PhD-FSTM-2023-097
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 19/09/2023 in Esch-sur-Alzette

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Hao CHENG
Born on 17 June 1993 in Shandong (China)

EFFICIENT AND SIDE-CHANNEL RESISTANT IMPLEMENTATIONS OF NEXT-GENERATION CRYPTOGRAPHY

Dissertation defence committee

Dr. Jean-Sébastien Coron, vice chairman
Professor, Université du Luxembourg

Dr. Robert Granger
Reader, University of Surrey

Dr. Volker Müller, chairman
Associate Professor, Université du Luxembourg

Dr. Daniel Page
Senior Lecturer, University of Bristol

Dr. Peter Y. A. Ryan, dissertation supervisor
Professor, Université du Luxembourg

ABSTRACT

The rapid development of emerging information technologies, such as quantum computing and the Internet of Things (IoT), will have or have already had a huge impact on the world. These technologies can not only improve industrial productivity but they could also bring more convenience to people's daily lives. However, these techniques have "side effects" in the world of cryptography – they pose new difficulties and challenges from theory to practice. Specifically, when quantum computing capability (i.e., logical qubits) reaches a certain level, Shor's algorithm will be able to break almost all public-key cryptosystems currently in use. On the other hand, a great number of devices deployed in IoT environments have very constrained computing and storage resources, so the current widely-used cryptographic algorithms may not run efficiently on those devices. A new generation of cryptography has thus emerged, including Post-Quantum Cryptography (PQC), which remains secure under both classical and quantum attacks, and LightWeight Cryptography (LWC), which is tailored for resource-constrained devices. Research on next-generation cryptography is of importance and utmost urgency, and the US National Institute of Standards and Technology in particular has initiated the standardization process for PQC and LWC in 2016 and in 2018 respectively.

Since next-generation cryptography is in a premature state and has developed rapidly in recent years, its theoretical security and practical deployment are not very well explored and are in significant need of evaluation. This thesis aims to look into the engineering aspects of next-generation cryptography, i.e., the problems concerning implementation efficiency (e.g., execution time and memory consumption) and security (e.g., countermeasures against timing attacks and power side-channel attacks). In more detail, we first explore efficient software implementation approaches for lattice-based PQC on constrained devices. Then, we study how to speed up isogeny-based PQC on modern high-performance processors especially by using their powerful vector units. Moreover, we research how to design sophisticated yet low-area instruction set extensions to further accelerate software implementations of LWC and long-integer-arithmetic-based PQC. Finally, to address the threats from potential power side-channel attacks, we present a concept of using special leakage-aware instructions to eliminate overwriting leakage for masked software implementations (of next-generation cryptography).

ACKNOWLEDGEMENTS

It has been a long journey to get from the start of my PhD studies to its completion, but this journey has always been pleasant. I have thoroughly enjoyed my PhD studies over the past four years, and it would never have been as wonderful as it has been without the help of many people around me.

First and foremost, I want to thank my supervisors Peter Y. A. Ryan and Johann Großschädl. I am very grateful to Peter for offering me the opportunity to be his PhD student and to work in the APSIA group, and for giving me freedom of choice when it came to research directions. He allowed me to attend any conference and school I wanted, and always did his best to provide assistance so that I could focus on my research better, which I appreciate deeply. Great thanks go to Johann, with whom I have worked most closely since I was just a master student. He helped me to develop my research skills and abilities from the ground up: from "how to write code neatly and use tools efficiently" to "how to explore research ideas and write papers", for which I am sincerely thankful. He always thought for me and always offered me any help he could, from work to daily life.

I want to thank Daniel Page, who was the host for my six months research visit at the University of Bristol and also my close collaborator, but more of a mentor, over the past two years. Many thanks to Dan for making my six months in Bristol so happy and memorable: his guidance on side-channel analysis, our daily discussions, our weekly group lunches, and the best farewell gift.

I want to thank my thesis supervision committee (CET: comité d'encadrement de thèse) members Jean-Sébastien Coron and Robert Granger, for continually giving me constructive suggestions from the very beginning of my PhD studies, and for encouraging me to broaden my horizons and to attempt high-risk high-gain research problems.

I want to thank Volker Müller, for serving on my thesis defense committee and chairing the defense, and for taking the time to read my thesis and giving me helpful feedback.

My thanks go to all my co-authors: Malik Alsahli, Alex Borgognoni, Luan Cardoso dos Santos, Daniel Dinu, Georgios Fotiadis, Christian Franck, Johann Großschädl, Ben Marshall, Daniel Page, Thinh Pham, Peter B. Rønne, Peter Y. A. Ryan, Jiaqi Tian. Special thanks go to Daniel Dinu and Thinh Pham; thanks to Daniel for helping me a lot with my first paper, and thanks also to Thinh for continuing to take care of our work even after starting a new job.

My thanks go to my colleagues. I want to thank Georgios Fotiadis, who is my close collaborator, my officemate, and my friend, for his theoretical support in our collaboration, and for enjoyable chats we had every day. I want to thank Aditya Damodaran, my officemate and my friend, for his enthusiastic help with the problems I encountered in my university life, and for the endless jokes he came up

with and shared. I want to thank Qingju Wang for the countless chats we had, and for the invaluable experience and advice she shared with me. I want to thank Peter B. Rønne, for providing me a lot of help when I was new in the APSIA group, and for giving me beneficial comments on my research work.

My thanks go to my friends. I want to thank Jiasheng Liang, Peng Jin, Jiaqi Tian, Yancong Yu, Qian Chen, and Daojiong Wang, for their constant support, and for the wonderful times we spent together.

Finally, my deepest appreciation goes to my parents. Words cannot even describe the love, understanding, encouragement, and support they have given me.

CONTENTS

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF FIGURES

Part I

Introductory Remarks

CHAPTER

1 INTRODUCTION

1.1 Next-generation cryptography

In this thesis, the term "next-generation cryptography" refers to Post-Quantum Cryptography (PQC) and LightWeight Cryptography (LWC).

