if they are zero [116,126]. For (3), sampling is based on the observation [145] that for a uniformly random number \(x \in [0, 2^k]\) we have \(\Pr(x = 0) = 1/2^k\), \(\Pr(x = 1) = 1/2^k\) and \(\Pr(x \in [2, 2^k)) = 1 - 1/2^k\). Therefore, for the appropriate value of \(k \in [1, 7]\), we can generate samples from the desired trinary distribution with \(\rho = 1/2^k\).

3.4 Configurable Lattice Crypto-Processor

The top-level architecture of our Sapphire crypto-processor is shown in Fig. 3-10. The efficient building blocks described in Sections 3.2 and 3.3 are integrated together with a 1 KB instruction memory and an instruction decoder to form the core of our crypto-processor. It can be programmed using 32-bit custom instructions to perform different polynomial arithmetic, transform and sampling operations, as well as simple branching. Details of programming the crypto-processor are provided in Appendix E.

We use dedicated clock gates for fine-grained power savings during program execution, and an interrupt pin is used to indicate completion of the program. The crypto-processor’s memory and data registers are accessed through a simple memory-mapped interface. Examples of programming the crypto-processor to implement Ring-LWE and Module-LWE computations are provided in Appendix E.
3.5 Implementation Results

3.5.1 System Architecture

The Sapphire crypto-processor is coupled with a low-power RISC-V micro-processor [43] (using the memory-mapped interface), as shown in Fig. 3-11, with 32 KB instruction memory and 64 KB data memory, which implements the RV32IM instruction set [42] and has Dhrystone performance similar to ARM Cortex-M0. The RISC-V core has a 1-cycle multiplier and a 32-cycle divider. When executing cryptographic workloads in the Sapphire core, the RISC-V core can be clock-gated using the \textit{wait-for-interrupt} (\texttt{wfi}) instruction. The processor is woken up by a dedicated interrupt from the Sapphire core, which is raised when the cryptographic operation is complete. Using the memory-mapped interface ensures that the cryptographic core can be accessed through simple load and store instructions, without requiring any custom instructions or changes to the compilation toolchain. While the cryptographic core is used to accelerate all lattice cryptography computations, the RISC-V processor is used for scheduling the cryptographic workloads as well as for compression and decompression.
of public keys and ciphertexts. The Keccak-f[1600] core inside Sapphire can be accessed standalone through RISC-V software, and is used to accelerate SHA-3 hashing and extendable output functions according to the requirements of the protocol.

Our test chip was fabricated in the TSMC 40nm low-power CMOS process, and the chip micrograph is shown in Fig. 3-12 with the key design components highlighted.

![Figure 3-11: Chip architecture with Sapphire crypto core and RISC-V micro-processor.](image)

<table>
<thead>
<tr>
<th>Chip Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
</tr>
<tr>
<td>Supply Voltage</td>
</tr>
<tr>
<td>Package</td>
</tr>
<tr>
<td>Die Size</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lattice Cryptography Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Area</td>
</tr>
<tr>
<td>Logic Gates</td>
</tr>
<tr>
<td>SRAM</td>
</tr>
<tr>
<td>Max. Frequency</td>
</tr>
<tr>
<td>Lattice Parameters</td>
</tr>
<tr>
<td>CS-PRNG</td>
</tr>
<tr>
<td>Hash Function</td>
</tr>
</tbody>
</table>

![Figure 3-12: Chip micrograph and test chip specifications.](image)
Figure 3-13: Effects of supply voltage scaling as measured from our test chip - (a) leakage current (b) average active current and maximum frequency.

The final placed-and-routed design of our Sapphire core consists of 106k logic gates (76 kGE for synthesized design) and 40.25 KB SRAM, with a total area of 0.28 mm$^2$ (logic and memory combined). Our test chip supports supply voltage scaling from 0.68 V to 1.1 V. Fig. 3-13 shows the effect of voltage scaling on leakage current, average active current and maximum operating frequency of our test chip.

Fig. 3-14 shows our test board and measurement setup. The test chip is housed in a QFN64 socket soldered to the board, an Opal Kelly XEM7001 FPGA development board [69] is used to interface with the chip, and a Keithley 2602A source meter [70] supplies power to the chip. Both the FPGA and the source meter are controlled from a host computer through USB and GPIB interfaces respectively. The FPGA is used to
transfer programs from the host computer to the instruction memory of our test chip. Also, a small ring-oscillator-based true random number generator [146] implemented on the FPGA is connected to our test chip through general purpose input-output (GPIO) pins for providing fresh random inputs to the `randombytes` function which is part of the NIST API. All lattice cryptography programs are written using custom instructions and compiled with a Perl script, while all RISC-V software is written in C and compiled using the `riscv-gcc` toolchain [147].

To measure the efficiency of our design, we have implemented the following NIST Round 2 lattice-based cryptography protocols on our test chip:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Lattice Problem</th>
<th>NIST Sec.</th>
<th>Parameter Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CCA-Secure Key Encapsulation Mechanism (KEM) Protocols</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NewHope</td>
<td>Ring-LWE</td>
<td>1</td>
<td>NewHope-512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>NewHope-1024</td>
</tr>
<tr>
<td>CRystals-Kyber</td>
<td>Module-LWE</td>
<td>1</td>
<td>Kyber-512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Kyber-768</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>Kyber-1024</td>
</tr>
<tr>
<td>Frodo</td>
<td>LWE</td>
<td>1</td>
<td>Frodo-640</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Frodo-976</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>Frodo-1344</td>
</tr>
<tr>
<td><strong>Digital Signature Protocols</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qTESLA</td>
<td>Ring-LWE</td>
<td>1</td>
<td>qTESLA-I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>qTESLA-III-size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>qTESLA-III-speed</td>
</tr>
<tr>
<td>CRystals-Dilithium</td>
<td>Module-LWE</td>
<td>-</td>
<td>Dilithium-I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Dilithium-II</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Dilithium-III</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Dilithium-IV</td>
</tr>
</tbody>
</table>

where NIST security levels 1-6 indicate brute-force security matching or exceeding that of AES-128, SHA3-256, AES-192, SHA3-384, AES-256 and SHA3-512 respectively.
3.5.2 Protocol Implementations and Evaluation Results

Next, we describe some key aspects of our protocol implementations along with timing and energy profiling results. All polynomial arithmetic, transforms and sampling operations are accelerated using custom programs running in the Sapphire core, and all SHA-3 computations utilize the Keccak core inside Sapphire. The RISC-V processor is used only to read / write data and programs from / to the cryptographic core (both when executing polynomial computations and when utilizing the fast Keccak core for SHA-3 operations), generate initial randomness using the \texttt{randombytes} function, encode / decode messages and compress / decompress public keys and ciphertexts. For polynomials which need to be read from the polynomial memory and encoded (or decoded and written to the polynomial memory), we directly post-process the outputs (or pre-process the inputs) of the crypto-processor’s internal memory, instead of first storing the data in intermediate temporary arrays and then processing them. This saves around 10-20% cycles in overall protocol run-time. Also, the internal clock gates of our crypto-processor are strategically enabled and disabled during program execution to reduce overall energy consumption.

For the NewHope and CRYSTALS-Kyber key exchange schemes, each of the CPA-secure public key encryption functions – \texttt{CPA-PKE.KeyGen}, \texttt{CPA-PKE.Encrypt} and \texttt{CPA-PKE.Decrypt} – has been written entirely (excluding encoding and decoding operations) using Sapphire custom instructions with each of the corresponding programs fitting completely in its 1 KB instruction memory. The CCA-secure key encapsulation functions – \texttt{CCA-KEM.KeyGen}, \texttt{CCA-KEM.Encaps} and \texttt{CCA-KEM.Decaps} – involve calls to SHA-3 and the CPA-PKE functions (according to Fujisaki-Okamoto transform [148]), which are scheduled in software. Since the signature schemes qTESLA and CRYSTALS-Dilithium both involve probabilistic rejection of intermediate values, the associated polynomial computations are split into multiple custom programs (instead of one each) for the \texttt{KeyGen}, \texttt{Sign} and \texttt{Verify} functions. These blocks of code are scheduled using RISC-V software, which also handles encoding and decoding. The only exception is the \texttt{KeyGen} step in qTESLA, where high-precision discrete Gaussian sampling using
Figure 3-15: Configurations of the Sapphire polynomial memory for different Ring-LWE and Module-LWE schemes.

large CDT is implemented in software, with SHA-3 in hardware. Since Module-LWE algorithms involve working with vectors or matrices of polynomials, it is particularly important to ensure that these polynomials fit inside the crypto-processor memory as much as possible (because reads and writes to the internal memory through software are not cheap). When multiplying the public matrix $A$ with the secret vector $s$, the matrix $A$ is generated through rejection sampling, one row at a time, following the just-in-time approach from [149]. This reduces memory footprint so that the entire computation can fit in the polynomial memory. Fig. 3-15 shows how we utilize the configurability of our Sapphire polynomial memory to support different ring dimensions for Ring-LWE and Module-LWE protocols.

Although our lattice crypto-processor architecture primarily targets Ring-LWE and Module-LWE schemes, we also implement the LWE-based Frodo KEM protocol to demonstrate its flexibility. Since LWE-based algorithms require large matrix multiplications, the arithmetic operations dominate total computation cost unlike Ring-LWE and Module-LWE where sampling is the most expensive operation. Since the matrix dimensions are not powers of two, we tile the rows or columns so that we can use the crypto-processor’s power-of-two-sized array operations effectively, as shown in Fig. 3-16. For Frodo-640, we split each 640-element array into two arrays of size 512 and 128. For Frodo-976, we simply use arrays of size 1024 with the last 48 elements zeroed out or ignored, as applicable. For Frodo-1344, we use arrays of size 1536, formed by splitting them into two arrays of size 1024 and 512, with the last 192
elements (of the 512-dimension array) zeroed out or ignored, as applicable. Clearly, the polynomial memory is split and accessed in non-uniform sizes for both Frodo-640 and Frodo-1344. However, this tiling scheme makes our version of Frodo incompatible with the reference software implementation.

Frodo involves three large matrix multiplications: $AS$, $S'A$ and $S'B$, where $A$, $S$, $S'$ and $B$ have dimensions $n \times n$, $n \times \bar{n}$, $\bar{m} \times n$ and $n \times \bar{n}$ respectively with $n \in \{640, 976, 1344\}$ and $\bar{m} = \bar{n} = 8$. We ensure that $S'$ is stored in row-major form and $B$ is stored in column-major form, which simplifies calculating $S'B$ using the schoolbook matrix multiplication technique. For calculating the matrix $AS$, we generate $A$ in row-major form (using rejection sampling, with zero chance of rejection since $q$ is a power of two) and $S$ in column major form (using CDT-based discrete Gaussian sampling) so that the same techniques still work. For $n \in \{640, 976\}$, the matrix $S$ is generated two columns at a time to reduce the number of outer loop iterations. Since both matrices $S'$ and $A$ are generated on-the-fly in row-major fashion, calculating $S'A$ is a bit more complex. Detailed pseudo-codes and discussions are provided in Appendix E. The $AS + E$ and $S'A + E'$ computations (shown in Fig. 3-17) require 10.9M and 9.9M cycles respectively for Frodo-640, 25.3M and 23.2M
Figure 3-17: Computation of the matrices $B = AS + E$ and $B' = S'A + E'$ in Frodo KEM, where the matrices $S, E$ are generated two columns at a time and $S', E'$ are generated two rows at a time.

cycles respectively for Frodo-976, and 67.1M and 62.7M cycles respectively for Frodo-1344, and constitute majority of the total cycle count. This is quite different from the Ring-LWE and Module-LWE schemes, where polynomial sampling accounts for 60-70% of the total computation cost. Please note that memory usage of Frodo-1344-CCA-KEM-Decaps exceeds the 64 KB processor data memory on our test chip; hence it was evaluated only in simulation, with power consumption extrapolated from measured power for Frodo-640 and Frodo-976.

In Tables 3.7 and 3.8, we compare cycle count and energy consumption of assembly-optimized Cortex-M4 software [137] with our hardware-accelerated implementation on our test chip operating at 0.68 V and 12 MHz, with average cycle counts for 100 executions. Clearly, our design achieves order of magnitude improvement in energy-efficiency and performance compared to state-of-the-art software, after accounting for voltage scaling. We note that Module-LWE, although a bit slower compared to Ring-LWE, offers parameters with better scalability in terms of security and efficiency compared to Ring-LWE. Also LWE-based key encapsulation is almost two orders of magnitude more expensive compared to its Ring-LWE and Module-LWE alternatives.
Table 3.7: Measured energy and performance of key encapsulation schemes

<table>
<thead>
<tr>
<th>Protocol</th>
<th>ARM Cortex-M4</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycles</td>
<td>Energy (μJ)</td>
</tr>
<tr>
<td>NewHope-512-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Encaps</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Decaps</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NewHope-1024-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>1,243,729</td>
<td>764.89</td>
</tr>
<tr>
<td>Encaps</td>
<td>1,963,184</td>
<td>1207.34</td>
</tr>
<tr>
<td>Decaps</td>
<td>1,978,982</td>
<td>1217.07</td>
</tr>
<tr>
<td>CRYSTALS-Kyber-v1-512-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>726,921</td>
<td>447.06</td>
</tr>
<tr>
<td>Encaps</td>
<td>987,864</td>
<td>607.54</td>
</tr>
<tr>
<td>Decaps</td>
<td>1,018,946</td>
<td>626.65</td>
</tr>
<tr>
<td>CRYSTALS-Kyber-v1-768-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>1,200,291</td>
<td>738.18</td>
</tr>
<tr>
<td>Encaps</td>
<td>1,446,284</td>
<td>889.46</td>
</tr>
<tr>
<td>Decaps</td>
<td>1,477,365</td>
<td>908.58</td>
</tr>
<tr>
<td>CRYSTALS-Kyber-v1-1024-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>1,771,729</td>
<td>1089.61</td>
</tr>
<tr>
<td>Encaps</td>
<td>2,142,912</td>
<td>1317.89</td>
</tr>
<tr>
<td>Decaps</td>
<td>2,188,917</td>
<td>1346.18</td>
</tr>
<tr>
<td>Frodo-640-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>81,293,476</td>
<td>49995.49</td>
</tr>
<tr>
<td>Encaps</td>
<td>86,178,252</td>
<td>52999.62</td>
</tr>
<tr>
<td>Decaps</td>
<td>87,170,982</td>
<td>53610.15</td>
</tr>
<tr>
<td>Frodo-976-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Encaps</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Decaps</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Frodo-1344-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Encaps</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Decaps</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 3.8: Measured energy and performance of digital signature schemes

<table>
<thead>
<tr>
<th>Protocol</th>
<th>ARM Cortex-M4 [137]</th>
<th>This work</th>
<th>ARM Cortex-M4 [137]</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycles</td>
<td>Energy (µJ)</td>
<td>Cycles</td>
<td>Energy (µJ)</td>
</tr>
<tr>
<td>qTESLA-I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>17,545,901</td>
<td>10790.73</td>
<td>4,846,949</td>
<td>203.13</td>
</tr>
<tr>
<td>Sign</td>
<td>6,317,445</td>
<td>3885.23</td>
<td>168,273</td>
<td>8.92</td>
</tr>
<tr>
<td>Verify</td>
<td>1,059,370</td>
<td>651.51</td>
<td>38,922</td>
<td>1.65</td>
</tr>
<tr>
<td>qTESLA-III-size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>58,227,852</td>
<td>35810.13</td>
<td>11,479,190</td>
<td>469.73</td>
</tr>
<tr>
<td>Sign</td>
<td>19,869,370</td>
<td>12219.66</td>
<td>348,429</td>
<td>18.43</td>
</tr>
<tr>
<td>Verify</td>
<td>2,297,530</td>
<td>1412.98</td>
<td>69,154</td>
<td>2.78</td>
</tr>
<tr>
<td>qTESLA-III-speed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>30,720,411</td>
<td>18893.05</td>
<td>11,898,241</td>
<td>482.42</td>
</tr>
<tr>
<td>Sign</td>
<td>11,987,079</td>
<td>7372.05</td>
<td>317,083</td>
<td>16.78</td>
</tr>
<tr>
<td>Verify</td>
<td>2,225,296</td>
<td>1368.56</td>
<td>67,712</td>
<td>2.62</td>
</tr>
<tr>
<td>CRYSTALS-Dilithium-I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>-</td>
<td>-</td>
<td>95,202</td>
<td>3.44</td>
</tr>
<tr>
<td>Sign</td>
<td>-</td>
<td>-</td>
<td>376,392</td>
<td>13.53</td>
</tr>
<tr>
<td>Verify</td>
<td>-</td>
<td>-</td>
<td>142,576</td>
<td>13.53</td>
</tr>
<tr>
<td>CRYSTALS-Dilithium-II</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>-</td>
<td>-</td>
<td>130,022</td>
<td>5.00</td>
</tr>
<tr>
<td>Sign</td>
<td>-</td>
<td>-</td>
<td>514,246</td>
<td>20.95</td>
</tr>
<tr>
<td>Verify</td>
<td>-</td>
<td>-</td>
<td>184,933</td>
<td>7.35</td>
</tr>
<tr>
<td>CRYSTALS-Dilithium-III</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>2,322,955</td>
<td>1428.62</td>
<td>167,433</td>
<td>6.54</td>
</tr>
<tr>
<td>Sign</td>
<td>9,978,000</td>
<td>6136.47</td>
<td>634,763</td>
<td>24.94</td>
</tr>
<tr>
<td>Verify</td>
<td>2,322,765</td>
<td>1428.50</td>
<td>229,481</td>
<td>9.03</td>
</tr>
<tr>
<td>CRYSTALS-Dilithium-IV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>-</td>
<td>-</td>
<td>223,272</td>
<td>8.17</td>
</tr>
<tr>
<td>Sign</td>
<td>-</td>
<td>-</td>
<td>815,636</td>
<td>30.01</td>
</tr>
<tr>
<td>Verify</td>
<td>-</td>
<td>-</td>
<td>276,221</td>
<td>10.91</td>
</tr>
</tbody>
</table>
3.5.3 Implementation of Kyber-v2 CCA-KEM

The specifications of CRYSTALS-Kyber CCA-KEM [117] were modified during NIST Round 2. Implementation of the initial version, which we call Kyber-v1, was described previously in Section 3.5. Here, we provide implementation results of the modified version, which we call Kyber-v2. From an implementation perspective, the most important change is in prime $q$ (changed from 7681 to 3329) and consequently the definition of NTT. For Kyber-v2, we have $q \equiv 1 \mod n$ but $q \not\equiv 1 \mod 2n$, that is, $\mathbb{Z}_q^*$ contains primitive 256-th roots of unity but not primitive 512-th roots. So, the NTT now decomposes a ring element $a \in \mathbb{Z}_q[x]/(x^{256}+1)$ as $(a \mod x^2 - \zeta, \cdots, a \mod x - \zeta^{511})$ instead of $(a \mod x - \zeta, \cdots, a \mod x - \zeta^{255})$, where $\{\zeta, \zeta^3, \cdots, \zeta^{253}, \zeta^{255}\}$ is the set of all the 256-th primitive roots of unity. In other words, each ring element is decomposed into 128 polynomials of degree 2 modulo $q$ instead of 256 polynomials of degree 1 modulo $q$. Therefore, polynomial multiplication in the ring now requires extension field arithmetic. Our Sapphire crypto core does not natively support this modified NTT representation. To solve this, we employ the “1-Round-Preprocess-then-NTT” or “1PtNTT” technique from [150]. Next, we briefly describe this technique and how it is used for polynomial multiplication.

Following [150], the 1PtNTT technique first divides polynomial $f(x) \in \mathbb{Z}_q[x]/(x^{256}+1)$ with 256 coefficients into two smaller polynomials $f_{\text{even}}(y) \in \mathbb{Z}_q[y]/(y^{128}+1)$ and $f_{\text{odd}}(y) \in \mathbb{Z}_q[y]/(y^{128}+1)$ with 128 coefficients each, where $f_{\text{even}}$ and $f_{\text{odd}}$ respectively contain the even and odd coefficients of $f$ and $y = x^2$, that is, $f(x) = f_{\text{even}}(x^2) + x \cdot f_{\text{odd}}(x^2)$. The 1PtNTT and 1PtNTT$^{-1}$ operations are defined as:

$$\hat{f} = 1\text{PtNTT}(f) = (\text{NTT}(f_{\text{even}}), \text{NTT}(f_{\text{odd}})) = (\hat{f}_{\text{even}}, \hat{f}_{\text{odd}})$$

$$f = 1\text{PtNTT}^{-1}(\hat{f}) = (\text{NTT}^{-1}(\hat{f}_{\text{even}}), \text{NTT}^{-1}(\hat{f}_{\text{odd}})) = (f_{\text{even}}, f_{\text{odd}})$$

where NTT refers to the traditional 128-point number theoretic transform (which is supported by our hardware architecture). Let $p(x) = f(x) \cdot g(x) \in \mathbb{Z}_q[x]/(x^{256}+1)$ be the product of the two polynomials, then $p(x) = p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2)$ where
\[ p_{\text{even}}(y) = f_{\text{even}}(y) \cdot g_{\text{even}}(y) + f_{\text{odd}}(y) \cdot (y \cdot g_{\text{odd}}(y)) \in \mathbb{Z}_q[y]/(y^{128} + 1) \]

\[ p_{\text{odd}}(y) = f_{\text{odd}}(y) \cdot g_{\text{even}}(y) + f_{\text{even}}(y) \cdot g_{\text{odd}}(y) \in \mathbb{Z}_q[y]/(y^{128} + 1) \]

Then, the equation \( p = 1\text{PtNTT}^{-1}(1\text{PtNTT}(f) \circ 1\text{PtNTT}(g)) \) is used for polynomial multiplication in the 1PtNTT domain, where

\[
1\text{PtNTT}(f) \circ 1\text{PtNTT}(g) = (\text{NTT}(f_{\text{even}}) \circ \text{NTT}(g_{\text{even}}) + \text{NTT}(f_{\text{odd}}) \circ \text{NTT}(g_{\text{odd}}),
\text{NTT}(f_{\text{odd}}) \circ \text{NTT}(g_{\text{even}}) + \text{NTT}(f_{\text{even}}) \circ \text{NTT}(g_{\text{odd}}))
\]

and \( g_{\text{odd}} \equiv y \cdot g_{\text{odd}}(y) \in \mathbb{Z}_q[y]/(y^{128} + 1) \) and \( \circ \) denotes coefficient-wise multiplication of polynomials. For further details, please refer to [150]. The following table summarizes the basic operation counts:

<table>
<thead>
<tr>
<th></th>
<th>NTT(_{128})</th>
<th>NTT(_{256})</th>
<th>(\rightarrow_{128})</th>
<th>(+_{128})</th>
<th>(+_{256})</th>
<th>(\odot_{128})</th>
<th>(\odot_{256})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kyber-v1</td>
<td>NTT / NTT(^{-1})</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NTT-based PolyMul</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Kyber-v2</td>
<td>1PtNTT / 1PtNTT(^{-1})</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1PtNTT-based PolyMul</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>4</td>
<td>-</td>
</tr>
</tbody>
</table>

where NTT\(_{128}\) / NTT\(_{256}\) denote traditional 128/256-point NTT, \(\rightarrow_{128}\) denotes 128-point polynomial circular left shift computation, \(+_{128}\) / \(+_{256}\) denote 128/256-point polynomial addition, \(\odot_{128}\) / \(\odot_{256}\) denote 128/256-point coefficient-wise multiplication) for polynomial forward / inverse transform and polynomial multiplication in Kyber-v1 and Kyber-v2 using NTT and 1PtNTT respectively. Clearly, 1PtNTT-based polynomial multiplication is more computationally expensive than the NTT-based approach; the difference was theoretically estimated by [150] to be 10-20%.

Next, we describe how we implement 1PtNTT-based polynomial arithmetic for Kyber-v2 on our Sapphire lattice crypto-processor. The polynomial memory is split into 64 polynomials of \(n = 128\) elements each, and a scaling factor of 19 is used for fast rejection sampling (rejection probability reduced from 0.19 to 0.03). Appendix E provides an example of programming the crypto-processor to compute \( p = f \cdot g \) for \( f, g \in \mathbb{Z}_{3329}[x]/(x^{256} + 1) \), where \( \hat{f} = (\hat{f}_{\text{even}}, \hat{f}_{\text{odd}}) \) is available in transform domain.
Overall, our 1PtNTT-based implementation requires $4,176 - 2,835 = 1,341$ additional cycles compared to our NTT-based implementation. However, NTT (\(g_{even}\)), NTT (\(g_{odd}\)) and NTT (\(\overrightarrow{g_{odd}}\)) computed above are not over-written so that they can be used for multiple such polynomial multiplications as required in Module-LWE. Therefore, the computation of NTT (\(g_{even}\)), NTT (\(g_{odd}\)) and NTT (\(\overrightarrow{g_{odd}}\)) gets amortized over all polynomial multiplications, and the number of additional cycles per polynomial multiplication is effectively $2,006 - 1,546 = 460$ after excluding the computation of NTT (\(g\)) / 1PtNTT (\(g\)). There are \(k^2, k^2 + k\) and \(k\) such polynomial multiplications in Kyber-CPA-PKE KeyGen, Encrypt and Decrypt respectively, thus leading to additional cycle counts in our CCA-KEM implementation.

The cycle counts and energy consumption (at 0.68 V and 12 MHz) of our hardware-accelerated Kyber-v2 CCA-KEM implementation are tabulated below. Compared to Kyber-v1 (see Table 3.7), the power consumption is slightly higher because we had to use the fully configurable modular multiplier instead of the pseudo-configurable one. Note that the cycle count of KeyGen is significantly lower due to the absence of public key compression, while the cycle counts of Encaps and Decaps are slightly higher due to the additional 1PtNTT-related computations described earlier.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Cycle Count</th>
<th>Energy ((\mu J))</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYSTALS-Kyber-v2-512-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>54,861</td>
<td>1.75</td>
</tr>
<tr>
<td>Encaps</td>
<td>134,965</td>
<td>3.89</td>
</tr>
<tr>
<td>Decaps</td>
<td>146,068</td>
<td>4.61</td>
</tr>
<tr>
<td>CRYSTALS-Kyber-v2-768-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>84,110</td>
<td>2.72</td>
</tr>
<tr>
<td>Encaps</td>
<td>184,080</td>
<td>5.39</td>
</tr>
<tr>
<td>Decaps</td>
<td>198,011</td>
<td>6.36</td>
</tr>
<tr>
<td>CRYSTALS-Kyber-v2-1024-CCA-KEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyGen</td>
<td>116,841</td>
<td>3.85</td>
</tr>
<tr>
<td>Encaps</td>
<td>236,886</td>
<td>7.10</td>
</tr>
<tr>
<td>Decaps</td>
<td>256,828</td>
<td>8.34</td>
</tr>
</tbody>
</table>
3.5.4 Implementation of Lattice-Based CCA-IBE

Identity-Based Encryption (IBE), first proposed in 1984 by Shamir [151], is a type of public key encryption where public keys of users are derived from their identities, e.g., e-mail, IP addresses, etc. Unlike traditional protocols where user public keys are obtained from certificates, IBE has the unique advantage of not requiring certificate storage and verification. A trusted third party, known as Private Key Generator (PKG), generates these keys, analogous to Certificate Authority (CA) in the traditional setting. Given security parameter \( \lambda \), an IBE scheme consists of the following four probabilistic polynomial time algorithms:

- **Setup** \( (1^\lambda) \rightarrow (mpk, msk) \) : used to generate master public key \( mpk \) and master secret key \( msk \) of the PKG.

- **Extract** \( (mpk, msk, ID) \rightarrow sk_{ID} \) : used by the PKG to generate secret key \( sk_{ID} \) of an user with identity \( ID \).

- **Encrypt** \( (mpk, ID, m) \rightarrow c \) : sender encrypts message \( m \) using \( mpk \) and receiver’s public key derived from their identity \( ID \), and outputs ciphertext \( c \).

- **Decrypt** \( (sk_{ID}, c) \rightarrow \{m, \bot\} \) : receiver decrypts ciphertext \( c \) using their secret key \( sk_{ID} \), and outputs either message \( m \) or \( \bot \) if the ciphertext is invalid.

The IBE scheme is correct if, for any message \( m \) and identity \( ID \), the following equality holds with overwhelming probability:

\[
\text{Decrypt} (sk_{ID}, \text{Encrypt} (mpk, ID, m)) = m
\]

The **Setup** and **Extract** steps are performed very infrequently. Once the keys are set up and stored, the **Encrypt** and **Decrypt** steps are used for ID-based encryption and decryption respectively.

The first lattice-based IBE crypto-system was proposed by Gentry *et al.* [89], but had ciphertexts of the order of millions of bits, thus making it impractical. Several improvements have been proposed over the past years, and the most efficient construction till date is the DLP-IBE scheme [152] which uses NTRU lattices for
Algorithm 3.1 IND-CPA-Secure ID-based Encryption [152]

Require: \(mpk, ID, m\)
Ensure: \((u, v, c) = IBE-CPA-Encrypt (mpk, ID, m)\)

1: \(r, e_1, e_2 \leftarrow \{−1, 0, 1\}^n; k \leftarrow \{0, 1\}^n\) (uniform)
2: \(u \leftarrow r \ast mpk + e_1 \in \mathcal{R}_q\)
3: \(v \leftarrow r \ast H(ID) + e_2 + \lceil q/2 \rceil \cdot k \in \mathcal{R}_q\)
4: \(v \leftarrow \lceil v/2^l \rceil\)
5: return \((u, v, c = m \oplus H'(k))\)

Algorithm 3.2 IND-CPA-Secure ID-based Decryption [152]

Require: \(sk_{ID}, (u, v, c)\)
Ensure: \(m = IBE-CPA-Decrypt (sk_{ID}, (u, v, c))\)

1: \(v \leftarrow 2^l \cdot v\)
2: \(w \leftarrow v - u \ast sk_{ID} \in \mathcal{R}_q\)
3: \(k \leftarrow \lceil w/q \rceil\)
4: return \(m = c \oplus H'(k)\)

key generation and Ring-LWE for encryption to achieve public keys of size \(O(n)\) and ciphertexts of size \(O(2n)\), where \(n\) is the degree of underlying polynomial ring \(\mathcal{R}_q\).

The Ring-LWE-based Encrypt and Decrypt functions of the DLP-IBE scheme are described in Algorithms 3.1 and 3.2. Details of the Setup and Extract algorithms are available in [152], and we exclude any discussion on them since only the Encrypt and Decrypt algorithms are expected to be executed by constrained embedded devices.

In the Encrypt step, coefficients of the error polynomials \(r, e_1\) and \(e_2\) are sampled from a discrete probability distribution with support \(\{-1, 0, 1\}\), and the coefficients of polynomial \(k\) are sampled uniformly from \(\{0, 1\}\). The distribution parameters directly affect security and efficiency of the IBE scheme, and we describe our parameter selection later, along with the choice of \(n\) and \(q\). \(H\) is a hash function which maps an arbitrary-length identity string \(ID\) to a polynomial in \(\mathcal{R}_q\), and \(H'\) is another hash function which converts \(k \in \mathcal{R}_q\) to a one-time pad of length \(m\text{len}\) (equal to the length of message \(m\)). The polynomial \(v\) is compressed by dropping \(l\) least significant bits of each of its coefficients. This causes negligible increase in decryption failure probability as long as \(l \leq \lceil \log_2 q \rceil - 3\), according to [152]. To verify that the decryption works correctly (with an infinitesimally small probability of failure), we note:
\[ w \approx r \ast H(ID) + e_2 + \lfloor q/2 \rfloor \cdot k - (r \ast mpk + e_1) \ast sk_{ID} \]
\[ = r \ast \{ H(ID) - mpk \ast sk_{ID} \} + e_2 - e_1 \ast sk_{ID} + \lfloor q/2 \rfloor \cdot k \]
\[ = r \ast s + e_2 - e_1 \ast sk_{ID} + \lfloor q/2 \rfloor \cdot k \]

since the master public key and user secret key satisfy the property: \( mpk \ast sk_{ID} + s = H(ID) \), where \( s \) is a short element in \( \mathcal{R}_q \) [152]. Decryption is correct as long as all coefficients of \( r \ast s + e_2 - e_1 \ast sk_{ID} \) lie in the range \((-q/4, q/4)\).

The original DLP-IBE scheme is only IND-CPA-secure, that is, *indistinguishable under chosen plaintext attacks*, so the same key-pair cannot be used for multiple encryptions. Here, we describe how to make this scheme IND-CCA2-secure, that is,

**Algorithm 3.3** IND-CCA2-Secure ID-based Encryption

Require: \( mpk, ID, m \)

Ensure: \((u, v, c, d) = \text{IBE-CCA-Encrypt} (mpk, ID, m)\)

1: \( k \overset{\$}{\leftarrow} \{0, 1\}^n \) (uniform)
2: \( r \leftarrow F(k, 0x00) \in \{-1,0,1\}^n \)
3: \( e_1 \leftarrow F(k, 0x01) \in \{-1,0,1\}^n \)
4: \( e_2 \leftarrow F(k, 0x02) \in \{-1,0,1\}^n \)
5: \( u \leftarrow r \ast mpk + e_1 \in \mathcal{R}_q \)
6: \( v \leftarrow r \ast H(ID) + e_2 + \lfloor q/2 \rfloor \cdot k \in \mathcal{R}_q \)
7: \( v \leftarrow \lfloor v/2^l \rfloor \)
8: \text{return} \((u, v, c = m \oplus H'(k), d = G(k))\)

**Algorithm 3.4** IND-CCA2-Secure ID-based Decryption

Require: \( sk_{ID}, (u, v, c, d) \)

Ensure: \( m = \text{IBE-CCA-Decrypt} (sk_{ID}, (u, v, c, d)) \)

1: \( v \leftarrow 2^l \cdot v \)
2: \( w \leftarrow v - u \ast sk_{ID} \in \mathcal{R}_q \)
3: \( k' \leftarrow \lfloor w/q \rfloor \)
4: \( r' \leftarrow F(k', 0x00) \in \{-1,0,1\}^n \)
5: \( e_1' \leftarrow F(k', 0x01) \in \{-1,0,1\}^n \)
6: \( e_2' \leftarrow F(k', 0x02) \in \{-1,0,1\}^n \)
7: \( u' \leftarrow r' \ast mpk + e_1' \in \mathcal{R}_q \)
8: \( v' \leftarrow r' \ast H(ID) + e_2' + \lfloor q/2 \rfloor \cdot k' \in \mathcal{R}_q \)
9: \( v' \leftarrow \lfloor v'/2^l \rfloor \)
10: \text{if} \( d = G(k') \text{ and } (u, v) = (u', v') \) \text{then}
11: \text{return} \( m = c \oplus H'(k') \)
12: \text{else}
13: \text{return} \perp
14: \text{end if}
indistinguishable under adaptive chosen ciphertext attacks, using the standard Fujisaki-
Okamoto transform [152,153]. The IND-CCA2-secure scheme allows key reuse so that
keys can be cached long-term in the sensor nodes. The key generation phase remains
unchanged, and the IND-CCA2-secure IBE scheme is described in Algorithms 3.3 and
3.4. The CCA-secure encryption deterministically derives the error polynomials \( r, e_1 \)
and \( e_2 \) from \( k \) instead of sampling them randomly like its CPA-secure counterpart.
Here, \( F \) is a hash function which generates error polynomials from \( k \), and \( G \) is another
hash function which computes a \( hlen \)-bit digest of the polynomial \( k \).

We choose parameters \( n = 1024 \) and \( q \approx 2^{23} \) for 128-bit security level, as rec-
ommended in [152] and [154]. To ensure that prime \( q \) allows efficient modular
multiplication, we choose \( q = 8380417 = 2^{23} - 2^{13} + 1 \) which supports fast Bar-
rett reduction due to its special structure. Also, \( q \equiv 1 \mod 2n \), thus allowing fast
polynomial multiplication using NTT. We explore two options for choosing the er-
ror probability distribution \( \Pr[x] \) for \( x \in \{-1,0,1\} \): (1) uniform distribution with
\( \Pr[x = -1] = \Pr[x = 0] = \Pr[x = 1] = 1/3 \), and (2) trinary distribution with
\( \Pr[x = -1] = \Pr[x = 1] = \rho/2 \) and \( \Pr[x = 0] = 1 - \rho \) for \( \rho \in \{1/2, 1/4, 1/8, \cdots \} \).
We use the methodology proposed in [155] to analyze security of the IBE scheme for
different error distributions with varying standard deviation (\( \sigma \)). In Table 3.9, we
show the security levels (in bits) provided by these distributions for our parameters
\((n,q) = (1024,8380417)\). Clearly, the uniform distribution provides highest security,
while security provided by the trinary distribution decreases with smaller \( \rho \). Since
sampling of error polynomials accounts for bulk of the computation cost of Ring-LWE,
we also analyze the number of pseudo-random bits required to generate samples from
these distributions as an indicator of their efficiency. For sampling a polynomial
coefficient from distribution (1), we need to generate 2 uniformly random bits and use
rejection sampling, that is, output \( -1, 0 \) and \( 1 \) when these bits are \( 00_2 \), \( 01_2 \) and \( 10_2 \)
respectively, and reject (and repeat the process with 2 more random bits) when they
are \( 11_2 \). Then, expected number of random bits to sample uniformly in \( \{-1,0,1\} \) is
\[
= 2 \cdot \frac{3}{4} + 4 \cdot \frac{1}{4} \cdot \frac{3}{4} + 6 \cdot \left(\frac{1}{4}\right)^2 \cdot \frac{3}{4} + 8 \cdot \left(\frac{1}{4}\right)^3 \cdot \frac{3}{4} + 10 \cdot \left(\frac{1}{4}\right)^4 \cdot \frac{3}{4} + \cdots \\
= 2 \cdot \frac{3}{4} \cdot \left\{ \sum_{i=1}^{\infty} i \cdot \left(\frac{1}{4}\right)^{i-1} \right\} = \frac{3}{2} \cdot \frac{1}{\left(1-\frac{1}{4}\right)^2} = \frac{8}{3}
\]
Table 3.9: Security of IBE scheme with different error distributions

<table>
<thead>
<tr>
<th>Distribution</th>
<th>$\rho$</th>
<th>$\sigma$</th>
<th>Security Level</th>
<th>Random Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>-</td>
<td>$\sqrt{2}/3$</td>
<td>143</td>
<td>$\approx 2731$</td>
</tr>
<tr>
<td>Trinary</td>
<td>$1/2$</td>
<td>$1/\sqrt{2}$</td>
<td>141</td>
<td>2048</td>
</tr>
<tr>
<td></td>
<td>$1/4$</td>
<td>$1/2$</td>
<td>134</td>
<td>3072</td>
</tr>
<tr>
<td></td>
<td>$1/8$</td>
<td>$1/2\sqrt{2}$</td>
<td>129</td>
<td>4096</td>
</tr>
</tbody>
</table>

and the total number of random bits required for sampling $n$ such polynomial coefficients is $8n/3$ on average. For sampling a polynomial coefficient from distribution (2) where $1/\rho$ is a power of two, we need to generate $\log_2(2/\rho)$ uniformly random bits and then output $-1$ when these bits are all zeros, $1$ when they are all ones, and $0$ otherwise. Rejection sampling is not necessary in this case, and sampling $n$ such polynomial coefficients always requires $n \log_2(2/\rho)$ random bits. We choose the trinary distribution with $\rho = 1/2$ because it requires the smallest number of random bits, as shown in Table 3.9. There is slight reduction in security compared to using the uniform distribution, but it still remains well above our target 128-bit security level.

For our NTT implementation, we choose the $n$-th and $2n$-th roots of unity modulo $q$ to be $\omega = 10730$ and $\psi = 1306$ respectively. We instantiate the hash functions $H : \{0,1\}^* \to \mathcal{R}_q$, $H' : \mathcal{R}_q \to \{0,1\}^{m\text{len}}$ and $F : \mathcal{R}_q \times \{0,1\}^8 \to \mathcal{R}_q$ using the SHA-3-based extendable output function SHAKE-256, and $G : \mathcal{R}_q \to \{0,1\}^{h\text{len}}$ using SHA3-256. We implemented the IBE scheme using our configurable lattice cryptography processor. The measured cycle counts and energy consumption (at 0.68 V and 12 MHz) of ID-based encryption decryption, both CPA-secure and CCA-secure, are reported in Table 3.10. Our hardware-accelerated CCA-IBE has low energy consumption and still fast enough for practical applications. It can also be integrated with TLS to enable efficient handshakes with post-quantum security [156].