1.1.1 Post-quantum cryptography

Quantum computing. The advent of quantum computing is a technological revolution that will soon have a massive impact on our daily lives and may even disrupt whole industries [KLM07]. In short, a quantum computer operates on so-called qubits (the "quantum analog" of bits), which can not only take the two states 0 and 1, but also be in a superposition of both states. A quantum computer with n qubits can be in an arbitrary superposition of up to 2^n states simultaneously, enabling it to process 2^n values in parallel or to store 2^n values in one step. For example, a quantum computer with about 50 logical qubits could solve certain complex optimization problems a lot faster than the most advanced classical supercomputer today. In the not-so-distant future, our daily life will start to get affected by large-scale quantum computers that are powerful enough to aid the discovery of new drugs or materials, organize the routes of millions of self-driving cars in metropolitan areas without introducing traffic jams, and improve the efficiency of national power grids [KLM07]. Unfortunately, quantum computing has also a destructive side because a large-scale quantum computer with a few thousand logical qubits would be able to break essentially every public-key cryptosystem in use today. This was discovered in the mid-90s by Peter Shor [Sho94], who developed a polynomial-time quantum algorithm to factor large integers, which could break the widely-used RSA cryptosystem [RSA78]. In addition, Shor's algorithm would also enable one to compute discrete logarithms in any group, hence, also in large elliptic curve groups, thereby breaking Elliptic Curve Cryptography (ECC). It is widely believed that the security of symmetric cryptography is by far less affected by the existence of adversaries equipped with quantum computing capabilities. When instantiated correctly, current symmetric cryptographic algorithms can offer resistance against large-scale quantum computers. Fortunately, there exists a diverse range of symmetric algorithms that have already received approval from internationally recognized standardization bodies, e.g, the block ciphers for symmetric encryption purposes and message authentication codes, for authentication and integrity verification. However, symmetric cryptography alone

cannot mitigate the threat posed by quantum computers, as it is usually combined with public-key cryptographic algorithms, such as key-exchange schemes or key encapsulation mechanisms (KEM), public-key encryption (PKE), and digital signature algorithms, all being vulnerable to large-scale quantum computers.

Put simply, a malicious actor with a large-scale quantum computer would have the ability to decrypt all traffic over communication channels secured using state-of-the-art cryptographic techniques. Even more troubling is that an adversary with sufficient storage capacity can already eavesdrop on and store encrypted data shared over the network and decrypt it after possessing a large-scale quantum computer, an attack known as "store now and decrypt later". Consequently, the threats posed by quantum computers are not just future concerns, but, on the contrary, solutions and countermeasures need to be developed, evaluated, standardized, and deployed as soon as possible.

There are two main solutions for developing quantum-resistant cryptographic systems. The first is to use the laws of quantum physics and quantum phenomena to derive secure cryptographic protocols, a research area known as quantum cryptography. The second approach is to develop cryptographic algorithms using current technologies, which are based on new mathematical problems that are computationally hard to solve for both the adversaries with existing technologies and also the quantum-enabled adversaries. The latter approach is known as post-quantum cryptography, which is the focus in this thesis.

Quantum cryptography. Quantum cryptography is at present limited to Quantum Key Establishment (QKE), also known as Quantum Key Distribution (QKD). Notable QKE protocols, such as the Bennett and Brassard BB84 protocol [BB14], have been studied for decades. It is shown that, in theory, such protocols provide information theoretic security, also known as unconditional security, i.e., the security of the scheme does not depend on the computational power of the adversary. QKE is not a stand-alone solution for the development of quantum-resistant communication systems, as it needs to be combined with other cryptographic algorithms, stemming from symmetric and public-key cryptography, in order to achieve the desired security properties, such as confidentiality, integrity, and authentication. Furthermore, QKE protocols do not appear to be a functional solution. Because, first, deployment of cryptographic systems based on QKE would require the procurement and installation of additional hardware components implementing QKE. Second, in order to exchange quantum information, it would also require the major intervention in infrastructures to install the appropriate fibres for establishing a quantum communication channel between two or more sides. Although QKE protocols have been extensively studied in the literature, the technological limitations of QKE (e.g., low key rates in large distances) raise doubts on whether QKE can be a realistic solution in practical applications. In particular, the US National Security Agency (NSA) in a Q&A report issued in August 2021 states "NSA does not consider QKD a practical security solution for protecting national security information"[1].

Post-quantum cryptography. PQC is a new branch of cryptography, which aims at building new cryptosystems based on mathematical problems that are believed to be hard (not only for classical computers but) even for large-scale quantum computers, while these cryptosystems are to be implemented using classical computers. PQC is therefore a crucial direction in cryptography and of great importance for the future of cybersecurity. The vision of PQC is to replace public-key cryptographic algorithms that are vulnerable to quantum computers with the new algorithms that do not require the integration of additional hardware components. From this point of view, it seems that PQC is a more realistic solution compared to QKE systems. However, when considering to use in practice the new cryptosystems and new hard mathematical problems, on which the security of various applications will rely, there are some challenges that need to be taken into consideration:

[1] https://media.defense.gov/2021/Aug/04/2002821837/-1/-1/1/Quantum_FAQs_20210804.PDF.

Table 1.1: NIST standardized PQC algorithms and fourth round candidates. SIKE is considered broken due to the attacks in [CD23, MMP+23, Rob23].

Type	Status	Algorithm
PKE/KEM	Winner	CRYSTALS-KYBER
	4th round candidates	Classic McEliece, BIKE, HQC, SIKE
Digital signature	Winner	CRYSTALS-DILITHIUM, FALCON, SPHINCS+

- The first challenge relates to the actions that need to be taken before large-scale adoption of cryptographic schemes in the application domains. History shows that this process takes a great deal of time, sometimes even decades. New cryptographic schemes need to undergo a long period of time or even years of cryptanalytic scrutiny before they can be trusted, first and foremost by the academic community. Standards then need to be issued for the schemes, and these standards need to be trusted by the industrial sector. Once the necessary trust has been established, the new schemes need to be deployed in final products. Note, for example, that ECC has been around since the 1980s, but it took more than two decades for elliptic curve cryptosystems to be deployed in practice.
- It is often the case that many quantum-resistant cryptosystems improve their security at the cost of reduced performance and/or increased storage requirements (e.g., large keys, large signatures). This is an important issue as it affects applications in resource/memory-constrained devices.
- Ensuring the correctness of the security properties, the modelling of the threat scenarios and trust assumptions need to be done carefully.

The above challenges suggest that the transition from classical cryptography to PQC should be accelerated, and the sooner we can get to the PQC era, the better. This is also enforced by a recent report from the US National Academies of Sciences, Engineering, and Medicine in 2019 [Nat19], stating:

> "**Key finding 10**: Even if a quantum computer that can decrypt current cryptographic ciphers is more than a decade off, the hazard of such a machine is high enough-and the time frame for transitioning to a new security protocol is sufficiently long and uncertain-that prioritization of the development, standardization, and deployment of post-quantum cryptography is critical for minimizing the chance of a potential security and privacy disaster."

NIST PQC standardization. Given the real-world threat posed by quantum computing, it is not surprising that research in the domain of *post-quantum cryptography* has gained momentum over the past years. In 2016, the US National Institute of Standards and Technology (NIST) announced a process to "solicit, evaluate, and standardize quantum-resistant public-key cryptographic algorithms" and published a call to submit proposals[2]. This call, whose submission deadline passed at the end of November 2017, covered the complete spectrum of public-key functionalities considered by the NIST, i.e., public-key encryption, key agreement, and digital signatures. A total of 72 candidates were submitted, of which 69 satisfied the minimum requirements for acceptability and entered the first round of a multi-year evaluation process, and candidates are from five different categories, i.e., code-based, hash-based, isogeny-based, lattice-based, multivariate. In early 2019, the NIST selected 26 of the submissions as candidates for the second round. About 18 months later, in July 2020, the number of candidates was further reduced to only 7, which entered the third round of NIST's evaluation process. In addition, the NIST also announced 8 so-called "alternate candidates", which could still become part of the standard after the third round (i.e., some of the alternate candidates may be considered in a fourth round

[2] https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

Table 1.2: NIST security categories for evaluating security of PQC candidate schemes.

NIST security level	Description
1	Security equivalent to the complexity of best-known attack for breaking AES-128.
2	Security equivalent to the complexity of best-known attack for finding collision for SHA-256/SHA3-256.
3	Security equivalent to the complexity of best-known attack for breaking AES-192.
4	Security equivalent to the complexity of best-known attack for finding collision for SHA-384/SHA3-384.
5	Security equivalent to the complexity of best-known attack for breaking AES-256.

[AAC+22]). In July 2022, NIST announced[3] that CRYSTALS-Kyber [SAB+21] was selected for standardization for public-key encryption and key-establishment algorithms, and CRYSTALS-DILITHIUM [LDK+21], Falcon [PFH+20], and SPHINCS+ [HBD+22] were selected for standardization for digital signature algorithms. In addition, BIKE [ABB+22], Classic McEliece [BCC+22], HQC [MAB+23], and SIKE[4] [JAC+22] were selected to advance to the fourth round, which is currently still ongoing. Table 1.1 summarizes the PQC algorithms that are selected as the first candidates to be standardized, as well as three additional KEM algorithms that are further evaluated by NIST in the fourth round of the competition. In the meantime, NIST released a call for additional digital signature schemes for the post-quantum cryptography standardization process[5], where the deadline for submissions was June 2023. 50 "onramp" submissions were received, where 40 of them met NIST submission requirements and were released[6].

Furthermore, NIST has published its own methodology for evaluating the security of candidate algorithms. The detailed security levels are shown in Table 1.2. Submitters are requested to provide a detailed security assessment of their candidate schemes, based on the security levels provided by NIST. The security of all NIST submissions is based on mathematical problems that are believed to be hard to solve even using quantum computers. These hard mathematical problems are classified into five different families of computational hardness assumptions. In this thesis, we focus on lattice-based and isogeny-based cryptographic primitives.

Lattice-based schemes. The security of most of the lattice-based schemes that are currently proposed in the literature is based on closely related problems, such as the Shortest Integer Solution (SIS) problem [Ajt96] (usually used for digital signature schemes) and the Learning With Errors (LWE) problem [Reg05] (usually used for PKE/KEM). Variants of these hard lattice problems exist, such as the Ring and Module versions of SIS and LWE, namely Ring-LWE, Ring-SIS and Module-LWE and Module-SIS. In general, when designing protocols, the Ring and Module versions are more commonly used, as they result in more practical schemes in terms of storage requirements. Lattice-based schemes have been extensively studied throughout the years, and one of their main advantages is their efficiency. Specifically, public key encryption schemes are as efficient, if not more so, than currently-used schemes such as RSA. On the downside, lattice-based schemes suffer from large keys and signature sizes. In the NIST compe-

[3] https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[4] Due to the recent attacks [CD23, MMP+23, Rob23], the SIKE team acknowledges that SIKE and SIDH are insecure and should not be used; see details at https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-4/submissions/sike-team-note-insecure.pdf.

[5] https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf.

[6] https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

Table 1.3: IoT connections (billions). Cellular IoT is also included in wide-area IoT. Source: Ericsson mobility report (June 2023) [Eri23, Figure 7].

IoT	2022	2028	CAGR
Wide-area IoT	2.9	6.0	13%
Cellular IoT	2.7	5.4	12%
Short-range IoT	10.2	28.7	19%
Total	13.2	34.7	18%

tition, lattice-based schemes are currently represented by CRYSTALS-KYBER, CRYSTALS-DILITHIUM, and FALCON, all of them being selected as the first PQC standards by NIST.

Isogeny-based schemes. The security of isogeny-based cryptosystems is based on the hardness of finding an isogeny between two isogenous (supersingular) elliptic curves and related problems. Isogeny-based cryptography is one of the newest additions in the PQC landscape, and as such it has been less studied than other classes of hardness assumptions. On the positive side, isogeny-based systems have very short keys compared to other post-quantum schemes in the literature. Their main disadvantage, when compared with other schemes, especially with lattice-based schemes, is their speed. In the NIST PQC competition, there was only one isogeny-based candidate, namely Supersingular Isogeny Key Encapsulation (SIKE), which is derived from the Supersingular Isogeny Diffie-Hellman (SIDH) key-exchange protocol [JD11, DJP14]. SIKE made it all the way to the fourth round of the NIST competition, before it was eventually broken in July 2022 [CD23]. Isogenies can be used for developing secure public key encryption schemes, as well as signature schemes, although the latter suffer from the heavy computational cost. A new digital signature scheme was recently submitted in the NIST standardization of additional PQC signature schemes, namely SQISign [DKL+20, DLLW23], which offers both very compact public keys and signatures.

1.1.2 Lightweight cryptography

Internet of Things. The Internet of Things, abbreviated as IoT, refers to the network that connects various kinds of physical objects (or "things") and enables them to communicate and transmit data with each other or central servers. The development of IoT started in early 2000s, and since then, an increasing number of "everyday objects" (e.g., household appliances, vehicles, wearable devices, and industrial/business machines) are integrating the advanced communication capabilities (e.g., Bluetooth, Wi-Fi, and Cellular). Today, Internet-enabled smart devices are present in almost every area of people's lives, for instances, healthcare, home automation, industrial production ("Industry 4.0"), transportation, and logistics. Different IoT devices exhibit big variations in computing power, data transmission speed and memory capacity. A typical example of the high-end IoT device is "intelligent vehicles", which possess the powerful processors to enhance safety and driving experience as well as the ultra-high data transmission rates to ensure real-time communication. At the opposite are the low-end battery-supplied sensors, which are equipped with small 8-bit, 16-bit, or 32-bit microcontrollers (MCU) of limited computing power. IoT can be viewed as a vast ecosystem teeming with devices of high diversity, and thus many different MCU platforms, wireless communication standards, and operating systems currently exist (to meet the requirements of different application domains or scenarios).