Table 3.10: Performance and Energy Consumption of IBE Implementation

<table>
<thead>
<tr>
<th>IBE Scheme</th>
<th>Encrypt Cycles</th>
<th>Encrypt Energy ($\mu$J)</th>
<th>Decrypt Cycles</th>
<th>Decrypt Energy ($\mu$J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IND-CPA-Secure</td>
<td>95,369</td>
<td>3.88</td>
<td>111,652</td>
<td>4.55</td>
</tr>
<tr>
<td>IND-CCA2-Secure</td>
<td>106,980</td>
<td>4.38</td>
<td>194,171</td>
<td>7.93</td>
</tr>
</tbody>
</table>
3.5.5 Comparison with Previous Work

In Table 3.11, we compare our design with existing hardware-accelerated implementations of NIST Round 2 lattice-based protocols. Our crypto-processor is significantly more energy-efficient when executing Ring-LWE and Module-LWE protocols compared to [111], while also being smaller in area. Efficiency of our design is also greater than or comparable to previous Ring-LWE hardware [104,109,157]. Furthermore, our design is the first to offer the flexibility to support multiple protocols (both key encapsulation and signatures) on the same chip, including NIST PQC candidates NewHope, Kyber, Frodo, qTesla and Dilithium at different security levels. Finally, we also compare our design with state-of-the-art elliptic curve cryptography hardware [72]. We observe that our implementation of lattice-based key encapsulation using NewHope-512 is almost an order of magnitude more efficient compared to Diffie-Hellman key exchange using the NIST P-256 elliptic curve at comparable pre-quantum security level.

Table 3.11: Comparison of our design with state-of-the-art hardware

<table>
<thead>
<tr>
<th>Design</th>
<th>Platform</th>
<th>Tech (nm)</th>
<th>VDD (V)</th>
<th>Protocol</th>
<th>Logic (kGE)</th>
<th>SRAM (KB)</th>
<th>Cycles</th>
<th>Energy (μJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>ASIC</td>
<td>40</td>
<td>0.68</td>
<td>NewHope-512-CCA-KEM-Encaps</td>
<td>106</td>
<td>40.25</td>
<td>136,077</td>
<td>3.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NewHope-1024-CPA-PKE-Encrypt</td>
<td></td>
<td></td>
<td>106,611</td>
<td>4.58</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Kyber-512-CCA-KEM-Encaps</td>
<td></td>
<td></td>
<td>131,698</td>
<td>3.58</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Kyber-768-CPA-PKE-Encrypt</td>
<td></td>
<td></td>
<td>94,440</td>
<td>3.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Kyber-768-CCA-KEM-Encaps</td>
<td></td>
<td></td>
<td>177,540</td>
<td>4.89</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Frodo-640-CCA-KEM-Encaps</td>
<td></td>
<td></td>
<td>11,609,668</td>
<td>431.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dilithium-II-Sign</td>
<td></td>
<td></td>
<td>514,246</td>
<td>20.95</td>
</tr>
<tr>
<td>Basu et al. [111] †</td>
<td>ASIC</td>
<td>65</td>
<td>1.2</td>
<td>NewHope-512-CCA-KEM-Encaps</td>
<td>1273</td>
<td>-</td>
<td>307,847</td>
<td>69.42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Kyber-512-CCA-KEM-Encaps</td>
<td>1341</td>
<td>-</td>
<td>31,669</td>
<td>6.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dilithium-II-Sign</td>
<td>1603</td>
<td>-</td>
<td>155,166</td>
<td>50.42</td>
</tr>
<tr>
<td>Albrecht et al. [109]</td>
<td>SLE 78</td>
<td>-</td>
<td>-</td>
<td>Kyber-768-CPA-PKE-Encrypt</td>
<td>-</td>
<td>-</td>
<td>4,747,291</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Kyber-768-CCA-KEM-Encaps</td>
<td>-</td>
<td>-</td>
<td>5,117,996</td>
<td>-</td>
</tr>
<tr>
<td>Oder et al. [104]</td>
<td>FPGA</td>
<td>-</td>
<td>-</td>
<td>NewHope-1024-Simple-Encrypt</td>
<td>-</td>
<td>-</td>
<td>179,292</td>
<td>-</td>
</tr>
<tr>
<td>Howe et al. [107]</td>
<td>FPGA</td>
<td>-</td>
<td>-</td>
<td>Frodo-640-CCA-KEM-Encaps</td>
<td>-</td>
<td>-</td>
<td>3,317,760</td>
<td>-</td>
</tr>
<tr>
<td>Fritzmann et al. [157]</td>
<td>FPGA</td>
<td>-</td>
<td>-</td>
<td>NewHope-1024-CPA-PKE-Encrypt</td>
<td>-</td>
<td>-</td>
<td>589,285</td>
<td>-</td>
</tr>
<tr>
<td>Hutter et al. [60] †</td>
<td>ASIC</td>
<td>130</td>
<td>1.2</td>
<td>Curve25519-ECDHE</td>
<td>50</td>
<td>-</td>
<td>1,622,354</td>
<td>113.56</td>
</tr>
<tr>
<td>Banerjee et al. [72]</td>
<td>ASIC</td>
<td>65</td>
<td>0.8</td>
<td>NIST-P256-ECDHE</td>
<td>149</td>
<td>6.75</td>
<td>680,000</td>
<td>24.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NIST-P256-ECDSA-Sign</td>
<td></td>
<td></td>
<td>180,000</td>
<td>6.48</td>
</tr>
</tbody>
</table>

† Only post-synthesis area and energy consumption reported by [111] and [60]
3.5.6 Side-Channel Analysis

Side-channel security is an important aspect of all public-key cryptography implementations. In order to prevent information leakage through timing side channels, the most important requirement is to ensure that the timing and memory access patterns of underlying computations are independent of the secret data being computed upon. In our implementation, this is achieved either by making the computations constant-time, e.g., binomial sampling, discrete Gaussian sampling, NTT and polynomial arithmetic, or by using rejection sampling, e.g., sampling numbers from \([0, q]\) or \([-\eta, \eta]\) or probabilistic rejection during signature schemes.

Our power side-channel measurement setup is shown in Fig. 3-18. Our test board has an 18 \(\Omega\) resistor connected in series between the power supply and the \(V_{DD}\) pin of our test chip. The voltage across this resistor, proportional to the chip’s current draw, is magnified using a non-inverting differential amplifier (consists of an AD8001 op-amp chip [77], with 6 dB flat gain up to 100 MHz, in the non-inverting configuration.

Figure 3-18: Power side-channel measurement setup.
with resistors of appropriate sizes) and then observed through a 2.5 GS/s Tektronix MDO3024 mixed domain oscilloscope [78].

The execution times of binomial sampling, discrete Gaussian sampling, NTT, polynomial coefficient-wise multiplication and addition (with \( n = 1024 \) and \( q = 12289 \)) were measured for 10,000 random executions to verify that these computations are indeed constant-time. The corresponding power waveforms and energy consumption histograms, measured from our test chip operating at 1.1 V and 12 MHz, are shown in Fig. 3-19. The energy consumption of each operation follows a narrow unimodal distribution which indicates protection against any obvious timing and simple power analysis side-channels (information leakage due to data-dependent timing or energy consumption usually leads to multimodal distributions) [158].

Typical simple power analysis (SPA) attacks on lattice cryptography implementations exploit information leakage through conditional branching or data-dependent execution times during the modular arithmetic computations in NTT or polynomial coefficient-wise multiplication [159–161]. As explained in Fig. 3-19, our implementation of polynomial arithmetic is constant-time. To quantitatively evaluate SPA resistance of our design, we perform a difference-of-means test [158,161,162] on three polynomial operations – NTT, coefficient-wise multiplication and coefficient-wise addition. In this test, we try to differentiate two sets of measurements – those with a particular coefficient (‘0’-th coefficient in our case) in the input polynomial set to 0 (denoted as set ‘0’ or \( S_0 \)) versus the same coefficient set to \( q - 1 \) (denoted as set ‘1’ or \( S_1 \)) – by comparing their means separately for each point in the mean power trace. The difference-of-means is calculated for increasing number of measurements and plotted as a function of the number of traces \( N \). The corresponding 99.99% confidence interval for having a zero difference of means between these two sets is calculated as \( t_c \cdot \sqrt{(\sigma_0^2 + \sigma_1^2)/N} \), where \( \sigma_0 \) and \( \sigma_1 \) are the standard deviations of the two sets \( S_0 \) and \( S_1 \) respectively and \( t_c \) is the critical t-statistic for \( N - 1 \) degrees of freedom and cumulative probability \( = 1 - (1 - 0.9999)/2 = 0.99995 \). As long as the absolute difference-of-means is smaller than the confidence interval, it is a strong indicator that the sets \( S_0 \) and \( S_1 \) are indistinguishable.
Figure 3-19: Measured power waveforms for different polynomial sampling, transform and arithmetic operations along with histograms of energy consumption for 10,000 measurements per operation, obtained from our test chip at 1.1 V and 12 MHz.
Figure 3-20: Difference-of-means test for polynomial number theoretic transform (NTT) with representative power traces from set $S_0$ (top left) and $S_1$ (top right), difference waveform (bottom left) and difference of means versus number of traces with 99.99% confidence interval (bottom right).

Figure 3-21: Difference-of-means test for polynomial coefficient-wise multiplication with representative power traces from set $S_0$ (top left) and $S_1$ (top right), difference waveform (bottom left) and difference of means versus number of traces with 99.99% confidence interval (bottom right).

Figure 3-22: Difference-of-means test for polynomial coefficient-wise addition with representative power traces from set $S_0$ (top left) and $S_1$ (top right), difference waveform (bottom left) and difference of means versus number of traces with 99.99% confidence interval (bottom right).
In Fig. 3-20, 3-21 and 3-22, we provide difference-of-means test results, over 1000 traces, for three polynomial operations (with $n = 1024$ and $q = 12289$). The red lines denote measured difference-of-means, and the dashed lines mark the 99.99% confidence interval for ideal zero difference-of-means.

The protocol implementations discussed earlier do not have any explicit countermeasures against differential power analysis (DPA) attacks. Although DPA attacks can be mitigated by using ephemeral keys (no re-use of public-private keypairs), it is still important to analyze how these protocols can be made DPA-secure. Since our crypto-processor is programmable, masking-based countermeasures can be implemented using the right mix of software and hardware acceleration. For example, we evaluate a masked version of NewHope-CPA-PKE. Following [139,163,164], the additively homomorphic property of Ring-LWE is exploited to randomize the decryption algorithm as a first-order DPA countermeasure. We observe that the masked decryption is about $3 \times$ less efficient compared to the unmasked version, both in terms of energy and performance. Further details are available in [130,131].

Typically, a non-specific fixed vs. random $t$-test [80] is performed to statistically quantify information leakage from a cryptographic algorithm implementation in software or hardware. In Fig. 3-23, we show $t$-test results for unmasked and masked NewHope-1024-CPA-PKE.Decrypt, over 10,000 measurements each, as obtained from our test chip. Unlike the unmasked version, the absolute $t$-value remains well below the threshold for the masked implementation.

Figure 3-23: Leakage test results for (a) unmasked and (b) masked NewHope-1024-CPA-PKE.Decrypt, with red dotted line indicating the $|t| = 4.5$ threshold.
3.6 Summary and Contributions

Lattice-based cryptography is currently a fast-evolving field of research with a variety of candidates being considered for post-quantum standardization as well as new protocols being proposed which exploit the special properties of lattices. In this chapter, we have presented an energy-efficient configurable lattice crypto-processor supporting different polynomial, prime field and discrete distribution parameters. Using this design we demonstrate not only NIST candidate lattice-based key encapsulation and digital signature schemes, e.g., NewHope, qTESLA, CRYSTALS-Kyber, CRYSTALS-Dilithium and Frodo, but also other novel protocols such as identity-based encryption and key encapsulation from lattices.

Efficient modular arithmetic, sampling and number theoretic transform together provide an order of magnitude improvement in performance and energy-efficiency compared to state-of-the-art software and hardware implementations. Several circuit, architecture and algorithm techniques are used to achieve this energy-efficient design. Our configurable modular arithmetic unit with optimized reduction circuitry for several commonly used prime moduli provides up to $3 \times$ energy savings. Our sampler design consists of an energy-efficient parallel-data-path Keccak core for pseudo-random number generation coupled with efficient post-processing algorithms to provide order of magnitude energy savings. Our number theoretic transform memory architecture is built entirely using single-port memories, thus providing 30% area savings. Mathematical properties of the constant factors used during polynomial multiplication are exploited to further compress the required memory by 37.5%.

All key building blocks in our design are constant-time, and we implement countermeasures to prevent timing and simple power analysis attacks. To prevent stronger differential power analysis attacks, masking-based countermeasures can also be implemented. Experimental validation results are provided.

Instead of being a hardware accelerator, our design acts an application-specific crypto-processor providing significantly more flexibility compared to previous work. All key protocol parameters are configurable at run time, and lattice-based algorithms
can be accelerated by executing programs built using a custom instruction set. Our crypto-processor is also integrated with a RISC-V micro-processor to provide further flexibility and support new applications using software-hardware co-design.
Chapter 4

Post-Quantum Cryptography using DTLS Engine and RISC-V

As discussed in Chapter 3, there have been a lot of recent advances in the design of custom hardware accelerators for post-quantum cryptography (PQC), e.g., lattice-based cryptography. However, there has been little work in exploring how existing pre-quantum RSA / ECC co-processors can be used to accelerate post-quantum algorithms (although not as efficiently as dedicated PQC accelerators) [109]. In this work, we implement several PQC algorithms through software-hardware co-design using the low-power RISC-V micro-processor and energy-efficient AES, SHA2 and ECC cryptographic accelerators in the custom chip described in Chapter 2, which was originally designed to accelerate the DTLS protocol. We implement the isogeny-based key encapsulation SIKE [165], where we utilize the modular arithmetic unit inside our ECC accelerator to speed up isogeny computations. We also accelerate lattice-based key encapsulation Kyber [117], Frodo [114] and ThreeBears [166] and hash-based signature SPHINCS+ [167] using our AES and SHA2 accelerators. In all cases, the most computationally expensive functions are accelerated in hardware, achieving up to an order of magnitude improvement in energy-efficiency over software implementations. The efficient mapping of modular arithmetic functions in SIKE software to our custom accelerator is done in collaboration with Siddharth Das.
4.1 Implementation of SIKE

The Supersingular Isogeny Key Encapsulation (SIKE) scheme [165] uses secret walks on isogeny graphs of supersingular elliptic curves to perform a Diffie-Hellman-like key exchange resistant to known quantum attacks. SIKE has the smallest key size among NIST Round 2 candidates, e.g., 330-byte public key and 346-byte ciphertext for SIKEp434 (uncompressed) at NIST post-quantum security level 1. However, SIKE is order of magnitude more computationally expensive than other PQC schemes [168], with 99% of the computation cost attributed to arithmetic modulo large primes, thus motivating our use of dedicated hardware for big-integer arithmetic.

In this work, we focus on SIKEp434 which is based on the finite field $\mathbb{F}_{p^2}$, a quadratic extension of the prime field $\mathbb{F}_p$, where $p = 2^{216^3} - 1$ is a 434-bit prime. Since all $\mathbb{F}_{p^2}$ arithmetic can be expressed in terms of $\mathbb{F}_p$, it suffices to look at $\mathbb{F}_p$ operations only. Table 4.1 summarizes the cycle counts (S/W) of various $\mathbb{F}_p$ arithmetic computations and the numbers of these operations in the KeyGen, Encaps and Decaps steps, as obtained from the publicly available optimized-C software implementation of SIKEp434 [165] profiled on our RISC-V processor. Here, fp_add, fp_sub, fp_neg, fp_div2, fp_corr, mul, sqr and rdc_mont denote modular addition, subtraction, negation, division by two, correction from $[0, 2p)$ to $[0, p)$, multiplication, squaring

<table>
<thead>
<tr>
<th>Operation</th>
<th>KeyGen</th>
<th>Encaps</th>
<th>Decaps</th>
<th>S/W Cycles</th>
<th>H/W+S/W Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp_add</td>
<td>11,247</td>
<td>17,367</td>
<td>19,268</td>
<td>1,198</td>
<td>314</td>
</tr>
<tr>
<td>fp_sub</td>
<td>17,949</td>
<td>23,585</td>
<td>28,582</td>
<td>775</td>
<td>286</td>
</tr>
<tr>
<td>fp_neg</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>335</td>
<td>-</td>
</tr>
<tr>
<td>fp_div2</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>608</td>
<td>-</td>
</tr>
<tr>
<td>fp_corr</td>
<td>440</td>
<td>872</td>
<td>874</td>
<td>775</td>
<td>-</td>
</tr>
<tr>
<td>mul</td>
<td>37,268</td>
<td>60,504</td>
<td>63,035</td>
<td>20,154</td>
<td>2,044</td>
</tr>
<tr>
<td>sqr</td>
<td>436</td>
<td>1,744</td>
<td>3,052</td>
<td>20,154</td>
<td>1,804</td>
</tr>
<tr>
<td>rdc_mont</td>
<td>28,952</td>
<td>47,072</td>
<td>50,420</td>
<td>14,457</td>
<td>1,470</td>
</tr>
</tbody>
</table>
and Montgomery reduction respectively. Clearly, fp_add, fp_sub, mul, sqr and rdc_mont account for bulk of the computation, therefore we optimize and accelerate these functions using the configurable modular arithmetic unit described earlier. The corresponding hardware-accelerated cycle counts (H/W+S/W) are also provided in Table 4.1, and our implementation details are described next. Since the output of Montgomery reduction lies in \([0,2p]\), all these functions operate in this range [169].

For modular addition \(a + b \mod 2p\), fp_add employs the constant-time technique of calculating \(a + b - 2p\) and then adding back \(2p\) or 0 depending on whether a borrow was generated or not. We split each input, zero-padded to 448 bits, into two chunks (lower 224 bits in \(lo[\cdot]\) and higher 224 bits in \(hi[\cdot]\)) and exploit the architecture of our 256-bit modular arithmetic unit to accelerate this computation using just 4 ADD operations, as shown in Fig. 4-1. In the first ADD step, the modular adder’s 256-bit input registers are set to \(x = 0x8\cdots0 \parallel lo[a], y = 0x8\cdots0 \parallel lo[b]\) and \(q = 0x8\cdots0 \parallel lo[2p]\). Since our modular adder always computes \(x + y\) and then subtracts \(q\) from this sum if the addition resulted in a carry, setting the most significant bits of \(x\) and \(y\) to 1 ensures that we get \(z = carry \parallel lo[a+b-2p]\), where \(carry\) can be -1, 0 or +1. To skip explicitly calculating carry propagation for the next ADD step, we pre-compute and store \(hi[2p] - 1, hi[2p]\) and \(hi[2p] + 1\), and then set \(x = 0x8\cdots0 \parallel hi[a], y = 0x8\cdots0 \parallel hi[b]\) and \(q = 0x8\cdots0 \parallel hi[2p] - carry\). The corresponding output is \(z = mask \parallel hi[a+b-2p]\), where \(mask\) is bit-wise AND-ed with \(2p\) and added to \(a+b-2p\) in the final two ADD steps to compute \(a + b \mod 2p\), as described in Fig. 4-1. Our accelerated fp_add takes 314 cycles in total.

For modular subtraction \(a - b \mod 2p\), fp_sub first calculates \(a - b\) and then adds \(2p\) or 0 depending on whether a borrow was generated or not. We follow a technique similar to fp_add to accelerate this computation using 2 SUB and 2 ADD operations, as shown in Fig. 4-1, in total 284 cycles. Once again, we utilize the fact that our modular subtractor computes \(x - y\) and then adds \(q\) to this difference if the subtraction resulted in a borrow. In the second SUB operation, the most significant bits of \(x\) and \(y\) are set to 0 and 1 respectively and the borrow from previous SUB step is adjusted into \(q\) to allow borrow propagation.
Although our modular multiplier computes $x \cdot y \mod q$, it can be used as a plain shift-and-add multiplier as long as $q = 0$, $\text{len}(x) + \text{len}(y) \leq 256$ bits and $q\text{len} = \max(\text{len}(x), \text{len}(y))$. To accelerate 435-bit $\times$ 435-bit multiplication $\text{mul}$, we once again zero-pad each input to 448 bits and then split them into two 224-bit parts. Our base hardware-accelerated 224-bit $\times$ 224-bit multiplication is shown in Fig. 4-2. The 224-bit numbers are split further into 96 bits and 128 bits as $A = A_12^{128} + A_0$ and $B = B_12^{128} + B_0$, and schoolbook multiplication $A \times B = A_1B_12^{256} + (A_0B_1 + A_1B_0)2^{128} + A_0B_0$ is used. Here, the multiplications $A_0 \times B_1$ and
\( A_1 \times B_0 \) are hardware-accelerated using our 256-bit multiplier as explained earlier, while \( A_0 \times B_0 \) and \( A_1 \times B_1 \) are computed in parallel in software using the RISC-V processor’s 32-bit ALU, which allows us to compute the final result in 490 cycles. For the 448-bit \( \times \) 448-bit product, we implement Karatsuba multiplication [170]. The 448-bit inputs \( a \) and \( b \) are split into equal parts as \( a = a_1 2^{224} + a_0 \) and \( b = b_1 2^{224} + b_0 \). Then, the final 896-bit result is computed as \( a \times b = p_2 2^{448} + p_1 2^{224} + p_0 \), where \( p_0 = a_0 b_0 \), \( p_2 = a_1 b_1 \) and \( p_1 = (a_0 + a_1)(b_0 + b_1) - p_0 - p_2 \). Together with the three 224-bit multiplications and other additions / subtractions, our hardware-accelerated \texttt{mul} implementation takes 2,044 cycles, which is 10 times faster than software. For squaring \texttt{sqr}, we compute \( a \times a = s_2 2^{448} + s_1 2^{224} + s_0 \), where \( s_0 = a_0^2 \), \( s_1 = 2a_0 a_1 \) and \( s_2 = a_1^2 \), which takes 1,804 cycles, again 11 times faster than software.

The final arithmetic function that we optimize is \texttt{rdc\_mont}, the Montgomery reduction of the outputs of \texttt{mul} and \texttt{sqr}. All multiplications in SIKEp434 are performed in Montgomery domain [171], that is, any number \( a \) is represented as \( aR \mod 2p \) with \( R = 2^{448} \). \texttt{rdc\_mont} converts the product of two such numbers \( c = (aR)(bR) \in [0, 4p^2) \) back to the Montgomery form \( d = (ab)R \mod 2p \) as follows:

\[
d = (c + (c \cdot p' \mod R) \cdot p) / R \in [0, 2p)
\]

where \( p' = -p^{-1} \mod R \). Since \( p = 2^{163} 3^{137} - 1 \) and \( R = 2^{448} \), this computation can
be further simplified as:

\[
d = (c + (c \cdot p' \mod 2^{448}) \cdot 2^{216}3^{137} - (c \cdot p' \mod 2^{448}))/2^{448}
\]

\[
= \lfloor (c + (c \cdot p' \mod 2^{448}) \cdot 2^{216}3^{137})/2^{448} \rfloor
\]

We re-structure the Comba-based Montgomery reduction algorithm from [172] for the 434-bit prime \( p \) and 128-bit radix so that we can use our 256-bit multiplier. The corresponding pseudo-code is shown in Fig. 4-3, where \( c, d \) and \( \hat{p} = p + 1 \) are shown as 32-bit arrays with 28, 14 and 14 elements respectively. Several multiplications are saved since the 216 least significant bits of \( \hat{p} \) are zeros. Once again, we perform arithmetic computations in parallel in software and in the accelerator. To efficiently interleave these computations, we re-order them as shown in Fig. 4-3. The modular reduction take 1,470 cycles overall, which is 10 times faster than software.

The overall cycle counts and energy consumption of SIKEp434, both software and hardware-accelerated versions, are reported in Table 4.2. All SHA3-related functions are performed in software since they account for less than 1% of the total computation.

![Algorithm: Comba-based Montgomery Reduction modulo p434](image)

Figure 4-3: Hardware-accelerated Montgomery reduction, where \( c \) is the 870-bit input, \( d \) is the 435-bit reduced output and \( \hat{p} = p + 1 \). Multiplications are shown in red, and the steps where \( d \) is calculated are shown in blue.
Table 4.2: Performance of SIKEp434

<table>
<thead>
<tr>
<th></th>
<th>S/W Only</th>
<th></th>
<th>S/W + H/W Accel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycles</td>
<td>Energy</td>
<td>Cycles</td>
</tr>
<tr>
<td>KeyGen</td>
<td>$1.22 \times 10^9$</td>
<td>48.83 mJ</td>
<td>$157 \times 10^6$</td>
</tr>
<tr>
<td>Encaps</td>
<td>$2.00 \times 10^9$</td>
<td>79.98 mJ</td>
<td>$257 \times 10^6$</td>
</tr>
<tr>
<td>Decaps</td>
<td>$2.13 \times 10^9$</td>
<td>85.30 mJ</td>
<td>$274 \times 10^6$</td>
</tr>
</tbody>
</table>

cost. With the fast arithmetic implementations described earlier, we achieve $\approx 8 \times$ reduction in energy consumption compared to RISC-V software. We are able to perform key encapsulation in 16.1 s while consuming 11.56 mJ of energy (at 0.8 V and 16 MHz), which is $3 \times$ faster and $9 \times$ more energy-efficient (after accounting for supply voltage scaling) than the optimized ARM Cortex-M4 implementation from [168] (with specialized FPU and DSP instructions) having 44.4 s key encapsulation time with 1.76 J energy consumption (at 24 MHz and 3.3 V) [138]. Our proposed techniques can also be easily extended to larger SIKE parameters without any change in the hardware.

4.2 Implementation of Other PQC Schemes

We also explore the lattice-based CCA-secure key encapsulation schemes Kyber, Frodo and ThreeBears, and the hash-based signature scheme SPHINCS+ at different security levels. Unlike SIKE, a significant portion of the computation costs (70%, 70%, 50% and 95% respectively) of these protocols is attributed to SHA3-based hashing and extendable output functions [168]. Although NIST has required the use of SHA3 as the symmetric primitive for PQC standardization for the sake of uniformity, PQC candidates have also proposed variants of their algorithms which use AES and SHA2 in order to benefit from widely deployed hardware accelerators, for example, Kyber-90s, Frodo-AES and SPHINCS+-SHA256. Following the same approach, we use our energy-efficient AES-128/256 and SHA2-256 hardware as a substitute for SHA3, similar to [109]. The SHA2-256 hash function is used as a drop-in replacement of SHA3-256 wherever 256-bit message digests are required. As a replacement for the SHAKE-256
extendable output function which absorbs an arbitrary-length byte-string and then squeezes out a specified number of bytes, we first use SHA2-256 to compute a 256-bit digest of the input, and then generate output bytes using AES-256 in counter mode with this 256-bit digest used as the key. Similarly, SHA2-256 and AES-128 are used instead of SHAKE-128, and SHA2-512 is used to replace SHA3-512 for Kyber-90s.

The energy consumption of these protocols (measured at 0.8 V and 16 MHz) at various security levels is shown in Figure 4-4 and compared with software. Clearly, using the AES and SHA2 accelerators provides 2-8× improvement in energy-efficiency compared to implementing SHA3 in software on the RISC-V, and has an up to an order of magnitude lower energy consumption compared to ARM Cortex-M4 [168,173]. Once again, our implementations are slower than dedicated accelerators [130,174], but we require only 30.8k (resp. 46.8k) logic gates (in addition to the RISC-V core), the combined area of AES-128/256 and SHA2-256 (resp. SHA2-256/512) modules. Please refer to [175] for further details and measurement results.

While PQC protocols are still being analyzed for security and efficiency, our work shows that existing embedded devices with standard cryptographic accelerators can still be used to reasonably speed up PQC implementations until new optimized PQC accelerators are designed. Although the core hardware accelerators are orders of magnitude more efficient than software, we were unable to achieve similar efficiency in our software-hardware co-design due to latency of the memory-mapped interface, which may be addressed in the future by using direct memory access interface.
4.3 Summary and Contributions

In this chapter, we have demonstrated energy-efficient "post-quantum" cryptography using software-hardware co-design with our "pre-quantum" DTLS accelerator described in Chapter 2. In particular, we have implemented some of the NIST PQC candidate schemes where majority of the computation cost is due to either big-integer arithmetic or hashing. We have re-purposed the modular arithmetic unit inside our ECC accelerator to speed up isogeny-based SIKE key encapsulation. We have also used the AES and SHA2 hardware primitives to substitute SHA3 computations and accelerated lattice-based Kyber, Frodo and ThreeBears key encapsulation and hash-based SPHINCS+ signatures. We have verified the correctness of our software-hardware co-design of these protocols by comparing their results with corresponding software-only counterparts. Overall, we have achieved up to an order of magnitude improvement in energy-efficiency compared to optimized software implementations.

While post-quantum cryptography protocols are still being studied for security and efficiency, with many new protocols expected to be proposed in the near future, our work shows that existing embedded devices with standard "pre-quantum" cryptographic accelerators can still be used to reasonably speed up "post-quantum" implementations until new optimized hardware accelerators are designed and integrated with them.
Chapter 5

Low-Power Elliptic Curve Pairing
Crypto-Processor

Elliptic curves are used as the de facto standard for traditional public key cryptography such as key establishment, digital signatures, authenticated key exchange and public key encryption. In Chapter 2, we looked at the TLS protocol which relies on elliptic curve cryptography (ECC) for many of its strong security guarantees. However, there also exists another realm of cryptographic applications beyond the traditional usage of elliptic curves, known as pairing-based cryptography (PBC).

Pairing was originally used as a tool to break discrete logarithms in certain classes of elliptic curves by reducing an instance of the elliptic curve discrete logarithm problem to an instance of the finite field discrete logarithm problem [176]. Now, pairings are widely used in many novel constructive cryptographic applications such as signature aggregation, functional encryption, multi-party key agreement and zero-knowledge protocols [177]. Only special elliptic curves can be used for pairing-based protocols, also known as pairing-friendly curves. This includes several well-known families such as Barreto-Naehrig (BN) curves, Barreto-Lynn-Scott (BLS) curves, Kachisa-Schaefer-Scott (KSS) curves and Miyaji-Nakabayashi-Takano (MNT) curves [37]. BN curves based on 254-bit and 256-bit prime fields have been widely used in PBC applications for the past several years, until recent advances in cryptanalysis reduced their security level from 128-bit to $\approx 100$-bit [178,179].
The BLS12-381 pairing-friendly elliptic curve, based on a 381-bit prime field, has been recently proposed for PBC applications at the 128-bit security level [180]. It is also part of ongoing standardization process led by the Internet Engineering Task Force (IETF) [181]. However, along with strong security, the new curve also has higher computational complexity, thus making it challenging to implement on low-power embedded devices. To address this challenge, we present a low-power BLS12-381 elliptic curve pairing crypto-processor. Our key contributions (described in detail in Sections 5.2, 5.3 and 5.4) are summarized as follows:

- Majority of the computation cost in BLS12-381-based cryptographic protocols is due to modular multiplication over the 381-bit prime field. We design an energy-efficient word-serial Montgomery modular multiplier which enables two orders of magnitude energy savings. Our modular arithmetic unit can be configured to support both the 381-bit base field and the 255-bit scalar field.

- We couple the modular arithmetic unit with memories, instruction decoding and control logic to provide the flexibility to implement various ECC and PBC protocols for IoT applications. Different protocols can be accelerated by executing programs built using a custom instruction set.

- We split the crypto-processor memory into a three-level hierarchy with dedicated clock gates which are automatically activated depending on the function under execution, thus providing up to 20% energy savings.

- Commonly used modular arithmetic and elliptic curve functions are implemented as micro-code in the form of digital logic which is 6× smaller than ROM.

- Several algorithm-architecture co-optimizations such as Karatsuba-style divide-conquer, pre-computations, sharing of pairing functions and special properties of the BLS12-381 curve are used to further provide up to 2× energy savings in different pairing-based protocols.

- We implement and experimentally validate algorithm-level countermeasures to protect our design from common timing and power side-channel attacks.
5.1 Background

5.1.1 Elliptic Curves and Pairings

Elliptic curve cryptography (ECC) [36] was introduced in Chapter 2 along with its applications in key exchange, digital signature and authentication protocols. Pairing-based cryptography (PBC) [37] is a variant of ECC which uses bilinear maps between special elliptic curves and finite fields to enable security applications beyond traditional key exchange and digital signatures. Pairings are fundamental to the construction of novel cryptographic algorithms and protocols such as signature aggregation, identity-based encryption (IBE), attribute-based encryption (ABE), multi-party key agreement, inner product encryption, etc [177]. Signature aggregation enables the combination of arbitrarily large number of signatures into a single signature to resolve communication bottleneck in mesh networks such as blockchain [182]. Identity-based encryption allows the derivation of public keys from user identities, e.g., emails, IP addresses, etc, thus avoiding expensive certificate-based public key validation. Attribute-based encryption enables users with only certain specified attributes or permissions to decrypt the ciphertext. Multi-party key agreement enables multiple devices to agree upon a shared secret, as an extension of the traditional two-party Diffie-Hellman key exchange. Functional encryption allows computation on encrypted data with a function embedded in the decryption key. In particular, pairing-based function-hiding inner product functional encryption [183] can be used for simple privacy-preserving data classification tasks, thus enabling a new paradigm in the field of secure computation.

Let $E : y^2 = x^3 + ax + b$ be an elliptic curve defined over prime field $\mathbb{F}_p$. Let $\mathbb{G}_1$ be a cyclic subgroup of $E(\mathbb{F}_p)$ of order $q$. Then, there also exists a cyclic subgroup $\mathbb{G}_2$ of $E(\mathbb{F}_{p^k})$ of order $q$, where the embedding degree $k$ is the smallest integer such that $q \mid (p^k - 1)$. Let $\mathbb{G}_T$ be a $q$-order subgroup of the multiplicative group $\mathbb{F}_{p^k}^\times$. Then, a pairing is defined by the map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ which satisfies the bilinearity property: $e(aP, bQ) = e(P, Q)^{ab}$, where $P \in \mathbb{G}_1$, $Q \in \mathbb{G}_2$, $a, b \in \mathbb{Z}_q$.

Special pairing-friendly elliptic curves are required to perform such computations, e.g., BN curves and BLS12 curves. Throughout this work, we consider the optimal
Ate pairing, which is known for its efficiency [184]. In this case, computing the pairing $e$ involves evaluating a rational function $f_{\lambda,Q}$, where $\lambda$ is a constant specific to the curve, at point $P$ followed by a final exponentiation:

$$e(P, Q) = f_{\lambda,Q}(P)^{(p^k - 1)/q}$$

The function $f_{\lambda,Q}$ can be computed efficiently using Miller’s algorithm [185]. Further details are available in [37] and [186], and also in Appendix C.

### 5.1.2 BLS12-381 Pairing-Friendly Curve

The BLS12-381 curve was proposed by Sean Bowe in 2017 [180] to enable several efficient cryptographic features in the Zcash crypto-currency protocol [187]. Since then, it has been widely adopted by rest of the crypto-currency and blockchain community including Ethereum, Chia Network, DFINITY and Algorand [181]. It has also been included in multiple IETF standard drafts such as “Pairing-Friendly Curves” [181], “Hashing to Elliptic Curves” [188] and “BLS Signatures” [189]. While several open-source cryptographic software libraries (e.g., Apache Milagro \(^1\), mcl \(^2\), MIRACL \(^3\), RELIC \(^4\) and zkcrypto \(^5\)) now support BLS12-381, efficient hardware-accelerated implementations are largely unexplored. This is our motivation behind designing a low-power crypto-processor for BLS12-381 to demonstrate security applications.

The base field prime $p$ and prime order $q$ for BLS12-381 are summarized here:

- $p = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0 \cdots f6b0f6241eabfffbeb153ffffb9efffffffffaab_h$
- $q = 0x73eda753299d7d483339d80809a1d80553bda402ffe5bfe \cdots ffffffff00000001_h$

[190] provides a comprehensive discussion on the BLS12-381 curve, parameter choices, mathematical properties and cryptographic applications.

\(^1\) https://github.com/apache/incubator-milagro-crypto
\(^2\) https://github.com/herumi/mcl
\(^3\) https://github.com/miracl/core
\(^4\) https://github.com/relic-toolkit/relic
\(^5\) https://github.com/zkcrypto/pairing
5.2 Hardware Implementation of Pairing

5.2.1 Prime Field Modular Arithmetic

As verified through software profiling, big integer prime field modular arithmetic, especially modular multiplication, accounts for more than 90% of the computation cost in pairing-based cryptography (very similar to traditional elliptic curve cryptography). In case of the BLS12-381 pairing group, we need to perform arithmetic over the 381-bit prime field $\mathbb{F}_p$ (also referred to as the base field) and the 255-bit prime field $\mathbb{F}_q$ (also referred to as the scalar field). Here, we discuss our hardware implementations of modular addition / subtraction, multiplication and inversion over $\mathbb{F}_p$ and $\mathbb{F}_q$.

**Modular Addition and Subtraction:** Our modular adder-subtractor design is shown in Fig. 5-1, and they consist of a pair of cascaded 381-bit adder-subtractors. The modulus can be selected between $p$ and $q$ using a multiplexer. The most significant 126 bits of the data-path are gated when operating over $\mathbb{F}_q$ instead of $\mathbb{F}_p$. Modular reduction is performed using conditional subtraction / addition, which are computed in the same cycle to avoid timing side-channel leakage.

**Modular Multiplication:** Modular multiplication is the most important computation in pairing implementations. Overall computation cost of any algorithm or protocol is typically estimated by the equivalent number of modular multiplications, thus making an efficient modular multiplier crucial to the implementation. Montgomery modular multiplication [171] is one of the standard techniques for such large prime fields. It replaces expensive divisions by the prime modulus with divisions by a carefully chosen constant $R = 2^r$, that is, simple right shifts by $r$ bits. Typically,

![Diagram](image)

Figure 5-1: Design of modular adder-subtractor for $\mathbb{F}_p$ and $\mathbb{F}_q$. 
$r$ is a multiple of the underlying implementation’s word size. We choose $R = 2^{384}$ for the base field $\mathbb{F}_p$ and $R = 2^{256}$ for the scalar field $\mathbb{F}_q$. Montgomery multiplication requires mapping the inputs to the Montgomery domain by multiplying them with $R$. Typically, this domain conversion is done only at beginning and end of a protocol, with all intermediate steps performed in Montgomery domain.

Previous work on pairing accelerators use either high-performance parallel pipelined multipliers with large area overhead [191, 192] or compact serial multipliers with lower energy-efficiency [193]. To balance area and energy-efficiency, we implement Montgomery modular multiplication using the coarsely integrated operand scanning (CIOS) approach [194, 195]. Instead of computing multiplication and reduction separately, the CIOS approach performs both operations together in an interleaved manner. Each input is split into $s$ words of size $w$ bits. The core CIOS loop requires $s(2s + 1)$ and $2(2s^2 + 2s + 1)$ multiplications and additions respectively, all with $w$-bit word size. Final output of the CIOS loop needs to be adjusted from modulo $2p$ to modulo $p$ using a conditional subtraction. This step is often skipped in software implementations for efficiency reasons, also known as lazy reduction, with subsequent arithmetic in the modulo $2p$ domain. We use our single-cycle modular adder (simply adding 0 modulo $p$) to perform this conditional subtraction efficiently and in constant time.

In order to identify the ideal word size for our application, we have profiled CIOS hardware architectures with word size $w \in \{16, 24, 32, 48, 64, 96\}$ (with $s =}\)

---

**Figure 5-2:** Synthesized area and simulated energy consumption profiling of CIOS Montgomery product in hardware with different word sizes $w \in \{16, 24, 32, 48, 64, 96\}$. 

108
Their area and simulated energy consumption (as obtained from post-synthesis simulation) are compared in Fig. 5-2. Clearly, the energy consumption saturates at 64-bit word size, with 50% and 25% lower energy than conventional 16-bit and 32-bit architectures respectively. Therefore, we implement CIOS in hardware with \(w = 64\) \((\Rightarrow s = 6)\), as shown in Fig. 5-3. We split zero-padded inputs into six 64-bit words and operate on them iteratively using a 64-bit \(\times\) 64-bit multiplier and a 128-bit + 64-bit + 64-bit adder, both utilizing carry-save structures for shorter critical path delay. Synthesized area of this design in TSMC 40nm low-power technology is 38.7k-gate, including input and output registers. The simulated energy consumption at 1.1 V is \(\approx 8.16\) nJ per Montgomery product in 168 cycles. In our implementation, modular squaring and modular multiplication have equal computation cost.

**Modular Inversion:** We implement modular inversion using exponentiation following Fermat’s theorem [36], which involves repeated squarings and multiplications depending on the binary representation of the exponent. Inversion in \(\mathbb{F}_p\) and \(\mathbb{F}_q\) require 608 and 417 modular multiplications respectively (including modular squarings).
5.2.2 Elliptic Curve and Pairing Computations

Similar to traditional ECC, prime field modular arithmetic lies at the core of PBC as well. However, unlike ECC, pairing computation is not limited to just $\mathbb{F}_p$ arithmetic. Pairings also require arithmetic over the extension fields of $\mathbb{F}_p$. For BLS12-381, these fields are constructed as follows:

$$
\mathbb{F}_{p^2} = \mathbb{F}_p[\alpha]/(\alpha^2 + 1)
$$

$$
\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[\beta]/(\beta^3 - 1 - \alpha)
$$

$$
\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[\gamma]/(\gamma^2 - \beta)
$$

This construction of the form $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$ is also known as *towered arithmetic*, as shown in Fig. 5-4. Extension field arithmetic over $\mathbb{F}_{p^2}/\mathbb{F}_p$, $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$ and $\mathbb{F}_{p^{12}}/\mathbb{F}_{p^6}$ involves manipulation of polynomials with coefficients in $\mathbb{F}_p$. We speed up the extension field multiplications, squarings and inversions by extensively using Karatsuba-style divide-and-conquer techniques [196, 197], which provide up to 35% improvement in energy-efficiency and performance of the pairing computation. Detailed derivation of the extension field arithmetic formulas is provided in Appendix C.
We use homogeneous projective coordinates for all elliptic curve point operations. To prevent side-channel attacks and other potential implementation vulnerabilities, we employ the optimized exception-free point doubling and complete point addition formulas from [198]. This ensures that the implementation is constant-time and the same point arithmetic formula works for all input points, thus avoiding data-dependent conditional executions.

Elliptic curve scalar multiplication (ECSM) is implemented using the double-and-add-always algorithm to prevent side-channel attacks [76]. ECSM computation with a 255-bit $\mathbb{F}_q$ scalar requires $4,847M_1 + 14,025A_1 + I_1$ (where $I_1 = 608M_1$) for $G_1$ and $4,337M_2 + 510S_2 + 10,200A_2 + I_2$ (where $I_2 = 4M_1 + 2A_1 + I_1$) for $G_2$, where $A_1$ (resp. $A_2$), $M_1$ (resp. $M_2$) and $I_1$ (resp. $I_2$) denote additions / subtractions, multiplications / squarings and inversions respectively in $\mathbb{F}_p$ (resp. $\mathbb{F}_{p^2}$). Depending on the application, ECSM performance can be further improved by using standard pre-computation-based techniques (memory-time trade-offs) such as windowing, comb, etc [36]. Please refer to Appendix C for detailed discussion.

The two main components of pairing computation are Miller Loop (ML) and Final Exponentiation (FE). The Miller Loop consists of a series of line computations based on binary representation of the curve parameter $u$. For BLS12-381, their computation costs, in terms of equivalent number of $\mathbb{F}_p$ multiplications, are $7,050M_1$ and $8,339M_1$ respectively. Therefore, a BLS12-381 pairing is equivalent to $15,389\mathbb{F}_p$ multiplications. Detailed formulas and algorithms are discussed in Appendix C.

\subsection*{5.2.3 Multi-Pairing}

Many practical pairing-based cryptographic protocols require evaluating the product of several pairings [177], also known as multi-pairing, as shown below:

$$\prod_{j=1}^{n} e(P_j, Q_j) = e(P_1, Q_1) \times e(P_2, Q_2) \times \cdots \times e(P_n, Q_n)$$

If one set of pairing inputs is shared, the multi-pairing can be simplified by sharing operations using the bilinearity property [37].
\[ \prod_{j=1}^{n} e(P, Q_j) = e(P, \sum_{j=1}^{n} Q_j) \quad \text{and} \quad \prod_{j=1}^{n} e(P_j, Q) = e(\sum_{j=1}^{n} P_j, Q) \]

so that the \( n \)-fold multi-pairing is reduced to just one pairing and \( n - 1 \) point additions (in \( G_2 \) and \( G_1 \) respectively). This is a significant saving in computation cost due to the elimination of \( n - 1 \) pairings and \( n - 1 \) multiplications in \( G_T \).

For the general case (without any shared inputs), the multi-pairing can be improved by sharing Miller Loop (ML) and Final Exponentiation (FE) computations across multiple pairing instances (please refer to Appendix C for detailed discussion). Fig. 5-5 compares the BLS12-381 multi-pairing computation cost for different values of \( n \) – (1) without any shared operations, (2) with shared FE only and (3) with shared ML and FE. Note that the horizontal and vertical axes in the plot are in logarithmic scale to the base of 2 and 10 respectively. Sharing the FE operation provides up to \( 2 \times \) savings in the compute cost compared to the conventional approach. Sharing both ML and FE provides another \( \approx 30\% \) savings. This will be particularly useful in the implementation of aggregate-signature verification, to be discussed in Section 5.4.

Figure 5-5: Computation cost of BLS12-381 multi-pairing for different number of pairings in the product \( (n) \) and with various optimizations.
5.2.4 Hashing to Points on $G_1$

Apart from multi-pairings, many pairing-based protocols also require mapping random $\mathbb{F}_p$ elements to points on the elliptic curve, known as the hash-to-curve operation. Here, we briefly outline our implementation of hashing to $G_1$ for BLS12-381. We follow the techniques proposed in [199] to adapt the simplified Shallue-van de Woestijne-Ulas (SWU) mapping [200,201] to the BLS12-381 curve. First, the simplified SWU map [188] is used to hash the $\mathbb{F}_p$ element to a point $(x_s, y_s)$ on the curve $E_s(\mathbb{F}_p) : y_s^2 = x^3 + A_s x_s + B_s$ ($A_s, B_s \neq 0$) which is 11-isogenous to $E(\mathbb{F}_p) : y^2 = x^3 + 4$. Next, this is transformed into a point $(x, y)$ on $E$ using the following 11-isogeny map [199]:

$$x = \frac{\sum_{i=0}^{11} k_{1,i} x_s^i}{\sum_{i=0}^{9} k_{2,i} x_s^i} \quad \text{and} \quad y = y_s \frac{\sum_{i=0}^{15} k_{3,i} x_s^i}{\sum_{i=0}^{14} k_{4,i} x_s^i}$$

The constants $A_s, B_s, k_{1,i}, k_{2,i}, k_{3,i}$ and $k_{4,i}$ for BLS12-381 $G_1$ are available in [188]. The computation costs of these two transformations are $1,234 M_1 + 15 A_1$ and $663 M_1 + 51 A_1$ respectively. So, cost of the hash-to-curve operation on $G_1$ is equivalent to $1,897 \mathbb{F}_p$ multiplications. Similar transformations can be applied to $G_2$ as well [188,199].

5.2.5 Point Arithmetic on Jubjub

As discussed earlier, our modular arithmetic unit supports both the 381-bit base field $\mathbb{F}_p$ and the 255-bit scalar field $\mathbb{F}_q$. Therefore, along with BLS12-381, we can also implement point arithmetic on the Jubjub elliptic curve [202] which is defined over $\mathbb{F}_q$. Jubjub is a twisted Edwards curve of the form $-x^2 + y^2 = 1 + dx^2y^2$ with $d = -(10240/10241)$. This curve supports a complete addition formula:

$$(x_1, y_1) + (x_2, y_2) = \left( \frac{x_1 y_2 + x_2 y_1}{1 + dx_1 x_2 y_1 y_2}, \frac{x_1 x_2 + y_1 y_2}{1 - dx_1 x_2 y_1 y_2} \right)$$

Following [203] and [204], we implement Jubjub point addition using extended coordinates $(X : Y : Z : T)$, where $x = X/Z$, $y = Y/Z$ and $xy = T/Z$. Constant-time Jubjub ECSM is implemented using the double-and-add-always method [76] and requires 4,755 multiplications and 4,590 additions in $\mathbb{F}_q$. 

113
5.3 Pairing Crypto-Processor

The top-level architecture of our pairing crypto-processor is shown in Fig. 5-6. The efficient building blocks described in Section 5.2 are integrated with a 15.375 KB data memory, a 1 KB instruction memory and an instruction decoder to form the core of our crypto-processor. It can be programmed using 32-bit custom instructions to perform different modular arithmetic, ECC-related and PBC-related operations and control functions including branching. Details about programming the crypto-processor are provided in Appendix E.

The crypto-processor data memory is hierarchical with three levels. First, the modular arithmetic unit (with modular adder / subtractor and Montgomery modular multiplier) is coupled with a small $8 \times 384$-bit register file $M_0$, which is implemented completely using flip-flops for efficiency. $M_0$ is the primary memory used for all $\mathbb{F}_p$ and $\mathbb{F}_p^2$ arithmetic computations. At the next level, a $64 \times 384$-bit SRAM $M_1$ is used to store all temporary variables required for $\mathbb{F}_p^{\phi}$, $\mathbb{F}_p^{\mu_2}$, $G_1$, $G_2$ and $G_T$ arithmetic computations. Finally, a $256 \times 384$-bit SRAM $M_2$ is used to store the top-level function inputs and outputs. While simple ECSM and pairing computations require only few of these 256 memory locations in $M_2$, having a large top-level memory is useful to support efficiency multi-pairings (where the number of inputs becomes $n$-fold).
and hash-to-curve maps (where a large number of constants are required for the isogeny map). Each memory module is dynamically clock gated based on the function under execution, providing up to 20% power savings. Fig. 5-7 shows the hierarchical memory clock gating during a snapshot of the final exponentiation computation (clock waveforms obtained from simulation). The modular arithmetic unit is operational continuously, while the memories are accessed only when data movement is required.

The modular arithmetic unit requires several constants in Montgomery domain, such as curve parameters and constants for Frobenius maps, twist maps and hash maps, stored in a $22 \times 384$-bit lookup table ($LUT_0$). The $F_p$ and $F_{p^2}$ functions are handled by the modular arithmetic unit. The $F_{p^6}$, $F_{p^{12}}$, $G_1$, $G_2$ and $G_T$ towered arithmetic functions discussed in Section 5.2 (which require $F_p$ and $F_{p^2}$ arithmetic) are implemented as optimized micro-codes stored in another $768 \times 24$-bit lookup table ($LUT_1$). To save area, these LUTs are implemented entirely using digital logic. Combined area of these two LUTs is only 6k-gate, which is 53k-gate and 34k-gate smaller than SRAM-based and ROM-based implementations respectively.

5.4 Implementation Results

5.4.1 System Architecture

As shown in Fig. 5-8, the pairing crypto-processor is integrated (through a memory-mapped interface) with a low-power RISC-V micro-processor [43], with 32 KB instruction memory and 64 KB data memory, which implements the RV32IM instruction set [42] and has Dhrystone performance similar to ARM Cortex-M0. The RISC-V
core has a 1-cycle multiplier and a 32-cycle divider. Through the same interface, the RISC-V is also coupled with accelerators for AES-128/256 and SHA2-256 occupying 12k-gate and 23k-gate respectively. The RISC-V, AES, SHA and pairing cores all have dedicated clock gates, which can be independently configured for power savings. When executing cryptographic workloads, the RISC-V core can be clock-gated using the \textit{wait-for-interrupt (wfi)} instruction. The processor is woken up by dedicated interrupts from the crypto cores, which are raised at the end of cryptographic execution. Using the memory-mapped interface ensures that the crypto cores can be accessed through simple load and store instructions, without requiring any custom instructions or changes to the compilation toolchain. While the crypto cores accelerate all cryptographic computations, the RISC-V processor is used for scheduling these cryptographic workloads as well as for processing their inputs and outputs.

Our test chip was fabricated in the TSMC 40nm LP CMOS process, and the chip micrograph is shown in Fig. 5-9 with the key design components highlighted. The final placed-and-routed design of our pairing crypto core consists of 112k logic gates (97 kGE for synthesized design) and 16 KB SRAM, with a total area of 0.2 mm$^2$ (logic and memory combined). Our test chip supports supply voltage scaling from 0.66 V to 1.1 V. It’s maximum operating frequency (for both RISC-V and the crypto cores) at 0.66 V and 1.1 V is 16 MHz and 90 MHz respectively.

Figure 5-8: Chip architecture with pairing crypto core and RISC-V micro-processor.
Fig. 5-10 shows our test board and measurement setup. The test chip is housed in a QFN64 socket soldered to the board, an Opal Kelly XEM7001 FPGA development board \cite{69} is used to interface with the chip, and a Keithley 2602A source meter \cite{70} supplies power to the chip. Both the FPGA and the source meter are controlled from a host computer through USB and GPIB interfaces respectively. The FPGA is used to transfer programs from the host computer to the instruction memory of our test chip. All pairing and elliptic curve cryptography programs are written using custom instructions and compiled with a Python script, while all RISC-V software is written in C and compiled using the \texttt{riscv-gcc} toolchain \cite{147}.

Energy consumption of the main cryptographic operations (all constant-time), measured from our test chip operating at 90 MHz and 1.1 V, are tabulated below:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{F}_p$ Multiplication</td>
<td>176 $\mu$s</td>
<td>17.6 nJ</td>
</tr>
<tr>
<td>$G_1$ ECSM</td>
<td>12.48 ms</td>
<td>108.00 $\mu$J</td>
</tr>
<tr>
<td>$G_2$ ECSM</td>
<td>32.81 ms</td>
<td>424.27 $\mu$J</td>
</tr>
<tr>
<td>$G_T$ Exponentiation</td>
<td>44.75 ms</td>
<td>376.86 $\mu$J</td>
</tr>
<tr>
<td>Miller Loop</td>
<td>17.07 ms</td>
<td>147.97 $\mu$J</td>
</tr>
<tr>
<td>Final Exponentiation</td>
<td>20.63 ms</td>
<td>173.71 $\mu$J</td>
</tr>
<tr>
<td>Hash-to-$G_1$ Map</td>
<td>3.78 ms</td>
<td>33.98 $\mu$J</td>
</tr>
<tr>
<td>Jubjub ECSM</td>
<td>10.49 ms</td>
<td>90.20 $\mu$J</td>
</tr>
</tbody>
</table>
5.4.2 Pairing-Based Protocol Implementations

To measure the efficiency of our design as well as to demonstrate its flexibility in supporting various security applications, we have implemented and profiled the following BLS12-381 pairing-based cryptography protocols on our test chip:

1. Short signature generation [205]
2. Short signature verification [205]
3. Signature aggregation [206]
4. Aggregate-signature verification [206]
5. Multi-signature verification [182,207]
6. Blind signature generation [207]
7. Identity-based signature generation [208]
8. Identity-based encryption [209]
9. Searchable public key encryption [210]
10. One round three party key agreement [211]
Table 5.1 provides brief descriptions of these protocols and their computational requirements (in terms of $G_1 / G_2$ point additions, $G_1 / G_2$ ECSMs, hash-to-$G_1$ maps, $G_T$ exponentiations, pairings and multi-pairings) are summarized in Table 5.2.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description and Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Signatures [205]</td>
<td>Boneh-Lynn-Shacham (BLS) pairing-based signatures are half the length of traditional</td>
</tr>
<tr>
<td></td>
<td>elliptic curve-based digital signatures; useful in low-bandwidth network applications</td>
</tr>
<tr>
<td>Signature Aggregation [206]</td>
<td>Enables aggregation of multiple pairing-based signatures into a single signature; used to</td>
</tr>
<tr>
<td></td>
<td>authenticate mesh networks and blockchains</td>
</tr>
<tr>
<td>Multi-Signatures [182,207]</td>
<td>Special case of signature aggregation where all signatures correspond to the same message</td>
</tr>
<tr>
<td>Blind Signatures [207]</td>
<td>Enables users to obtain signatures without revealing the message to the signer; used</td>
</tr>
<tr>
<td></td>
<td>to secure digital currency schemes</td>
</tr>
<tr>
<td>Identity-Based Signatures [208]</td>
<td>Signatures with public keys derived from digital identities of users; used to simplify</td>
</tr>
<tr>
<td></td>
<td>public key distribution and verification</td>
</tr>
<tr>
<td>Identity-Based Encryption [209]</td>
<td>Encryption with public keys derived from digital identities of users; used as alternative</td>
</tr>
<tr>
<td></td>
<td>to digital certificate-based authentication</td>
</tr>
<tr>
<td>Searchable Public Key Encryption  [210]</td>
<td>Enables public key encryption with the ability to search from a list of specified keywords</td>
</tr>
<tr>
<td></td>
<td>without decrypting the ciphertext; may be used for encrypted network traffic inspection</td>
</tr>
<tr>
<td>One Round Three Party Key Agreement [211]</td>
<td>Extension of traditional two-party Diffie-Hellman key exchange to enable three parties to compute a shared secret using only a single round of communications; may be extended to larger number of parties using a tree structure of two-party and three-party key agreements</td>
</tr>
</tbody>
</table>
Table 5.2: Computational requirements of pairing-based protocol implementations

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Computational Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Short sign. gen.</td>
<td>$1 \times \text{Hash-to-}G_1 \text{ Map} + 1 \times G_1 \text{ ECSM}$</td>
</tr>
<tr>
<td>(2) Short sign. verif.</td>
<td>$1 \times \text{Hash-to-}G_1 \text{ Map} + 2 \times \text{Pairing}$</td>
</tr>
<tr>
<td>(3) Sign. aggregate</td>
<td>$(n - 1) \times G_1 \text{ Point Addition}$</td>
</tr>
</tbody>
</table>
| (4) Agg.-sign. verif.     | $n \times \text{Hash-to-}G_1 \text{ Map} + 1 \times \text{Pairing}$  
  $+ 1 \times n\text{-fold Multi-Pairing}$ |
| (5) Multi-sign. verif.    | $1 \times \text{Hash-to-}G_1 \text{ Map} + 2 \times \text{Pairing}$  
  $+ (n - 1) \times G_2 \text{ Point Addition}$ |
| (6) Blind sign. gen.      | $1 \times \text{Hash-to-}G_1 \text{ Map} + 3 \times G_1 \text{ ECSM}$  
  $+ 1 \times F_q \text{ Inversion}$ |
| (7) ID-based sign. gen.   | $1 \times \text{Hash-to-}G_1 \text{ Map} + 1 \times \text{Pairing}$  
  $+ 3 \times G_1 \text{ ECSM} + 1 \times G_1 \text{ Point Addition}$ |
| (8) ID-based encrypt.     | $1 \times \text{Hash-to-}G_1 \text{ Map} + 1 \times \text{Pairing}$  
  $+ 2 \times G_1 \text{ ECSM}$ |
| (9) Searchable encrypt.   | $1 \times \text{Hash-to-}G_1 \text{ Map} + 1 \times \text{Pairing}$  
  $+ 2 \times G_1 \text{ ECSM}$ |
| (10) 1-rnd 3-party key agt.| $2 \times G_1 \text{ ECSM} + 1 \times G_2 \text{ ECSM}$  
  $+ 1 \times \text{Pairing}$ |

Figure 5-11: Pairing-based protocol implementation benchmarks.
Aggregate-signature verification is implemented using the multi-pairing technique from Section 5.2 with shared Miller Loop and Final Exponentiation. Multi-signature is a special case of aggregate signature where all signatures correspond to the same message. Verification of blind signature and ID-based signature works similar to short signature verification, so they are not implemented separately. Wherever possible, \( \mathbb{G}_T \) exponentiations are replaced with \( \mathbb{G}_1 \) or \( \mathbb{G}_2 \) ECSMs using bilinear property [212].

Fig. 5-11 compares the energy consumption (at 0.66 V) of our hardware-accelerated implementations of these protocols with RISC-V software. Signature aggregation, aggregate-signature verification and multi-signature verification are implemented for \( n = 16 \) signatures and the energy consumption per signature is reported. Clearly, hardware acceleration provides two orders of magnitude improvement (about \( 130-140 \times \)) in performance and energy-efficiency compared to software.

### 5.4.3 Implementation of Blind Polynomial Evaluation

Apart from the traditional applications of pairings in signatures, public key encryption and key exchange (discussed in the previous sub-section), we also implement one of the modern pairing-based protocols – verifiable blind evaluation of polynomials. This protocol is one of the integral components of zero-knowledge succinct non-interactive arguments of knowledge, also known as zk-SNARKs, which are used to guarantee privacy in blockchain and cryptocurrency mechanisms such as Zcash [213].

The setup phase of this protocol pre-computes \((E_1(1), E_1(s), \ldots, E_1(s^d))\) and \((E_2(\alpha), E_2(\alpha s), \ldots, E_2(\alpha s^d))\) for \( s, \alpha \in \mathbb{F}_q \), where \( E_1(x) = xG_1 \in \mathbb{G}_1 \) and \( E_2(x) = xG_2 \in \mathbb{G}_2 \). The prover computes \( \beta_1 = E_1(P(s)) = \sum_{i=0}^{d} c_i s^i G_1 \) and \( \beta_2 = E_2(\alpha P(s)) = \alpha \sum_{i=0}^{d} c_i s^i G_2 \), where \( P(x) = \sum_{i=0}^{d} c_i x^i \) is the polynomial to be evaluated (\( c_i \in \mathbb{F}_q \)). Then, the verifier checks whether \( e(\beta_1, \alpha G_2) = e(G_1, \beta_2) \). Clearly, the prover needs to perform significantly more computation compared to the verifier, especially for large degree polynomials. The prover needs to compute \( d + 1 \) ECSMs each in \( \mathbb{G}_1 \) and \( \mathbb{G}_2 \) along with \( d \) point additions each in \( \mathbb{G}_1 \) and \( \mathbb{G}_2 \). With hardware acceleration, this requires \( \approx 45.5 \) ms per polynomial coefficient. The verifier only needs to compute two pairings, which requires \( \approx 75.5 \) ms in total.
5.4.4 Comparison with Previous Work

Table 5.3 compares our design with previous work on pairing accelerators. This work is the first to demonstrate the newly proposed high security BLS12-381 curve in hardware, while previous works [191–193] implement lower security BN curves. Our design is an order of magnitude more energy-efficient than the embedded-scale accelerator in [193], enabled, in part, by our choice of modular arithmetic architecture. Compared to the high-performance accelerators in [191,192], our design is an order of magnitude smaller with significantly lower power consumption due to power-performance trade-offs. We also implement side-channel countermeasures for stronger security and provide the flexibility to accelerate a variety of pairing-based protocols in hardware.

<table>
<thead>
<tr>
<th>Table 5.3: Comparison of our pairing crypto-processor with previous work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tech (nm)</strong></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Voltage (V)</td>
</tr>
<tr>
<td>Freq (MHz)</td>
</tr>
<tr>
<td>Total Area</td>
</tr>
<tr>
<td>Logic Gates</td>
</tr>
<tr>
<td>Logic Gates</td>
</tr>
<tr>
<td>Avg. Power</td>
</tr>
<tr>
<td>Pairing Curve</td>
</tr>
<tr>
<td>Side-Channel Countermeasures</td>
</tr>
<tr>
<td>Pairing Energy</td>
</tr>
<tr>
<td>Multi-Pairing Energy (n = 16)</td>
</tr>
<tr>
<td>Hardware Acceleration Support</td>
</tr>
</tbody>
</table>

a Post-synthesis results reported by [193]  
b [192] implemented FDSOI body-bias tuning for low energy  
c [192] reported multi-pairing performance with n = 7, extrapolated to n = 16
5.4.5 Side-Channel Analysis

All modular arithmetic and elliptic curve operations accelerated by our crypto-processor are constant-time. To ensure security against timing and simple power analysis (SPA) attacks [76], the following techniques are used in our ECSM and pairing implementations, as discussed in Section 5.2:

- complete point addition formulas [198]
- double-and-add-always technique [214]

All the benchmarks and measurement results discussed earlier were based on our SPA-secure (and constant-time) implementations. In high security use cases, it can also be configured to protect against stronger differential power analysis (DPA) attacks [76]. The following techniques are used in our DPA-secure implementations:

- randomized projective coordinates [215, 216], where elliptic curve projective points \((X : Y : Z)\) in both ECSM and pairing are transformed into the form \((\lambda X : \lambda Y : \lambda Z)\) with non-zero random \(\lambda \in \mathbb{F}_p\)
- ECSM with random scalar splitting [214], where secret scalar \(k \in \mathbb{F}_q\) is split into two parts \(r, k - r \in \mathbb{F}_q\) so that the ECSM is computed as \(kP = rP + (k - r)P\)
- pairing with random exponents and bilinear property [216], computed as \(e(aP, bQ) = e(P, Q)^{ab} = e(P, Q)\) with random \(a \in \mathbb{F}_q\) and \(b = a^{-1} \mod q\)

The first technique is practically free, requiring only a few \(\mathbb{F}_p\) multiplications. The second technique can be significantly simplified by using the multi-exponentiation technique (Shamir’s trick) [36], where \(2P\) is pre-computed and both scalars \(r\) and \(k - r\) are processed simultaneously to share the point doubling step and merge the point addition steps. The third technique is relatively expensive, requiring one \(G_1\) ECSM, one \(G_2\) ECSM and one \(\mathbb{F}_q\) inversion. All three also require generation of one random element in \(\mathbb{F}_p\) or \(\mathbb{F}_q\), which is performed using the SHA2-256 accelerator in our chip (SHA2-256-HMAC-DRBG [65] followed by rejection sampling). Fig. 5-12 compares the energy consumption of our implementations of BLS12-381 \(G_1\) ECSM
and pairing with SPA and DPA countermeasures. DPA-secure ECSM is only 10% more expensive compared to SPA-secure, thus making it a very attractive option for applications requiring reuse of the same secret scalar. However, DPA-secure pairing is $2.3\times$ more expensive than SPA-secure, so it should be used carefully. We note that all these countermeasures can be implemented without making any changes to the hardware design, by utilizing the programmability of our crypto-processor.
Fig. 5-13 shows our power side-channel measurement setup. Our test board has an 18 Ω resistor connected in series between the chip’s $V_{DD}$ pin and power supply. The voltage across this resistor, proportional to the chip’s current draw, is magnified using a differential amplifier (AD8001 op-amp chip [77], with 6 dB flat gain up to 100 MHz, in non-inverting configuration with resistors of appropriate sizes) and then observed using a 2.5 GS/s Tektronix MDO3024 mixed domain oscilloscope [78].

Fig. 5-14 and Fig. 5-15 show power traces from our constant-time SPA-secure hardware-accelerated BLS12-381-based $\mathbb{G}_1$ ECSM and pairing respectively. Their constant-time behaviour was also verified with 10,000 random executions. Fig. 5-16
Figure 5-16: Difference-of-means test with 99.99% confidence interval for SPA-secure implementations of (a) $G_1$ ECSM and (b) pairing.

shows the difference-of-means test results for $G_1$ ECSM and pairing over 1,000 traces with 99.99% confidence interval. We observe that measured absolute difference-of-means is smaller than confidence interval, thereby indicating SPA resistance.

To validate the side-channel security of our DPA-secure implementations, we have performed experimental leakage assessment tests, also known as *non-specific fixed vs. random* $t$-tests [80]. For the $t$-test, power waveforms are divided into two sets $Q_0$ (for fixed input) and $Q_1$ (for random input) of sizes $N_0$ and $N_1$ respectively, where $N_0 + N_1 = N$ is the total number of measurements. The $t$-test statistic is defined as $t = (\mu_0 - \mu_1) / \sqrt{\frac{\sigma_0^2}{N_0} + \frac{\sigma_1^2}{N_1}}$, where $\mu_0$, $\mu_1$ and $\sigma_0^2$, $\sigma_1^2$ are means and standard variances of $Q_0$, $Q_1$ respectively. Such $t$-values are determined for increasing number $N$ of measurements, and $|t| > 4.5$ indicates information leakage. In Fig. 5-17, we show $t$-test results for DPA-secure $G_1$ ECSM and pairing, over 10,000 measurements each.

Figure 5-17: Leakage test results for DPA-secure implementations of (a) $G_1$ ECSM and (b) pairing, with red dotted line indicating the $|t| = 4.5$ threshold.
5.5 Summary and Contributions

Pairing-based cryptography has recently gained a lot of interest in the context of IoT security applications due to the novel cryptographic constructions enabled by pairings, e.g., signature aggregation, identity-based encryption, functional encryption, etc. In this chapter, we have presented a low-power programmable crypto-processor to accelerate ECC and PBC using the recently proposed BLS12-381 pairing-friendly elliptic curve (also IETF standardization candidate). Our design is the first hardware accelerator to support the BLS12-381 pairing group.

Our pairing hardware implementation enables more than two orders of magnitude improvement in performance and energy-efficiency compared to embedded software. Several circuit, architecture and algorithm techniques are used to achieve this energy-efficient design. A 64-bit word-serial Montgomery modular arithmetic unit provides up to 50% energy savings compared to traditional designs with smaller word sizes. Karatsuba-style divide-and-conquer techniques are used to reduce energy consumption of the pairing computation by 35%. Strategically sharing computations between the Miller Loop and the Final Exponentiation gives another 30% energy savings. A hierarchical memory architecture with dedicated clock gates is used to achieve additional 20% reduction in energy consumption. Special properties of the BLS12-381 curve are exploited to further provide up to $2 \times$ improvement in performance and energy-efficiency of different pairing-based algorithms.

All key building blocks in our design are constant-time and we also implement several countermeasures against timing and power side-channel attacks, including both simple power analysis and differential power analysis. Experimental validation results are provided.

Our crypto-processor can be programmed with a set of custom instructions to accelerate many different pairing-based security protocols. Owing to the flexibility of our cryptographic core, new protocols, algorithm optimizations and side-channel countermeasures can also be easily realized using the same design.
Chapter 6

Efficient Privacy-Preserving Computation from Pairings

With the exponential growth in cloud computing in the recent years, we have witnessed the emergence of “computing as a service”, where big data computation is outsourced to cloud servers. While the powerful cloud infrastructure enables computationally complex data analysis, there are also growing concerns about privacy of the data being processed. Therefore, there is significant interest in the field of “secure outsource computation” or “computation on encrypted data”. This includes various cryptographic tools [217] such as secure multi-party computation, homomorphic encryption, functional encryption and secret sharing, which all serve the same purpose of allowing outsourced computation (under different threat models, network constraints and security requirements) without revealing the data being computed upon. In this work, we implement efficient functional encryption based on pairings. Traditional encryption schemes allow users to either decrypt the original data or recover no information at all. Functional encryption allows users to compute a function of the encrypted data during the process of decryption, but the original data is never revealed [218]. One of the most practical constructions of functional encryption computes inner products of secret vectors, also know as inner product encryption. In this chapter, we discuss efficient algorithms along with software and hardware-accelerated implementation results for the function-hiding inner product encryption scheme [183] based on the BLS12-381 pairing group.
Details of our algorithm optimizations and implementation results are discussed in Sections 6.2, and 6.3. We employ fast elliptic curve scalar multiplication using skew Frobenius map, scalar decomposition and comb pre-computation to enable $3.5 \times$ speedup in encryption. We use fast multi-pairing with shared Miller loop and final exponentiation along with power tree-based table construction for discrete logarithm together for $3 \times$ speedup in decryption. We also present two example applications for privacy-preserving biomedical sensor data classification and privacy-preserving wireless fingerprint-based indoor localization.

### 6.1 Background

In this section, we provide a brief introduction to pairing-based functional encryption and its efficient construction. We use bold lower-case symbols to denote vectors and bold upper-case symbols to denote matrices. The binary representation of a $t$-bit integer $k$ is denoted as $(k_{t-1}, \cdots, k_1, k_0)_2$, where $k_i \in \{0, 1\} \ \forall \ 0 \leq i < t$. The set of all integers is denoted as $\mathbb{Z}$ and the quotient ring of integers modulo $q$ is denoted as $\mathbb{Z}_q$. For two vectors $\mathbf{v}$ and $\mathbf{w}$, their inner product is written as $\langle \mathbf{v}, \mathbf{w} \rangle$. For a matrix $\mathbf{B}$, its transpose is denoted as $\mathbf{B}^T$ and its determinant is denoted as $\det(\mathbf{B})$. The general linear group of $n \times n$ invertible matrices over $\mathbb{Z}_q$ is denoted by $\text{GL}_n(\mathbb{Z}_q)$. The bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is a function which maps two elements from groups $\mathbb{G}_1$ and $\mathbb{G}_2$ to the target group $\mathbb{G}_T$, where all three groups are of prime order $q$. Here, $\mathbb{G}_1$ and $\mathbb{G}_2$ are elliptic curve groups with generator points $G_1 \in \mathbb{G}_1$ and $G_2 \in \mathbb{G}_2$ respectively, while $\mathbb{G}_T$ is the subgroup of a large extension field. The group operations in $\mathbb{G}_1$ and $\mathbb{G}_2$ are written additively, while the group operation in $\mathbb{G}_T$ is written multiplicatively. For vector $\mathbf{v} = (v_1, \cdots, v_n) \in \mathbb{Z}_q^n$, the corresponding vector of group elements $(v_1 G_1, \cdots, v_n G_n)$ is denoted by $\mathbf{v} G_1$. For scalar $k \in \mathbb{Z}_q$ and vectors $\mathbf{v}, \mathbf{w} \in \mathbb{Z}_q^n$, we also use the expressions $k \cdot (\mathbf{v} G_1) = k \mathbf{v} G_1$ and $(\mathbf{v} G_1) + (\mathbf{w} G_1) = (\mathbf{v} + \mathbf{w}) G_1$ (these apply to $\mathbb{G}_2$ as well). Finally, the bilinear pairing operation can be applied to such vectors as:

$$e(\mathbf{v} G_1, \mathbf{w} G_2) = \prod_{i=1}^{i=n} e(v_i G_1, w_i G_2) = e(G_1, G_2) \sum_{i=1}^{i=n} v_i \cdot w_i = e(G_1, G_2)^{\langle \mathbf{v}, \mathbf{w} \rangle}$$
6.1.1 Functional Encryption

In traditional cryptographic encryption schemes, the recipient either decrypts the data if they possess the correct secret key, or they are unable to decrypt it and hence learns nothing at all about the encrypted data. Functional encryption (FE), proposed in 2011 by Boneh, Sahai and Waters [218], is a type of encryption scheme where decryption allows a user to obtain a function of the encrypted data and nothing else. Given the encryption of a message $x$ and the secret key embedding a function $f$, the decryption output is $f(x)$.

6.1.2 Inner Product Encryption (IPE)

Several cryptographic schemes such as identity-based encryption (IBE), attribute-based encryption (ABE), searchable encryption, etc, whose pairing-based constructions and implementations were discussed in Chapter 5, can be viewed as special cases of functional encryption. In this chapter, we are going to exclusively focus on functional encryption schemes which embed the inner product functionality.

In an inner product encryption (IPE) scheme, the secret keys and the ciphertexts are respectively associated with vectors $x \in \mathbb{Z}_q^n$ and $y \in \mathbb{Z}_q^n$ of length $n$ with each element in $\mathbb{Z}_q$. Then, for secret key $sk_x$ corresponding to vector $x$ and ciphertext $ct_y$ corresponding to vector $y$, decryption outputs the value $\langle x, y \rangle \in \mathbb{Z}_q$.

6.1.3 Function-Hiding Inner Product Encryption (FHIPE)

Several constructions of inner product encryption schemes based on elliptic curves and pairings have been proposed by [183,219–222]. In this work, we are going to focus on the function-hiding inner product encryption (FHIPE) scheme proposed by Kim et al. [183], which is the most efficient construction till date. Along with efficiency, it also hides the underlying function, that is, both vectors $x$ and $y$ remain secret to the decryptor. This makes it ideal for privacy-preserving computation applications, for example, when the decryption is performed by a third party cloud server which cannot be trusted with the plaintext vectors, as will be discussed later.
The FHIPE scheme of Kim et al. [183] consists of the following four algorithms, where \( \lambda \in \mathbb{N} \) is a security parameter, \( n \) is a positive integer and \( S \) is a polynomial-sized subset of \( \mathbb{Z}_q \) (\(|S| = \text{poly}(\lambda)\)):

- **Setup** \((1^\lambda, S) \rightarrow (pp, msk)\) : for security parameter \( \lambda \), the setup algorithm samples a matrix \( B \in \mathbb{GL}_n(\mathbb{Z}_q) \) and outputs public parameters \( pp = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, e, S) \) corresponding to the bilinear map, along with master secret key \( msk = (pp, \mathbb{G}_1, \mathbb{G}_2, B, B^*) \), where \( B^* = \det(B) \cdot (B^{-1})^T \).

- **KeyGen** \((msk, x) \rightarrow sk_x\) : the key generation algorithm outputs secret key \( sk_x = (k_1, k_2) = (\alpha \cdot \det(B) \mathbb{G}_1, \alpha \cdot x \cdot B \mathbb{G}_1) \) corresponding to vector \( x \in \mathbb{Z}_q^n \), where \( \alpha \in \mathbb{Z}_q \) is a uniformly random element.

- **Encrypt** \((msk, y) \rightarrow ct_y\) : the encryption algorithm generates ciphertext \( ct_y = (c_1, c_2) = (\beta \mathbb{G}_2, \beta \cdot y \cdot B^* \mathbb{G}_2) \) corresponding to vector \( y \in \mathbb{Z}_q^n \), where \( \beta \in \mathbb{Z}_q \) is a uniformly random element.

- **Decrypt** \((pp, sk_x, ct_y) \rightarrow z \in S \cup \{\bot\}\) : the decryption algorithm computes:

\[
d_1 = e(k_1, c_1) = e(G_1, G_2)^{\alpha \beta \cdot \det(B)}
\]

and,
\[
d_2 = e(k_2, c_2) = e(G_1, G_2)^{\alpha \beta \cdot x \cdot B \cdot (B^*)^T \cdot y^T}
\]

Since \( B \cdot (B^*)^T = \det(B) \cdot I_{n \times n} \), where \( I_{n \times n} \) is the \( n \times n \) identity matrix, we have \( d_2 = e(G_1, G_2)^{\alpha \beta \cdot \det(B) \cdot \langle x, y \rangle} \). Therefore, \( d_2 = d_1^{\langle x, y \rangle} \), and the decryptor checks whether there exists \( z \in S \) such that \( d_2 = d_1 \). It outputs \( z = \langle x, y \rangle \) if it is able to find such a value, else outputs \( \bot \). The correctness of this decryption holds only when the plaintext vectors \( x, y \) satisfy the property \( \langle x, y \rangle \in S \).

The **Setup** and **KeyGen** steps can be performed once for setting up a particular functionality and application. Once the keys are set up and stored, only the **Encrypt** and **Decrypt** steps are used for encryption and decryption respectively. Note that implementing the decryption algorithm requires solving the discrete logarithm problem (DLP) over the set \( S \), thus putting constraints on the size of set \( S \) and consequently the length \( n \) of the vectors \( x, y \) and the maximum possible values of their elements, as will be discussed next.
6.2 Optimized FHIPE Encryption and Decryption

6.2.1 Analysis of Computation Cost

The two most computationally expensive steps in the FHIPE scheme are Encrypt and Decrypt. Since key setup is done once per application, its cost gets amortized over several subsequent invocations of encryption and decryption. Before optimizing and implementing encryption and decryption, we first analyze their computation costs:

- **Cost of Encrypt:**
  - Multiplication of $1 \times n$ row vector $y$ with $n \times n$ matrix $B^*$
  - Sampling of uniformly random scalar $\beta \in \mathbb{Z}_q$
  - Multiplication of each element of the $1 \times n$ row vector $y \cdot B^*$ with $\beta$
  - Elliptic curve scalar multiplication (ECSM) of point $G_2 \in G_2$ by scalar $\beta$ and also by each element of the $1 \times n$ row vector $\beta \cdot y \cdot B^*$

- **Cost of Decrypt:**
  - Pairing $d_1 = e(k_1, c_1)$
  - Multi-pairing $d_2 = e(k_2, c_2) = \prod_{i=1}^{n} e(k_{2,i}, c_{2,i})$, where $k_{2,i}$ and $c_{2,i}$ denote the $i$-th elements of vectors $k_2$ and $c_2$ respectively
  - Solution of the discrete logarithm $d_2 = d_1^z$ over $G_T$ with $z \in S$

All matrix and vector arithmetic are performed modulo $q$. Also, let the elements of vectors $x$ and $y$ be bounded as $x_i \leq B_x$ and $y_i \leq B_y$ respectively. Then, we have

$$\langle x, y \rangle \leq nB_xB_y \Rightarrow nB_xB_y < q \quad \text{since} \quad \langle x, y \rangle \in S \subset \mathbb{Z}_q$$

However, in practice, we will need to have $nB_xB_y \ll q$ (to be discussed later) so that the discrete logarithm can be computed efficiently for successful decryption. For encryption, we have two options: (1) multiply the elements of $y \cdot B^*$ by $\beta$ and then perform ECSM of $G_2$ by these scalars (and also by $\beta$), or (2) compute the point $\beta G_2$ and then perform ECSM of $\beta G_2$ by the elements of $y \cdot B^*$. We will choose the first option so that multiple invocations of Encrypt always involve ECSM computations with
different scalars (even if $y$ remains the same), thus inherently providing a side-channel countermeasure through scalar randomization [76].