The overall IoT market is enormous and growing steadily. The latest Ericsson mobile report (June 2023) [Eri23], in particular Table 1.3, expects the number of IoT connections around the world will increase in the next years with a Compound Annual Growth Rate (CAGR) of about 18%. The total number of IoT connections is 13.2 billion in 2022 and is predicted to reach even 34.7 billion in 2028. In addition, [Pre23] shows that the global microcontroller market (including 8-bit, 16-bit, and 32-bit MCUs)

was valued at USD 28.2 billion in 2022, and is expected to reach USD 58.2 billion by 2030, growing at a CAGR of 9.5%. Also, according to the statistics in [Pre23], in the MCU product market in 2021, 8-bit, 16-bit, and 32-bit microcontrollers account for 24%, 35%, and 41% of the market share, respectively. Two most related MCU platforms in this thesis are 8-bit and 32-bit. 8-bit microcontrollers (e.g., AVR, PIC) are one of the most popular electronic components, and they are broadly used in automotive and industrial applications as well as digital signal processing. Besides, due to the increasing demands for more powerful computing capabilities (in IoT environment) and the decreasing unit prices, the product market size of 32-bit microcontrollers grew greatly in the past years, and is poised to expand at a CAGR of 11.7% from 2022 to 2030 [Pre23]. At present, the ARM architecture is the undisputed leader in the 32-bit platforms, despite intense competition from ESP32 and RISC-V.

Based on the above, it is not surprising that the security of IoT raises lots of concerns, especially, given that 1) computing capabilities of many IoT devices such as miniature sensors and actuators are very limited and 2) the physical access to many IoT devices cannot be restricted (i.e., they are more vulnerable to some implementation attacks compared to classical computers like PCs and laptops).

Lightweight cryptography and NIST LWC standardization. Due to the fact that current NIST cryptographic standards were designed to perform well on general-purpose computers, their performance might be unsatisfactory when implemented on such small computing devices. As a result, there is a need of *lightweight cryptography*, which can be very generally defined as "cryptographic primitives, schemes, and protocols tailored to (extremely) constrained environments". After a series of exploratory workshops in 2015 and 2016 and a report [MBTM17] summarizing the context and goals, NIST initiated a selection process for lightweight cryptography via an associated call [NIS18] released in 2018. The process scope involves two specific forms of cryptographic functionalities, with each submission specifying a suite of algorithms with *required* support for an Authenticated Encryption with Associated Data (AEAD) API [NIS18, Section 3.1], plus *optional* support for a hash function API [NIS18, Section 3.2]. The submitters are allowed to submit a family of AEAD algorithms, and the primary AEAD member is required to have a at least 128-bit key length, a at least 96-bit nonce length, and a at least 64-bit tag length. Besides, the AEAD algorithms shall have a minimum strength of 112 bits of security. Although the term is open to a more general interpretation, the call defines lightweight as "*tailored for resource-constrained devices*" [NIS18, Section 1]. This implies that the said algorithms should, e.g., be 1) efficient on constrained hardware and software platforms (versus existing standards), 2) efficient for short messages, and 3) amenable to countermeasures against implementation attacks. A total of 57 candidates were submitted by the (extended) deadline of March 2019, of which 56 satisfied the acceptance criteria and were considered as "proper and complete". These 56 proposals entered a multi-round evaluation process that took (almost) four years altogether. By the end of the first round the number of candidates was reduced to 32 and then, in the second round, a further 22 algorithms were eliminated from NIST's evaluation process. The remaining 10 candidates made it to the third and final round, in which they were extensively scrutinized for security and efficiency over a period of more than 1.5 years. Finally in February 2023, NIST announced that it decided to standardize the Ascon family [DEMS21] for lightweight cryptography applications[7], and [TMC+23] explains this selection.

1.2 Cryptographic engineering

According to [Koç09], cryptographic engineering refers to the theory and practice of engineering of cryptographic systems, and a cryptographic engineer designs, implements, tests, validates, and sometimes reverse-engineers or attempts to break cryptographic systems. In other words, cryptographic engineering studies the deployment of various cryptographic algorithms in different environments,

[7] https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon.

and it is usually concerned with issues such as the efficiency and security arising from the implementation of cryptographic algorithms. Cryptographic engineering is a fast-moving and multidisciplinary field, e.g., see a most recent call for papers[8] of Conference on Cryptographic Hardware and Embedded Systems (CHES), a prestigious conference in the area of cryptographic engineering.

1.3 Contributions and organization

Broadly speaking, this thesis aims at providing a feasible solution to well connect the two objects above, i.e., next-generation cryptography and cryptographic engineering (with more background information presented in Chapter 2 of Part I). Concretely speaking, we make contributions to addressing engineering difficulties of deploying next-generation cryptography in the real world, covering four specifically-different topics:

1. Lattice-based cryptosystems are considered the most promising PQC candidates for use in constrained devices due to both their low computational cost and reasonably small lengths of key, ciphertext, and signature. We evaluate the performance of constant-time implementations of three NIST PQC candidates[9] on 8-bit AVR microcontrollers, namely NTRU [CDH+19], NTRU Prime [BBC+20c], and ThreeBears [Ham19], through implementing the hand-optimized assemblers for their performance-critical components. We describe these works in Part II.

    - Most of the work related to AvrNtru [CGRR21], a lightweight implementation of NTRU (more accurately the NTRUEncrypt) on AVR, was done during the master thesis. Though the associated paper was improved, polished, and finally published during the author's PhD studies, it is reasonable to not include it in this thesis.
    - In Chapter 3, we describe a lightweight implementation of NTRU Prime, more concretely, the streamlined NTRU Prime using the parameter set `sntrup653` which meets the NIST PQC security level 1. The associated paper is [CDG+19].
    - In Chapter 4, we describe the speed-optimized and the memory-optimized implementations of BabyBear, i.e., an instantiation of ThreeBears using a parameter set fulfilling the NIST PQC security level 1. The associated paper is [CGRR20].

2. The field of isogeny-based cryptography has grown very rapidly in the past decade. Various isogeny-based cryptosystems have appeared in the literature, among which the most popular two key-establishment algorithms are SIKE [JAC+22] and CSIDH [CLM+18]. They are also the two target algorithms[10] in Part III. Compared to the PQC algorithms of other categories (e.g., lattice-based, code-based), their secret and public keys are much shorter, but the computational cost, namely the execution time, is usually one or several orders of magnitude higher. In Part III, we study how to utilize the large computing power from modern vector instruction sets to make the implementation of isogeny-based cryptography more efficient.

    - Rather than directly researching the complex field of isogeny-based cryptography, we first develop a vectorized implementation of the relatively-simpler classic ECDH scheme X25519 [Ber06]. It is based on the consideration that X25519 and isogeny-based cryptography such as SIKE and CSIDH work on some common underlying arithmetic (e.g., Montgomery curve arithmetic); X25519 therefore is more suitable for the first step in the sense of easier mathematical understanding and lighter engineering efforts. In [CGT+20], we describe an AVX2

[8] https://ches.iacr.org/2024/callforpapers.php.
[9] While we were conducting this series of research work, the NIST PQC standardization was in its second round.
[10] While we were conducting this series of research work, the NIST PQC standardization was in its third round.

batch implementation of X25519, but we do not include this work in this thesis since X25519 does not belong to the domain of next-generation cryptography.