For Encrypt, majority of the computation is due to $n + 1$ point multiplications (ECSMs) in $G_2$. For Decrypt, compute is dominated by $n$-fold multi-pairing along with solving the DLP. Next, we discuss different techniques to reduce this cost. In this work, we will give special attention to bilinear maps based on the BLS12-381 pairing-friendly elliptic curve [180,181], as discussed in detail in Chapter 5. For better readability, we once again summarize the key parameters:

- The base field is $F_p$, where $p$ is the following 381-bit prime:
  
  $0x1a0111ea397fe69a4b1ba7b6434bacad764774b84f38512bf6730d2a0 \cdots$
  
  $\cdots f6b0f6241eabfffeb153ffffbfefefefefefefefefefefefefefefefefefefefefefefefefefef
h$

- $G_1$ is a $q$-order subgroup of $E(F_p) : y^2 = x^3 + 4$

- $G_2$ is a $q$-order subgroup of $E'(F_{p^2}) : y^2 = x^3 + 4 (1 + \alpha)$

- $G_T$ is a $q$-order cyclotomic subgroup (containing the $q$-th roots of unity) of $F_{p^{12}}^*$

- Order $q$ of all three groups is the following 255-bit prime:
  
  $0x73eda753299d7d483339d80809a1d80553bda402ffe5bfeffffff00000001\_h$

- The BLS12 family-of-curves parameter is $u = -0xd201000000010000\_h$, which is a 64-bit integer, so that $p = \frac{1}{3}(u - 1)^2(u^4 - u^2 + 1) + u$ and $q = u^4 - u^2 + 1$

Further details about the BLS12-381 curve are available in [180] and [190].

Next, we discuss various algorithm optimizations we perform to achieve efficient FHIPE encryption and decryption based on BLS12-381. We present detailed results of our software implementation on three different platforms with variety of performance metrics: (1) RISC-V RV32IM at 90 MHz (on the test chip from Chapter 5 with 64 KB RAM), (2) ARM Cortex-M7 at 600 MHz (on Teensy 4.0 [223] with 1 MB RAM) and (3) Intel Cascade Lake at 2.4 GHz (on dodeca-core Intel Xeon Silver 4214R CPU [224] with 32 GB RAM). Our software library is written entirely in C with the underlying multi-precision arithmetic using 32-bit limbs for the 32-bit architectures (1)-(2) and 64-bit limbs for the 64-bit architecture (3). We also present hardware-software co-design results using the crypto-processor from Chapter 5.
6.2.2 Efficient FHIPE Encryption and its Implementation

In the FHIPE scheme, encryption requires $n$-dimensional matrix-vector multiplication in $\mathbb{F}_q$ along with $n + 1$ point multiplications (ECSMs) in $\mathbb{G}_2$, which is a $q$-order subgroup of the elliptic curve $E'(\mathbb{F}_{p^2})$. Here, $E'$ is a sextic twist (of degree 6) of the curve $E$, and the corresponding isomorphism is [37, 225]:

$$
\Psi_6 : \begin{cases}
E'(\mathbb{F}_{p^2}) & \rightarrow E(\mathbb{F}_{p^2}) \\
(x, y) & \rightarrow (x \xi^{-1/3}, y \xi^{-1/2})
\end{cases}
$$

where $x, y \in \mathbb{F}_{p^2}$, $\xi = 1 + \alpha \in \mathbb{F}_{p^2}$ is a cubic and quadratic non-residue in the field of definition of the twist and the polynomial $X^6 - \xi$ is irreducible over $\mathbb{F}_{p^2}$. Note that the isomorphism $\Psi_6$ is slightly different for the BLS12-381 curve, which has an M-type twist, compared to the traditionally used BN curves with D-type twists.

**Skew Frobenius Map:** The twist curve $E'$ supports the skew Frobenius map [226–228] shown below:

$$
\hat{\phi} : \begin{cases}
E'(\mathbb{F}_{p^2}) & \rightarrow E'(\mathbb{F}_{p^2}) \\
(x, y) & \rightarrow (x \xi^{p(p-1)/6}, y \xi^{-3(p-1)/6})
\end{cases}
$$

where $x, y \in \mathbb{F}_{p^2}$ and the skew Frobenius map satisfies $\hat{\phi}(P) = pP \forall P \in E'(\mathbb{F}_{p^2})$, that is, it easily maps any point in $\mathbb{G}_2$ to its $p$-th scalar multiple.

**Computation Analysis:** Any element $z$ in $\mathbb{F}_{p^2} = \mathbb{F}_p[\alpha]/(\alpha^2 + 1)$ can be written as $z = z_0 + z_1 \alpha$, where $z_0, z_1 \in \mathbb{F}_p$. Then, the $p$-th power of $z$ is $z^p = \bar{z} = z_0 - z_1 \alpha$, that is, a simple conjugation [186]. Further, from Chapter 5, we already have the constant $\delta = \xi^{(p-1)/6} \in \mathbb{F}_{p^2}$ used in the final exponentiation step of pairing computation. Then, we can easily pre-compute $\delta^{-1} \in \mathbb{F}_{p^2}$, and we note that $\delta^{-1} = \delta' (1 + \alpha) = \delta' \xi$ where $\delta' \in \mathbb{F}_p$ is a scalar. With this information, we can now re-write the map $\hat{\phi}$ as:

$$
\hat{\phi}(x, y) = (\delta'^2 \xi^2 \bar{x}, \delta'^3 \xi^3 \bar{y})
$$

We use this expression to analyze the computation cost of $\hat{\phi}$. Clearly, each conjugation
requires only one negation in \( \mathbb{F}_p \). Also, multiplications by \( \xi^2, \xi^3 \) require only additions, subtractions and negations in \( \mathbb{F}_p \), as shown below:

\[
\begin{align*}
\xi \bar{z} & = (1 + \alpha) \cdot (z_0 - z_1\alpha) = (z_0 + z_1) + (z_0 - z_1)\alpha \\
\xi^2 \bar{z} & = (1 + \alpha)^2 \cdot (z_0 - z_1\alpha) = 2\alpha \cdot (z_0 - z_1\alpha) = 2z_1 + 2z_0\alpha \\
\xi^3 \bar{z} & = (1 + \alpha)^3 \cdot (z_0 - z_1\alpha) = 2(\alpha - 1) \cdot (z_0 - z_1\alpha) = 2(z_1 - z_0) + 2(z_0 + z_1)\alpha
\end{align*}
\]

Since we can pre-compute both the scalars \( 2\delta^{'2} \in \mathbb{F}_p \) and \( 2\delta^{'3} \in \mathbb{F}_p \), we only need to perform four modular multiplications. Overall, the computation cost of \( \tilde{\phi} \) is \( 4M_1 + 2A_1 \) where \( A_1 \) and \( M_1 \) respectively denote modular addition (also equivalent to subtraction or negation) and modular multiplication in \( \mathbb{F}_p \). Clearly, this is significantly cheaper than explicitly computing the point multiplication by \( p \) which would require several thousand \( \mathbb{F}_p \) multiplications.

**Proposed Fast ECSM in \( \mathbb{G}_2 \):** Now, we describe our proposed technique to speed up \( \mathbb{G}_2 \) point multiplication in FHE encryption, based on the skew Frobenius map. For the BLS12-381 pairing-friendly elliptic curve, we have \( q \mid (p - u) \) since \( p = \frac{1}{3}(u - 1)^2(u^4 - u^2 + 1) + u \) and \( q = u^4 - u^2 + 1 \). For any point \( P \in \mathbb{G}_2 \), we know that \( qP = O \), where \( O \) is the point at infinity. Therefore, we have:

\[
(p - u)P = O \Rightarrow \hat{\phi}(P) = pP = uP \Rightarrow \hat{\phi}(\hat{\phi}(P)) = p^2P = u^2P
\]

Since \( q \) is 255-bit long and \( u^2 \) is 128-bit long, any scalar \( k \in \mathbb{F}_q \) can be decomposed as \( k = k^{(1)} + k^{(2)}u^2 \) where \( k^{(1)}, k^{(2)} \) are both half the bit-size of \( k \) (roughly 128-bit long):

\[
kP = (k^{(1)} + k^{(2)}u^2)P = k^{(1)}P + k^{(2)}\hat{\phi}(\hat{\phi}(P))
\]

This allows us to decompose one 255-bit scalar multiplication in \( \mathbb{G}_2 \) into two smaller 128-bit scalar multiplications. Finally, we apply the multi-exponentiation technique (also known as Shamir’s trick) from [36,229,230] to efficiently combine them into one simultaneous 128-bit multi-scalar multiplication, as shown in Algorithm 6.1. In lines 13-14, we have used a dummy point addition with \( T_{dummy} \) to prevent timing side-channel attacks. The computation cost of this approach is compared with traditional
Algorithm 6.1 Fast point multiplication in $\mathbb{G}_2$ (for BLS12-381)

Require: $P \in \mathbb{G}_2$, and $k^{(1)} = (k_{127}^{(1)}, \ldots, k_1^{(1)}, k_0^{(1)})_2$, $k^{(2)} = (k_{127}^{(2)}, \ldots, k_1^{(2)}, k_0^{(2)})_2$ such that $k = k^{(1)} + k^{(2)} u^2$ for scalar $k \in \mathbb{F}_q$

Ensure: $T = kP \in \mathbb{G}_2$

1: $Q \leftarrow \hat{\phi}(\hat{\phi}(P))$
2: $R \leftarrow P + Q$
3: $T \leftarrow \mathcal{O}$
4: $T_{dummy} \leftarrow \mathcal{O}$
5: for $(i = 127; i \geq 0; i = i - 1)$ do
6: $T \leftarrow 2T$
7: if $k_i^{(1)} = 1$ and $k_i^{(2)} = 1$ then
8: $T \leftarrow T + R$
9: else if $k_i^{(1)} = 0$ and $k_i^{(2)} = 1$ then
10: $T \leftarrow T + Q$
11: else if $k_i^{(1)} = 1$ and $k_i^{(2)} = 0$ then
12: $T \leftarrow T + P$
13: else
14: $T_{dummy} \leftarrow T + P$
15: end if
16: end for
17: return $T$

ECSM in terms of the number of modular additions and multiplications (in $\mathbb{F}_p$) using our Python reference implementation. While the traditional ECSM would require $14,643 M_1 + 43,617 A_1$, our proposed technique requires $8,317 M_1 + 22,021 A_1$, which is 43% faster ($\approx 1.8 \times$). Projective coordinate formulas (see Chapter 5) are used for all elliptic curve point operations.

Constant-Time Scalar Decomposition: The scalar decomposition can be performed using long division by $u^2$. The binary version of traditional long division [231] is not constant-time due to the use of conditional subtractions and additions. Algorithm 6.2 shows our modified constant-time version which prevents timing side-channel leakage of information about the secret scalar. In line 8, the binary mask $\beta$ is derived from the borrow generated when subtracting $u^2$ from $k^{(1)}$. Conditional subtractions and additions are avoided by using bit-wise masked operands in lines 9-10. Since both the quotient $k^{(2)}$ and the remainder $k^{(1)}$ are 128-bit long (after zero padding, if required), the conditional in line 7 is set to “$i < 128$” and 128-bit additions and
Algorithm 6.2 Proposed constant-time scalar decomposition based on binary long division for efficient FHIPE Encrypt (for BLS12-381)

Require: 255-bit scalar \( k = (k_{254}, \cdots, k_1, k_0)_2 \in \mathbb{Z}_q \), and 128-bit constant \( u^2 \)

Ensure: 128-bit scalars \( k^{(1)} \) and \( k^{(2)} \) such that \( k = k^{(1)} + k^{(2)} u^2 \)

1: \( k^{(1)} \leftarrow 0 \)
2: \( k^{(2)} \leftarrow 0 \)
3: \( \alpha \leftarrow 2^{127} \)
4: for \( (i = 254; i \geq 0; i = i - 1) \) do
5: \( k^{(1)} \leftarrow k^{(1)} << 1 \)
6: \( k^{(1)} \leftarrow k^{(1)} + k_i \)
7: if \( i < 128 \) then
8: \( \beta \leftarrow \neg(k^{(1)} < u^2) \)
9: \( k^{(1)} \leftarrow k^{(1)} - (\beta \& u^2) \)
10: \( k^{(2)} \leftarrow k^{(2)} + (\beta \& \alpha) \)
11: \( \alpha \leftarrow \alpha >> 1 \)
end if
12: end for
13: return \((k^{(1)}, k^{(2)})\)

subtractions are used throughout Algorithm 6.2. The total number of such 128-bit arithmetic operations required is 383 additions and 256 subtractions, along with 255 left shifts and 128 right shifts.

**Efficient FHIPE Encryption:** In Algorithm 6.3, we construct an efficient version of FHIPE Encrypt by combining our proposed point multiplication technique (Algorithm 6.1) with the comb pre-computation technique from [36,62]. The comb method significantly speeds up the ECSM computation over fixed base points by pre-computing several points and processing multiple bits of the scalar simultaneously. Since all \( \mathbb{G}_2 \) point multiplications required in Encrypt are on the same point \( \mathbb{G}_2 \), this method fits perfectly in this case. Comb method with window size \( w \) speeds up the ECSM by \( \approx w \) times using \( 2^w - 1 \) pre-computed points [36]. Since we use the comb method in conjunction with multi-exponentiation (to integrate with Algorithm 6.1), we will need \( 2^{2w} - 1 \) pre-computed points, including those corresponding to \( \mathbb{G}_2 \), \( \hat{\phi}(\hat{\phi}(\mathbb{G}_2)) \) and their combinations. For the BLS12-381 curve, each point in \( \mathbb{G}_2 \) requires \( 4 \times 381 = 1524 \) bits of storage. In Algorithm 6.3, we set \( w = 2 \) so that 15 points need \( \approx 2.8 \) KB memory. In line 1, these points are denoted as \( R[1], R[2], \cdots, R[15] \), where
Algorithm 6.3 Efficient constant-time FHIPE Encrypt using fast point multiplication in $G_2$ and comb pre-computations (for BLS12-381)

Require: Vector $\mathbf{y} \in \mathbb{Z}_q^n$, matrix $\mathbf{B}^* \in \mathbb{GL}_n(\mathbb{Z}_q)$, and generator $G_2 \in G_2$

Ensure: $(c_1, c_2) = (\beta G_2, \beta \cdot \mathbf{y} \cdot \mathbf{B}^* G_2)$ for uniformly random $\beta \in \mathbb{Z}_q$

1. Pre-compute and store $R[1], R[2], \cdots, R[15] \in G_2$ where $R[2^3a_3 + 2^2a_2 + 2a_1 + a_0] = a_3 2^{64} \hat{\phi}(\hat{\phi}(G_2)) + a_2 \hat{\phi}(\hat{\phi}(G_2)) + a_1 2^{64} G_2 + a_0 G_2$ for $a_0, a_1, a_2, a_3 \in \{0, 1\}$
2. Generate uniformly random 255-bit $\beta \in \mathbb{Z}_q$ through rejection sampling
3. $c_1 \leftarrow \beta G_2$
4. $(k_1, k_2, \cdots, k_n) \leftarrow \beta \cdot \mathbf{y} \cdot \mathbf{B}^*$
5. $T_j \leftarrow \mathcal{O}$ for $1 \leq j \leq n$
6. $T_{\text{dummy}} \leftarrow \mathcal{O}$
7. for $(j = 1; j \leq n; j = j + 1)$ do
8. Decompose $k_j \in \mathbb{Z}_q$ into two 128-bit parts $k_j^{(1)} = (k_{j,127}, \cdots, k_{j,1}, k_{j,0})$ and $k_j^{(2)} = (k_{j,127}, \cdots, k_{j,1}, k_{j,0})$ such that $k_j = k_j^{(1)} + k_j^{(2)}$ using Algorithm 6.2
9. for $(i = 63; i \geq 0; i = i - 1)$ do
10. $T_j \leftarrow 2T_j$
11. $a \leftarrow 2^3k_{j,i+64} + 2^2k_{j,i} + 2k_{j,i+1} + k_{j,i}$
12. if $a = 0$ then
13. $T_{\text{dummy}} \leftarrow T_j + G_2$
14. else
15. $T_j \leftarrow T_j + R[a]$
16. end if
17. end for
18. end for
19. $c_2 \leftarrow (T_1, T_2, \cdots, T_n)$
20. return $ct_y = (c_1, c_2)$

$R[2^3a_3 + 2^2a_2 + 2a_1 + a_0] = a_3 2^{64} \hat{\phi}(\hat{\phi}(G_2)) + a_2 \hat{\phi}(\hat{\phi}(G_2)) + a_1 2^{64} G_2 + a_0 G_2$ for $a_0, a_1, a_2, a_3 \in \{0, 1\}$ (with $a_0, a_1, a_2, a_3$ not all zero). Since $R[1] = G_2$ is already available, we pre-compute the remaining 14 points. Note that $w = 3$ and $w = 4$ would require $\approx 11.7$ KB and $\approx 47.4$ KB memory respectively, both too expensive for resource-constrained embedded devices. The matrix-vector arithmetic in line 4 requires $n(n + 1)$ multiplications and $n(n - 1)$ additions in $\mathbb{F}_q$. In line 8, each element $k_j$ of the $1 \times n$ row vector $\beta \cdot \mathbf{y} \cdot \mathbf{B}^*$, written as a 255-bit scalar in $\mathbb{F}_q$, is decomposed into 128-bit components by performing constant-time long division by $u^2$ (Algorithm 6.2) to obtain the quotient $k_j^{(2)}$ and the remainder $k_j^{(1)}$.

We analyze the computation cost of our proposed approach using our Python reference implementation. Pre-computing the 14 points requires $11,221M_1 + 10,787A_1$, 139
but this needs to be performed only once and quickly gets amortized over subsequent encryption computations. Assuming that the cost of $\mathbb{F}_q$ arithmetic is equivalent to $\mathbb{F}_p$ (in a real implementation, $\mathbb{F}_q$ arithmetic is slightly less expensive than $\mathbb{F}_p$ since $q < p$), Encrypt requires $(14,643 + 4,138n + n(n + 1))M_1 + (43,617 + 10,956n + n(n - 1))A_1$ to encrypt a $1 \times n$ row vector. The matrix-vector multiplication introduces an $n^2$ term in the cost expression, but the remaining ECSM-related terms still dominate for reasonable vector sizes ($n < 4 \times 10^3$). In comparison, the baseline Encrypt from [183] requires $(14,643(n + 1) + n(n + 1))M_1 + (43,617(n + 1) + n(n - 1))A_1$ (without skew Frobenius map and comb; for fair comparison, we assume the baseline is constant-time and we still use the point arithmetic costs from Appendix C), thus making our approach up to $3.5 \times$ more efficient. Fig. 6-1 shows how computation cost (in terms of $M_1$ since $A_1 \ll M_1$) varies with different $n$, both for the baseline and our proposed approach. We observe a linear dependence with $n$ for $n \leq 10^3$.

**Software Implementation:** We implement Encrypt with our optimized constant-time approach (Algorithm 6.3) in software on the three platforms – (1) RISC-V RV32IM at 90 MHz, (2) ARM Cortex-M7 at 600 MHz and (4) Intel Cascade Lake at 2.4 GHz. All the pre-computed comb points $R[1], R[2], \cdots, R[15]$ and the secret
matrix $B^*$ are stored in Montgomery domain to facilitate efficient modular arithmetic. The random scalar $\beta$ is also considered to be sampled in Montgomery domain.

Our software implementation results are shown in Table 6.1. The measured execution times are reported for $n \in \{5, 10, 25, 50, 75, 100, 250, 500, 750, 1000\}$. On platforms (1) and (2), the available data memory allows for profiling up to $n = 25$ and $n = 100$ respectively, limited by the storage required by $n \times n$ matrix $B^*$. Energy consumption for the RISC-V at 1.1 V is also reported. The $\text{Encrypt}$ computation can be divided into three categories – matrix-vector arithmetic, scalar decompositions and elliptic curve point multiplications. From our software evaluation results, we observe that elliptic curve point multiplications (in $\mathbb{G}_2$) account for 99% of the total computation cost, while matrix-vector arithmetic and scalar decompositions together account for the remaining 1%. This not only justifies our choice of optimizing the $\mathbb{G}_2$ ECSM but also motivates the use of hardware acceleration.

**Hardware-Accelerated Implementation:** We implement $\text{Encrypt}$ on the custom chip from Chapter 5 using hardware-software co-design, where all modular arithmetic (including those required for matrix-vector arithmetic) and elliptic curve operations
are accelerated using our BLS12-381 pairing crypto-processor, and the RISC-V general-purpose micro-processor is used to handle control flow, scheduling, data movement and the scalar decompositions. The measured execution time and energy consumption (at 90 MHz and 1.1 V) are shown in Table 6.2. This is two orders of magnitude more efficient compared to software-only implementation on RISC-V (Table 6.1).

6.2.3 Efficient FHIPE Decryption and its Implementation

In the FHIPE scheme, decryption requires one pairing, one \( n \)-fold multi-pairing and solving a discrete logarithm in \( \mathbb{G}_T \) where the exponent belongs to a polynomial-sized subset \( S \) of \( \mathbb{Z}_q \). We set \( S = \{0, 1, \ldots, s - 1\} \) so that \( |S| = s \).

**Pairing and Multi-Pairing:** For fast multi-pairing, the Miller loop and final exponentiation computations are shared, as discussed in Section 5.2.3. The point and line arithmetic formulas discussed in Appendix C are used for both pairing and multi-pairing.

**Solving the Discrete Logarithm:** For solving discrete logarithms, we note that using Pollard’s rho algorithm [232] will require \( O(\sqrt{q}) \) iterations irrespective of \( s \), while using Pollard’s kangaroo algorithm [232], which is \( O(\sqrt{s}) \), requires computing several \( \mathbb{G}_T \) exponentiations. Therefore, we are going to use the *baby-step giant-step algorithm* [233] (also used by [183]), which is \( O(\sqrt{s}) \) and requires only \( \mathbb{G}_T \) squarings and multiplications, as discussed next.

**Baby-Step Giant-Step Algorithm:** This is based on a time-memory trade-off. To find \( z \in S \subset \mathbb{Z}_q \) such that \( d_2 = d_1^z \) for \( d_1, d_2 \in \mathbb{G}_T \), the exponent is written as

\[
z = i\alpha + j \quad \text{where} \quad \alpha = \lceil \sqrt{s} \rceil, \quad 0 \leq i < \alpha, \quad 0 \leq j < \alpha
\]

\[
\Rightarrow \quad d_1^{i\alpha+j} = d_2 \quad \Rightarrow \quad d_1^j = d_2 (d_1^{-\alpha})^i
\]

The baby-step giant-step algorithm first pre-computes \( d_1^j \) for all \( j \in [0, \alpha) \) and then searches for the value of \( i \in [0, \alpha) \) which satisfies the above relation, as shown in
Algorithm 6.4 Solving discrete logarithm using baby-step giant-step (from [233])

Require: $d_1, d_2 \in G_T$ and $S = \{0, 1, \ldots, s - 1\} \subset \mathbb{Z}_q$

Ensure: $z \in S$ if $d_2 = d_1^z$

1: $\alpha \leftarrow \lceil \sqrt{s} \rceil$
2: Pre-compute and store a table of pairs $(j, d_1^j)$ for $0 \leq j < \alpha$
3: $t_0 \leftarrow d_1^{-\alpha}$
4: $t_1 \leftarrow d_2$
5: for $(i = 0; i < \alpha; i = i + 1)$ do
6: if $t_1$ is the second component ($d_1^j$) of any pair in the table then
7: $z \leftarrow i\alpha + j$
8: return $z$
9: else
10: $t_1 \leftarrow t_1 \cdot t_0$
11: end if
12: end for
13: return $\bot$

Algorithm 6.4. Note that the table lookup in line 6 is executed $\lfloor z/\alpha \rfloor \leq \alpha$ times, and the $G_T$ multiplication in line 10 is executed $\lfloor z/\alpha \rfloor - 1 < \alpha$ times, depending on the discrete logarithm result $z$. Clearly, $s$ must still be small enough to allow the algorithm to execute in reasonable time. The associated memory requirement is the table of $\alpha$ elements in $G_T$, which requires $\alpha \times 12 \times 381$ bits $\approx 0.56\alpha$ KB.

Proposed Fast Lookup Table Construction: The conventional lookup table construction involves computing $d_1^2, d_1^3, \ldots, d_1^{\alpha}$ through $\alpha - 1$ repeated field multiplications ($d_1^0$ and $d_1^1$ are trivial). However, in case of FHIPE, we can do much better owing to special properties of the target field. We are going to utilize fast Granger-Scott cyclotomic squarings in $G_T$ [234], discussed in Appendix C. We note that $M_{12} \equiv 54M_1 + 224A_1$ and $cS_{12} \equiv 18M_1 + 107A_1$, so cyclotomic squarings are about 3 times cheaper than multiplications in $G_T$. To compute the powers $d_1^2, d_1^3, \ldots, d_1^{\alpha}$ using a combination of squarings and multiplications, we use Knuth’s power tree [231]. While the power tree is traditionally used for fast evaluation of a single power with minimum number of multiplications, we observe that it also fits perfectly in our application since it allows the strategic use of squarings. In fact, all even powers are generated through squarings and all odd powers through multiplications of previously computed values in the tree. Fig. 6-2 compares repeated multiplications with the power tree for $\alpha = 8$. 

143
Algorithm 6.5 Proposed fast lookup table construction using power tree

Require: Field element $d_1 \in G_T$ and integer $\alpha \geq 2$
Ensure: List of pre-computed field elements $T = \{1, d_1, \cdots, d_1^\alpha\}$

1: $T[0] \leftarrow 1$
2: $T[1] \leftarrow d_1$
3: Initialize power tree with root node 1 at level 0
4: $l \leftarrow 1$
5: while $T$ has $< \alpha + 1$ elements do
6: for each node $n$ in level $l - 1$ of power tree do
7: if $2n \leq \alpha$ then
8: Add node $2n$ to tree at level $l$ as child of node $n$
9: $T[2n] \leftarrow T[n]^2$
10: end if
11: for each node $m$ at levels 0 to $l - 2$ in the path from root to node $n$ do
12: if $n + m \leq \alpha$ and tree does not already contain node $n + m$ then
13: Add node $n + m$ to tree at level $l$ as child of node $n$
14: $T[n + m] \leftarrow T[n] \cdot T[m]$
15: end if
16: end for
17: end for
18: $l \leftarrow l + 1$
19: end while
20: return $T$

Figure 6-2: Computation of $d_1^2, d_1^3, \cdots, d_1^\alpha$ for $d_1 \in G_T$ and $\alpha = 8$ using (left) repeated multiplications and (right) power tree with squarings and multiplications. Red arrows indicate $G_T$ multiplications and green arrows indicate cheaper $G_T$ cyclotomic squarings.
Algorithm 6.5 outlines our fast lookup table construction using power tree. The number of squarings (resp. multiplications) is $\frac{\alpha}{2}$ (resp. $\frac{\alpha}{2} - 1$) and $\frac{\alpha - 1}{2}$ (resp. $\frac{\alpha - 1}{2}$) when $\alpha$ is even and odd respectively. On average, total computation cost of our power tree-based approach is $\approx 34\%$ lower than using repeated multiplications.

**Fast Lookup using Hash Table:** In Algorithm 6.4, apart from lookup table construction in line 2, another potentially time-consuming operation is the table lookup in line 5. Table lookup using brute force search requires retrieving and comparing all $\alpha$ elements of the table in the worst case. As suggested by [235], this can be improved by using hash tables [120]. Here, we provide our efficient hash table construction tailored for FHIPE and $G_T$ elements in BLS12-381.

For $hlen$-bit hashes, the minimum probability $p$ of encountering a collision upon hashing $N$ inputs is given by [28]:

$$p \approx 1 - e^{-N^2 / 2^hlen + 1}$$

In our case, $N = \alpha = \lceil \sqrt{s} \rceil$ as we will need to hash each element of the lookup table $\{1, d_1, \ldots, d_\alpha^{-1}\}$. The theoretically calculated collision probabilities corresponding to different values of $\alpha$ and $hlen$ are tabulated below:

<table>
<thead>
<tr>
<th>$hlen$</th>
<th>$\alpha = 16$</th>
<th>$\alpha = 32$</th>
<th>$\alpha = 64$</th>
<th>$\alpha = 128$</th>
<th>$\alpha = 256$</th>
<th>$\alpha = 512$</th>
<th>$\alpha = 1024$</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>0.78%</td>
<td>3.08%</td>
<td>11.75%</td>
<td>39.35%</td>
<td>86.47%</td>
<td>99.97%</td>
<td>100%</td>
</tr>
<tr>
<td>16</td>
<td>0.19%</td>
<td>0.78%</td>
<td>3.08%</td>
<td>11.75%</td>
<td>39.35%</td>
<td>86.47%</td>
<td>99.97%</td>
</tr>
<tr>
<td>18</td>
<td>0.05%</td>
<td>0.19%</td>
<td>0.78%</td>
<td>3.08%</td>
<td>11.75%</td>
<td>39.35%</td>
<td>86.47%</td>
</tr>
<tr>
<td>20</td>
<td>0.01%</td>
<td>0.05%</td>
<td>0.19%</td>
<td>0.78%</td>
<td>3.08%</td>
<td>11.75%</td>
<td>39.35%</td>
</tr>
<tr>
<td>22</td>
<td>&lt;0.01%</td>
<td>0.01%</td>
<td>0.05%</td>
<td>0.19%</td>
<td>0.78%</td>
<td>3.08%</td>
<td>11.75%</td>
</tr>
<tr>
<td>24</td>
<td>&lt;0.01%</td>
<td>&lt;0.01%</td>
<td>0.01%</td>
<td>0.05%</td>
<td>0.19%</td>
<td>0.78%</td>
<td>3.08%</td>
</tr>
<tr>
<td>26</td>
<td>&lt;0.01%</td>
<td>&lt;0.01%</td>
<td>&lt;0.01%</td>
<td>0.01%</td>
<td>0.05%</td>
<td>0.19%</td>
<td>0.78%</td>
</tr>
</tbody>
</table>

Depending on $\alpha$, we will choose $hlen$ so that collision probability is less than 1\%. In case collisions occur, chaining techniques [120] can be used with very little additional memory overhead. Size of the hash table is $2^hlen$, which is quite large (much larger than storing just the list of $\alpha$ elements in $G_T$, hence the time-memory trade-off). So, this method is suitable only for systems with ample storage available.
To speed up hashing, compressed form of the $G_T$ element is used as input. For $a = b_0 + b_1 \gamma + b_2 \gamma^2 + b_3 \gamma^3 + b_4 \gamma^4 + b_5 \gamma^5 \in G_T$ (where $b_0, \ldots, b_5 \in \mathbb{F}_{p^2}$), the elements $b_1$, $b_2$, $b_4$ and $b_5$ completely define $a \neq 1$ [236]. Therefore, we can simply hash $b_1 \| b_2 \| b_4 \| b_5$ instead of $b_0 \| b_1 \| b_2 \| b_3 \| b_4 \| b_5$ (where $\|$ denotes concatenation), thus reducing the hash input size by 33%. The trivial case of $a = 1$ can be handled separately. The CRC32 checksum is used for hashing $G_T$ elements followed by truncation to generate an index in the hash table as follows:

$$\text{table\_index} = \text{CRC32} ( b_1 \| b_2 \| b_4 \| b_5 ) \mod 2^\text{hlen}$$

For each $\alpha \in \{ 16, 32, 64, 128, 256, 512, 1024 \}$, we performed 1,000 random trials with $\text{hlen}$ that theoretically provides $\approx 1\%$ collision probability. The results are shown below, and agree very well with theoretical values. We rarely observed more than one collision during the same trial, thus confirming that chaining overhead is negligible.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{hlen}$</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>No. of collisions</td>
<td>8</td>
<td>8</td>
<td>11</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

**Constant-Time Loop:** In Algorithm 6.4, the number of loop iterations depends on the discrete logarithm result $z$. The number of table lookups and $G_T$ multiplications can be easily determined through timing or simple power analysis, thus leading to a side-channel. For privacy-preserving applications, it is desirable to keep $z$ secret to the decryptor (evaluator). For constant-time, we need to go through all $\alpha$ iterations of the loop irrespective of whether a matching table entry has been found (this prevents leaking $i = \lfloor z / \alpha \rfloor$) and returning $z$ only after all iterations are complete. If hash tables are not used, the brute-force table lookup should also go through all table entries irrespective of whether a match is found (this prevents leaking $j = z \mod \alpha$).

**Efficient FHIPE Decryption:** In Algorithm 6.6, we construct an efficient version of FHIPE Decrypt by combining our proposed tree-based fast lookup table generation technique (Algorithm 6.5) along with multi-pairing using shared Miller loop and final exponentiation (Section 5.2.3). Lines 6-13 implement a constant-time loop. In lines 9-10, we use dummy output calculation to prevent side-channel leakage.
Algorithm 6.6 Efficient constant-time FHIPE Decrypt using fast multi-pairing and tree-based lookup table construction (for BLS12-381)

**Require:** $k_1 \in G_1$, $c_1 \in G_2$, $k_2 = (k_{2,1}, \ldots, k_{2,n}) \in G_1^n$, $c_2 = (c_{2,1}, \ldots, c_{2,n}) \in G_2^n$, $S = \{0, 1, \ldots, s - 1\} \subset \mathbb{Z}_q$ and $\alpha = \lceil \sqrt{s} \rceil$

**Ensure:** $z \in S$ such that $e(k_2, c_2) = e(k_1, c_1)^z$

1: $d_1 \leftarrow e(k_1, c_1)$
2: $d_2 \leftarrow e(k_2, c_2) = e(k_{2,1}, c_{2,1}) \times \cdots \times e(k_{2,n}, c_{2,n})$
3: Generate table of pairs $(j, d_j^1)$ for $0 \leq j < \alpha$ and $d_1^\alpha$ using Algorithm 6.5
4: $t_0 \leftarrow d_1^{-\alpha}$
5: $t_1 \leftarrow d_2$
6: $z \leftarrow \bot$
7: for $(i = 0; i < \alpha; i = i + 1)$ do
8: if $t_1$ is the second component ($d_j^1$) of any pair in the table then
9: $z \leftarrow i\alpha + j$
10: else
11: $z_{\text{dummy}} \leftarrow i\alpha + j_{\text{dummy}}$
12: end if
13: $t_1 \leftarrow t_1 \cdot t_0$
14: end for
15: **return** $z$

Based on our Python reference implementation, the equivalent number of $F_p$ multiplications required in lines 1, 2, 3, 4 and 13 are $15,389M_1$ (pairing), $(10,571 + 4,860n)M_1$ ($n$-fold multi-pairing), $\approx 36\alpha M_1$ (power tree-based table construction), $705M_1$ ($F_{p^{12}}$ inversion) and $54M_1$ ($F_{p^{12}}$ multiplication) respectively. Therefore, the cost of Decrypt is $\approx (15,389 + 10,571 + 4,860n + 36\alpha + 705 + 54\alpha)M_1 = (26,665 + 4,860n + 90\alpha)M_1$ on average. In comparison, the baseline Decrypt from [183] requires $\approx 15,389(n+1) + 54(\alpha - 1) + 705 + 54\alpha)M_1 = (16,040 + 15,389n + 108\alpha)M_1$ on average (without fast multi-pairing and tree-based table construction; for fair comparison, we assume the baseline is constant-time and we still use the towered arithmetic costs from Appendix C), thus making our approach up to $3\times$ more efficient. Fig. 6-3 shows how computation cost varies with different $n$ (with $\alpha = 256 \Rightarrow s = 65,536$), both for the baseline and our proposed approach. We note that dependence on $\alpha$ is weak for reasonably sized $n$, that is, multi-pairing dominates over discrete logarithm in terms of arithmetic cost. On the other hand, discrete logarithm dominates memory cost. Hash table can be used for fast lookup in line 8, provided enough memory is available.
Software Implementation: We implement Decrypt with our optimized constant-time approach (Algorithm 6.6) using our software C library on the three platforms – (1) RISC-V RV32IM at 90 MHz, (2) ARM Cortex-M7 at 600 MHz and (4) Intel Cascade Lake at 2.4 GHz. Our software implementation results without and with hash table are shown in Tables 6.3 and 6.4 respectively. The measured execution times are reported for $n \in \{5, 10, 25, 50, 75, 100, 250, 500, 750, 1000\}$ and $\alpha \in \{16, 32, 64, 128, 256, 512, 1024\}$. On platforms (1) and (2), we profile decryption for the pairs of values $(n, \alpha)$ for which there is enough storage. The available data memory allows for profiling up to $\alpha = 64$ and $\alpha = 512$ without hash table on platforms (1) and (2) respectively, and up to $\alpha = 16$ and $\alpha = 32$ with hash table on platforms (1) and (2) respectively. Energy consumption for the RISC-V at 1.1 V is also reported.