- In Chapter 5, we present the throughput-optimized and latency-optimized constant-time vectorized implementations of CSIDH with AVX-512. The associated paper is [CFG+21].
- In Chapter 6, we show the throughput-optimized and latency-optimized AVX-512 vectorized implementations of SIKE, which have fixed instruction sequences. The associated paper is [CFGR22].

3. RISC-V is an open ISA specification, and a central tenet of the ISA is modularity: a general-purpose ISA can be augmented with a set of special-purpose, standard or non-standard (i.e., custom) extensions. There already exists a standard crypto (K) extension [RVK22, RVK23], and it currently targets speeding up major symmetric cryptosystems, particularly, AES and SHA-2. In Part IV, we study the Instruction Set Extension (ISE) design for accelerating respectively LWC and multi-precision integer arithmetic, whose dedicated instructions are not offered in the current RISC-V K extension.

   - In Chapter 7, we present the RISC-V ISE design for all the ten finalists of NIST LWC standardization[11]. We evaluate the finalists from an ISE design and ISE-assisted implementation perspective. The associated paper is [CGM+23].
   - In Chapter 8, we come up with the RISC-V ISE design for multi-precision integer arithmetic (which is the underlying arithmetic for many public key cryptosystems, e.g., RSA, ECC, and isogeny-based cryptography), and take two different data representations into account. The associated paper is [CFG+23].

4. In Part V, we study the security aspect of the cryptographic implementation, more specifically the power side-channel analysis and the related leakage elimination.

   - In Chapter 9, we design a RISC-V ISE that can prevent architectural and micro-architectural overwriting leakage, and we show two different realizations taking into account the optimization respectively for latency and for area. The associated paper is [CP23].

Finally, in Chapter 10 of Part VI, we provide a summary, discuss the (potential) impact of this thesis, and present some ideas for future work.

1.4 Publications

The research papers produced during the research of this thesis are listed below.

Lightweight implementaton of lattice-based cryptography. The research focus during the PhD studies months 1 to 6 was: lightweight and constant-time software implementation of lattice-based cryptographic schemes on low-end microcontrollers. Three papers were produced on this topic.

[11] While we were conducting this research wrok, the NIST LWC standardization was in its final round.

[CDG+19] **A lightweight implementation of NTRU Prime for the post-quantum internet of things.**
Hao Cheng, Daniel Dinu, Johann Großschädl, Peter B. Rønne, Peter Y. A. Ryan.
WISTP 2019

[CGRR20] **Lightweight post-quantum key encapsulation for 8-bit AVR microcontrollers.**
Hao Cheng, Johann Großschädl, Peter B. Rønne, Peter Y. A. Ryan.
CARDIS 2020

[CGRR21] **AVRNTRU: lightweight NTRU-based post-quantum cryptography for 8-bit AVR microcontrollers.**
Hao Cheng, Johann Großschädl, Peter B. Rønne, Peter Y. A. Ryan.
DATE 2021

Vectorized implementation of isogeny-based cryptography. The research focus during the PhD studies months 7 to 24 was: vectorized and constant-time software implementation of isogeny-based (and classic ECC) cryptographic schemes on high-end processors. Three papers were produced on this topic.

[CGT+20] **High-throughput elliptic curve cryptography using AVX2 vector instructions.**
Hao Cheng, Johann Großschädl, Jiaqi Tian, Peter B. Rønne, Peter Y. A. Ryan.
SAC 2020

[CFG+21] **Batching CSIDH group actions using AVX-512.**
Hao Cheng, Georgios Fotiadis, Johann Großschädl, Peter Y. A. Ryan, Peter B. Rønne.
IACR TCHES 2021

[CFGR22] **Highly vectorized SIKE for AVX-512.**
Hao Cheng, Georgios Fotiadis, Johann Großschädl, Peter Y. A. Ryan.
IACR TCHES 2022

Efficient cryptographic instruction set extension design. The research focus during the PhD studies months 25 to 36 was: efficient RISC-V cryptographic instruction set extension design, specifically, for lightweight symmetric cryptography and for multi-precision integer arithmetic. Two papers were produced on this topic.

[CGM+23] **RISC-V instruction set extensions for lightweight symmetric cryptography.**
Hao Cheng, Johann Großschädl, Ben Marshall, Daniel Page, Thinh Pham.
IACR TCHES 2023

[CFG+23] **RISC-V instruction set extensions for multi-precision integer arithmetic.**
Hao Cheng, Georgios Fotiadis, Johann Großschädl, Daniel Page, Thinh Pham, Peter Y. A. Ryan.
to be submitted

Side-channel leakage analysis and elimination. The research focus during the PhD studies months 37 to 42 (involving a *6-month research visit* at the University of Bristol) was: side-channel leakage analysis and elimination; specifically, the design of leakage-focused instructions for masked implementation, which prevents architectural and micro-architectural overwriting leakage. One paper was produced on this topic.

[CP23] **eLIMInate: a Leakage-focused ISE for Masked Implementation.**
Hao Cheng, Daniel Page.
under review

Other work. During the PhD studies, we also provided assistance to our collaborators on other topics: for example, software implementation of lightweight symmetric cryptography. One paper was produced.

[ABC+22] **Lightweight permutation-based cryptography for the ultra-low-power internet of things.**
Malik Alsahli, Alex Borgognoni, Luan Cardoso dos Santos, Hao Cheng, Christian Franck, Johann Großschädl.
SECITC 2022

CHAPTER

2

BACKGROUND

The two essential properties of a cryptographic implementation are efficiency and security. The evaluation of cryptographic implementations differs on various platforms, e.g., low-end microcontrollers vs. high-end processors. In this Chapter, Section 2.1 describes the different platforms on which we experimented next-generation cryptography, after which Section 2.2 explains the trade-off about cryptographic implementations. Then, Section 2.3 provides more information about the efficiency of cryptographic implementations and Section 2.4 discusses security aspects (i.e., resistance to physical attacks).