For each $\alpha \in \{16, 32, 64, 128, 256, 512, 1024\}$, the hash output length $hlen$ (in bits) is chosen to have theoretical collision probability $\approx 1\%$. Then, the hash table size is $2^{hlen+1}$ bytes, where table entries are 2 bytes each. For implementation without hash table, the power table size is $576\alpha$ bytes, where $\mathbb{F}_p$ elements are stored in 384 bits. Note that we need to have enough data memory to store both the power table and the hash table when using hash table for discrete logarithm.
Table 6.3: BLS12-381 FHIPE Decrypt software evaluation results without hash table

(1) RISC-V RV32IM at 90 MHz

<table>
<thead>
<tr>
<th>Time</th>
<th>α</th>
<th>n</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>19.74 s</td>
<td>28.72 s</td>
<td>55.65 s</td>
<td>100.54 s</td>
<td>145.43 s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>32</td>
<td>20.30 s</td>
<td>29.28 s</td>
<td>56.21 s</td>
<td>101.10 s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>64</td>
<td>21.44 s</td>
<td>30.42 s</td>
<td>57.35 s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Energy</th>
<th>α</th>
<th>n</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>179 mJ</td>
<td>261 mJ</td>
<td>506 mJ</td>
<td>914 mJ</td>
<td>1.32 J</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>32</td>
<td>184 mJ</td>
<td>266 mJ</td>
<td>511 mJ</td>
<td>919 mJ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>64</td>
<td>195 mJ</td>
<td>276 mJ</td>
<td>521 mJ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(2) ARM Cortex-M7 at 600 MHz

<table>
<thead>
<tr>
<th>Time</th>
<th>α</th>
<th>n</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.14 s</td>
<td>1.78 s</td>
<td>3.14 s</td>
<td>5.65 s</td>
<td>8.15 s</td>
<td>11.67 s</td>
<td>25.79 s</td>
<td>51.27 s</td>
<td>76.53 s</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.17 s</td>
<td>1.81 s</td>
<td>3.18 s</td>
<td>5.68 s</td>
<td>8.18 s</td>
<td>11.71 s</td>
<td>25.82 s</td>
<td>51.31 s</td>
<td>76.56 s</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.25 s</td>
<td>1.88 s</td>
<td>3.25 s</td>
<td>5.75 s</td>
<td>8.25 s</td>
<td>11.78 s</td>
<td>25.90 s</td>
<td>51.38 s</td>
<td>76.64 s</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>1.40 s</td>
<td>2.03 s</td>
<td>3.40 s</td>
<td>5.90 s</td>
<td>8.41 s</td>
<td>11.93 s</td>
<td>26.05 s</td>
<td>51.53 s</td>
<td>76.79 s</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>1.73 s</td>
<td>2.37 s</td>
<td>3.74 s</td>
<td>6.24 s</td>
<td>8.74 s</td>
<td>12.27 s</td>
<td>26.38 s</td>
<td>51.87 s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>2.52 s</td>
<td>3.15 s</td>
<td>4.52 s</td>
<td>7.02 s</td>
<td>9.53 s</td>
<td>13.05 s</td>
<td>27.17 s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Energy</th>
<th>α</th>
<th>n</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>135 ms</td>
<td>195 ms</td>
<td>377 ms</td>
<td>678 ms</td>
<td>987 ms</td>
<td>1.30 s</td>
<td>3.09 s</td>
<td>6.22 s</td>
<td>9.23 s</td>
<td>12.37 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>139 ms</td>
<td>199 ms</td>
<td>381 ms</td>
<td>682 ms</td>
<td>991 ms</td>
<td>1.31 s</td>
<td>3.10 s</td>
<td>6.22 s</td>
<td>9.24 s</td>
<td>12.38 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>148 ms</td>
<td>208 ms</td>
<td>389 ms</td>
<td>690 ms</td>
<td>1.00 s</td>
<td>1.31 s</td>
<td>3.11 s</td>
<td>6.23 s</td>
<td>9.25 s</td>
<td>12.39 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>165 ms</td>
<td>225 ms</td>
<td>406 ms</td>
<td>708 ms</td>
<td>1.02 s</td>
<td>1.33 s</td>
<td>3.12 s</td>
<td>6.25 s</td>
<td>9.26 s</td>
<td>12.40 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>205 ms</td>
<td>265 ms</td>
<td>446 ms</td>
<td>748 ms</td>
<td>1.06 s</td>
<td>1.37 s</td>
<td>3.16 s</td>
<td>6.29 s</td>
<td>9.30 s</td>
<td>12.44 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>302 ms</td>
<td>362 ms</td>
<td>543 ms</td>
<td>845 ms</td>
<td>1.15 s</td>
<td>1.47 s</td>
<td>3.26 s</td>
<td>6.38 s</td>
<td>9.40 s</td>
<td>12.54 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>572 ms</td>
<td>632 ms</td>
<td>813 ms</td>
<td>1.11 s</td>
<td>1.42 s</td>
<td>1.74 s</td>
<td>3.53 s</td>
<td>6.65 s</td>
<td>9.67 s</td>
<td>12.81 s</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

(3) Intel Cascade Lake at 2.4 GHz

<table>
<thead>
<tr>
<th>Time</th>
<th>α</th>
<th>n</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>175 ms</td>
<td>245 ms</td>
<td>427 ms</td>
<td>728 ms</td>
<td>1.30 s</td>
<td>1.20 s</td>
<td>3.42 s</td>
<td>6.53 s</td>
<td>9.54 s</td>
<td>12.65 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>179 ms</td>
<td>253 ms</td>
<td>431 ms</td>
<td>732 ms</td>
<td>1.31 s</td>
<td>1.21 s</td>
<td>3.43 s</td>
<td>6.55 s</td>
<td>9.56 s</td>
<td>12.67 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>188 ms</td>
<td>260 ms</td>
<td>438 ms</td>
<td>738 ms</td>
<td>1.32 s</td>
<td>1.22 s</td>
<td>3.44 s</td>
<td>6.57 s</td>
<td>9.58 s</td>
<td>12.69 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>205 ms</td>
<td>275 ms</td>
<td>450 ms</td>
<td>750 ms</td>
<td>1.33 s</td>
<td>1.23 s</td>
<td>3.45 s</td>
<td>6.59 s</td>
<td>9.60 s</td>
<td>12.71 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>225 ms</td>
<td>295 ms</td>
<td>465 ms</td>
<td>765 ms</td>
<td>1.34 s</td>
<td>1.24 s</td>
<td>3.46 s</td>
<td>6.61 s</td>
<td>9.62 s</td>
<td>12.73 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>265 ms</td>
<td>332 ms</td>
<td>494 ms</td>
<td>794 ms</td>
<td>1.35 s</td>
<td>1.25 s</td>
<td>3.47 s</td>
<td>6.63 s</td>
<td>9.64 s</td>
<td>12.75 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>345 ms</td>
<td>389 ms</td>
<td>535 ms</td>
<td>835 ms</td>
<td>1.36 s</td>
<td>1.26 s</td>
<td>3.48 s</td>
<td>6.65 s</td>
<td>9.66 s</td>
<td>12.77 s</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

The Decrypt computation can be divided into three categories – pairing plus multi-pairing, lookup table construction and table-based discrete logarithm. From our software evaluation results, we observe that pairing and multi-pairing account for 20-90% of the total computation cost, while table construction and table-based discrete logarithm together account for the remaining 10-80%, depending on the relative values of \( n \) and \( \alpha \). Multi-pairing dominates the computation cost for very large \( n \). For small \( n \) and large \( \alpha \), the table construction and discrete logarithm computation account for a relatively large fraction of the total cost. Clearly, the benefit of using hash tables is marginal, especially for small \( \alpha \) or large \( n \). Except for extremely large \( \alpha \), hash tables should be avoided in order to save memory cost.
Table 6.4: BLS12-381 FHIPE Decrypt software evaluation results with hash table

(1) RISC-V RV32IM at 90 MHz

<table>
<thead>
<tr>
<th>Time</th>
<th>𝛼 𝑛</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>19.74 s</td>
<td>28.72 s</td>
<td>55.65 s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>168 mJ</td>
<td>244 mJ</td>
<td>473 mJ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(2) ARM Cortex-M7 at 600 MHz

<table>
<thead>
<tr>
<th>Time</th>
<th>𝛼 𝑛</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.14 s</td>
<td>1.78 s</td>
<td>3.14 s</td>
<td>5.65 s</td>
<td>8.15 s</td>
<td>11.67 s</td>
<td>25.79 s</td>
<td>51.27 s</td>
<td>76.53 s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.17 s</td>
<td>1.81 s</td>
<td>3.18 s</td>
<td>5.68 s</td>
<td>8.18 s</td>
<td>11.71 s</td>
<td>25.82 s</td>
<td>51.31 s</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

(3) Intel Cascade Lake at 2.4 GHz

<table>
<thead>
<tr>
<th>Time</th>
<th>𝛼 𝑛</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>135 ms</td>
<td>195 ms</td>
<td>377 ms</td>
<td>678 ms</td>
<td>987 ms</td>
<td>1.30 s</td>
<td>3.09 s</td>
<td>6.22 s</td>
<td>9.23 s</td>
<td>12.37 s</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>140 ms</td>
<td>200 ms</td>
<td>381 ms</td>
<td>682 ms</td>
<td>992 ms</td>
<td>1.31 s</td>
<td>3.10 s</td>
<td>6.22 s</td>
<td>9.24 s</td>
<td>12.38 s</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>148 ms</td>
<td>208 ms</td>
<td>389 ms</td>
<td>691 ms</td>
<td>1.00 s</td>
<td>1.32 s</td>
<td>3.11 s</td>
<td>6.23 s</td>
<td>9.25 s</td>
<td>12.39 s</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>166 ms</td>
<td>226 ms</td>
<td>407 ms</td>
<td>708 ms</td>
<td>1.02 s</td>
<td>1.33 s</td>
<td>3.12 s</td>
<td>6.25 s</td>
<td>9.26 s</td>
<td>12.40 s</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>205 ms</td>
<td>265 ms</td>
<td>446 ms</td>
<td>748 ms</td>
<td>1.06 s</td>
<td>1.37 s</td>
<td>3.16 s</td>
<td>6.29 s</td>
<td>9.30 s</td>
<td>12.44 s</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>293 ms</td>
<td>353 ms</td>
<td>534 ms</td>
<td>836 ms</td>
<td>1.14 s</td>
<td>1.46 s</td>
<td>3.25 s</td>
<td>6.37 s</td>
<td>9.39 s</td>
<td>12.53 s</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>542 ms</td>
<td>602 ms</td>
<td>783 ms</td>
<td>1.08 s</td>
<td>1.39 s</td>
<td>1.71 s</td>
<td>3.50 s</td>
<td>6.62 s</td>
<td>9.64 s</td>
<td>12.78 s</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: BLS12-381 FHIPE Decrypt hardware-software co-design results

without hash table

<table>
<thead>
<tr>
<th>Time</th>
<th>𝛼 𝑛</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>144 ms</td>
<td>205 ms</td>
<td>387 ms</td>
<td>692 ms</td>
<td>996 ms</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>166 ms</td>
<td>227 ms</td>
<td>410 ms</td>
<td>714 ms</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>235 ms</td>
<td>296 ms</td>
<td>478 ms</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Energy</th>
<th>𝛼 𝑛</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.22 mJ</td>
<td>1.74 mJ</td>
<td>3.29 mJ</td>
<td>5.88 mJ</td>
<td>8.47 mJ</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.41 mJ</td>
<td>1.93 mJ</td>
<td>3.48 mJ</td>
<td>6.07 mJ</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.99 mJ</td>
<td>2.52 mJ</td>
<td>4.06 mJ</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

with hash table

<table>
<thead>
<tr>
<th>Time</th>
<th>𝛼 𝑛</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>143 ms</td>
<td>203 ms</td>
<td>386 ms</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Energy</th>
<th>𝛼 𝑛</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.22 mJ</td>
<td>1.72 mJ</td>
<td>3.28 mJ</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Hardware-Accelerated Implementation: We implement Decrypt on the custom chip from Chapter 5 using hardware-software co-design, where all modular arithmetic and elliptic curve operations are accelerated using the crypto-processor, and the RISC-V is used to handle control flow, scheduling, data movement, hash table generation and iterative table lookups. The measured execution time and energy consumption (at 90 MHz and 1.1 V) are shown in Table 6.5. This is two orders of magnitude more efficient compared to software-only implementation on RISC-V (Tables 6.3 and 6.4).

6.2.4 Memory Cost

Table 6.6 shows the minimum data memory required to store the inputs, outputs, pre-computed tables and some intermediate values in FHIPE Encrypt and Decrypt using Algorithms 6.3 and 6.6 respectively. Additional memory (in the range of few KB) is also required to store temporary variables, depending on the implementation.

Table 6.6: Minimum memory requirement of BLS12-381 FHIPE implementation

<table>
<thead>
<tr>
<th>( n )</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>FHIPE Encrypt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 KB</td>
<td>9 KB</td>
<td>29 KB</td>
<td>94 KB</td>
<td>198 KB</td>
<td>341 KB</td>
<td>2 MB</td>
<td>8 MB</td>
<td>17 MB</td>
<td>31 MB</td>
<td></td>
</tr>
<tr>
<td>FHIPE Decrypt without hash table</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \alpha )</td>
<td>16</td>
<td>22</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13 KB</td>
<td>22 KB</td>
<td>40 KB</td>
<td>76 KB</td>
<td>148 KB</td>
<td>292 KB</td>
<td>580 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>15 KB</td>
<td>31 KB</td>
<td>42 KB</td>
<td>78 KB</td>
<td>157 KB</td>
<td>294 KB</td>
<td>582 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>22 KB</td>
<td>43 KB</td>
<td>49 KB</td>
<td>97 KB</td>
<td>169 KB</td>
<td>301 KB</td>
<td>589 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>34 KB</td>
<td>55 KB</td>
<td>61 KB</td>
<td>109 KB</td>
<td>181 KB</td>
<td>313 KB</td>
<td>601 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>46 KB</td>
<td>66 KB</td>
<td>73 KB</td>
<td>120 KB</td>
<td>192 KB</td>
<td>325 KB</td>
<td>613 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>57 KB</td>
<td>66 KB</td>
<td>84 KB</td>
<td>155 KB</td>
<td>263 KB</td>
<td>407 KB</td>
<td>624 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>128 KB</td>
<td>137 KB</td>
<td>155 KB</td>
<td>272 KB</td>
<td>380 KB</td>
<td>524 KB</td>
<td>812 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>245 KB</td>
<td>254 KB</td>
<td>272 KB</td>
<td>272 KB</td>
<td>380 KB</td>
<td>524 KB</td>
<td>812 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>750</td>
<td>362 KB</td>
<td>371 KB</td>
<td>389 KB</td>
<td>425 KB</td>
<td>542 KB</td>
<td>641 KB</td>
<td>929 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>479 KB</td>
<td>488 KB</td>
<td>506 KB</td>
<td>542 KB</td>
<td>542 KB</td>
<td>758 KB</td>
<td>1 MB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| \( \alpha \) | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | | | |
| 5 | 45 KB | 552 KB | 552 KB | 2 MB | 8 MB | 32 MB | 129 MB | | | |
| 10 | 47 KB | 554 KB | 561 KB | 2 MB | 8 MB | 32 MB | 129 MB | | | |
| 25 | 54 KB | 573 KB | 585 KB | 2 MB | 8 MB | 32 MB | 129 MB | | | |
| 50 | 66 KB | 585 KB | 596 KB | 2 MB | 8 MB | 32 MB | 129 MB | | | |
| 75 | 78 KB | 596 KB | 667 KB | 2 MB | 8 MB | 32 MB | 129 MB | | | |
| 100 | 89 KB | 667 KB | 784 KB | 2 MB | 8 MB | 32 MB | 129 MB | | | |
| 250 | 160 KB | 784 KB | 901 KB | 2 MB | 8 MB | 32 MB | 129 MB | | | |
| 500 | 277 KB | 784 KB | 1018 KB | 2 MB | 8 MB | 32 MB | 129 MB | | | |
| 750 | 394 KB | 901 KB | | 2 MB | | | | | | |
| 1000 | 511 KB | 1018 KB | | | | | | | |
For FHIPE Encrypt, while $G_2$ ECSMs dominate computation cost, majority of the memory requirement is due to storing the $n \times n$ matrix $B^* \in \mathbb{GL}_n(\mathbb{Z}_q)$. To solve this problem, we may derive inspiration from lattice-based cryptography, where the storage overhead due to large matrix-vector multiplications is reduced by deterministically sampling elements of the matrix from a seed every time instead of one-time sampling from the same seed and storing the entire matrix. Similar approach may be followed to sample $B^*$ on-the-fly one column at a time, thus reducing memory cost from $O(n^2)$ to $O(n)$. The associated computational overhead is less than 1%. However, this would also require re-evaluating the security proof due to changes in the Setup and KeyGen phases (we need $B = det(B^*) \cdot (B^*^{-1})^T$ and $k_1 = \alpha \cdot det(B^*)G_1$ for correctness).

### 6.2.5 Communication Cost

Table 6.7 shows the sizes of ciphertexts (with $n + 1$ points each in $G_2$) for different vector sizes $n$. Apart from cryptographic computations, wireless communications also account for a significant fraction of the energy consumption on embedded devices. The communication cost of FHIPE is directly related to the ciphertext size and it can be halved by using elliptic curve point compression [190]. For point compression, only the $x$-coordinate is transmitted ($x \in \mathbb{F}_{p^2}$ for $G_2$ point). Decompression requires evaluating the $y$-coordinate as $y = \pm \sqrt{x^3 + 4(1 + \alpha)} \in \mathbb{F}_{p^2}$. One additional bit is transmitted along with the $x$-coordinate to indicate whether to select the positive or the negative value of $y$ [190]. Since the order of extension field $\mathbb{F}_{p^2}$ is $p^2 \equiv 9 \mod 16$, the square root can be computed using a specialized version of Tonelli-Shanks algorithm [188].

Uncompressed and compressed $G_2$ points are 192 bytes and 96 bytes respectively. The computation cost of decompressing a $G_2$ point is $367M_2 + 761S_2 + A_2 + A_1 \equiv 2623M_1 + 4121A_1$. Depending on the vector size $n$, point decompression adds 25-55% overhead to the decryption cost.

<table>
<thead>
<tr>
<th>$n$</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (KB)</td>
<td>1.12</td>
<td>2.06</td>
<td>4.88</td>
<td>9.56</td>
<td>14.25</td>
<td>18.94</td>
<td>47.06</td>
<td>93.94</td>
<td>140.81</td>
<td>187.69</td>
</tr>
</tbody>
</table>
6.3 Privacy-Preserving Computation

Here, we discuss examples of privacy-preserving computation that can be performed using pairing-based function-hiding inner product encryption. We consider two applications – (1) biomedical sensor data classification and (2) wireless fingerprint-based indoor localization, and describe how these computations are mapped to FHIPE.

Fig. 6-4 shows the system diagram for a typical application scenario, with an IoT device encrypting private data $y$ using FHIPE Encrypt and a cloud server evaluating the inner product $\langle x, y \rangle$ using FHIPE Decrypt with a key embedding $x$. We provide implementation results based on the software C library and hardware-software co-design from Section 6.2. It can be improved further using assembly optimizations, to be explored in future work.

6.3.1 Biomedical Sensor Data Classification

We consider linear classification of data [237] with input feature vector $x$. For weight vector $w$, the input is classified to one of two classes depending on whether the inner
product $\langle \mathbf{x}, \mathbf{w} \rangle$ is above or below a threshold $T$. This is equivalent to dividing a high-dimensional input space into two parts with a hyperplane.

For classification tasks outsourced to a server, it is important to protect the input data as well as the pre-trained classifier weights. Maintaining data confidentiality ensures the client’s privacy, while hiding the weights preserves intellectual property. In several applications, e.g., medical data classification, the classifier weights are obtained through training over sensitive data, so there may also be a privacy requirement in hiding the weights. Next, we discuss how this can be achieved using FHIPE.

In the FHIPE setting, the encryptor sends $ct_x = \text{Encrypt} (msk, \mathbf{x})$ and the decryptor evaluates $z = \langle \mathbf{x}, \mathbf{w} \rangle = \text{Decrypt} (pp, sk_w, ct_x)$, where $sk_w = \text{KeyGen} (msk, \mathbf{w})$ is the decryption key embedding (hiding) the classifier weights. The decryptor then classifies the encrypted data to class $C_0$ or $C_1$ depending on whether $z \leq T$ and $z > T$ respectively. The decryptor (evaluator) is assumed to have prior knowledge of the classification threshold $T$. Both the input vector $\mathbf{x}$ and the weight vector $\mathbf{w}$ remain hidden to the decryptor. The FHIPE scheme inherently supports both positive and negative vector elements. While we have previously discussed evaluating non-negative inner product $z$ using discrete logarithm, the baby-step giant-step algorithm can be very easily adjusted to support both negative and positive exponents.

As a practical example of such linear data classification, we consider heartbeat categorization using the open-source PTB Diagnostic ECG Database [238] from PhysioNet [239, 240]. In particular, we use the post-processed and segmented version of this database, with each segment containing one heartbeat, available on Kaggle [241] (we need to correct an alignment error in the post-processed data before using it for training and validating the classifier). It consists of 14,552 electrocardiogram (ECG) samples at 125 Hz sampling frequency, and each sample is a 188-dimensional vector of floating point numbers. Out of these, 4,046 samples correspond to normal heartbeats (class $C_0$), while the remaining 10,506 are abnormal heartbeats affected by arrhythmia and myocardial infarction (class $C_1$). We scale and quantize the dataset from 64-bit floating point to 4-bit unsigned integers. Fig. 6-5 shows two randomly chosen samples, one from each class, before and after quantization. For training and cross-validating
the classifier, 500 samples from each class are used as the validation set while the remaining samples are used as the training set.

We train a single-layer perceptron neural network over this dataset using gradient descent [242], with 79% training accuracy and 72% validation accuracy. The weights are all 10-bit signed integers. The perceptron bias is negated to obtain the linear classifier threshold $T$ (an 8-bit positive integer). For FHIPE with this classifier, we have $n = 188$ and $\alpha = \sqrt{188 \times 2^4 \times 2^{10}} \approx 1755$. The encryption and decryption times are 1.11 s and 3.60 s respectively on Intel Cascade Lake at 2.4 GHz (hash table used for decryption). Due to memory limitations, we are unable to implement both encryption and decryption on the RISC-V RV32IM and ARM Cortex-M7 platforms. Although storage requirement can be reduced by downsampling the dataset to smaller $n$, we do not follow this because it drastically lowers classification accuracy.

![Figure 6-5: Quantization of normal and abnormal ECG samples from [239].](image)

(a) before quantization (64-bit floating point)

(b) after quantization (4-bit unsigned integer)
As another practical example, we consider electroencephalogram (EEG) classification using the open-source Epileptic Seizure Recognition Data Set [243] from UCI Machine Learning Repository [244]. It consists of 11,500 samples, and each sample is a 178-dimensional vector of quantized 12-bit signed integers corresponding to 1 second duration. Out of these, 9,200 samples correspond to normal recordings (class $C_0$), while the remaining 2,300 are recordings of seizure activity (class $C_1$). For training and cross-validating the classifier, 250 samples from each class are used as the validation set while the remaining samples are used as the training set.

Once again, we train a single-layer perceptron neural network over this dataset using gradient descent, with 78% training accuracy and 77% validation accuracy. The weights are all 9-bit signed integers. The perceptron bias is negated to obtain the
linear classifier threshold $T$ (a 14-bit positive integer). For FHIPE with this classifier, we have $n = 178$ and $\alpha = \sqrt{178 \times 2^{12} \times 2^9} \approx 19321$. The encryption and decryption times are 1.05 s and 3.76 s respectively on Intel Cascade Lake at 2.4 GHz (hash table used for decryption). Due to memory limitations, we are unable to implement both encryption and decryption on RISC-V RV32IM and ARM Cortex-M7 platforms.

Unlike the ECG classifier, we observe that this EEG classifier can be downscaled up to $10 \times$ without affecting classification accuracy. Fig. 6-6 shows two randomly chosen samples, one from each class, before and after downsampling. We train another similar perceptron neural network over this downsampled dataset, with 78% training accuracy and 79% validation accuracy. The weights are all 5-bit signed integers. The perceptron bias is negated to obtain the linear classifier threshold $T$ (an 11-bit positive integer). For FHIPE with this classifier, we have $n = 18$ and $\alpha = \sqrt{18 \times 2^{12} \times 2^5} \approx 1536$. The encryption time is 27.07 s, 1.68 s and 103 ms respectively on RISC-V RV32IM at 90 MHz, ARM Cortex-M7 at 600 MHz and Intel Cascade Lake at 2.4 GHz respectively. With hardware-software co-design using our pairing crypto-processor and RISC-V RV32IM together at 90 MHz, we require 188 ms for encryption. The decryption time is 1.99 s on Intel Cascade Lake at 2.4 GHz (with hash table). Due to memory limitations, we are unable to implement decryption on RISC-V RV32IM and ARM Cortex-M7 platforms.

We have explored only data classification in this work. Privacy-preserving training of such classifiers may also be implemented in the context of FHIPE. Other linear classifiers, such as logistic regression and support vector machine [237], are also suitable. Classification tasks such as cardiovascular disease, city traffic congestion and digit recognition, discussed by [245] in the context of generic inner product function encryption, also fit well in the context of FHIPE. A privacy-preserving quadratic classifier was constructed by [246] for handwritten digit recognition using a similar pairing-based scheme, but it is not function-hiding. It was pointed out by [247] that data classification using non-function-hiding inner product functional encryption schemes may leak information about the encrypted input vector since the decryptor may compute inner products with several chosen weight vectors and solve a linear
system of equations to determine a vector statistically close to the input. The FHIPE scheme is considered secure against such leakage, thus making it a very attractive choice for data classification with both encrypted inputs and hidden weights.

### 6.3.2 Wireless Fingerprint-Based Indoor Localization

Indoor localization systems use wireless networks to determine approximate locations of people, objects or electronic devices in areas where traditional satellite-based positioning technologies either have low precision or do not work at all, e.g., inside buildings, airports, parking garages, alleyways and underground. One of the most popular indoor localization techniques is based on WiFi fingerprints [248,249] due to the availability of WiFi access points in such indoor locations.

Let us consider an area with $N$ WiFi access points, each with a unique public identifier $AP_j$ for $1 \leq j \leq N$, e.g., MAC addresses. At any location $(x_i, y_i)$, let the RSSI (Received Signal Strength Indicator) values corresponding to these $N$ access points be $v_i = (v_{i,1}, v_{i,2}, \cdots, v_{i,N})$, which act as wireless fingerprints. In the setup phase, the service provider collects and stores $(x_i, y_i)$ and $v_i$ for $M$ locations of interest into a database: $D = \langle i, (x_i, y_i), v_i \rangle^M_{i=1}$. The service provider stores this database in their server and publishes the list of access point identifiers $T_{AP} = \{AP_j\}^N_{j=1}$. In the operating phase, a client measures the wireless fingerprint $v = (v_1, v_2, \cdots, v_N)$ at its location and sends it to the server. Then, the server computes squared Euclidean distances $d_i$ between $v$ and each $v_i$ stored in its database, where:

$$d_i = \|v - v_i\|^2 = \sum_{j=1}^{N} (v_j - v_{i,j})^2 = \sum_{j=1}^{N} v_j^2 + \sum_{j=1}^{N} (-2 v_j v_{i,j}) + \sum_{j=1}^{N} v_{i,j}^2$$

The $k$ nearest neighbors of the client (from the database) are determined as the locations $\{(x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2}), \cdots, (x_{i_k}, y_{i_k})\}$ (where $1 \leq i_1, i_2, \cdots, i_k \leq M$) with smallest distances $d_i$. For $k = 1$, this gives the nearest location $(x_{i_1}, y_{i_1})$. For $k > 1$, the client’s approximate location may be estimated by computing the centroid of these nearest neighbors as: $(x_c, y_c) = (\frac{1}{k} \sum_{l=1}^{k} x_{i_l}, \frac{1}{k} \sum_{l=1}^{k} y_{i_l})$. Finally, the server either sends this information back to the client or uses it to perform a specific action.
From a privacy and security perspective, it is important to keep the client’s wireless fingerprint and estimated location confidential. It is also equally important to keep the service provider’s database private. We discuss how this can be achieved in the FHIPE setting. The service provider creates $M$ decryption keys corresponding to its database entries as $sk_{v'_i} = \text{KeyGen}(msk, v'_i = (\sum_{j=1}^{N} v_{i,j}^2, -2v_{i,1}, -2v_{i,2}, \cdots, -2v_{i,N}, 1))$ for $1 \leq i \leq M$, and shares it with the server. The client sends its encrypted fingerprint $ct_{v'} = \text{Encrypt}(msk, v' = (1, v_1, v_2, \cdots, v_N, \sum_{j=1}^{N} v_j^2))$ to the server. The server then computes all $M$ distance metrics $d_i = \langle v', v'_i \rangle = \text{Decrypt}(pp, sk_{v'_i}, ct_{v'})$ for $1 \leq i \leq M$. Then, it identifies location indices $1 \leq i_1, i_2, \cdots, i_k \leq M$ of the client’s $k$ nearest neighbors and acts based on this information (as specified by the service provider), e.g., sends messages, recommendations, ads, etc to the client. Neither the service provider’s database nor the client’s fingerprint is revealed in this process. The coordinates corresponding to location indices $1 \leq i \leq M$ (including the nearest neighbors $i_1, i_2, \cdots, i_k$) are kept secret throughout this process.

As an example, we simulate FHIPE-based privacy-preserving indoor localization using the WiFi heat map generated by an open-source tool [250]. Our example scenario with simulated heat map and layout of $N = 4$ access points and $M = 9$ database locations is shown in Fig. 6-7. The service provider’s database entries of RSSI values (in dBm) corresponding to each access point at all these locations are obtained using the simulated WiFi heat map, as tabulated below:

<table>
<thead>
<tr>
<th></th>
<th>$i = 1$</th>
<th>$i = 2$</th>
<th>$i = 3$</th>
<th>$i = 4$</th>
<th>$i = 5$</th>
<th>$i = 6$</th>
<th>$i = 7$</th>
<th>$i = 8$</th>
<th>$i = 9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_{i,1}$</td>
<td>-38</td>
<td>-42</td>
<td>-46</td>
<td>-42</td>
<td>-45</td>
<td>-47</td>
<td>-46</td>
<td>-47</td>
<td>-48</td>
</tr>
<tr>
<td>$v_{i,2}$</td>
<td>-46</td>
<td>-42</td>
<td>-38</td>
<td>-47</td>
<td>-45</td>
<td>-42</td>
<td>-48</td>
<td>-47</td>
<td>-46</td>
</tr>
<tr>
<td>$v_{i,3}$</td>
<td>-46</td>
<td>-47</td>
<td>-48</td>
<td>-42</td>
<td>-45</td>
<td>-47</td>
<td>-38</td>
<td>-42</td>
<td>-46</td>
</tr>
<tr>
<td>$v_{i,4}$</td>
<td>-48</td>
<td>-47</td>
<td>-46</td>
<td>-47</td>
<td>-45</td>
<td>-42</td>
<td>-46</td>
<td>-42</td>
<td>-38</td>
</tr>
<tr>
<td>$d_i$</td>
<td>164</td>
<td>102</td>
<td>108</td>
<td>102</td>
<td>40</td>
<td>22</td>
<td>108</td>
<td>22</td>
<td>4</td>
</tr>
</tbody>
</table>
Figure 6-7: Example indoor localization scenario and simulated WiFi heat map with $N = 4$ access points \{AP_1, \ldots, AP_4\} and $M = 9$ database locations \{L_1, \ldots, L_9\}.

We assume that the client is located at bottom left quadrant of the area of interest, as shown by the green icon in Fig. 6-7. The client’s simulated wireless fingerprint is $\mathbf{v} = (-47, -45, -45, -39)$. The computed distances $d_i$ are also shown in the same table. As expected, the client’s nearest neighbor is obtained as $L_9$, and the next nearest neighbors are $L_6$ and $L_8$. The vector size for each of the FHIPE decryption keys and encrypted fingerprint is $n = N + 2 = 6$. Since all RSSI values are negative, they can be encoded as 6-bit unsigned integers (after ignoring signs) for key generation and encryption. Since the RSSI values lie between $v_{\text{min}} = -55$ and $v_{\text{max}} = -30$, we have $\alpha = \sqrt{N \times (v_{\text{max}} - v_{\text{min}})^2} = \sqrt{4 \times (55 - 30)^2} = 50$. The encryption time is 9.02 s, 561 ms and 35 ms respectively on RISC-V RV32IM at 90 MHz, ARM Cortex-M7 at 600 MHz and Intel Cascade Lake at 2.4 GHz respectively. The decryption time is 22.51 s, 1.31 s and 155 ms respectively on these platforms (hash table not used for decryption). With hardware-software co-design using our pairing crypto-processor and RISC-V RV32IM together at 90 MHz, we require 62 ms and 247 ms for encryption and decryption respectively. For localization with more access points, it may be useful to consider RSSI with higher precision (up to one place of decimal) at the cost of increased memory requirement (due to higher $\alpha$).
6.4 Summary and Contributions

In this chapter, we have presented algorithm optimizations and efficient software and hardware-accelerated implementation results for BLS12-381 pairing-based function-hiding inner product encryption (FHIPE). Fast elliptic curve scalar multiplication using skew Frobenius map, scalar decomposition and comb pre-computation enables $3.5 \times$ reduction in encryption cost. Fast multi-pairing with shared Miller loop and final exponentiation along with power tree-based table construction for discrete logarithm together lead to $3 \times$ speedup in decryption. While previous work has explored FHIPE implementation only on server-scale computing platforms, we have also performed extensive profiling of encryption and decryption with different parameters on embedded-scale platforms. Our results demonstrate that FHIPE can be realized on embedded systems, including resource-constrained devices, with some limitations on the FHIPE parameters as well as memory constraints. Our results show that dedicated hardware accelerators, such as the pairing crypto-processor from Chapter 5, can provide orders of magnitude gains in performance and energy-efficiency of FHIPE implemented on embedded devices. We discuss two practical applications of privacy-preserving computation based on FHIPE – biomedical sensor data classification and wireless fingerprint-based indoor localization. While secure computation is usually considered to be too computationally expensive for embedded devices [217], our results confirm that optimized algorithms and hardware acceleration together make pairing-based inner product functional encryption a very attractive option for privacy-preserving computation applications in low-power IoT systems.
Chapter 7

Conclusions and Future Directions

7.1 Summary and Conclusions

With unprecedented growth in the number of wireless-connected embedded devices, there are also increasing security concerns and demand for efficient implementations of cryptographic primitives. While advances in semiconductor technology and integrated circuit innovations have led to the design of extremely powerful micro-processors in modern times, these general-purpose computing systems are still not capable of efficiently handling cryptographic tasks. This is especially true for public key cryptography which often involves multi-precision arithmetic, polynomial manipulations and reduction with large prime moduli. As pointed out by some of the early works in this domain [251,252], specialized hardware is required to accelerate such complex cryptographic functions with low energy consumption and high performance. This has been the inspiration and motivation for our work.

Along with advances in circuits, architectures and semiconductor technologies, we have also witnessed the development and widespread adoption of many new cryptographic algorithms over the past decade. Elliptic curve cryptography (ECC) [32,36] is now the standard public key primitive for key exchange, authentication and digital signatures with significantly smaller key sizes compared to their predecessors based on integer factorization and finite field discrete logarithms. Along with standard encryption algorithms, e.g., AES [29,51] and hash functions, e.g., SHA [30,31], standard
security protocols such as Transport Layer Security (TLS) [33, 44] use ECC-based authenticated key exchange to secure our daily Internet communications. Elliptic curves supporting special bilinear pairing maps, also known as pairing-based cryptography (PBC) [37, 177, 181], are now used to enable new primitives such as signature aggregation, identity-based encryption, attribute-based encryption and functional encryption. Most recently, we have also seen the rise of post-quantum cryptography which is considered to be secure against future quantum adversaries [94–96]. This includes several primitives such as lattices [82, 83, 97], isogenies [165], etc. Among these, lattice-based cryptography (LBC) [114–118] is widely considered to be the most promising due to its efficiency and extensive security analysis. In the near future, we expect to see network protocols using a combination of several such cryptographic tools to secure their communications and to enable novel security applications. Furthermore, with the proliferation of embedded devices in Internet of Things (IoT) networks, implementations of these cryptographic algorithms need to be energy-efficient and protected against physical attacks.

In this work, we have demonstrated that we can indeed realize next-generation sophisticated public key cryptography algorithms in embedded systems with low energy consumption and reduced design cost. Our design of custom hardware accelerators for elliptic curve cryptography, lattice-based cryptography and pairing-based cryptography achieve orders of magnitude improvement in energy-efficiency and performance compared to state-of-the-art software and hardware implementations. These accelerators are also integrated with a low-power embedded micro-processor to demonstrate a wide variety of security protocols using efficient hardware-software co-design. Several algorithm-level countermeasures have also been implemented to protect our cryptographic hardware from common timing and power side-channel attacks. Our designs have been fabricated in the form of test chips in 65nm and 40nm low-power CMOS processes. Functionality of the cryptographic accelerators are verified using test vectors, power consumption measured experimentally and side-channel countermeasures validated using standard statistical tests. The test chips are also integrated into custom-built printed circuit boards for system-level demonstration. To achieve our
design objectives, we have used several circuit, architecture and algorithm techniques:

- **Elliptic Curve Cryptography:** Wide data-path adders are used to reduce control circuitry in the modular arithmetic unit, leading to reduced latency and lower energy consumption. Data gating is used to save energy when operating over smaller prime fields. Memory-time trade-offs, such as windowing and comb pre-computations, are used to speed up elliptic curve scalar multiplications, which directly benefits the TLS authentication handshakes. This is further improved by suitable choice of coordinate representation along with the design of a dedicated modular inverter, which is an area-energy trade-off.

- **Pairing-Based Cryptography:** For pairing computations, the word size for Montgomery modular arithmetic is chosen strategically to provide the right balance between energy consumption, latency and area. Karatsuba-style divide-and-conquer techniques are used for efficient extension field arithmetic. Several components of the pairing algorithm are shared to provide energy savings, while special properties of the pairing-friendly elliptic curve are used to speed up pairing-based cryptography protocols.

- **Lattice-Based Cryptography:** Low-power modular arithmetic is implemented using Barrett reduction, which also allows the configurability to support multiple prime fields. Parallel data-path architectures for pseudo-random number generation using symmetric primitives, such as block ciphers and hash functions, along with optimized post-processing algorithms are used to design energy-efficient discrete distribution samplers. A single port memory-based number theoretic transform architecture is used to provide area savings.

Custom clock gating of both logic and memory modules is used extensively for power savings. The clock gates are designed to activate automatically based on the cryptographic function under execution. Voltage scaling is used to demonstrate energy-performance trade-offs. Our designs also support the compression of public keys and ciphertexts to reduce communication overheads at the cost of increased computation, depending on the application requirements.
This work demonstrates that domain-specific hardware is critical to enabling computationally expensive cryptographic algorithms and novel security applications on embedded devices. While an efficient algorithm may have great asymptotic complexity, the constant factors become important when realizing them in software and hardware. Therefore, algorithm-architecture co-optimizations must be performed for efficient implementation – circuits and architectures are equally important to accelerate efficient algorithms. These designs must not only be efficient but also side-channel-secure, with countermeasures implemented at the circuit, architecture and/or algorithm levels. Along with computation cost, it is also important to reduce communication overheads by optimizing at the network protocol level. Finally, it is desirable for the designs to have flexibility in order to support a variety of cryptographic primitives and easily update implementations on-the-fly.

7.2 Future Directions

There are many exciting new directions in the field of cryptography and hardware security. Here are some possible extensions of this work:

- **More Post-Quantum**: The architectural techniques discussed in this work can be applied to many other public key algorithms including isogeny-based cryptography, code-based cryptography and also lattice-based cryptography using learning with rounding and NTRU. Since several among these algorithms are expected to be standardized by NIST, efficient and side-channel-secure hardware implementations will be critical to enable commercial adoption.

- **More Lattices**: Lattice-based homomorphic encryption and functional encryption also use number theoretic transform and discrete distribution sampling, but with very different parameters compared to post-quantum lattice-based key encapsulation and signatures. It will be interesting to see if some of the design optimizations in our lattice crypto-processor may be used to accelerate other lattice-based primitives and enable efficient computations on encrypted data.
• **Physical Attacks**: While we have implemented only algorithm-level side-channel countermeasures, it will be interesting to explore circuit-level and architecture-level defenses. More sophisticated and invasive physical attacks, e.g., fault injection attacks, should be investigated. The use of machine learning techniques for side-channel analysis should also be explored.

• **Circuit Primitives**: It will be interesting to see how on-chip sources of randomness such as true random number generators (TRNGs) and physically unclonable functions (PUFs) can be efficiently integrated with such cryptographic hardware accelerators. This is especially important for applications such as smart cards, identification tags and biomedical devices.

• **New Technologies**: While all our designs are fabricated in silicon technology, it will be very useful to explore implementations using novel technologies such as carbon nanotubes and 2D materials.

• **Optimized Software**: Assembly-optimized software can be used to further speed up the function-hiding inner product encryption scheme on the server side.

• **Formal Verification**: Since these cryptographic accelerators implement very complex algorithms, it will be important to develop a robust formal verification framework to validate them.
Appendix A

List of Abbreviations

- NIST National Institute of Standards and Technology
- IETF Internet Engineering Task Force
- TLS Transport Layer Security
- DTLS Datagram Transport Layer Security
- AES Advanced Encryption Standard
- SHA Secure Hash Algorithm
- RSA Rivest-Shamir-Adelman
- ECC Elliptic Curve Cryptography
- PBC Pairing-Based Cryptography
- PQC Post-Quantum Cryptography
- LBC Lattice-Based Cryptography
- PKE Public Key Encryption
- KEM Key Encapsulation Mechanism
- IBE Identity-Based Encryption
- SPA Simple Power Analysis
- DPA Differential Power Analysis
- CPA Correlation Power Analysis
- RAM Random Access Memory
- ASIC Application-Specific Integrated Circuit
- FPGA Field-Programmable Gate Array
- CMOS Complementary Metal Oxide Semiconductor
- SRAM Static Random Access Memory
- IoT Internet of Things
Appendix B

Mathematical Preliminaries

The work discussed in this thesis requires some mathematical background to define the technical terms and theoretical concepts. In particular, abstract algebraic structures, such as groups, rings and fields, are integral to the construction of all the cryptographic schemes discussed in this work. Furthermore, the security of these cryptographic algorithms is based on some well-known computational hardness assumptions. In this Appendix, we provide a quick background on both of these aspects. For further details, please refer to [28].

Groups, Rings and Fields: The most important algebraic constructs used in cryptography are finite groups, rings and fields. They all consist of non-empty sets of a finite number of elements along with one or two operations which can be used to generate one or more elements of the set from another.