2.1 Platform

2.1.1 AVR

8-bit AVR microcontrollers is currently widely used in the embedded realm (e.g., smart cards, wireless sensor nodes). The AVR architecture is based on the modified Harvard memory model, it follows the RISC philosophy, and was originally developed by Atmel Corporation (now part of Microchip Technology, Inc.). It features 32 general-purpose working registers (i.e., `R0` to `R31`) of 8-bit width, which are directly connected to the Arithmetic Logic Unit (ALU). Standard arithmetic/logical instructions have a two-address format, which means they can read two independent 8-bit operands from two of the working registers and write the result back to one of them. Since AVR is a "Harvard-based" architecture, it uses separate memories, buses, and address spaces for program and data to maximize performance and parallelism. Three pairs of working registers can operate as 16-bit pointers (`X`, `Y`, and `Z`) to access data memory, whereby five addressing modes are supported. Furthermore, the pointer `Z` can be used to read from (and write to) program memory. The current revision of the AVR instruction set supports 129 instructions in total, and each of them has fixed latency [AVR21]. Examples of instructions that are frequently used in our software in this thesis are addition (`ADD`/`ADC`) and subtraction (`SUB`/`SBC`); they take a single cycle. On the other hand, the multiplication (`MUL`) and also the load (`LD`) and store (`ST`) instructions are more expensive since they have a latency of two clock cycles.

2.1.2 AVX-512

AVX-512 is the latest generation of the Intel Advanced Vector eXtensions (AVX) and enriches the x64 execution environment by 32 512-bit registers (`zmm0`–`zmm31`) and various 512-bit instructions. AVX-512

consists of multiple extensions, whereby AVX-512 Foundation (AVX-512F) is the core extension with a 32-bit vector multiplier. Starting with Cannon Lake (Palm Cove microarchitecture), Intel integrated the so-called Integer Fused Multiply-Add extension (AVX-512IFMA, or simply IFMA) into AVX-512 [Int18b], which was specifically designed to speed up public-key cryptographic software relying on large integer arithmetic. Intel described IFMA in [Int18b] as "two new instructions for big number multiplication for acceleration of RSA vectorized SW and other Crypto algorithms (Public key) performance". Concretely, the two new IFMA instructions `vpmadd52luq` and `vpmadd52huq` multiply a pair of eight packed unsigned 52-bit integers (one located in each 64-bit element of two 512-bit vectors) to obtain eight intermediate products, each being 104 bits long. Then, either the lower 52 bits (`vpmadd52luq`) or the upper 52 bits (`vpmadd52huq`) of these products are added to the eight packed unsigned 64-bit integers of a 512-bit destination register, which holds the final result. Compared to the `vpmuludq` and `vpmuldq` multiply instruction of AVX-512F, the IFMA extension does not only offer a wider multiplier of 52 bits, but also combines vector multiplication and vector addition into a single instruction.

2.1.3 RISC-V

RISC-V (see, e.g., [Wat16]) is an Instruction Set Architecture (ISA) specification which emerged from academic roots; it now enjoys a significant role in educational and research activities, and industrial deployment across a range of use-cases and sectors. At least two features make RISC-V an attractive option. First, the design is open in the sense it can be implemented or modified by anyone, with neither license nor royalty requirements. This fact has contributed to 1) a rich community organized around the RISC-V International non-profit, 2) availability of supporting infrastructure such as compilation tool-chains, and 3) a range of (typically open source) compliant implementations. Second, it adopts strongly RISC-oriented design principles but is highly modular: a sparse, general-purpose base ISA, e.g., RV32I [RV19, Chapter 2], RV64I [RV19, Chapter 5], can be augmented with special-purpose (or even domain-specific), standard and non-standard extensions. In RV32I and RV64I base ISAs, there are 32 integer general-purpose registers (`x0` – `x31`) in total[1], some of which serve a special purpose, e.g., as stack pointer (register `sp`, namely `x2`) or to hold the return address of a function call (registers `a0` – `a7`, namely `x10` – `x17`), and `x0` is hard-wired to `0`. These general-purpose registers, together with the program counter (`pc`), constitute the architectural (i.e., user-visible) state. There is no further state besides these 33 registers; in particular, there are no ALU status bits or "flags" that get set or cleared when an instruction produces a carry/borrow, overflow/underflow, negative, or zero result. Hence, there are no add-with-carry, subtract-with-borrow, or branch-on-bit-set/clear instructions in RISC-V. Furthermore, RISC-V (unlike ARM) does not support conditional (or predicated) instruction-execution, which simplifies both the micro-architecture and the code-generation back-end of compilers.

2.2 Trade-off

Due to the requirements of different applications, characteristics of different algorithms, and different target platforms, it is very difficult to design and develop a one-size-fits-all implementation. Instead, implementers usually try to find and reach the optimal trade-off for a specific scenario or use case. Figure 2.1 shows some metrics about the trade-off of crypto implementation that implementer could or should consider.

[1] There is a reduced version of RV32I designed for embedded systems, namely RV32E, which reduces the integer register count to 16 general-purpose registers (`x0` – `x15`) [RV19, Chapter 3].

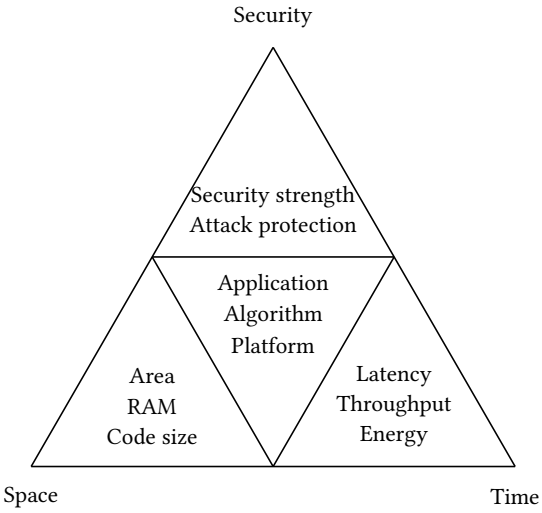


Figure 2.1: Trade-off of cryptographic implementations.

2.3 Efficiency

Roughly classified, there are three categories in cryptographic implementations, namely HardWare (HW) implementation, SoftWare (SW) implementation, and a hybrid of HW and SW, e.g., co-processor, Application Specific Instruction Processor (ASIP), and ISE. In line with the scope and the contributions of this thesis, we focus only on software implementation and on the ISE-based hybrid approach.

2.3.1 Software

The efficiency of a software cryptographic implementation may be evaluated from two classic dimensions, namely time (e.g., execution latency/throughput) and space (e.g., data/instruction footprint). In addition, some other metrics derived from timings are also popularly used, in particular, energy consumption.