A group \((G, \cdot)\) is comprised of a non-empty set \(G\) of elements along with a binary operator \(\cdot\) satisfying the following properties:

1. the group operation is associative, that is, \(a \cdot (b \cdot c) = (a \cdot b) \cdot c \forall a, b, c \in G\)
2. there exists an identity element \(e \in G\) such that \(a \cdot e = e \cdot a = a \forall a \in G\)
3. for each \(a \in G\) there exists an element \(a^{-1} \in G\), known as the inverse of \(a\), such that \(a \cdot a^{-1} = a^{-1} \cdot a = e\)

A group \(G\) is abelian if it is commutative, that is, \(a \cdot b = b \cdot a \forall a, b \in G\). A group \(G\) is finite if \(|G|\) is finite. The number of elements in a finite group is called its order. A
group $G$ is cyclic if there exists an element $g \in G$ such that any element $a \in G$ can be expressed as $a = g^i$, where $i$ is an integer and $g^i$ is computed as

$$g^i = g \cdot g \cdot \cdots \cdot g$$

The order of an element $a \in G$ is the least positive integer $t$ such that $a^t = e$, if such an integer exists. If no such integer exists, then the order of that element is $\infty$.

A ring $(R, +, \times)$ is comprised of a non-empty set $R$ of elements along with two binary operators $+$ and $\times$ (often referred to as “addition” and “multiplication” respectively) satisfying the following properties:

1. $(R, +)$ is an abelian group with additive identity element denoted as $0$
2. the operation $\times$ is associative, that is, $a \times (b \times c) = (a \times b) \times c$ $\forall$ $a, b, c \in R$
3. there exists a multiplicative identity, denoted as $1$ ($\neq 0$), such that $1 \times a = a \times 1 = a \forall a \in R$
4. the operation $\times$ is distributive over $+$, that is, $a \times (b + c) = (a \times b) + (a \times c)$ and $(b + c) \times a = (b \times a) + (c \times a)$ $\forall$ $a, b, c \in R$

A ring $R$ is commutative if $a \times b = b \times a$ $\forall$ $a, b \in R$. A ring $R$ is finite if $|R|$ is finite.

A field $(F, +, \times)$ is a commutative ring where all non-zero elements have multiplicative inverses. A field $F$ is finite if $|F|$ is finite. The number of elements in a finite field is called its order. It can be proven that a finite field $F$ always contains $p^m$ elements, where $p$ is a prime and $m \geq 1$ is an integer. This field is also referred to as a Galois field, denoted by $\mathbb{F}_{p^m}$ or $GF(p^m)$. The characteristic of field $\mathbb{F}_q$, of order $q = p^m$, is $p$. It can also be proven that the set of integers modulo a prime $p$ is a finite field of characteristic $p$, denoted by $\mathbb{Z}_p$.

For commutative ring $R$, a polynomial in $x$ over $R$ can be written as $f(x) = a_m x^m + \cdots + a_2 x^2 + a_1 x + a_0$, where $a_0, a_1, \cdots, a_m \in R$ and $m \geq 0$. The set of all such polynomials in $x$ with coefficients from $R$ form a ring, known as the polynomial ring and denoted by $R[x]$. The two ring operations are polynomial addition and multiplication with coefficient arithmetic in $R$. 

172
Let $F[x]$ be the ring of polynomials in $x$ with coefficients in a field $F$. The set of polynomials in $F[x]$ of degree less than $\deg(f(x))$ forms a commutative ring denoted by $F[x]/f(x)$. Polynomial addition and multiplication in this ring is performed modulo $f(x)$. A polynomial $f(x) \in F[x]$ (of degree $\geq 1$) is irreducible over $F$ if it cannot be expressed as the product of two polynomials in $F[x]$. If $f(x)$ is irreducible over $F$, then $F[x]/f(x)$ is a field. In particular, the field $\mathbb{F}_p[x]/f(x)$ is referred to as an extension field of $\mathbb{F}_p$, also denoted by $\mathbb{F}_{p^m}$, with its elements being polynomials of degree less than $m = \deg(f(x))$ with coefficients in $\mathbb{F}_p$. The order of this field is $p^m$.

**Computational Hardness Assumptions:** The security of many cryptographic protocols is based on the intractability of several well-studied computational problems. A computational problem, with carefully chosen parameters, is considered intractable if it cannot be solved in polynomial time for a non-negligible fraction of all of its possible inputs. Although there are no concrete proofs of intractability, the computational hardness assumptions in modern cryptosystems are based on extensive complexity analysis of the current state-of-the-art algorithms which attempt to solve these problems. Here, we briefly discuss two such instances which were used to construct some of the earliest known cryptosystems and continue to be used today – the integer factorization problem and the discrete logarithm problem.

The **integer factorization problem** states the following: given a positive integer $n$, it is difficult to find its prime factorization $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, where the $p_i$ are distinct primes and each $e_i \geq 1$. Currently, the fastest algorithm which can solve this problem using classical computing is the general number field sieve (GNFS) [253] with time complexity $L_n[1/3, (\sqrt[3]{64/9})] = exp\left((\sqrt[3]{64/9} + o(1))(\ln n)^{1/3}(\ln \ln n)^{2/3}\right)$. However, with quantum computing, Shor’s algorithm [81] can solve this problem with time complexity $O((\log n)^3)$ and space complexity $O(\log n)$. The integer factorization problem and its variants form the basis of many widely used cryptographic protocols including the famous Rivest-Shamir-Adleman (RSA) public key encryption and digital signature schemes [254] and the Rabin public key encryption scheme [255].

The **discrete logarithm problem** states the following: given a finite cyclic group $G$ of order $n$ with generator $g \in G$ and element $h \in G$, it is difficult to find unique
integer $x$ such that $h = g^x$ and $0 \leq x \leq n - 1$. The discrete logarithm problem in the multiplicative subgroup of $\mathbb{Z}_p$ (also denoted $\mathbb{Z}_p^*$), where $p$ is a prime, is closely related to the integer factorization problem and the fastest algorithm which can solve this problem is the GNFS. With quantum computers, Shor’s algorithm can also be used to solve DLP in polynomial time. The discrete logarithm problem (DLP) and its variants form the basis of many widely used cryptographic protocols including the famous Diffie-Hellman (DH) key exchange [256], the ElGamal public key encryption and digital signature schemes [257] and the digital signature algorithm (DSA) [32].

In Chapter 2, we discuss elliptic curve cryptography (ECC) based on a variant of DLP based on elliptic curve groups, also known as the elliptic curve discrete logarithm problem (ECDLP) [36]. This is used to construct elliptic curve Diffie-Hellman (ECDH) key exchange and elliptic curve digital signature algorithm (ECDSA). Compared to DLP, protocols based on ECDLP have smaller ciphertext and key sizes at the same security level. In Chapters 5 and 6, we discuss pairing-based cryptography (PBC) [37] which maps elliptic curve group elements to finite field elements, acting as a bridge between ECDLP and DLP. In Chapter 3 and 4, we discuss lattice-based cryptography (LBC) built on hardness of the “learning with errors” (LWE) problem [82] and its variants, believed to be secure against quantum adversaries.
Appendix C

BLS12-381 Pairing Formulas

In this Appendix, we provide detailed mathematical derivations and formulas for towered arithmetic and elliptic curve point and line arithmetic over BLS12-381.

Pairing Computation

Let $E : y^2 = x^3 + ax + b$ be an elliptic curve defined over prime field $\mathbb{F}_p$. Let $G_1$ be a cyclic subgroup of $E(\mathbb{F}_p)$ of order $q$. Then, there also exists a cyclic subgroup $G_2$ of $E(\mathbb{F}_{p^k})$ of order $q$, where the embedding degree $k$ is the smallest integer such that $q | (p^k - 1)$. Let $G_T$ be a $q$-order subgroup of the multiplicative group $\mathbb{F}_{p^k}^\ast$. Then, a pairing is defined by the bilinear map $e : G_1 \times G_2 \rightarrow G_T$ with the following properties:

- Bilinearity: $e(aP, bQ) = e(P, Q)^{ab}$, where $P \in G_1$, $Q \in G_2$, $a, b \in \mathbb{Z}_q$
- Non-degeneracy: $\forall P \in G_1 \setminus \{O\} \exists Q \in G_2 : e(P, Q) \neq 1$
- Computability: $e(P, Q)$ can be computed efficiently

Many different definitions of the pairing function $e$ are available in literature, e.g., Weil pairing, Tate pairing, Ate pairing, optimal Ate pairing, etc [37, 186]. In this work, we consider the optimal Ate pairing, which is known for its efficiency [184]. In this case, computing the pairing $e$ involves evaluating a rational function $f_{\lambda, Q}$, where $\lambda$ is a constant specific to the curve, at point $P$ followed by a final exponentiation:

$$e(P, Q) = f_{\lambda, Q}(P)^{(p^k - 1)/q}$$
The function \( f_{\lambda, Q} \) can be computed efficiently (in polynomial time) by utilizing the following property identified by Miller [185]:

\[
f_{i+j, Q} = f_{i, Q} \cdot f_{j, Q} \cdot \frac{l_{iQ,jQ}}{v_{(i+j)Q}}
\]

This relation allows the use of a double-and-add approach (similar to elliptic curve scalar multiplication) to compute \( f_{\lambda, Q} \) by evaluating a series of straight lines intersecting the curve at desired points – straight line \( l_{iQ,jQ} \) through \( iQ, jQ \) and \( -(i + j)Q \); and vertical line \( v_{(i+j)Q} \) through \( (i+j)Q \) and \( -(i+j)Q \). This is known as Miller’s Algorithm. Further details are available in [37] and [186].

Since \( \mathbb{G}_1 \) is defined over \( \mathbb{F}_p \), it has an efficient representation. However, \( \mathbb{G}_2 \) is defined over \( \mathbb{F}_{p^k} \), which complicates the pairing computation since the Miller functions are all defined over \( \mathbb{F}_{p^k} \). Both BN and BLS12 curves have embedding degree \( k = 12 \). Fortunately, they also possess an efficiently computable isomorphism \( \Psi_d \) that allows mapping points between \( E(\mathbb{F}_{p^k}) \) and its twist curve \( E'(\mathbb{F}_{p^k/d}) \), where \( d \) is the degree of the twist:

\[
\Psi_d : E'(\mathbb{F}_{p^k/d}) \rightarrow E(\mathbb{F}_{p^k})
\]

The BN and BLS12 curves both support sextic twist \( (d = 6) \), thus allowing us to compute the Miller functions in \( \mathbb{F}_{p^2} \) instead of \( \mathbb{F}_{p^{12}} \). This not only compresses the elements of \( \mathbb{G}_2 \) but also reduces the computational complexity of the pairing map.

Then, the pairing map for BN and BLS12 curves can be redefined as

\[
e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T : E(\mathbb{F}_p) \times E'(\mathbb{F}_{p^2}) \rightarrow \mathbb{F}_{p^{12}}^*
\]

Let \( \xi \in \mathbb{F}_{p^2} \) be such that \( X^6 - \xi \) is irreducible over \( \mathbb{F}_{p^2} \). Then, for \( E : y^2 = x^3 + ax + b \) there are two possible forms of the twist curve and corresponding isomorphism:

- D-type twist: \( E'(\mathbb{F}_{p^2}) : y^2 = x^3 + b/\xi \) and \( \Psi_6 : (x, y) \rightarrow (x \xi^{1/3}, y \xi^{1/2}) \)
- M-type twist: \( E'(\mathbb{F}_{p^2}) : y^2 = x^3 + b \xi \) and \( \Psi_6 : (x, y) \rightarrow (x \xi^{-1/3}, y \xi^{-1/2}) \)

Specific parameter sets for BN and BLS12 curves and their twists will be discussed next. The consequence of having D-type versus M-type twists will be discussed later.
Pairing-Friendly Elliptic Curves

An elliptic curve $E(\mathbb{F}_p)$ is *pairing-friendly* if the following two conditions hold [37]:

- cardinality of $E(\mathbb{F}_p)$, also written as $\#E(\mathbb{F}_p)$, has a prime factor $q \geq \sqrt{p}$
- embedding degree of $E$ with respect to $q$ is less than $\log_2(r)/8$

It is highly unlikely that a randomly chosen elliptic curve satisfies these properties.

As recommended in [37], we need to find sets of three integers $(p, q, t)$ to construct pairing-friendly ordinary elliptic curves with desired embedding degree $k$, while satisfying the following properties:

- $p$ is prime or power of a prime
- $q$ is prime
- $t$ is co-prime to $p$
- $q \mid (p + 1 - t)$
- $q \mid (p^k - 1)$ and $q \nmid (p^i - 1) \forall 1 \leq i < k$
- $4q - t^2 = Dz^2$ for sufficiently small positively integer $D$ and integer $z$

Typically, these three parameters are defined as polynomials to generate *families of pairing-friendly curves* [258]. Here, we consider two very popular curve families:

**Barreto-Naehrig (BN) Curves:** Proposed by Barreto and Naehrig in 2005 [259], this family of pairing-friendly elliptic curves is defined as:

$E : y^2 = x^3 + b, \ b \neq 0$

- $p(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1$
- $q(u) = 36u^4 + 36u^3 + 18u^2 + 6u + 1$
- $t(u) = 6u^2 + 1$

The embedding degree of this family is $k = 12$, and discriminant $D = 3$.

One of the most widely used curves in the BN family is the BN-254 curve $E : y^2 = x^3 + 2$ defined by $u = -0x408000000000001h$ [260]. Here, $p$ and $q$ are both 254-bit primes. The extension field arithmetic is constructed as: $\mathbb{F}_{p^2} = \mathbb{F}_p[\alpha]/(\alpha^2 + 1)$,
\( \mathbb{F}_{p^6} = \mathbb{F}_{p^2}[\beta]/(\beta^3 - 1 - \alpha) \) and \( \mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[\gamma]/(\gamma^2 - \beta) \). The corresponding twist curve is defined as \( E'(\mathbb{F}_{p^2}) : y^2 = x^3 + 2/(1 + \alpha) = x^3 + (1 - \alpha) \), which is of D-type.

Another recently proposed curve in the BN family is the BN-462 curve \( E : y^2 = x^3 + 5 \) defined by \( u = 0x4001fffffffffffffffbbff_h \) [179]. Here, \( p \) and \( q \) are 462-bit primes. The extension field arithmetic is constructed as: \( \mathbb{F}_{p^2} = \mathbb{F}_p[\alpha]/(\alpha^2 + 1) \), \( \mathbb{F}_{p^6} = \mathbb{F}_{p^2}[\beta]/(\beta^3 - 2 - \alpha) \) and \( \mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[\gamma]/(\gamma^2 - \beta) \). The corresponding twist curve is defined as \( E'(\mathbb{F}_{p^2}) : y^2 = x^3 + 2/(2 + \alpha) = x^3 + (2 - \alpha) \), which is of D-type.

**Barreto-Lynn-Scott (BLS) Curves:** Proposed by Barreto, Lynn and Scott in 2002 [261], this includes several families of pairing-friendly elliptic curves with different embedding degrees such as BLS12, BLS24, BLS48, etc. In particular, the BLS12 family is defined as:

\[
E : y^2 = x^3 + b , \quad b \neq 0
\]

- \( p(u) = \frac{1}{3}(u - 1)^2(u^4 - u^2 + 1) + u \)
- \( q(u) = u^4 - u^2 + 1 \)
- \( t(u) = u + 1 \)

The embedding degree of this family is \( k = 12 \), and discriminant \( D = 3 \).

A recently proposed curve in the BLS12 family is the BLS12-381 curve \( E : y^2 = x^3 + 4 \) defined by \( u = -0xd201000000010000_h \) [180]. Here, \( p \) is a 381-bit prime while \( q \) is a 255-bit prime. The extension field arithmetic is constructed as: \( \mathbb{F}_{p^2} = \mathbb{F}_p[\alpha]/(\alpha^2 + 1) \), \( \mathbb{F}_{p^6} = \mathbb{F}_{p^2}[\beta]/(\beta^3 - 1 - \alpha) \) and \( \mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[\gamma]/(\gamma^2 - \beta) \). The corresponding twist curve is defined as \( E'(\mathbb{F}_{p^2}) : y^2 = x^3 + 4(1 + \alpha) \), which is of M-type.

**Pairing Curve Security:** The security of pairing-friendly elliptic curves relies on the hardness of the following discrete logarithm problems:

- elliptic curve discrete logarithm problem (ECDLP) over \( \mathbb{G}_1 \) and \( \mathbb{G}_2 \)
- finite field discrete logarithm problem (FFDLP) over \( \mathbb{G}_T \)

The best-known algorithms for solving these discrete logarithm problems are based on Pollard’s rho algorithm and the index calculus algorithm [181]. Till 2016, the standard
choice for pairing-based cryptography implementations at 128-bit security level was the BN-254 curve (or its nearest neighbour 254-bit and 256-bit BN curves). In 2016, Kim and Barbulescu proposed the extended tower number field sieve (exTNFS) algorithm [178] which drastically reduced the computational complexity of solving FFDLP. The exTNFS attack affected the security of BN and BLS curves with embedding degrees divisible by 6. As a result, the security level of BN-254 was reduced from 128-bit down to ≈100-bit [181]. The Internet Engineering Task Force (IETF) is currently standardizing pairing-friendly curves [181], and it recommends using BLS12-381 and BN-462 instead of BN-254 for standard security applications. The estimated security levels of BLS12-381 and BN-462 are ≈126-bit and ≈134-bit respectively. The BLS12-381 curve is preferred for computational efficiency, while the BN-462 curve can be used for higher security use cases.

Towered Arithmetic for BLS12-381

Extension Field Arithmetic in $\mathbb{F}_{p^2}$

Here, we discuss our implementation of arithmetic over the quadratic extension field $\mathbb{F}_{p^2} = \mathbb{F}_p[\alpha]/(\alpha^2 + 1)$ for the BLS12-381 pairing groups. The associated computation costs are provided in terms of additions / subtractions ($A_1$), multiplications / squarings ($M_1$) and inversions ($I_1$) in $\mathbb{F}_p$.

Addition in $\mathbb{F}_{p^2}/\mathbb{F}_p$:

\[(x_0 + x_1\alpha) + (y_0 + y_1\alpha) = (x_0 + y_0) + (x_1 + y_1)\alpha\]

$\Rightarrow$ Cost ($A_2$) = $2A_1$

Multiplication in $\mathbb{F}_{p^2}/\mathbb{F}_p$:

\[(x_0 + x_1\alpha) \cdot (y_0 + y_1\alpha) = (x_0y_0 - x_1y_1) + (x_0y_1 + x_1y_0)\alpha\]

Here, $(x_0y_1 + x_1y_0)$ is calculated using the Karatsuba method [196] as:

\[(x_0y_1 + x_1y_0) = (x_0 + x_1) \cdot (y_0 + y_1) - (x_0y_0 + x_1y_1)\]

to reduce the number of multiplications at the cost of extra additions.

$\Rightarrow$ Cost ($M_2$) = $3M_1 + 5A_1$
Squaring in \( F_{p^2}/F_p \):
\[
(x_0 + x_1 \alpha)^2 = (x_0^2 - x_1^2) + (2x_0 x_1) \alpha
\]
Here, \((x_0^2 - x_1^2)\) is calculated using the Complex method \([196]\) as:
\[
(x_0^2 - x_1^2) = (x_0 + x_1) \cdot (x_0 - x_1)
\]
to reduce the number of multiplications at the cost of extra additions.
\(\Rightarrow\) Cost \((S_2) = 2M_1 + 3A_1\)

Inversion in \( F_{p^2}/F_p \):
\[
(x_0 + x_1 \alpha)^{-1} = (x_0 - x_1 \alpha) \cdot (x_0^2 + x_1^2)^{-1}
\]
\(\Rightarrow\) Cost \((I_2) = 4M_1 + 2A_1 + I_1\)

Multiplication with \( \xi = (1 + \alpha) \) in \( F_{p^2}/F_p \):
\[
(1 + \alpha) \cdot (x_0 + x_1 \alpha) = (x_0 - x_1) + (x_0 + x_1) \alpha
\]
This operation will be essential in later computations.
\(\Rightarrow\) Cost \((m_\xi) = 2A_1\)

Multiplication with \( \xi^{-1} = 2^{-1} (1 - \alpha) \) in \( F_{p^2}/F_p \):
\[
2^{-1} (1 - \alpha) \cdot (x_0 + x_1 \alpha) = 2^{-1} (x_0 + x_1) + 2^{-1} (x_1 - x_0) \alpha
\]
This operation will also be essential in later computations. We assume \(2^{-1} \mod p\) is available as a pre-computed constant.
\(\Rightarrow\) Cost \((m_{\xi^{-1}}) = 2M_1 + 2A_1\)

For all the above computations, we have utilized \(\alpha^2 \equiv -1 \mod (\alpha^2 + 1)\). Also, small scalar multiples of \( F_p \) elements are computed using repeated additions.

**Extension Field Arithmetic in \( F_{p^6} \)**

Here, we discuss our implementation of arithmetic over the cubic extension field \( F_{p^6} = F_{p^2}[\beta]/(\beta^3 - 1 - \alpha) \) for the BLS12-381 pairing groups. The associated computation costs are provided in terms of additions / subtractions \((A_2)\), multiplications \((M_2)\), squarings \((S_2)\), inversions \((I_2)\) and \( \xi \)-multiplications \((m_\xi)\) in \( F_{p^2} \).

**Addition in \( F_{p^6}/F_{p^2} \):**
\[
(x_0 + x_1 \beta + x_2 \beta^2) + (y_0 + y_1 \beta + y_2 \beta^2) = (x_0 + y_0) + (x_1 + y_1) \beta + (x_2 + y_2) \beta^2
\]
\(\Rightarrow\) Cost \((A_6) = 3A_2\)
Multiplication in $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$:

$$(x_0 + x_1 \beta + x_2 \beta^2) \cdot (y_0 + y_1 \beta + y_2 \beta^2) = x_0 y_0 + \xi (x_1 y_2 + x_2 y_1) + (x_0 y_1 + x_1 y_0 + \xi x_2 y_2) \beta + (x_0 y_2 + x_2 y_0 + \xi x_1 y_1) \beta^2$$

Here, $(x_1 y_2 + x_2 y_1), (x_0 y_1 + x_1 y_0 + \xi x_2 y_2)$ and $(x_0 y_2 + x_2 y_0 + \xi x_1 y_1)$ are calculated using the Karatsuba method [196] as:

$$(x_1 y_2 + x_2 y_1) = (x_1 + x_2) \cdot (y_1 + y_2) - (x_1 y_1 + x_2 y_2)$$

$$(x_0 y_1 + x_1 y_0 + \xi x_2 y_2) = (x_0 + x_1) \cdot (y_0 + y_1) - (x_0 y_0 + x_1 y_1 - \xi x_2 y_2)$$

$$(x_0 y_2 + x_2 y_0 + \xi x_1 y_1) = (x_0 + x_2) \cdot (y_0 + y_2) - (x_0 y_0 + x_2 y_2 - x_1 y_1)$$

to reduce the number of multiplications at the cost of extra additions.

$\Rightarrow$ Cost $(M_6) = 6M_2 + 15A_2 + 2m_\xi$

Sparse Multiplication in $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$:

$$(x_0 + x_1 \beta + x_2 \beta^2) \cdot (y_0 + y_1 \beta + y_2 \beta^2) = \xi (x_1 y_2 + x_2 y_1) + (x_0 y_1 + \xi x_2 y_2) \beta + (x_0 y_2 + \xi x_1 y_1) \beta^2$$

Once again, $(x_1 y_2 + x_2 y_1)$ is calculated using the Karatsuba method [196] as:

$$(x_1 y_2 + x_2 y_1) = (x_1 + x_2) \cdot (y_1 + y_2) - (x_1 y_1 + x_2 y_2)$$

to reduce the number of multiplications at the cost of extra additions. Note that this formula is slightly different compared to BN curves [197] due to the presence of an M-type twist in BLS12-381 as opposed to a D-type twist.

$\Rightarrow$ Cost $(sM_6) = 5M_2 + 6A_2 + 2m_\xi$

Squaring in $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$:

$$(x_0^2 + x_1 \beta + x_2 \beta^2)^2 = x_0^2 + 2\xi x_1 x_2 + (2x_0 x_1 + \xi x_2^2) \beta + (x_1^2 + 2x_0 x_2) \beta^2$$

Here, $(x_1^2 + 2x_0 x_2)$ is calculated using the Chung-Hasan method [196] as:

$$(x_1^2 + 2x_0 x_2) = (x_0 + x_1 + x_2)^2 - (2x_0 x_1 + 2x_1 x_2 + x_0^2 + x_2^2)$$

to reduce the number of multiplications at the cost of extra additions.

$\Rightarrow$ Cost $(S_6) = 2M_2 + 3S_2 + 9A_2 + 2m_\xi$

Inversion in $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$:

$$(x_0 + x_1 \beta + x_2 \beta^2)^{-1} = ((x_0^2 - \xi x_1 x_2) + (\xi x_2^2 - x_0 x_1) \beta + (x_1^2 - x_0 x_2) \beta^2) \cdot (\xi x_1 (x_1^2 - x_0 x_2) + x_0 (x_0^2 - \xi x_1 x_2) + \xi x_2 (\xi x_2^2 - x_0 x_1))^{-1}$$

$\Rightarrow$ Cost $(I_6) = 9M_2 + 3S_2 + 5A_2 + I_2 + 4m_\xi$

181
Multiplication with $\beta$ in $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$:

$$\beta \cdot (x_0 + x_1 \beta + x_2 \beta^2) = \xi x_2 + x_0 \beta + x_1 \beta^2$$

This operation will be essential in later computations.

$$\Rightarrow \text{Cost} (m_\beta) = m_\xi$$

For all the above computations, we have utilized $\beta^3 \equiv \xi \mod (\beta^3 - \xi)$. Also, small scalar multiples of $\mathbb{F}_{p^2}$ elements are computed using repeated additions. The sparse multiplication will be useful in Miller line computations, to be discussed later.

**Extension Field Arithmetic in $\mathbb{F}_{p^{12}}$**

Here, we discuss our implementation of arithmetic over the quadratic extension field $\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[\gamma]/(\gamma^2 - \beta)$ for the BLS12-381 pairing groups. There is some similarity with our construction of $\mathbb{F}_{p^2}/\mathbb{F}_p$. The associated computation costs are provided in terms of additions / subtractions ($A_6$), multiplications ($M_6$), squarings ($S_6$), inversions ($I_6$) and $\beta$-multiplications ($m_\beta$) in $\mathbb{F}_{p^6}$.

**Addition in $\mathbb{F}_{p^{12}}/\mathbb{F}_{p^6}$:**

$$(x_0 + x_1 \gamma) + (y_0 + y_1 \gamma) = (x_0 + y_0) + (x_1 + y_1) \gamma$$

$$\Rightarrow \text{Cost} (A_{12}) = 2A_6$$

**Multiplication in $\mathbb{F}_{p^{12}}/\mathbb{F}_{p^6}$:**

$$(x_0 + x_1 \gamma) \cdot (y_0 + y_1 \gamma) = (x_0 y_0 + \beta x_1 y_1) + (x_0 y_1 + x_1 y_0) \gamma$$

Again, $(x_0 y_1 + x_1 y_0)$ is calculated using the Karatsuba method [196] as:

$$(x_0 y_1 + x_1 y_0) = (x_0 + x_1) \cdot (y_0 + y_1) - (x_0 y_0 + x_1 y_1)$$

to reduce the number of multiplications at the cost of extra additions.

$$\Rightarrow \text{Cost} (M_{12}) = 3M_6 + 5A_6 + m_\beta$$

**Squaring in $\mathbb{F}_{p^{12}}/\mathbb{F}_{p^6}$:**

$$(x_0 + x_1 \gamma)^2 = (x_0^2 + \beta x_1^2) + (2x_0 x_1) \gamma$$

Here, $2x_0 x_1$ is calculated using the Karatsuba method [196] as:

$$2x_0 x_1 = (x_0 + x_1)^2 - (x_0^2 + x_1^2)$$

to reduce the number of multiplications at the cost of extra additions.

$$\Rightarrow \text{Cost} (S_{12}) = 3S_6 + 4A_6 + m_\beta$$
**Inversion in** \( \mathbb{F}_{p^{12}}/\mathbb{F}_{p^6} \):

\[
(x_0 + x_1 \gamma)^{-1} = (x_0 - x_1 \gamma) \cdot (x_0^2 - \beta x_1^2)^{-1}
\]

\(\Rightarrow\) Cost \((I_{12}) = 2M_6 + 2S_6 + 2A_6 + I_6\)

Apart from towered arithmetic, few more optimized extension field operations are required specifically for pairing computation, which will be discussed later.

**Elliptic Curve Computations for BLS12-381**

**Point and Line Arithmetic in** \( \mathcal{G}_1 \) and \( \mathcal{G}_2 \)

Here, we discuss our implementations of point doubling, point addition, chord line and tangent line computations which will be essential for both elliptic curve scalar multiplication and pairing on BLS12-381.

**Point Arithmetic:** Homogeneous projective coordinates are used, where any point is written in the form \((X : Y : Z)\) and the corresponding affine representation is \((X/Z , Y/Z)\). The doubling and addition formulas are:

- For point \(P = (X_1 : Y_1 : Z_1)\) on the curve \(E : y^2 = x^3 + b\), the exception-free doubling formula for \((X_3 : Y_3 : Z_3) = 2P\) is \([198]\):

  \[
  X_3 = 2X_1Y_1(Y_1^2 - 9bZ_1^2) \\
  Y_3 = (Y_1^2 - 9bZ_1^2)(Y_1^2 + 3bZ_1^2) + 24bY_1^2Z_1^2 \\
  Z_3 = 8Y_1^3Z_1
  \]

  which can be computed using 6 multiplications \((M)\), 2 squarings \((S)\), 9 additions / subtractions \((A)\) and 1 multiplication by \(3b\) \((m_{3b})\), as shown in Algorithm C.1.

- For points \(P = (X_1 : Y_1 : Z_1)\) and \(Q = (X_2 : Y_2 : 1)\) on the curve \(E : y^2 = x^3 + b\), the complete mixed addition formula for \((X_3 : Y_3 : Z_3) = P + Q\) is \([198]\):

  \[
  X_3 = (X_1Y_2 + X_2Y_1)(Y_1Y_2 - 3bZ_1) - 3b(Y_1 + Y_2Z_1)(X_1 + X_2Z_1) \\
  Y_3 = (Y_1Y_2 + 3bZ_1)(Y_1Y_2 - 3bZ_1) + 9bX_1X_2(X_1 + X_2Z_1) \\
  Z_3 = (Y_1 + Y_2Z_1)(Y_1Y_2 + 3bZ_1) + 3X_1X_2(Y_1X_2 + X_2Y_1)
  \]

  which can be computed using 11 multiplications \((M)\), 13 additions / subtractions \((A)\) and 2 multiplications by \(3b\) \((m_{3b})\), as shown in Algorithm C.2.
Algorithm C.1 Exception-free projective point doubling on prime-order $j$-invariant 0 short Weierstrass curve $E : y^2 = x^3 + b$ [198]

Require: Point $P = (X_1 : Y_1 : Z_1)$ on $E : Y^2Z = X^3 + bZ^3$

Ensure: $(X_3 : Y_3 : Z_3) = 2P$

1: $t_0 \leftarrow Y_1^2$
2: $Z_3 \leftarrow t_0 + t_0$
3: $Z_3 \leftarrow t_0 + t_0$
4: $Z_3 \leftarrow Z_3 + Z_3$
5: $t_1 \leftarrow Y_1 \cdot Z_1$
6: $t_2 \leftarrow Z_1^2$
7: $t_2 \leftarrow (3b) \cdot t_2$
8: $X_3 \leftarrow t_2 \cdot Z_3$
9: $Y_3 \leftarrow t_0 + t_2$
10: $Z_3 \leftarrow t_1 \cdot Z_3$
11: $t_1 \leftarrow t_2 + t_2$
12: $t_2 \leftarrow t_1 + t_2$
13: $t_0 \leftarrow t_0 - t_2$
14: $Y_3 \leftarrow t_0 \cdot Y_3$
15: $Y_3 \leftarrow X_3 + Y_3$
16: $t_1 \leftarrow X_1 \cdot Y_1$
17: $X_3 \leftarrow t_0 \cdot t_1$
18: $X_3 \leftarrow X_3 + X_3$
19: return $(X_3 : Y_3 : Z_3)$

Algorithm C.2 Complete mixed point addition on prime-order $j$-invariant 0 short Weierstrass curve $E : y^2 = x^3 + b$ [198]

Require: Points $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : 1)$ on $E : Y^2Z = X^3 + bZ^3$

Ensure: $(X_3 : Y_3 : Z_3) = P + Q$

1: $t_0 \leftarrow X_1 \cdot X_2$
2: $t_1 \leftarrow Y_1 \cdot Y_2$
3: $t_2 \leftarrow X_2 + X_2$
4: $t_3 \leftarrow X_3 + Y_1$
5: $t_4 \leftarrow X_1 + Y_1$
6: $t_5 \leftarrow t_3 \cdot t_4$
7: $t_6 \leftarrow t_0 + t_1$
8: $t_7 \leftarrow Y_2 \cdot Z_1$
9: $t_8 \leftarrow t_4 \cdot Y_1$
10: $t_9 \leftarrow t_4 \cdot t_5$
11: $t_10 \leftarrow t_3 \cdot t_4$
12: $t_11 \leftarrow t_3 \cdot t_6$
13: $t_12 \leftarrow t_0 + t_6$
14: $t_13 \leftarrow t_0 + t_1$
15: $t_14 \leftarrow Y_2 \cdot Z_1$
16: $t_15 \leftarrow t_1 + t_2$
17: $t_16 \leftarrow t_1 - t_2$
18: $t_17 \leftarrow t_3 \cdot t_1$
19: $t_18 \leftarrow t_3 \cdot t_1$
20: $t_19 \leftarrow t_3 \cdot t_1$
21: $t_20 \leftarrow t_3 \cdot t_1$
22: $t_21 \leftarrow t_1 \cdot Z_3$
23: $t_22 \leftarrow t_1 \cdot Z_3$
24: $t_23 \leftarrow t_1 + Y_3$
25: $t_24 \leftarrow t_1 + Y_3$
26: $t_25 \leftarrow t_1 + Y_3$
27: return $(X_3 : Y_3 : Z_3)$

All arithmetic is performed in the underlying field: $\mathbb{F}_p$ for $\mathbb{G}_1$ and $\mathbb{F}_{p^2}$ for $\mathbb{G}_2$. Therefore, $A = A_1$, $M = S = M_1$ for $\mathbb{G}_1$ and $A = A_2$, $M = M_2$, $S = S_2$ for $\mathbb{G}_2$. Also, note that $b = 4$ for $\mathbb{G}_1$ and $b = 4 \xi$ for $\mathbb{G}_2$. So, multiplications by $3b$ can be easily performed using repeated additions and multiplications in $\mathbb{F}_p$.

Algorithm C.3 shows a constant-time implementation of elliptic curve scalar multiplication (ECSM) on $\mathbb{G}_1$ and $\mathbb{G}_2$ based on the simple left-to-right double-and-add algorithm from [36]. In lines 4, 6 and 8, the previously discussed doubling and addition formulas (Algorithms C.1 and C.2) are used. In lines 7-8, a dummy point addition with $T_{\text{dummy}}$ is used to prevent side-channel attacks, also known as double-and-add-always [76]. In lines 11-13, the homogeneous projective point $(X_T : Y_T : Z_T)$ is converted back to its affine form $(X_T/Z_T, Y_T/Z_T)$ using one inversion and two multiplications. When using Algorithm C.3, the ECSM computation with a 255-bit $\mathbb{F}_q$ scalar requires $4,847M_1 + 14,025A_1 + I_1$ (where $I_1 = 608M_1$) for $\mathbb{G}_1$ and
Algorithm C.3 Constant-time elliptic curve scalar multiplication on $G_1$ (resp. $G_2$) using mixed coordinates (for BLS12-381) (adapted from [36])

Require: $P = (x_P, y_P) \in G_1$ (resp. $G_2$) and scalar $k = (k_{254}, \cdots, k_1, k_0)_2 \in \mathbb{F}_q$

Ensure: $(x_T, y_T) = kP \in G_1$ (resp. $G_2$)

1: $T = (X_T : Y_T : Z_T) \leftarrow (0 : 1 : 0)$
2: $T_{dummy} \leftarrow (0 : 1 : 0)$
3: for $(i = 254; i \geq 0; i = i - 1)$ do
4: $T \leftarrow 2T$ using Algorithm C.1
5: if $k_i = 1$ then
6: $T \leftarrow T + (x_P : y_P : 1)$ using Algorithm C.2
7: else
8: $T_{dummy} \leftarrow T + (x_P : y_P : 1)$ using Algorithm C.2
9: end if
10: end for
11: $Z_T \leftarrow Z_T^{-1}$
12: $x_T \leftarrow X_T \cdot Z_T$
13: $y_T \leftarrow Y_T \cdot Z_T$
14: return $(x_T, y_T)$

$4,337M_2 + 510S_2 + 10,200A_2 + I_2$ (where $I_2 = 4M_1 + 2A_1 + I_1$ and we consider $m_\xi = 2A_1 \equiv A_2$) for $G_2$. Depending on the application, performance of this ECSM operation can be further improved by using standard pre-computation-based techniques (memory-time trade-offs) such as windowing, comb, etc [36].

**Line Arithmetic:** A major component of Millers’ algorithm is the computation of chord and tangent lines on the elliptic curve. While we could directly use generic point arithmetic formulas for our BLS12-381 implementation, this doesn’t work for line arithmetic. This is because BLS12-381 has an M-type twist, while previously used pairing curves, e.g., BN curves, have D-type twists. Therefore, we explicitly derive chord and tangent line computation formulas for the BLS12-381 curve as building blocks of our pairing implementation. First, the line equations are derived using one or two points on $E'(\mathbb{F}_p^2)$. Since the pairing result is in $\mathbb{F}_{p^{12}}^*$, the line equations must be in $\mathbb{F}_{p^{12}}$. Therefore, points on the twist curve $E'(\mathbb{F}_p^2)$ are converted to their equivalent representations on $E(\mathbb{F}_{p^{12}})$ using the isomorphism $\Psi_6$ discussed earlier:

- Affine point $(x, y) \in E'(\mathbb{F}_p^2)$ is transformed to $(x \xi^{-1/3}, y \xi^{-1/2}) = (x \gamma^{-2}, y \gamma^{-3}) \in E(\mathbb{F}_{p^{12}})$ since $\gamma^2 = \beta$ and $\beta^3 = \xi$ in $\mathbb{F}_{p^{12}} \Rightarrow \gamma^6 = \xi$
• Similarly, projective point \((X : Y : Z) \in E'(\mathbb{F}_p^2)\) is also transformed into 
\((X \xi^{-1/3} : Y \xi^{-1/2} : Z) = (X \gamma^{-2} : Y \gamma^{-3} : Z) \in E(\mathbb{F}_{p^2})\)

Now, slope of the chord through points \(T (X_T \gamma^{-2} : Y_T \gamma^{-3} : Z_T) \in E(\mathbb{F}_{p^2})\) and 
\(Q (x_Q \gamma^{-2} : y_Q \gamma^{-3}) \equiv (x_Q \gamma^{-2} : y_Q \gamma^{-3} : 1) \in E(\mathbb{F}_{p^2})\) (resp. the tangent at point \(T\) 
when \(T = Q\)) is \(\lambda / \gamma\), where:
\[
\lambda = \frac{Y_T - y_Q Z_T}{X_T - x_Q Z_T} \quad \text{(resp.} \quad \lambda = \frac{3X_T^2}{2Y_T Z_T})
\]
and, equation of the corresponding straight line is:
\[
y = \frac{\lambda}{\gamma} \cdot (x - \frac{X_T}{Z_T} \gamma^{-2}) + \frac{Y_T}{Z_T} \gamma^{-3} = \frac{\lambda}{\gamma} \cdot x - \frac{(\lambda X_T - Y_T)}{Z_T} \cdot \frac{1}{\gamma^3}
\]
Since \(\gamma^2 = \beta\) and \(\beta^3 = \xi\), we have \(\gamma^{-1} = \beta^2 \gamma / \xi\) and \(\gamma^{-3} = \gamma^3 / \xi\). Therefore, the explicit line equations evaluated at point \(P (x_p, y_p) \in E(\mathbb{F}_p)\) are obtained as:
• Chord: \(l_{T,Q}(P) = y_p D - x_p N \cdot \frac{\beta^2}{\xi} + (N x_Q - D y_Q) \cdot \frac{\gamma^3}{\xi}\)
• Tangent: \(l_{T,T}(P) = 2y_p Y_T Z_T - 3x_p X_T^2 \cdot \frac{\beta^3}{\xi} + (Y_T^2 - 12Z_T^2) \cdot \frac{\gamma^3}{\xi}\)

where \(N = (Y_T - y_Q Z_T)\) and \(D = (X_T - x_Q Z_T)\).

Note that any element in \(\mathbb{F}_{p^2} / \mathbb{F}_{p^2}/\mathbb{F}_{p^2}\) can be written as \(a = (b_0 + b_2 \beta + b_4 \beta^2) + (b_1 + b_3 \beta + b_5 \beta^2) \gamma\) where \(b_0, \ldots, b_5 \in \mathbb{F}_p^2\). Then, the above chord and tangent lines 
are sparse elements in \(\mathbb{F}_{p^2}\) of the form:
\[
l = b_0 + b_5 \beta^2 \gamma + b_3 \gamma^3 = b_0 + (b_3 \beta + b_5 \beta^2) \gamma \quad \text{since} \quad \gamma^2 = \beta
\]

Therefore, these lines can be computed entirely using \(\mathbb{F}_p^2\) arithmetic, and they can be 
evaluated at any point \(P (x_p, y_p) \in E(\mathbb{F}_p)\) using \(\mathbb{F}_p\) arithmetic. We observe that this 
sparse form of the chord and tangent lines in BLS12-381 is quite different from BN 
curves due to the presence of a different type of twist.

Along with line evaluations \(l_{T,T}(P)\) and \(l_{T,Q}(P)\) in \(\mathbb{F}_{p^2}\), the doubling and addition 
steps in Miller’s algorithm also require computation of projective points \((X_{2T} : Y_{2T} : 
Z_{2T}) = 2T\) and \((X_{T+Q} : Y_{T+Q} : Z_{T+Q}) = T + Q\) respectively in \(E'(\mathbb{F}_p^2)\):

186
• Point addition (with chord line evaluation):

\[ X_{T+Q} = DM \]
\[ Y_{T+Q} = N(x_Q D^2 Z_T - M) - y_Q D^3 Z_T \]
\[ Z_{T+Q} = D^3 Z_T \]

• Point doubling (with tangent line evaluation):

\[ X_{2T} = 2X_T Y_T (Y_T^2 - 36 \xi Z_T^2) \]
\[ Y_{2T} = (Y_T^2 + 36 Z_T^2) - 12(12 \xi Z_T^2)^2 \]
\[ Z_{2T} = 8Y_T^3 Z_T \]

where \( N = (Y_T - y_Q Z_T), \ D = (X_T - x_Q Z_T) \) and \( M = (N^2 Z_T - X_T D^2 - x_Q D^2 Z_T) \).

---

**Algorithm C.4** Point and line evaluation in Miller addition step for BLS12-381

**Require:** Points \( T = (X_T : Y_T : Z_T) \in E'(\mathbb{F}_p^2), \ Q = (x_Q, y_Q) \in E'(\mathbb{F}_p^2) \) and \( P = (x_P, y_P) \in E(\mathbb{F}_p) \)

**Ensure:** \((X_R : Y_R : Z_R) = T + Q \in E'(\mathbb{F}_p^2) \) and \( l_{T,Q}(P) = b_0 + (b_3 \beta + b_5 \beta^2) \gamma \in \mathbb{F}_{p^2} \)

1: \( t_0 \leftarrow x_Q \cdot Z_T \)
2: \( t_1 \leftarrow y_Q \cdot Z_T \)
3: \( N \leftarrow Y_T - t_1 \)
4: \( D \leftarrow X_T - t_0 \)
5: \( t_2 \leftarrow D^2 \)
6: \( t_3 \leftarrow X_T + t_0 \)
7: \( t_4 \leftarrow t_2 \cdot t_3 \)
8: \( M \leftarrow N^2 \)
9: \( M \leftarrow M \cdot Z_T \)
10: \( X_R \leftarrow D \cdot M \)
11: \( Z_R \leftarrow t_2 \cdot Z_T \)
12: \( Y_R \leftarrow y_P \cdot D \)
13: \( Y_R \leftarrow Y_R - M \)
14: \( Y_R \leftarrow Y_R \cdot N \)
15: \( t_2 \leftarrow t_2 \cdot t_0 \)
16: \( t_3 \leftarrow t_2 \cdot t_1 \)
17: \( Y_R \leftarrow Y_R - t_3 \)
18: \( Z_R \leftarrow t_2 \cdot M - t_3 \)
19: \( b_3 \leftarrow t_1 \cdot t_3 \)
20: \( b_5 \leftarrow (-x_P) \cdot N \)
21: \( b_5 \leftarrow \xi^{-1} \cdot b_5 \)
22: \( t_2 \leftarrow t_2 \cdot D \)
23: \( t_3 \leftarrow t_2 \cdot t_1 \)
24: \( b_3 \leftarrow t_0 - t_1 \)
25: \( b_3 \leftarrow t_1 \cdot t_3 \)
26: \( t_2 \leftarrow D^2 \)
27: \( t_3 \leftarrow X_T + t_0 \)
28: \( t_0 \leftarrow t_0 + t_0 \)
29: \( Z_R \leftarrow Y_R \cdot t_0 \)
30: \( Z_R \leftarrow Z_R + Z_R \)
31: \( Z_R \leftarrow Z_R + Z_R \)
32: \( b_0 \leftarrow y_P \cdot t_0 \)
33: \( b_5 \leftarrow (-3x_P) \cdot X_2 \)
34: \( b_5 \leftarrow \xi^{-1} \cdot b_5 \)
35: \( b_3 \leftarrow Y_2 - t_1 \)
36: \( b_3 \leftarrow \xi^{-1} \cdot b_3 \)
37: \( \text{return} \) \( 2T = (X_R : Y_R : Z_R) \) and \( l_{T,T}(P) = b_0 + (b_3 \beta + b_5 \beta^2) \gamma \)

---

**Algorithm C.5** Point and line evaluation in Miller doubling step for BLS12-381

**Require:** Point \( T = (X_T : Y_T : Z_T) \in E'(\mathbb{F}_p^2) \) and \( P = (x_P, y_P) \in E(\mathbb{F}_p) \)

**Ensure:** \((X_R : Y_R : Z_R) = 2T \in E'(\mathbb{F}_p^2) \) and \( l_{T,T}(P) = b_0 + (b_3 \beta + b_5 \beta^2) \gamma \in \mathbb{F}_{p^2} \)

1: \( X_2 \leftarrow X_T^2 \)
2: \( Y_2 \leftarrow Y_T^2 \)
3: \( Z_2 \leftarrow Z_T^2 \)
4: \( t_0 \leftarrow X_T + Y_T \)
5: \( t_0 \leftarrow t_0 - X_2 \)
6: \( t_0 \leftarrow t_0 - Y_2 \)
7: \( t_0 \leftarrow t_0 - Y_2 \)
8: \( t_1 \leftarrow \xi \cdot Z_2 \)
9: \( t_1 \leftarrow t_1 + t_1 \)
10: \( t_1 \leftarrow t_1 + t_1 \)
11: \( t_3 \leftarrow t_1 + t_1 \)
12: \( t_1 \leftarrow t_3 + t_1 \)
13: \( t_3 \leftarrow t_1 + t_1 \)
14: \( t_2 \leftarrow t_3 + t_1 \)
15: \( t_2 \leftarrow t_2 + t_1 \)
16: \( t_2 \leftarrow t_2 + t_1 \)
17: \( t_2 \leftarrow t_2 + t_1 \)
18: \( t_2 \leftarrow t_2 + t_1 \)
19: \( t_3 \leftarrow t_0 + t_0 \)
20: \( t_3 \leftarrow t_0 + t_0 \)
21: \( t_3 \leftarrow t_0 + t_0 \)
22: \( Y_R \leftarrow Y_2 + t_2 \)
23: \( Y_R \leftarrow Y_R + t_2 \)
24: \( Y_R \leftarrow Y_R + t_0 \)
25: \( Y_R \leftarrow Y_R + t_0 \)
26: \( t_0 \leftarrow t_0 + t_0 \)
27: \( t_0 \leftarrow t_0 + t_0 \)
28: \( t_0 \leftarrow t_0 + t_0 \)
29: \( Z_R \leftarrow Y_2 \cdot t_0 \)
30: \( Z_R \leftarrow Z_R + Z_R \)
31: \( Z_R \leftarrow Z_R + Z_R \)
32: \( b_0 \leftarrow y_P \cdot t_0 \)
33: \( b_5 \leftarrow (-3x_P) \cdot X_2 \)
34: \( b_5 \leftarrow \xi^{-1} \cdot b_5 \)
35: \( b_3 \leftarrow Y_2 - t_1 \)
36: \( b_3 \leftarrow \xi^{-1} \cdot b_3 \)
37: \( \text{return} \) \( 2T = (X_R : Y_R : Z_R) \) and \( l_{T,T}(P) = b_0 + (b_3 \beta + b_5 \beta^2) \gamma \)
Using these formulas, our implementations of the Miller addition and doubling step computations are shown in Algorithms C.4 and C.5 respectively. In lines 19-20 of Algorithm C.4 and lines 32-33 of Algorithm C.5, the multiplications by \(y_P, (-x_P)\) and \((-3x_P)\) require \(2M_1\) each. The cost of multiplication by \(\xi^{-1}\) is \(m_{\xi^{-1}}\) as discussed earlier. In Algorithm C.5, the quantities \(2X_TY_T\) and \(2Y_TZ_T\) are computed using the Karatsuba method [196] as:

\[
2X_TY_T = (X_T + Y_T)^2 - (X_T^2 + Y_T^2)
\]

\[
2Y_TZ_T = (Y_T + Z_T)^2 - (Y_T^2 + Z_T^2)
\]

While the cost of directly computing \(2X_TY_T\) or \(2Y_TZ_T\) is \(M_2 + A_2 \equiv 3M_1 + 7A_1\), the above method requires \(S_2 + 3A_2 \equiv 2M_1 + 9A_1\) (since \(X_T^2, Y_T^2, Z_T^2\) are already computed at the beginning). This method is clearly more efficient as \(M_1 \gg A_1\). Overall, the computation costs of Algorithms C.4 and C.5 are \(12M_2 + 2S_2 + 4M_1 + 7A_2 + 2m_{\xi^{-1}} + A_1\) and \(2M_2 + 7S_2 + 4M_1 + 22A_2 + 2m_{\xi^{-1}} + m_\xi + 3A_1\) respectively.

**Pairing Computation on BLS12-381**

Algorithm C.6 shows the complete optimal Ate pairing computation for BLS12-381 (adapted from the corresponding version for BN curves [262]). Our implementation is based on the towered arithmetic and line evaluation formulas described earlier, along with well-known speedup techniques from previous work [262–265]. The pairing computation is divided into two parts – Miller Loop and Final Exponentiation. Referring to the notation \(e(P, Q) = f_{\lambda, Q}(P)^{(p^k - 1)/q}\) from earlier, the Miller Loop (lines 1-16) computes \(f_{\lambda, Q}(P)\) using Miller’s algorithm [185]. Here, \(\lambda = u\) (unlike BN curves, where \(\lambda = 6u + 2\)). Next, the Final Exponentiation (line 17) raises this value to the power \((p^k - 1)/q\) (or its multiple, if required) to get the final result. Here, \(k = 12\). Due to efficiency reasons (to be discussed later), the exponent is \(3(p^{12} - 1)/q\) instead of \((p^{12} - 1)/q\) for BLS12-381. This does not affect correctness of the algorithm as the output is still an element in \(G_T\) and bilinearity is retained.
**Algorithm C.6** Optimal Ate pairing for BLS12-381 (adapted from [262])

**Require:** \( P(x_P, y_P) \in \mathbb{G}_1, Q(x_Q, y_Q) \in \mathbb{G}_2 \) and \(|u| = (u_{63}, \cdots, u_1, u_0)_2\)

**Ensure:** \( f = e(P, Q) \in \mathbb{G}_T \)

1. \( T \leftarrow 2Q \) and \( f \leftarrow l_{Q,Q}(P) \) using Algorithm C.5
2. \( T \leftarrow T + Q \) and \( l_P \leftarrow l_{T,Q}(P) \) using Algorithm C.4
3. \( f \leftarrow f \cdot l_P \)
4. **for** \((i = 61; i \geq 0; i = i - 1)\) **do**
5. \( T \leftarrow 2T \) and \( l_P \leftarrow l_{T,T}(P) \) using Algorithm C.5
6. \( f \leftarrow f^2 \)
7. \( f \leftarrow f \cdot l_P \)
8. **if** \( u_i = 1 \) **then**
9. \( T \leftarrow T + Q \) and \( l_P \leftarrow l_{T,Q}(P) \) using Algorithm C.4
10. \( f \leftarrow f \cdot l_P \)
11. **end if**
12. **end for**
13. \( f \leftarrow \tilde{f} \)
14. \( f \leftarrow f^{3(p^{12} - 1)/q} \)
15. **return** \( f \)

For BLS12-381, we have \( u < 0 \). So, we iterate over bits in the 64-bit binary representation of \(|u|\). Since \( u_{63} = u_{62} = 1 \), we write out the first two loop iterations in a compact form in lines 2-4. In line 14, \( \tilde{f} \) denotes conjugation in \( \mathbb{F}_{p^{12}} \) (requires 6 \( \mathbb{F}_p \) subtractions). This is required to compensate for \( u < 0 \), and we have followed [262] to replace expensive \( \mathbb{F}_{p^{12}} \) inversion with simple \( \mathbb{F}_{p^{12}} \) conjugation. The \( \mathbb{F}_{p^{12}} \) squaring in line 7 is performed using the formulas discussed earlier. In lines 4, 8 and 11, the \( f \cdot l_P \) multiplications can be simplified significantly due to the sparse nature of the line element \( l_P \). Note that BN curves also allow such sparse multiplications in \( \mathbb{F}_{p^{12}} \), but we cannot use them directly for BLS12-381 due to the presence of a different twist.

**Sparse Multiplication in \( \mathbb{F}_{p^{12}}/\mathbb{F}_{p^6}/\mathbb{F}_{p^2} \):** For \( f = c_0 + c_1 \gamma \) (where \( c_0, c_1 \in \mathbb{F}_{p^6} \)) and \( l_P = b_0 + (b_3 \beta + b_5 \beta^2) \gamma \) (where \( b_0, b_3, b_5 \in \mathbb{F}_{p^2} \)), the sparse multiplication \( f \cdot l_P \) in Algorithm C.6 for BLS12-381 can be written as:

\[
f \cdot l_P = (c_0 + c_1 \gamma) \cdot (b_0 + (b_3 \beta + b_5 \beta^2) \gamma) = b_0 c_0 + \beta c_1 \cdot (b_3 \beta + b_5 \beta^2) + (b_0 c_1 + c_0 \cdot (b_3 \beta + b_5 \beta^2)) \gamma
\]

To compute \( c_0 \cdot (b_3 \beta + b_5 \beta^2) \), we use the sparse \( \mathbb{F}_{p^6} \) multiplication formula discussed earlier. For \( c_1 \cdot (b_3 \beta + b_5 \beta^2) \), we again use the Karatsuba method [196] as:

\[
b_0 c_1 + c_0 \cdot (b_3 \beta + b_5 \beta^2) = (b_0 + b_3 \beta + b_5 \beta^2) \cdot (c_0 + c_1) - (b_0 c_0 + c_1 \cdot (b_3 \beta + b_5 \beta^2))
\]
to reduce the number of multiplications at the cost of extra additions. Calculating $b_0c_0$ and $b_0c_1$ involves three $\mathbb{F}_{p^2}$ multiplications each.

$\Rightarrow \text{Cost} \left(sM_{12}\right) = 3M_2 + M_6 + sM_6 + 4A_6 + m_\beta$

We developed a Python reference implementation to validate and profile our BLS12-381 formulation. Total computation cost of the BLS12-381 Miller Loop (lines 1-14 in Algorithm C.6), in terms of equivalent number of $\mathbb{F}_p$ multiplications, is $7,050M_1$ (including 63 Miller doubling and 5 Miller addition steps).

**Final Exponentiation:** The exponent $3(p^{12} - 1)/q$ is factored as [266]:

$$\frac{3(p^{12} - 1)}{q} = \left(\frac{p^6 - 1}{q}\right) \cdot \left(\frac{p^2 + 1}{q}\right)$$

As shown in Algorithm C.7, the easy part is calculated as:

$$f^{(p^6-1)(p^2+1)} = \left(f^{(p^6-1)}\right)^{(p^2+1)} = \left(f^{p^6} \cdot f^{-1}\right)^{(p^2+1)} = \left(f \cdot f^{-1}\right)^p \cdot \left(f \cdot f^{-1}\right)$$

Here, we have used the property that exponentiation by $p^6$ in $\mathbb{F}_{p^{12}}/\mathbb{F}_{p^6}$ is equivalent to conjugation. Since $p$ is the characteristic of $\mathbb{F}_{p^{12}}$, the $p$-th and $p^2$-th powers of any element in $\mathbb{F}_{p^{12}}$ can be easily computed using the Frobenius map. For any element $a = (b_0 + b_2\beta + b_4\beta^2) + (b_1 + b_3\beta + b_5\beta^2)\gamma = b_0 + b_1\gamma + b_2\gamma^2 + b_3\gamma^3 + b_4\gamma^4 + b_5\gamma^5 \in \mathbb{F}_{p^{12}}/\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$ (where $b_0, \cdots, b_5 \in \mathbb{F}_{p^2}$), its $p$-th and $p^2$-th powers are given by [197]:

- $a^p = b_0^p + b_2^p\gamma + b_4^p(\gamma^2)^p + b_2^p(\gamma^3)^p + b_4^p(\gamma^4)^p + b_5^p(\gamma^5)^p$
  
  $$= \bar{b}_0 + b_1\delta\gamma + \bar{b}_2\delta^2\gamma^2 + b_3\delta^3\gamma^3 + \bar{b}_4\delta^4\gamma^4 + b_5\delta^5\gamma^5$$
  
  where $\delta = \xi^{(p-1)/6}$ since $b^p = \bar{b} \forall b \in \mathbb{F}_{p^2}$ and $\gamma^p = (\gamma^6)^{(p-1)/6} \gamma = \delta \gamma$.

- $a^{p^2} = b_0^{p^2} + b_2^{p^2}\gamma^{p^2} + b_4^{p^2}(\gamma^2)^{p^2} + b_2^{p^2}(\gamma^3)^{p^2} + b_4^{p^2}(\gamma^4)^{p^2} + b_5^{p^2}(\gamma^5)^{p^2}$
  
  $$= b_0 + b_1\omega\gamma + b_2(\omega - 1)\gamma^2 - b_3\gamma^3 - b_4\omega\gamma^4 - b_5(\omega - 1)^2\gamma^5$$
  
  where $\omega = \xi^{(p^2-1)/6}$ since $b^{p^2} = (b^p)^p = \bar{b} = b \forall b \in \mathbb{F}_{p^2}$, $\gamma^{p^2} = (\gamma^6)^{(p^2-1)/6} \gamma = \omega \gamma$ and $\omega$ is a primitive sixth root of unity satisfying $\omega^{3} = -1$ and $\omega^{2} - \omega + 1 = 0$.

We can get further speedup by pre-computing $\delta, \cdots, \delta^5, \omega$ and $(\omega - 1)$. Then, the $p$-th and $p^2$-th power Frobenius computations cost $25M_1 + 47A_1$ and $8M_1 + 7A_1$ respectively.
Algorithm C.7 Easy part of BLS12-381 final exponentiation (adapted from [264])

Require: \( f \in \mathbb{F}_{p^{12}} \)
Ensure: \( g = f^{((p^6-1)(p^2+1))} \in G_{\Phi_6}(\mathbb{F}_{p^2}) \)

1. \( t_0 \leftarrow f^{-1} \)
2. \( t_1 \leftarrow f \)
3. \( t_0 \leftarrow t_1 \cdot t_0 \)
4. \( t_1 \leftarrow t_0^{p^2} \)
5. \( g \leftarrow t_1 \cdot t_0 \)
6. return \( g \)

Overall, the cost of Algorithm C.7, in terms of equivalent number of \( \mathbb{F}_p \) multiplications, is 821\(M_1\), as validated using our Python reference implementation.

The output of the easy part of final exponentiation is an element in \( G_{\Phi_6}(\mathbb{F}_{p^2}) \), which is a cyclotomic subgroup of \( \mathbb{F}_{p^{12}}^* \) [234]. Elements in \( G_{\Phi_6}(\mathbb{F}_{p^2}) \) have special properties which enable faster arithmetic useful in the hard part of final exponentiation. Cyclotomic inversion is equivalent to raising to the power \( p^6 \), which is the same as conjugation in \( \mathbb{F}_{p^{12}}/\mathbb{F}_{p^6} \), that is, inversion is practically free. Cyclotomic squaring is less expensive compared to \( \mathbb{F}_{p^{12}} \), as discussed next.

For any element \( a = (b_0 + b_2\beta + b_4\beta^2) + (b_1 + b_3\beta + b_5\beta^2) \gamma \in G_{\Phi_6}(\mathbb{F}_{p^2}) \) (where \( b_0, \ldots, b_5 \in \mathbb{F}_{p^2} \)), its square is computed using Granger and Scott’s method [234] as:

\[
a^2 = (B_0 + B_2\beta + B_4\beta^2) + (B_1 + B_3\beta + B_5\beta^2) \gamma \in G_{\Phi_6}(\mathbb{F}_{p^2})
\]

where \( B_0, \ldots, B_5 \in \mathbb{F}_{p^2} \) are given by [234]:

\[
\begin{align*}
B_0 &= 3(b_2^2\xi + b_0^2) - 2b_0 \\
B_2 &= 3(b_4^2\xi + b_2^2) - 2b_2 \\
B_4 &= 3(b_5^2\xi + b_4^2) - 2b_4 \\
B_1 &= 3((b_2 + b_5)^2 - (b_2^2 + b_5^2)) + 2b_1 \\
B_3 &= 3((b_0 + b_3)^2 - (b_0^2 + b_3^2)) + 2b_3 \\
B_5 &= 3((b_4 + b_1)^2 - (b_4^2 + b_1^2)) + 2b_5
\end{align*}
\]

Again, the Karatsuba method [196, 197] has been used to reduce the number of multiplications at the cost of extra additions.

\[ \Rightarrow \text{Cost } (cS_{12}) = 9S_2 + 30A_2 + 4m_\xi \]

Clearly, this is much faster than traditional \( \mathbb{F}_{p^{12}} \) squaring \((S_{12} = 3S_6 + 4A_6 + m_\beta \equiv 6M_2 + 9S_2 + 39A_2 + 7m_\xi > cS_{12})\).
The hard part of final exponentiation involves arithmetic in the cyclotomic group which has the same structure for all BLS12-based pairing maps irrespective of the associated twist curves. Therefore, we follow the exponentiation technique for BLS12 proposed in [267]. First, exponent of the hard part is written as a degree-3 polynomial in $p$ as shown below [267]:

$$3 \left( \frac{p^4 - p^2 + 1}{q} \right) = \lambda_0 + \lambda_1 p + \lambda_2 p^2 + \lambda_3 p^3$$

where

$$\lambda_0 = u^5 - 2u^4 + 2u^2 - u + 3$$
$$\lambda_1 = u^4 - 2u^3 + 2u - 1$$
$$\lambda_2 = u^3 - 2u^2 + u$$
$$\lambda_3 = u^2 - 2u + 1$$

Clearly, the factor of 3 in the exponent ensures that all four coefficients are integers, thus justifying the choice of this unconventional form of the exponent.

Next, the exponentiation is performed using a sequence of $\mathbb{F}_{p^{12}}$ multiplications, $\mathbb{F}_{p^{12}}$ Frobenius computations (of $p$-th, $p^2$-th and $p^3$-th powers), $G_{\Phi_6}(\mathbb{F}_{p^2})$ cyclotomic squarings and $G_{\Phi_6}(\mathbb{F}_{p^2})$ cyclotomic exponentiations (by $u$ and $u/2$).

Algorithm C.8 outlines our implementation of $G_{\Phi_6}(\mathbb{F}_{p^2})$ cyclotomic exponentiation by $u$, and similar approach can be used for exponentiation by $u/2$ as well. In line 3, cyclotomic squaring uses Granger-Scott method [234]. The conjugation in line 8

---

**Algorithm C.8** Cyclotomic exponentiation by $u$ for BLS12-381

**Require:** $f \in G_{\Phi_6}(\mathbb{F}_{p^2})$ and $|u| = (u_{63}, \cdots, u_1, u_0)_2$

**Ensure:** $g = f^u \in G_{\Phi_6}(\mathbb{F}_{p^2})$

1. $g \leftarrow f$
2. for $(i = 62; i \geq 0; i = i - 1)$ do
3.   $g \leftarrow g^2$ using Granger-Scott method [234]
4.   if $u_i = 1$ then
5.     $g \leftarrow g \cdot f$
6.   end if
7. end for
8. $g \leftarrow \bar{g}$
9. return $g$

192
Algorithm C.9 Hard part of BLS12-381 final exponentiation (adapted from [267])

Require: $f \in G_{\Phi_6}(\mathbb{F}_{p^2})$

Ensure: $g = f^{3(p^4-p^2+1)/q} \in G_T$

1: $t_0 \leftarrow f^2$
2: $t_1 \leftarrow t_0^{2u}$
3: $t_2 \leftarrow t_1^{u/2}$
4: $t_3 \leftarrow \bar{f}$
5: $t_1 \leftarrow t_3 \cdot t_4$
6: $t_1 \leftarrow \bar{t}_1$
7: $t_1 \leftarrow t_1 \cdot t_2$
8: $t_2 \leftarrow t_1^p$
9: $t_3 \leftarrow t_2^p$
10: $t_1 \leftarrow \bar{t}_1$
11: $t_3 \leftarrow t_3 \cdot t_4$
12: $t_2 \leftarrow t_2^p$
13: $t_1 \leftarrow t_1^p$
14: $t_2 \leftarrow t_2^p$
15: $t_1 \leftarrow t_1 \cdot t_2$
16: $t_1 \leftarrow t_1 \cdot t_2$
17: $t_2 \leftarrow t_2 \cdot t_0$
18: $t_2 \leftarrow t_2 \cdot t_0$
19: $t_2 \leftarrow t_2 \cdot f$
20: $t_1 \leftarrow t_1 \cdot t_2$
21: $t_2 \leftarrow t_2^p$
22: $g \leftarrow t_1 \cdot t_2$
23: return $g$

adjusts for $u < 0$. While $\mathbb{F}_{p^{12}}$ multiplications are used in line 5, these occur very infrequently (only when $u_i = 1$) due to the low Hamming weights of $u$ (also $u/2$). Since $u$ is a publicly known value, it is perfectly acceptable to have branching dependent on the bits of binary representation of $u$ (or $u/2$) in these exponentiations, similar to the Miller Loop. Using our Python reference implementation, the computation costs of cyclotomic exponentiation by $u$ and $u/2$, in terms of equivalent number of $\mathbb{F}_p$ multiplications, are $1,404M_1$ and $1,386M_1$ respectively.

Algorithm C.9 shows how the hard part of final exponentiation is computed for BLS12-381. The squaring in line 1 is based on Granger-Scott method [234]. The exponentiations in lines 2, 3, 8, 9 and 17 are based on Algorithm C.8. The Frobenius computations in lines 13, 14, 15 and 21 are based on the formulas discussed earlier. Overall, the cost of Algorithm C.9, in terms of equivalent number of $\mathbb{F}_p$ multiplications, is $7,518M_1$, as validated using our Python reference implementation. Therefore, the total cost of BLS12-381 pairing, including Miller Loop and Final Exponentiation, is $15,389M_1$, in terms of equivalent number of $\mathbb{F}_p$ multiplications.

To complete our analysis, we also developed and profiled Python reference implementations of pairings on the BN-254 and BN-462 curves. The pairing computation costs are tabulated next, in terms of equivalent number of $\mathbb{F}_p$ multiplications in the corresponding base prime field. The normalized cost is calculated assuming that the cost of multiplication is related to square of the prime size. We note that the pairing cost for BN-254 is very close to previous work [262, 264], thus confirming that our reference implementation is reasonably efficient and hence suitable for such comparison.
<table>
<thead>
<tr>
<th></th>
<th>BN-254</th>
<th>BLS12-381</th>
<th>BN-462</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prime Size</td>
<td>254b</td>
<td>381b</td>
<td>462b</td>
</tr>
<tr>
<td>Security Level</td>
<td>≈100-bit (not recommended)</td>
<td>≈126-bit (IETF optimistic)</td>
<td>≈134-bit (IETF conservative)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Pairing Computation Cost</strong></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Miller Loop</td>
<td>6,781M₁,254b</td>
<td>7,050M₁,381b</td>
<td>12,194M₁,462b</td>
</tr>
<tr>
<td>Final Exponentiation</td>
<td>5,094M₁,254b</td>
<td>8,339M₁,381b</td>
<td>8,439M₁,462b</td>
</tr>
<tr>
<td>Optimal Ate Pairing</td>
<td>11,875M₁,254b</td>
<td>15,389M₁,381b</td>
<td>20,633M₁,462b</td>
</tr>
<tr>
<td>Normalized Cost</td>
<td>0.34</td>
<td>1.00</td>
<td>1.97</td>
</tr>
</tbody>
</table>

Note: $M_{1,254b}$, $M_{1,381b}$ and $M_{1,462b}$ denote multiplications in respective base field ($\mathbb{F}_p$)

We observe that a pairing on the BLS12-381 curve is $\approx 3 \times$ more expensive than BN-254 and $\approx 2 \times$ less expensive than BN-462. The increase in computation cost is a direct result of higher security level. Clearly, BLS12-381 provides the right balance between efficiency and security.

**Multi-Pairing**

Multi-pairing involves evaluating the product of several pairings [177], as shown below:

$$\prod_{j=1}^{n} e(P_j, Q_j) = e(P_1, Q_1) \times e(P_2, Q_2) \times \cdots \times e(P_n, Q_n)$$

If one set of pairing inputs is shared, the multi-pairing can be simplified by sharing operations using the bilinearity property [37]:

$$\prod_{j=1}^{n} e(P, Q_j) = e(P, \sum_{j=1}^{n} Q_j) \quad \text{and} \quad \prod_{j=1}^{n} e(P_j, Q) = e(\sum_{j=1}^{n} P_j, Q)$$

so that the $n$-fold multi-pairing is reduced to just one pairing and $n - 1$ point additions (in $G_2$ and $G_1$ respectively). This is a significant saving in computation cost due to the elimination of $n - 1$ pairings and $n - 1$ multiplications in $G_T$.

For the general case (without any shared inputs), the multi-pairing can be improved by sharing Miller Loop and Final Exponentiation computations across multiple pairing...
instances. In Algorithm C.6, the Miller Loop updates the accumulator $f$ as $f \leftarrow f^2 \cdot l(P)$, where $l(P)$ is the evaluated line. In case of multi-pairing, where $n$ such lines are evaluated, the accumulator can be updated as $f \leftarrow f^2 \cdot \prod_{j=1}^{n} l(P_j)$. This ensures that the accumulator is squared only once per iteration of the Miller Loop instead of $n$ times [268]. Furthermore, only one Final Exponentiation needs to be used to map the output of this shared Miller Loop to the target group $G_T$ [37]. This is outlined in Algorithm C.10 for BLS12-381.

Algorithm C.10 Multi-pairing for BLS12-381 (adapted from [37,268])

Require: $P_1, P_2, \cdots, P_n \in G_1$, $Q_1, Q_2, \cdots, Q_n \in G_2$ and $|u| = (u_{63}, \cdots, u_1, u_0)_2$

Ensure: $f = \prod_{j=1}^{n} e(P_j, Q_j) \in G_T$

1: $f \leftarrow 1$
2: for $(j = 1; j \leq n; j = j + 1)$ do
3: $T_j \leftarrow Q_j$
4: end for
5: for $(i = 62; i \geq 0; i = i - 1)$ do
6: if $i < 62$ then
7: $f \leftarrow f^2$
8: end if
9: for $(j = 1; j \leq n; j = j + 1)$ do
10: $T_j \leftarrow 2T_j$ and $f \leftarrow f \cdot l_{T_j,T_j}(P_j)$
11: if $u_i = 1$ then
12: $T_j \leftarrow T_j + Q_j$ and $f \leftarrow f \cdot l_{T_j,Q_j}(P_j)$
13: end if
14: end for
15: end for
16: $f \leftarrow \bar{f}$
17: $f \leftarrow f^{3(p^{12}-1)/q}$
18: return $f$
Appendix D

Test Chip Design and Validation

Pin Description: All three test chips discussed in Chapters 2, 3 and 5 have the same external interface and follow very similar pinout. Their input/output pins and functionality are listed below:

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Type</th>
<th>Description</th>
<th>Abbrev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDD_CORE</td>
<td>PG</td>
<td>Core supply voltage</td>
<td>VC</td>
</tr>
<tr>
<td>VDD_IO</td>
<td>PG</td>
<td>I/O driver supply voltage</td>
<td>VP</td>
</tr>
<tr>
<td>VSS</td>
<td>PG</td>
<td>Shared ground for both core and I/O</td>
<td>VS</td>
</tr>
<tr>
<td>CLK</td>
<td>I</td>
<td>System clock</td>
<td>CK</td>
</tr>
<tr>
<td>RST_N</td>
<td>I</td>
<td>System reset (active-low)</td>
<td>RS</td>
</tr>
<tr>
<td>GPIO [15:0]</td>
<td>IO</td>
<td>GPIO input/output</td>
<td>G0-Gf</td>
</tr>
<tr>
<td>UART_TX</td>
<td>O</td>
<td>UART output</td>
<td>UT</td>
</tr>
<tr>
<td>UART_RX</td>
<td>I</td>
<td>UART input</td>
<td>UR</td>
</tr>
<tr>
<td>SPI_SCK</td>
<td>O</td>
<td>SPI clock</td>
<td>SK</td>
</tr>
<tr>
<td>SPI_SSB</td>
<td>O</td>
<td>SPI peripheral selection (active-low)</td>
<td>SS</td>
</tr>
<tr>
<td>SPI_MOSI</td>
<td>O</td>
<td>SPI controller-to-peripheral data</td>
<td>MO</td>
</tr>
<tr>
<td>SPI_MISO</td>
<td>I</td>
<td>SPI peripheral-to-controller data</td>
<td>MI</td>
</tr>
<tr>
<td>SD_CLK</td>
<td>O</td>
<td>SD clock</td>
<td>DK</td>
</tr>
<tr>
<td>SD_CFG [2:0]</td>
<td>I</td>
<td>SD clock divider configuration</td>
<td>C0-C2</td>
</tr>
<tr>
<td>SD_CMD</td>
<td>IO</td>
<td>SD command</td>
<td>CD</td>
</tr>
<tr>
<td>SD_DATA [3:0]</td>
<td>IO</td>
<td>SD data</td>
<td>D0-D3</td>
</tr>
</tbody>
</table>

† PG: power-ground, I: input, O: output, IO: inout
The core supply voltage is 0.8-1.2 V for the DTLS chip (Chapter 2), 0.68-1.1 V for the lattice chip (Chapter 3) and 0.66-1.1 V for the pairing chip (Chapter 5). The I/O driver supply voltage is 2.5 V for the DTLS chip and 3.3 V for the lattice and pairing chips. Peripherals supported by the chips are GPIO (general purpose input-output), UART (universal asynchronous receiver-transmitter) and SPI (serial peripheral interface). An SD (secure digital) interface is used to load programs into the on-chip RISC-V processor’s instruction and data memory.

**GPIO:** The sixteen GPIO pins can be accessed through RISC-V software using the `gpio_port` register at memory location `0xAAAB_0000`. The pins can be individually configured as input or output using the `gpio_ddr` data direction register at memory location `0xAAAB_0004`.

**UART:** The UART output (TX) and input (RX) can be provided through RISC-V software using the `uart_tx` and `uart_rx` registers at memory locations `0x2001_0000` and `0x2001_0004` respectively. Depending on the desired baud rate, the UART clock divider can be set using the the `uart_div` register at memory location `0x2001_0010`.

**SPI:** The SPI controller interface can be accessed through RISC-V software using the `spi_mosi`, `spi_miso` and `spi_enable` registers at memory locations `0x2002_0000`, `0x2002_0004` and `0x2002_0008` respectively. The SPI clock divider can be set using the `spi_div` register at memory location `0x2002_000C`.

**SD:** The SD card interface is used in the 4-bit wide data bus mode where the SD_CMD pin is used for commands and the four SD_DATA pins are used for data transfer 4 bits at a time. The SD clock is derived from the system clock using a logarithmic clock divider. The divider can be configured externally using the three SD_CFG pins. For 3-bit configuration value $D$, the SD clock frequency is $f_{CLK} / 2^D$, where $f_{CLK}$ is the system clock frequency.
FPGA Setup for Pre-Silicon Validation: All three designs discussed in Chapters 2, 3 and 5 were tested using a Digilent Nexys Video board [269] for verification prior to chip tape-out. The board contains an Artix XC7A200T-1 FPGA (with 33,650 logic slices, 365 BRAMs and 740 DSP slices; each logic slice has four 6-input LUTs and 8 flip-flops; each BRAM is 36 Kb) and the designs were synthesized and implemented using Xilinx Vivado. The FPGA setup is shown below:

The following schematic shows how our chip designs were emulated using FPGA:

The 100 MHz external clock was scaled appropriately using a mixed-mode clock manager (MMCM) to meet the timing constraints of each design.
**Test Setup for Post-Silicon Validation:** The test chips were housed in a QFN64 socket [270] soldered to a custom-designed printed circuit board. An Opal Kelly XEM7001 board [69], containing an Artix XC7A15T-1 FPGA (with 2,600 logic slices, 25 BRAMs and 45 DSP slices; each logic slice has four 6-input LUTs and 8 flip-flops; each BRAM is 36 Kb), was used to interface with the test chips. A Keithley 2602A source meter [70] was used to supply power (both core and I/O) to the chip. The FPGA board and the source meter were both controlled from a host computer through USB and GPIB interfaces respectively.

The test setup schematic is shown below. The SD card containing the program (both data and instructions) was emulated on the FPGA, with the memory emulated using BRAM and the control circuitry emulated using logic cells and in-out buffers. Clocking and reset logic was also implemented on the FPGA. GPIO[15] was used as a software-controlled debug signal output connected to the FPGA clocking logic, with an active-high pulse gating the chip clock on the next cycle. GPIO[7:0] were connected to LEDs on the test board for debug purposes. The FPGA portion of the test setup was synthesized and implemented using Xilinx Vivado.
Programming the RISC-V with Hardware Accelerators: All RISC-V software is written in C and compiled using the riscv-gcc toolchain [147]. The software is implemented bare-metal, that is, without any operating system. The low-level driver software library from [43] is used for handling exceptions, controlling interrupts, reading different performance counters, accessing the internal real-time clock, printing characters through UART and interfacing with other peripherals such as SPI and GPIO. The in-built hardware performance counters are used to profile total number of instructions executed and data memory usage. The hardware accelerators are accessed at the RISC-V software-level through a memory-mapped interface.

The memory-mapped locations are read from and written to using pointers. Reading from these memory-mapped locations are compiled into load instructions, while writing to them are compiled into store instructions. Since the RISC-V RV32 is a 32-bit architecture, ⌈N/32⌉ load / store accesses are required when the memory-mapped location corresponds to N bits of data. For example, the 256-bit registers in the DTLS cryptographic engine are accessed using the following two functions:

```c
void write_reg ( unsigned long *addr, unsigned long *data ) {
    *(addr ) = data[0];
    *(addr+1) = data[1];
    *(addr+2) = data[2];
    *(addr+3) = data[3];
    *(addr+4) = data[4];
    *(addr+5) = data[5];
    *(addr+6) = data[6];
    *(addr+7) = data[7];
}

void read_reg ( unsigned long *addr, unsigned long *data ) {
    data[0] = *(addr );
    data[1] = *(addr+1);
    data[2] = *(addr+2);
    data[3] = *(addr+3);
    data[4] = *(addr+4);
    data[5] = *(addr+5);
    data[6] = *(addr+6);
    data[7] = *(addr+7);
}
```
data[4] = *(addr+4);
data[5] = *(addr+5);
data[6] = *(addr+6);
data[7] = *(addr+7);
}

Similarly, the $256 \times 32$-bit instruction memories in both the lattice crypto-processor and the pairing crypto-processor are loaded using the following function:

```c
void write_imem ( unsigned long *data, int instr_count ) {
    for (int i = 0; i < instr_count; i++) {
        *(imem + i) = data[i];
    }
}
```

The RISC-V processor’s internal real-time clock (a software-controllable free-running 64-bit counter) is used to measure execution timings. For example, in order to measure the execution time of a function `foo` averaged over 100 iterations, the following code snippet is used:

```c
uint64_t time_start, time_end, time_avg = 0;
for (int i = 0; i < 100; i++) {
    time_start = getTime();
    foo(···);
    time_end = getTime();
    time_avg += (time_end - time_start);
}

time_avg /= 100;
```

Here, `getTime()` is a function in the driver library which returns the value of the 64-bit counter. The control overhead has negligible impact on cycle count.