On low-end constrained microcontrollers. With some unique advantages such as low unit costs and small volume, low-end MCUs are in large-scale use in the constrained IoT environment. Many different popular ISAs exist, e.g., 8-bit AVR, 16-bit MSP430, 32-bit ARM, 32-bit RISC-V, and so on. Energy consumption is a very important metric for low-end constrained MCUs, due to the fact that a great number of devices, such as many (industrial, agricultural, and medical) sensors, are supplied just by batteries. In some scenarios, with a small (non-rechargeable) battery, a sensor is expected to work for years or even decades. In general, the energy E can be computed from power P and time t, i.e., $E = P \cdot t$, and power is usually determined by the device itself (e.g., different operating modes, which are not crypto-related). Therefore, for a cryptographic implementer, the execution time of cryptographic implementations, specifically the cycle count[2], should be of great concern. However, the acceleration of cryptographic software on low-end MCUs is usually not an easy task due to limited computing capabilities of the hardware. Some manufactured devices are also extremely memory-constrained, e.g., C0 and C1 devices defined in RFC 7228 (see Table 2.1), which makes things even more tricky. As a result, the cryptographic applications to be deployed on such low-end microcontrollers should be considered/developed to possess reasonable running times, and at the same time, to work within the memory capacity, in order to preserve RAM/ROM space for other applications.

[2]The software wall time is computed with both the software cycles and the processor frequency, where the processor frequency is usually not crypto-related.

Table 2.1: Classes of constrained devices from RFC 7228 [BEK14, Table 1].

Device	Data size (e.g., RAM)	Code size (e.g., Flash)
Class 0, C0	$\ll$ 10 kB	$\ll$ 100 kB
Class 1, C1	$\sim$ 10 kB	$\sim$ 100 kB
Class 2, C2	50 kB	250 kB

On high-end processors. The computing and storage resources of high-end processors are usually sufficient enough for a cryptographic primitive. On such platforms, from an efficiency perspective, the priority metric of cryptographic software is its execution time/throughput. But we note that even though RAM and disk space for program code are abundant, the size of caches should be considered. Memory accesses are only fast if the code and data fits into the first-level cache, which is often relatively small compared to, e.g., RAM.

2.3.2 Instruction set extension

An ISE is usually designed to enhance the ISA to improve the software performance for single or multiple specific functions/applications. It allows developers to combine the flexibility of software with the dedicated computing power of hardware and produce a more attractive overall trade-off. As opposed to co-processors and ASIPs, ISEs usually have a lower overhead in hardware but a smaller speed-up in software.

Instruction level. Again, the ISE can also be evaluated from two classic dimensions, namely latency (i.e., the dimension of time) and area (i.e., the dimension of space). Instruction latency is important for the software side in the sense that it directly affects the execution time of software using the ISE. The area overhead reflects the extra cost needed on the hardware side to support the implemented instructions.

Software level. It is also important to evaluate an ISE by checking the speed-up of software before and after using this ISE, i.e., how much performance improvement the ISE actually provides.

2.4 Security

Aside from discovering and utilizing the potential flaws of cryptographic algorithms themselves, there exist Side-Channel Attacks (SCA) based on the information collected from some *side-channels* of cryptographic implementations, e.g., timing, power, and ElectroMagnetic (EM) emanations. According to [Por18], SCA can be explained as "A computer system runs operations in a conceptual abstract machine, that takes some inputs and provides some outputs; side-channel attacks are all about exploiting the difference between that abstract model and the real thing". In line with the scope and the contributions of this thesis, we focus only on timing-based side-channel attacks and power side-channel attacks.

2.4.1 Timing attack

Broadly, a timing attack is an attack where the attacker attempts to get information on the private key and/or other secrets by measuring and analyzing the running time of related operations. The first example of a timing attack can be traced back to 1996 by Kocher, who in [Koc96] showed an attack on RSA implementations. A unique feature of timing attacks that differentiates them from other side-channel attacks is they can be applied remotely [BB03, BT11], therefore for cryptographic implementations it is in some sense a basic requirement to be resistant against timing attacks.

Constant-time implementation. Although alternatives exist (e.g., masking), we focus on the most popular and widely-used approach, i.e., *constant-time implementations*. In a cryptographic software implementation, there are some operations deserving special attention: memory accesses, conditional jumps, integer divisions and modulo computations, shifts and rotations, and multiplications [Por18]. Making a cryptographic software implementation constant-time requires not only implementation tricks (e.g., rewriting ordinary conditional statements such as "if-then-else", using constant-time versions for certain operations such as modular reduction) but also optimizations on cryptographic algorithms themselves (e.g., using Montgomery ladder for computing scalar multiplications on Montgomery elliptic curves [Mon87], using dummy isogenies in CSIDH class group action [MCR19]). We classify constant-time implementations in to two categories. One has a weak constant-time property that given different inputs, the cryptographic implementation might possess *different operation/instruction sequences*, but they do not leak any secrets and/or any sensitive information. In contrast, the other with a strong constant-time property is that given different inputs, the cryptographic implementation always possesses *the same operation/instruction sequences*.

2.4.2 Power side-channel attack

A power side-channel attack is performed by measuring and analyzing the power consumption of related operations on hardware. It is based on the fact that cryptosystems operate on binary data, i.e., 0s and 1s, and CMOS (Complementary Metal-Oxide Semiconductor) hardware transitions between these two states by switching transistors, which causes dynamic power consumption. Unlike timing attacks which may be applied remotely, power side-channel attacks require the attacker to be physically close to the target device. Given that 1) the physical access to many embedded devices cannot be effectively restricted, 2) embedded devices are usually equipped with limited resources (e.g., power budget, computing capability), and 3) embedded devices are simpler in architecture and composition than classical computers, power side-channel attacks are more of a concern for embedded devices. Power side-channel attacks are one of the most studied attacks amongst different side-channel attacks on crypto implementations, and various analysis approaches exist: Simple Power Analysis (SPA), Differential Power Analysis (DPA), Correlation Power Analysis (CPA), Template Attacks (TA), Test Vector Leakage Assessment (TVLA), and so on.

Masking. In this thesis, we focus on DPA [KJJ99] and variants thereof. Two categories of classic countermeasures are hiding [MOP07, Chapter 7] and masking [MOP07, Chapter 10]. The latter, i.e., masking, is more popular, and there are various masking schemes as well as numerous associated research papers in the literature. For instance, for AES there are different masking schemes such as [RP10, Cor14, BFG+17, CGZ20, CGGS22]. Taking d-th order Boolean masking as a concrete example, a variable x is represented by $d+1$ statistically independent *shares* $\hat{x}_i$, i.e., $\hat{x} = \langle \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_d \rangle$. The value of x equals to the xor of all shares. In such a way, the variable x is concealed by masks, and the attacker can now only observe the power information of shares and masked implementations. For more details, we refer to [MOP07, Chapter 10].

Part II

Lightweight Implementation of Lattice-Based Cryptography

CHAPTER

3

LIGHTWEIGHT NTRU PRIME

This Chapter is based on our paper [CDG+19]. While we were conducting the research work described in this Chapter, the NIST PQC standardization process was in its second round.