Both the lattice crypto-processor and the pairing crypto-processor are designed with their respective custom instruction sets (please refer to Appendix E for further
These custom instructions are used to write assembly-style programs to accelerate cryptographic algorithms, and they are translated into a C header file using a Perl or Python script. The header file contains an array of 32-bit instruction words corresponding to the program. This array is then written to the instruction memory using the `write_imem` function described earlier. For example:

```c
write_imem (crypto_prog, crypto_prog_len);
```

loads the program stored in the array `crypto_prog` containing `crypto_prog_len` instructions. The header file is then combined with the top-level program and functions which interface with the crypto-processor, together compiled to a binary using `riscv-gcc` compiler, as shown below:
Appendix E

Crypto-Processor Programming

Lattice Crypto-Processor Programming

Here, we briefly describe all the custom instructions supported by our configurable lattice crypto-processor (from Chapter 3). Apart from the polynomials stored in its memory and the 256-bit seed registers r0 and r1, these are the core internal registers that can also be manipulated: 24-bit temporary registers reg and tmp; 16-bit counter registers c0 and c1; and 2-bit flag register to store comparison results (-1, 0 or +1). Most polynomial arithmetic instructions specify inputs and outputs using 7-bit addresses poly, poly_src and poly_dst in the polynomial memory, appropriately chosen based on the configured value of polynomial size n. Following is the list of instructions along with short descriptions:

<table>
<thead>
<tr>
<th>Configuration: set parameters and clock gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>config (n, q)</td>
</tr>
<tr>
<td>clock_config (keccak, ntt, sampler)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Operations: register assignments and arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0 = #VAL / c0 + #VAL / c0 - #VAL</td>
</tr>
<tr>
<td>c1 = #VAL / c1 + #VAL / c1 - #VAL</td>
</tr>
<tr>
<td>reg = #VAL / tmp</td>
</tr>
<tr>
<td>tmp = #VAL / tmp (OP) reg</td>
</tr>
</tbody>
</table>

where #VAL can be any unsigned integer of appropriate size, and (OP) is one of the following operations: {ADD, SUB, MUL, AND, OR, XOR, RSHIFT, LSHIFT}
### Register-Polynomial Operations: register and polynomial interactions

- \( \text{reg} = \max_{\text{elems}} \text{(poly)} \)
- \( \text{reg} = \sum_{\text{elems}} \text{(poly)} \)
- \( \text{reg} = (\text{poly})[\#\text{VAL}] / (\text{poly})[c0] / (\text{poly})[c1] \)
- \((\text{poly})[\#\text{VAL}] / (\text{poly})[c0] / (\text{poly})[c1] = \text{reg}\)

### Transforms: number theoretic transform and related computations

- \( \text{transform (mode, poly}_{-\text{dst}}, poly_{-\text{src}}) \)
- \( \text{mult}_{-\text{psi}} \text{(poly)} \)
- \( \text{mult}_{-\text{psi}_{\text{inv}}} \text{(poly)} \)

Where \( \text{mode} \) is one of the following: \{DIF\text{\_NTT}, DIF\text{\_INTT}, DIT\text{\_NTT}, DIT\text{\_INTT}\}

### Sampling: polynomial sampling from various distributions

- \( \text{bin}_{-\text{sample}} \text{(prng, seed, c0, c1, k, poly)} \)
- \( \text{cdt}_{-\text{sample}} \text{(prng, seed, c0, c1, r, s, poly)} \)
- \( \text{rej}_{-\text{sample}} \text{(prng, seed, c0, c1, poly)} \)
- \( \text{uni}_{-\text{sample}} \text{(prng, seed, c0, c1, eta, bitlen, poly)} \)
- \( \text{tri}_{-\text{sample}_{\text{1}}} \text{(prng, seed, c0, c1, m, poly)} \)
- \( \text{tri}_{-\text{sample}_{\text{2}}} \text{(prng, seed, c0, c1, m0, m1, poly)} \)
- \( \text{tri}_{-\text{sample}_{\text{3}}} \text{(prng, seed, c0, c1, rho, poly)} \)

Where \( \text{prng} \) can be SHAKE-128 or SHAKE-256, \( \text{seed} \) can be \( r0 \) or \( r1 \), and \( k, r, s, \text{eta}, \text{bitlen, m, m0, m1, rho} \) are the distribution parameters

### Polynomial Computations: polynomial initialization and other operations

- \( \text{init} \text{(poly)} \)
- \( \text{poly}_{-\text{copy}} \text{(poly}_{-\text{dst}}, poly_{-\text{src}}) \)
- \( \text{poly}_{-\text{op}} \text{(op, poly}_{-\text{dst}}, poly_{-\text{src}}) \)
- \( \text{shift}_{-\text{poly}} \text{(ring, poly}_{-\text{dst}}, poly_{-\text{src}}) \)

Where \( \text{op} \) can be one of the following: \{ADD, SUB, MUL, BITREV, CONST\_ADD, CONST\_SUB, CONST\_MUL, CONST\_AND, CONST\_OR, CONST\_XOR, CONST\_RSHIFT, CONST\_LSHIFT\}, and \( \text{ring} \) can be either \( x^N+1 \) or \( x^N-1 \)

### Comparison and Branching: simple branching operations

- \( \text{flag = eq}_{-\text{check}} \text{(poly, poly)} \)
- \( \text{flag = inf}_{-\text{norm}_{-\text{check}}} \text{(poly, bound)} \)
- \( \text{flag = compare} \text{(reg / tmp / c0 / c1, #VAL)} \)

If \( \text{flag} == / != -1 / 0 / +1 \) goto \langle \text{label} \rangle

Where the \( \text{flag} \) register stores -1, 0 and +1 for the register comparison result being “lesser than”, “equal to” and “greater than” respectively, and it stores 1 or 0 depending on whether the equality check and infinity norm check has passed or failed respectively

### Miscellaneous: other instructions

- \( \text{nop} \) for no operation
- \( \text{end} \) indicates end of program

---

206
SHA-3 Computations: hashing operations

sha3_init
sha3_256_absorb (poly)
sha3_512_absorb (poly)
sha3_256_absorb (r0 / r1)
sha3_512_absorb (r0 / r1)
r0 / r1 = sha3_256_digest
r0 || r1 = sha3_512_digest

where the seed registers are used to store the hash outputs – either r0 or r1 for SHA-3-256, and both r0 and r1 together for SHA-3-512

Next, some example code snippets are provided to show our implementation of lattice cryptography computations using this custom instruction set.

Ring-LWE: The following instructions generate (sample) the polynomials $a, s, e \in \mathcal{R}_q$ and calculate $a \cdot s + e$, which is a typical computation in NewHope-1024 [115]:

```plaintext
config (n = 1024, q = 12289)
# sample_a
rej_sample (prng = SHAKE-128, seed = r0, c0 = 0, c1 = 0, poly = 0)
# sample_s, sample_e
bin_sample (prng = SHAKE-256, seed = r1, c0 = 0, c1 = 0, k = 8, poly = 1)
bin_sample (prng = SHAKE-256, seed = r1, c0 = 0, c1 = 1, k = 8, poly = 2)
# ntt_s
mult_psi (poly = 1)
transform (mode = DIF_NTT, poly_dst = 4, poly_src = 1)
# a_mul_s
poly_op (op = MUL, poly_dst = 0, poly_src = 4)
# intt_a_mul_s
transform (mode = DIT_INTT, poly_dst = 5, poly_src = 0)
mult_psi_inv (poly = 5)
# a_mul_s_plus_e
poly_op (op = ADD, poly_dst = 1, poly_src = 5)
```
The `config` instruction is first used to configure the protocol parameters \( n \) and \( q \) which, in this example, are the parameters from NewHope-1024. For \( n = 1024 \), the polynomial memory is divided into 8 polynomials, which are accessed using the `poly` argument in all instructions. For sampling, the seed can be chosen from a pair of 256-bit registers \( r0 \) and \( r1 \), while two 16-bit registers \( c0 \) and \( c1 \) are used as counters for sampling multiple polynomials from the same seed. For coefficient-wise operations `poly_op`, the `poly_src` argument indicates the first source polynomial while the `poly_dst` argument denotes the second source (and destination) polynomial.

**Module-LWE:** Similarly, the following instructions are used to generate matrix of polynomials \( A \in \mathbb{R}_{q}^{2 \times 2} \) and vectors of polynomials \( s, e \in \mathbb{R}_{q}^{2} \), and calculate \( A \cdot s + e \), which is a typical computation in CRYSTALS-Kyber-v1-512 [117]:

```plaintext
config (n = 256, q = 7681)

# sample_s
bin_sample (prng = SHAKE-256, seed = r1, c0 = 0, c1 = 0, k = 3, poly = 4)
binsample (prng = SHAKE-256, seed = r1, c0 = 0, c1 = 1, k = 3, poly = 5)

# sample_e
bin_sample (prng = SHAKE-256, seed = r1, c0 = 0, c1 = 2, k = 3, poly = 24)
bin_sample (prng = SHAKE-256, seed = r1, c0 = 0, c1 = 3, k = 3, poly = 25)

# ntt_s
mul_psi (poly = 4)
transform (mode = DIF_NTT, poly_dst = 16, poly_src = 4)
mul_psi (poly = 5)
transform (mode = DIF_NTT, poly_dst = 17, poly_src = 5)

# sample_A0
rej_sample (prng = SHAKE-128, seed = r0, c0 = 0, c1 = 0, poly = 0)
rej_sample (prng = SHAKE-128, seed = r0, c0 = 1, c1 = 0, poly = 1)

# A0_mul_s
poly_op (op = MUL, poly_dst = 0, poly_src = 16)
poly_op (op = MUL, poly_dst = 1, poly_src = 17)
```

208
init (poly = 20)
poly_op (op = ADD, poly_dst = 20, poly_src = 0)
poly_op (op = ADD, poly_dst = 20, poly_src = 1)

# sample_A1
rej_sample (prng = SHAKE-128, seed = r0, c0 = 0, c1 = 1, poly = 0)
rej_sample (prng = SHAKE-128, seed = r0, c0 = 1, c1 = 1, poly = 1)

# A1_mul_s
poly_op (op = MUL, poly_dst = 0, poly_src = 16)
poly_op (op = MUL, poly_dst = 1, poly_src = 17)
init (poly = 21)
poly_op (op = ADD, poly_dst = 21, poly_src = 0)
poly_op (op = ADD, poly_dst = 21, poly_src = 1)

# intt_A_mul_s
transform (mode = DIT_INTT, poly_dst = 8, poly_src = 20)
mult_psi_inv (poly = 8)
transform (mode = DIT_INTT, poly_dst = 9, poly_src = 21)
mult_psi_inv (poly = 9)

# A_mul_s_plus_e
poly_op (op = ADD, poly_dst = 24, poly_src = 8)
poly_op (op = ADD, poly_dst = 25, poly_src = 9)

For $n = 256$, the polynomial memory is divided into 32 polynomials, which are again accessed using the poly argument. The init instruction is used to initialize a specified polynomial with all zero coefficients. The matrix $A$ is generated one row at a time, following a just-in-time approach [149] instead of generating and storing all the rows together, to save memory.

**LWE:** Unlike Ring-LWE and Module-LWE, our crypto-processor doesn’t directly support LWE computations. Therefore, we use hardware-software co-design with the crypto core and RISC-V, as discussed next. Frodo [114] involves three large matrix multiplications: $AS$, $S'A$ and $S'B$, where $A$, $S$, $S'$ and $B$ have dimensions $n \times n$, $n \times \bar{n}$,
\(\bar{m} \times n\) and \(n \times \bar{n}\) respectively with \(n \in \{640, 976, 1344\}\) and \(\bar{m} = \bar{n} = 8\). We ensure that \(S'\) is stored in row-major form and \(B\) is stored in column-major form, which simplifies calculating \(S'B\) using the schoolbook matrix multiplication technique. The \texttt{poly\_op} instruction is used to coefficient-wise multiply a row of the multiplier matrix with a column of the multiplicand matrix, and the \texttt{sum\_elems} instruction computes the sum of its elements to generate one element of the output matrix. For calculating the matrix \(AS\), we generate \(A\) in row-major form (using rejection sampling, with zero chance of rejection since \(q\) is a power of two) and \(S\) in column major form (using CDT-based discrete Gaussian sampling) so that the same techniques still work. For \(n \in \{640, 976\}\), the matrix \(S\) is generated two columns at a time to reduce the number of outer loop iterations, as illustrated in the pseudo-code below:

```c
#if (n == 1344)
for (j = 0; j < nbar; j = j + 1) {
#else
for (j = 0; j < nbar/2; j = j + 2) {
#endif
  cdt_sample (prng = SHAKE-256, seed = r1, ..., poly = 0)
  #if (n != 1344)
  cdt_sample (prng = SHAKE-256, seed = r1, ..., poly = 1)
  #endif
  for (i = 0; i < n; i = i + 1) {
    rej_sample (prng = SHAKE-128, seed = r0, ..., poly = 4)
    #if (n != 1344)
    poly_copy (poly_dst = 5, poly_src = 4)
    #endif
    poly_op (op = MUL, poly_dst = 4, poly_src = 0)
    AS[i][j] = sum_elems (poly = 4)
    #if (n != 1344)
    poly_op (op = MUL, poly_dst = 5, poly_src = 1)
  ```
AS[i][j+1] = sum_elems (poly = 5)
#endif
}
}

Since both matrices $S'$ and $A$ are generated on-the-fly in row-major fashion, this makes calculating $S'A$ a bit complicated. We multiply each element of the $i$-th row of $A$ with the $i$-th element of the $j$-th row of $S'$ to generate a partial sum. These $i$ partial sums are incrementally added together to compute the $j$-th row of the output matrix $S'A$. For $n \in \{640, 976\}$, we generate $S$ two columns at a time to reduce the number of outer loop iterations. The corresponding pseudo-code is shown below:

```c
#if (n == 1344)
for (j = 0; j < nbar; j = j + 1) {
#else
for (j = 0; j < nbar/2; j = j + 2) {
#endif
    cdt_sample (prng = SHAKE-256, seed = r1, ..., poly = 0)
    init (poly = 6)
    #if (n != 1344)
    cdt_sample (prng = SHAKE-256, seed = r1, ..., poly = 1)
    init (poly = 7)
    #endif
    for (i = 0; i < n; i = i + 1) {
        rej_sample (prng = SHAKE-128, seed = r0, ..., poly = 4)
        reg = (poly = 0)[i]
        poly_op (op = CONST_MUL, poly_dst = 2, poly_src = 4)
        poly_op (op = ADD, poly_dst = 6, poly_src = 2)
        #if (n != 1344)
        reg = (poly = 1)[i]
```
where the \( \text{reg} = (\text{poly})[i] \) instruction is used to save the \( i \)-th element of the array in the register \( \text{reg} \), the \text{init} \( (\text{poly}) \) instruction creates an array of zeros and the \text{CONST_MUL} \) operation multiplies each element of an array with the value in \( \text{reg} \).

**Module-LWE:** For CRYSTALS-Kyber-v2-512 [117] (discussed in Section 3.5.3), given polynomials \( f, g \in \mathbb{Z}_{3329}[x]/(x^{256} + 1) \), the following set of instructions compute \( p = f \cdot g \) where \( \hat{f} = (\hat{f}_{\text{even}}, \hat{f}_{\text{odd}}) \) is already available in the transform domain \( (\hat{f}_{\text{even}}, \hat{f}_{\text{odd}}, g_{\text{even}}, g_{\text{odd}}, p_{\text{even}}, p_{\text{odd}}) \) stored in locations 0, 32, 1, 33, 32, 36 respectively in the polynomial memory and the \text{shift_poly} \) instruction is used to compute \( g_{\text{odd}} \):

```
config (n = 128, q = 3329)
# lptntt_g = (ntt_g_even, ntt_g_odd) and ntt_shift_g_odd
shift_poly (ring = x^N+1, poly_dst = 31, poly_src = 33)
mult_psi (poly = 1)
transform (mode = DIF_NTT, poly_dst = 34, poly_src = 1)
mult_psi (poly = 33)
transform (mode = DIF_NTT, poly_dst = 2, poly_src = 33)
mult_psi (poly = 31)
transform (mode = DIF_NTT, poly_dst = 35, poly_src = 31)
poly_copy (poly_dst = 3, poly_src = 35)
# f_mul_g
poly_copy (poly_dst = 36, poly_src = 0)
poly_copy (poly_dst = 4, poly_src = 32)
poly_op (op = MUL, poly_dst = 0, poly_src = 34)
poly_op (op = MUL, poly_dst = 32, poly_src = 3)
```
poly_op (op = ADD, poly_dst = 0, poly_src = 32)
poly_op (op = MUL, poly_dst = 4, poly_src = 34)
poly_op (op = MUL, poly_dst = 36, poly_src = 2)
poly_op (op = ADD, poly_dst = 4, poly_src = 36)

#### 1ptnttinv_f_mul_g
transform (mode = DIT_INTT, poly_dst = 32, poly_src = 0)
mult_psi_inv (poly = 32)
transform (mode = DIT_INTT, poly_dst = 36, poly_src = 4)
mult_psi_inv (poly = 36)

Apart from the computations tabulated in Section 3.5.3, this implementation also involves poly_copy operations which require a small but finite number of cycles.

### Pairing Crypto-Processor Programming

Here, we briefly describe all the custom instructions supported by our BLS12-381 pairing crypto-processor (from Chapter 5). Apart from prime field elements stored in data memory, these are the core internal registers that can also be manipulated: two 256-bit scalar registers $r_0$ and $r_1$, two 16-bit counter registers $i$ and $j$, and a 1-bit flag register to store comparison results (0 or 1). Most arithmetic instructions are of the form instr dst, src1, src2 or instr dst, src, where src, src1, src2 and dst are 8-bit addresses in memory $M_2$ indicating the input and output operands. Note that locations in memories $M_0$ and $M_1$ are internal to the crypto-processor. Following is the list of instructions along with their short descriptions:

<table>
<thead>
<tr>
<th>Configuration:</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_mod_p / set_mod_q</td>
</tr>
<tr>
<td>to set prime field modulus to $p / q$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = #VAL / i + #VAL / i - #VAL$</td>
</tr>
<tr>
<td>$j = #VAL / j + #VAL / j - #VAL$</td>
</tr>
<tr>
<td>register assignments and arithmetic, where $#VAL$ can be any 16-bit unsigned integer</td>
</tr>
</tbody>
</table>
### Register-Memory Operations:

- `load_scalar r0 / r1, src`
  
  to load scalar registers from memory locations

### Memory Operations:

- `set dst, zero / one / m_one_p / m_one_q`
  
  to set memory location to 0 / 1 / \( R \mod p \) / \( R \mod q \) (Montgomery constants)

- `copy dst, src` and `multicopy dst, src, \#NUM`
  
  to copy data across memory locations, where \( \#NUM \in \{1, 2, \cdots, 16\} \) indicates the number of consecutive locations to be copied

- `to_mont / from_mont dst, src`
  
  to perform conversion to and from Montgomery domain

---

### \( \mathbb{F}_p \) Arithmetic:

- `fp_add / fp_sub / fp_mul dst, src1, src2`
  
  for modular addition / subtraction / multiplication

- `fp_add_one / fp_neg / fp_sqr / fp_inv / fp_sqrt / fp_invsqrt dst, src`
  
  to compute \( (x + 1) / (-x) / x^2 / x^{-1} \mod p \) (resp. \( q \)) and \( x^{(p+1)/4} / x^{(p-3)/4} \mod p \)

- `fp_fr2_mul / fp_fr3_mul dst, src`
  
  to multiply with \( 2\delta^2 / 2\delta^3 \) (see Section 6.2.2)

- `fp_omega_mul / fp_omegam1_mul dst, src`
  
  to multiply with \( \omega / \omega - 1 \) (see Appendix C)

- `fp_ha_mul / fp_hb_mul / fp_hc_mul / fp_hat11_mul dst, src`
  
  to multiply with \( A_s / B_s / (-11)^{3/2} \mod p / (11 A_s) \mod p \) (see Section 5.2.4)

- `fp_jdt2_mul dst, src`
  
  to multiply with \( 2d = -2(10240/10241) \mod q \) (see Section 5.2.5)

- `fp_small_scalar_mul dst, src, \#SCALAR`
  
  to multiply with small scalar \( \#SCALAR \in \{3, 4, \cdots, 63\} \)

---

### \( \mathbb{F}_{p^2} \) Arithmetic:

- `fp2_add / fp2_sub / fp2_mul dst, src1, src2`
  
  for addition / subtraction / multiplication in \( \mathbb{F}_{p^2} \)

- `fp2_neg / fp2_sqr / fp2_inv / fp2_conj dst, src`
  
  for negation / squaring / inversion / conjugation in \( \mathbb{F}_{p^2} \)

- `fp2_xi_mul / fp2_fourxi_mul / fp2_invxi_mul dst, src`
  
  to multiply with \( \xi / 4\xi / \xi^{-1} \) (see Appendix C)

- `fp2_delta_mul / fp2_delta2_mul / fp2_delta3_mul / fp2_delta4_mul / fp2_delta5_mul dst, src`
  
  to multiply with \( \delta / \delta^2 / \delta^3 / \delta^4 / \delta^5 \) (see Appendix C)

- `fp2_small_scalar_mul dst, src, \#SCALAR`
  
  to multiply with small scalar \( \#SCALAR \in \{3, 4, \cdots, 63\} \)
<table>
<thead>
<tr>
<th>Arithmetic:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{F}_p^6 )</td>
</tr>
<tr>
<td>( \text{fp6_add / fp6_sub / fp6_mul \ dst, src1, src2} )</td>
</tr>
<tr>
<td>for addition / subtraction / multiplication in ( \mathbb{F}_p^6 )</td>
</tr>
<tr>
<td>( \text{fp6_neg / fp6_sqr / fp6_inv \ dst, src} )</td>
</tr>
<tr>
<td>for negation / squaring / inversion in ( \mathbb{F}_p^6 )</td>
</tr>
<tr>
<td>( \text{fp6_sparse_mul \ dst, src1, src2} )</td>
</tr>
<tr>
<td>for sparse multiplication (see Appendix C)</td>
</tr>
<tr>
<td>( \text{fp6_beta_mul \ dst, src} )</td>
</tr>
<tr>
<td>to multiply with ( \beta ) (see Appendix C)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{F}_{p^{12}} ) and ( \mathbb{G}_T )</td>
</tr>
<tr>
<td>( \text{fp12_add / fp12_sub / fp12_mul \ dst, src1, src2} )</td>
</tr>
<tr>
<td>for addition / subtraction / multiplication in ( \mathbb{F}_{p^{12}} )</td>
</tr>
<tr>
<td>( \text{fp12_sqr / fp12_inv / fp12_conj \ dst, src} )</td>
</tr>
<tr>
<td>for squaring / inversion / conjugation in ( \mathbb{F}_{p^{12}} )</td>
</tr>
<tr>
<td>( \text{fp12_sparse_mul \ dst, src1, src2} )</td>
</tr>
<tr>
<td>for sparse multiplication (see Appendix C)</td>
</tr>
<tr>
<td>( \text{fp12_frobenius_p1 / fp12_frobenius_p2 \ dst, src} )</td>
</tr>
<tr>
<td>for ( p )-th / ( p^2 )-th power Frobenius computation (see Appendix C)</td>
</tr>
<tr>
<td>( \text{fp12_cyclo_sqr \ dst, src} )</td>
</tr>
<tr>
<td>for cyclotomic squaring in ( \mathbb{G}_T )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{G}_1 ) Curve</td>
</tr>
<tr>
<td>( \text{ecp_dbl \ dst, src} ) and ( \text{ecp_add \ dst, src1, src2} )</td>
</tr>
<tr>
<td>for point doubling and addition in ( \mathbb{G}_1 ) (see Algorithms C.1 and C.2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{G}_2 ) Curve</td>
</tr>
<tr>
<td>( \text{ecp2_dbl \ dst, src} ) and ( \text{ecp2_add \ dst, src1, src2} )</td>
</tr>
<tr>
<td>for point doubling and addition in ( \mathbb{G}_2 ) (see Algorithms C.1 and C.2)</td>
</tr>
<tr>
<td>( \text{ecp2_frobenius_1 / ecp2_frobenius_1 \ dst, src} )</td>
</tr>
<tr>
<td>to compute skew Frobenius maps ( \hat{\phi}(P) / \hat{\phi}(\hat{\phi}(P)) ) in ( \mathbb{G}_2 ) (see Section 6.2.2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miller Line</td>
</tr>
<tr>
<td>( \text{miller_add / miller_dbl \ dst, src1, src2} )</td>
</tr>
<tr>
<td>for Miller addition / doubling steps (see Algorithms C.4 and C.5)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Map:</td>
</tr>
<tr>
<td>( \text{hash_map \ dst, src} )</td>
</tr>
<tr>
<td>to transform ( \mathbb{F}_p ) element into point on ( E_s ) (see Section 5.2.4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jubjub Curve</td>
</tr>
<tr>
<td>( \text{jubjub_add_proj / jubjub_add_mix \ dst, src1, src2} )</td>
</tr>
<tr>
<td>for extended projective / mixed coordinate point addition on Jubjub (see Section 5.2.5)</td>
</tr>
</tbody>
</table>
Comparison and Branching:

- **check_zero**
  - Sets flag to 1 (or 0) if register $z$ in the modular arithmetic unit is zero (or not).

- **check i == / < / > #VAL**
  - **check j == / < / > #VAL**
  - Sets flag to 1 (or 0) if the condition is satisfied (or not), where #VAL can be any 16-bit unsigned integer.

- **check r0[i] / r1[i] / r0[j] / r1[j] == 0 / 1**
  - **check r0[i] == 0 / 1 && r1[i] == 0 / 1**
  - **check r0[j] == 0 / 1 && r1[j] == 0 / 1**
  - Sets flag to 1 (or 0) if the condition is satisfied (or not).

- **cmov dst, src1, src2**
  - Moves data from memory location src1 or src2 to dst (in constant-time) depending on whether register $z$ in the modular arithmetic unit is zero or non-zero.

- **cjump +/-#VAL1, +/-#VAL0**
  - Jumps to instruction located at +/-#VAL0 or +/-#VAL1 relative to current instruction depending on whether flag is 0 or 1, where #VAL0 and #VAL1 are 8-bit unsigned integers.

- **jump +/-#VAL**
  - Jumps unconditionally to instruction located at +/-#VAL relative to current instruction, where #VAL is an 8-bit unsigned integer.

Miscellaneous:

- **nop** for no operation

- **end** indicates end of program

Next, some example code snippets are provided to show our implementation of elliptic curve and pairing-related computations using this custom instruction set.

**SPA-Secure ECSM in $\mathbb{G}_1$:** This code snippet performs constant-time ECSM in $\mathbb{G}_1$ according to Algorithm C.3, with scalar $k$ in memory location 0, point $P$ in locations 1-2, point $T$ in locations 2-5 and point $T_{dummy}$ in locations 6-8:

1. set_mod_p
2. load_scalar r0, 0
3. to_mont 1, 1
4. to_mont 2, 2
5. set 3, zero
6. set 4, m_one_p
7. set 5, zero
8. i = 254
9. ecp_dbl 3, 3
10. check r0[i] == 1
11. cjump +3, +1
12. ecp_add 6, 3, 1
13. jump +3
14. ecp_add 3, 3, 1
15. jump +1
16. check i == 0
17. cjump +3, +1
18. i = i - 1
19. jump -10
20. fp_inv 5, 5
21. fp_mul 3, 3, 5
22. fp_mul 4, 4, 5
23. from_mont 3, 3
24. from_mont 4, 4

216
**SPA-Secure ECSM in** $G_2$: Similarly, constant-time ECSM in $G_2$ is implemented as shown below, with scalar $k$ in memory location 0, point $P$ in locations 1-4, point $T$ in locations 5-10 and point $T_{dummy}$ in locations 11-16:

1. set_mod_p 12. set 10, zero 23. $i = i - 1$
2. load_scalar r0, 0 13. $i = 254$ 24. jump -10
3. to_mont 1, 1 14. ecp2_dbl 5, 5 25. fp2_inv 9, 9
4. to_mont 2, 2 15. check r0[i] == 1 26. fp2_mul 5, 5, 9
5. to_mont 3, 3 16. cjump +3, +1 27. fp2_mul 7, 7, 9
6. to_mont 4, 4 17. ecp2_add 11, 5, 1 28. from_mont 5, 5
7. set 5, zero 18. jump +3 29. from_mont 6, 6
8. set 6, zero 19. ecp2_add 5, 5, 1 30. from_mont 7, 7
9. set 7, m_one_p 20. jump +1 31. from_mont 8, 8
10. set 8, zero 21. check i == 0
11. set 9, zero 22. cjump +3, +1

**DPA-Secure ECSM in** $G_1$: This is the DPA-secure version of $G_1$ ECSM discussed in Section 5.4.5, with scalar $k$ in memory location 0, random $r$ in location 1, random $\lambda$ in location 15, point $P$ in locations 2-3, point $2P$ pre-computed in locations 5-6, point $T$ in locations 8-9 and point $T_{dummy}$ in locations 11-13:

1. set_mod_r 16. fp_mul 9, 9, 15 28. check r0[i] == 1 && r1[i] == 1
2. fp_sub 0, 0, 1 17. $i = 254$ 29. cjump +1, +2
3. set_mod_p 18. ecp_dbl 8, 8 30. ecp_add 8, 8, 5
4. load_scalar r0, 0 19. check r0[i] == 0 && r1[i] == 0 31. check i == 0
5. load_scalar r1, 1 20. cjump +1, +2 32. cjump +3, +1
6. to_mont 2, 2 21. ecp_add 11, 8, 2 33. $i = i - 1$
7. to_mont 3, 3 22. check r0[i] == 0 && r1[i] == 1 34. jump -16
8. set 4, m_one_p 23. cjump +1, +2 35. fp_inv 10, 10
9. ecp_dbl 5, 2 24. ecp_add 8, 8, 2 36. fp_mul 8, 8, 10
10. fp_inv 7, 7 25. check r0[i] == 1 && r1[i] == 0 37. fp_mul 9, 9, 10
11. fp_mul 5, 5, 7 26. cjump +1, +2 38. from_mont 8, 8
12. fp_mul 6, 6, 7 27. ecp_add 8, 8, 2 39. from_mont 9, 9
13. set 8, zero
14. set 9, m_one_p
15. set 10, zero

217
**Optimal Ate Pairing:** This is our implementation of Miller Loop and Final Exponentiation in the optimal Ate pairing discussed in Algorithm C.6, with curve parameter $u$ in memory location 0, point $P$ in locations 5-6, point $Q$ in locations 1-4, point $T$ in locations 7-12, sparse line $l_P$ in locations 13-24 and output $f$ in locations 12-23:

**Miller Loop:**
1. set_mod_p
2. load_scalar r0, 0
3. to_mont 1, 1
4. to_mont 2, 2
5. to_mont 3, 3
6. to_mont 4, 4
7. to_mont 5, 5
8. to_mont 6, 6
9. copy 37, 1
10. copy 38, 2
11. copy 39, 3
12. copy 40, 4
13. set 41, m_one_p
14. set 42, zero
15. miller dbl 7, 1, 37
16. multicopy 25, 13, 12
17. miller add 7, 1, 7
18. fp12_sparse_mul 25, 25, 13
19. i = 61
20. miller dbl 7, 1, 7
21. fp12_sqr 25, 25
22. fp12_sparse_mul 25, 25, 13
23. check r0[i] == 1
24. cjump +1, +3
25. miller add 7, 1, 7
26. fp12_sparse_mul 25, 25, 13
27. check i == 0
28. cjump +3, +1
29. i = i - 1
30. jump -10
31. fp12_conj 25, 25

**Final Exponentiation:**
32. fp12_inv 37, 25
33. fp12_conj 49, 25
34. fp12_mul 37, 37, 49
35. fp12_frobenius_p2 49, 37
36. fp12_cyclo_sqr 37, 25
37. fp12_mul 25, 37, 49
38. fp12_cyclo_sqr 37, 25
39. multicopy 49, 37, 12
40. i = 62
41. fp12_cyclo_sqr 49, 49
42. check r0[i] == 1
43. cjump +1, +2
44. fp12_mul 49, 37, 49
45. check i == 0
46. cjump +3, +1
47. i = i - 1
48. jump -7
49. fp12_conj 49, 49
50. multicopy 61, 49, 12
51. i = 62
52. fp12_cyclo_sqr 61, 61
53. check r0[i] == 1
54. cjump +1, +2
55. fp12_mul 61, 49, 61
56. check i == 1
57. cjump +3, +1
58. i = i - 1
59. jump -7
60. fp12_conj 61, 61
61. fp12_conj 73, 25
62. fp12_mul 49, 73, 49
63. fp12_conj 49, 49
64. fp12_mul 49, 49, 61
65. multicopy 61, 49, 12
66. i = 62
67. fp12_cyclo_sqr 61, 61
68. check r0[i] == 1
69. cjump +1, +2
70. fp12_mul 61, 49, 61
71. check i == 0
72. cjump +3, +1
73. i = i - 1
74. jump -7
75. fp12_conj 61, 61
76. multicopy 73, 61, 12
77. i = 62
78. fp12_cyclo_sqr 73, 73
79. check r0[i] == 1
80. cjump +1, +2
81. fp12_mul 73, 61, 73
82. check i == 0
83. cjump +3, +1
84. i = i - 1
85. jump -7
86. fp12_conj 73, 73
87. fp12_conj 49, 49
88. fp12_mul 73, 49, 73
89. fp12_conj 49, 49
90. fp12_frobenius_p1 49, 49
91. fp12_frobenius_p2 49, 49
92. fp12_frobenius_p2 61, 61
93. fp12_mul 49, 49, 61
94. multicopy 61, 73, 12
95. i = 62
96. fp12_cyclo_sqr 61, 61
97. check r0[i] == 1
98. cjump +1, +2
99. fp12_mul 61, 73, 61
100. check i == 0
101. cjump +3, +1
102. i = i - 1
103. jump -7
104. fp12_conj 61, 61
105. fp12_mul 61, 61, 37
106. fp12_mul 61, 61, 25
107. fp12_mul 49, 61, 61
108. fp12_frobenius_p1 61, 73
109. fp12_mul 12, 49, 61
110. from_mont 12, 12
111. from_mont 13, 13
112. from_mont 14, 14
113. from_mont 15, 15
114. from_mont 16, 16
115. from_mont 17, 17
116. from_mont 18, 18
117. from_mont 19, 19
118. from_mont 20, 20
119. from_mont 21, 21
120. from_mont 22, 22
121. from_mont 23, 23
**Hash-to-Curve on** $G_1$: The following code snippet implements constant-time hashing to points on $G_1$ discussed in Section 5.2.4, where the input is in memory location 0, the intermediate point on isogeny curve in locations 1-2, isogeny map constants $k_{1,0-11} / k_{2,0-10} / k_{3,0-15} / k_{4,0-15}$ in locations 10-21 / 22-31 / 32-47 / 48-62 and the output point in locations 3-4:

1. set_mod_p
2. to_mont 0, 0
3. hash_map 1, 0
4. fp_mul 5, 1, 21
5. fp_add 5, 5, 20
6. fp_mul 5, 5, 1
7. fp_add 5, 5, 19
8. fp_mul 5, 5, 1
9. fp_add 5, 5, 18
10. fp_mul 5, 5, 1
11. fp_add 5, 5, 17
12. fp_mul 5, 5, 1
13. fp_add 5, 5, 16
14. fp_mul 5, 5, 1
15. fp_add 5, 5, 15
16. fp_mul 5, 5, 1
17. fp_add 5, 5, 14
18. fp_mul 5, 5, 1
19. fp_add 5, 5, 13
20. fp_mul 5, 5, 1
21. fp_add 5, 5, 12
22. fp_mul 5, 5, 1
23. fp_add 5, 5, 11
24. fp_mul 5, 5, 1
25. fp_add 5, 5, 10
26. fp_add 6, 1, 31
27. fp_mul 6, 6, 1
28. fp_add 6, 6, 30
29. fp_mul 6, 6, 1
30. fp_add 6, 6, 29
31. fp_mul 6, 6, 1
32. fp_add 6, 6, 28
33. fp_mul 6, 6, 1
34. fp_add 6, 6, 27
35. fp_mul 6, 6, 1
36. fp_add 6, 6, 26
37. fp_mul 6, 6, 1
38. fp_add 6, 6, 25
39. fp_mul 6, 6, 1
40. fp_add 6, 6, 24
41. fp_mul 6, 6, 1
42. fp_add 6, 6, 23
43. fp_mul 6, 6, 1
44. fp_add 6, 6, 22
45. fp_mul 6, 6, 1
46. fp_add 6, 6, 21
47. fp_mul 6, 6, 1
48. fp_add 6, 6, 20
49. fp_mul 6, 6, 1
50. fp_add 6, 6, 19
51. fp_mul 6, 6, 1
52. fp_add 6, 6, 18
53. fp_mul 6, 6, 1
54. fp_add 6, 6, 17
55. fp_mul 6, 6, 1
56. fp_add 6, 6, 16
57. fp_mul 6, 6, 1
58. fp_add 6, 6, 15
59. fp_mul 6, 6, 1
60. fp_add 6, 6, 14
61. fp_mul 7, 7, 1
62. fp_add 7, 7, 38
63. fp_mul 7, 7, 1
64. fp_add 7, 7, 37
65. fp_mul 7, 7, 1
66. fp_add 7, 7, 36
67. fp_mul 7, 7, 1
68. fp_add 7, 7, 35
69. fp_mul 7, 7, 1
70. fp_add 7, 7, 34
71. fp_mul 7, 7, 1
72. fp_add 7, 7, 33
73. fp_mul 7, 7, 1
74. fp_add 7, 7, 32
75. fp_mul 7, 7, 1
76. fp_add 7, 7, 31
77. fp_mul 8, 8, 1
78. fp_add 8, 8, 61
79. fp_mul 8, 8, 1
80. fp_add 8, 8, 60
81. fp_mul 8, 8, 1
82. fp_add 8, 8, 59
83. fp_mul 8, 8, 1
84. fp_add 8, 8, 58
85. fp_mul 8, 8, 1
86. fp_add 8, 8, 57
87. fp_mul 8, 8, 1
88. fp_add 8, 8, 56
89. fp_mul 8, 8, 1
90. fp_add 8, 8, 55
SPA-Secure ECSM on Jubjub: This code snippet performs constant-time ECSM on Jubjub, with scalar $k$ in memory location 0, point $P$ in locations 1-2, point $T$ in locations 4-7 and point $T_{dummy}$ in locations 8-11:

1. set_mod_q
2. load_scalar r0, 0
3. to_mont 1, 1
4. to_mont 2, 2
5. fp_mul 3, 1, 2
6. set 4, zero
7. set 5, m_one_q
8. set 6, zero
9. set 7, m_one_q
10. i = 254
11. jubjub_add_proj 4, 4, 4
12. check r0[i] == 1
13. cjump +3, +1
14. jubjub_add_mix 8, 1, 4
15. jump +3
16. jubjub_add_mix 4, 1, 4
17. jump +1
18. check i == 0
19. cjump +3, +1
20. i = i - 1
21. jump -10
22. fp_inv 7, 7
23. fp_mul 4, 4, 7
24. fp_mul 5, 5, 7
25. from_mont 4, 4
26. from_mont 5, 5
Bibliography


228


233


241


242


[270] 3M Company. 3M™ Textool™ Open-Top Sockets for QFN Applications, 0.5 mm Pitch, 64 Pos, Even Row, 9x9 Pkg. https://www.3m.com/3M/en_US/p/d/b5005034035.