3.1 Introduction

NTRU Prime. *NTRU Prime* is a family of lattice-based crypto schemes developed by Bernstein, Chuengsatiansup, Lange, and van Vredendaal [BCLV17], who drew inspiration from the 20-year old classical NTRU cryptosystem [HPS98]. There are two variants of NTRU Prime; one is *Streamlined NTRU Prime*, which uses the quotient $h = g/(3f)$ of two secret polynomials g, f as public key (similar to the classical NTRU), while the other, *NTRU LPRime*, has public keys of the form $h = e + Af$, where e, f are secret and A is public (like in cryptosystems based on the Ring Learning With Errors (RLWE) problem [LPR10], e.g., NewHope [ADPS16]). In essence, NTRU Prime can be seen as an attempt to improve the security of the classical NTRU encryption algorithm (and other lattice-based cryptosystems) by avoiding rings with "worrisome" structure and using extension fields of the form $\mathcal{R}/q = (\mathbb{Z}/q)[x]/(x^p - x - 1)$ instead, where p is prime. Multiplication in such fields can be efficiently implemented through several layers of Karatsuba's technique [KO63], which makes NTRU Prime relatively fast on 64-bit processors with vector instructions. Concretely, the designers of NTRU Prime describe in [BCLV17] a highly-optimized implementation of the field multiplication using Intel's AVX2 vector instructions that executes 16 separate multiplications of integers modulo 2^{16} in a SIMD-parallel way. NTRU Prime is among the 26 candidates in the second round of NIST's evaluation process. This second round will focus on evaluating the candidates' performance across a wide variety of systems and platforms, which includes "not only big computers and smart phones, but also devices that have limited processor power" [AAA+20].

Research on software optimization techniques that enable fast implementations of (Streamlined) NTRU Prime has, until now, been limited to 64-bit Intel processors with AVX2 vector engine. When using a parameter set for 128 bits of post-quantum security, the AVX2 implementation introduced in [BCLV17] requires 59,600 clock cycles for encryption (i.e., "encapsulation" of a 256-bit key) on an Intel Haswell processor, while the decryption ("decapsulation") is 63.5% more costly and takes 97,452 cycles. The only performance figures for NTRU Prime on small platforms (e.g., 8, 16, or 32-bit microcontrollers) we are aware of were reported in a recent paper on `pqm4` [PQM4], a testing and benchmarking toolsuite

for NIST PQC candidates on ARM Cortex-M4 devices. Due to the lack of an optimized ARM implementation, the authors of [PQM4] resorted to the reference C code provided by the designers of NTRU Prime, which requires 54.9 million clock cycles for encapsulation and 166.5 million cycles for decapsulation (these cycle counts were determined with Streamlined NTRU Prime and parameters for 128-bit post-quantum security). However, both results do not allow one to reason about the actual performance of NTRU Prime on microcontrollers since the aim of a reference C implementation is to promote the understanding of an algorithm rather than achieving high speed. Therefore, not much is known on how to optimize NTRU Prime for a small microcontroller and what execution time a carefully-tuned assembler implementation could achieve.

Contributions. In this work we present a highly-optimized implementation of Streamlined NTRU Prime for 8-bit AVR microcontrollers that we developed from scratch to reach high speed and resistance against timing attacks. We chose 8-bit AVR as evaluation platform for two reasons. First, the 8-bit AVR architecture remains very popular in devices with increased security requirements, e.g., smart cards and (wireless) sensor nodes. Second, 8-bit AVR microcontrollers are among the most resource-limited of all currently used computing platforms, which implies that if NTRU Prime can be implemented to run with acceptable speed on an AVR device, it can also be implemented to run satisfactorily on more powerful 16 and 32-bit microcontrollers (e.g., an ARM Cortex-M), whereas the opposite is not necessarily true. The implementation we describe in the next sections is not purely optimized for speed, but strives for a balance between performance and other metrics of interest for low-end devices used in the Internet of Things (IoT), in particular binary code size. Therefore, we decided to refrain from full loop unrolling and other optimization techniques that are likely to increase the code size significantly (especially on an 8-bit device) for marginal performance benefits. We also restrict our arsenal of polynomial multiplication algorithms to the basic (i.e., recursive) Karatsuba variant and the schoolbook method for the same reason. Recent results by Kannwischer et al. [KRS19] show that a combination of Karatsuba's technique with the asymptotically faster Toom-Cook algorithm [Too63, Coo66] can slightly reduce the multiplication time, e.g., by 17.4% for polynomials of degree 701 (excluding the reduction of coefficients), but only at the expense of almost doubled stack usage and significantly increased implementation complexity. On the other hand, our Karatsuba/schoolbook multiplication is simple to implement and has the further advantage of enabling compact code size (see Section 3.4) while remaining competitive in terms of performance.

3.2 Background

NTRU Prime is introduced in [BCLV17] as a high-security *prime-degree large-Galois-group inert-modulus* ideal-lattice-based cryptosystem. A distinguishing feature of NTRU Prime is the use of an irreducible non-cyclotomic polynomial P; the designers recommend to choose a polynomial P of prime degree p with a large Galois group. More specifically, they suggest $P = x^p - x - 1$ and recommend to take a prime modulus q such that P is irreducible modulo q, which means q is inert in the ring $\mathcal{R} = \mathbb{Z}[x]/P$ and $\mathcal{R}/q = (\mathbb{Z}/q)[x]/P$ is actually a field. Due to the prime degree of P, the only subfields of $(\mathbb{Z}/q)[x]/P$ are $\mathbb{Z}/q$ and the entire field $(\mathbb{Z}/q)[x]/P$. Furthermore, the requirement of a large Galois group implies that P has, at most, a few roots in any field of reasonable degree, which makes automorphism computations hard. Finally, since q is an inert prime, there are no ring homomorphisms from $(\mathbb{Z}/q)[x]/P$ to any smaller non-0 ring.

The NTRU Prime family of Key Encapsulation Mechanisms (KEMs) specified in [BCLV17, BCLV19] consists of Streamlined NTRU Prime and NTRU LPrime, but we only consider the former since it is more implementation-friendly. Streamlined NTRU Prime is similar to classical NTRU, but adopts a rounding technique in the encapsulation and, as explained above, uses a field instead of a ring.

Notation and parameters. A parameter set for Streamlined NTRU Prime consists of the triple (p, q, w), which defines the main algebraic structures. The parameter p is the degree of the irreducible polynomial $P = x^p - x - 1$ and is prime; the parameter sets given in [BCLV19] use 653, 761, and 857. Also the modulus q, which represents the characteristic of the field $\mathcal{R}/q = (\mathbb{Z}/q)[x]/P$, is a prime with typical values of 4621, 4591, and 5167, respectively, for the three degrees considered in [BCLV19]. The weight parameter w is a positive integer that defines the number of non-0 coefficients of certain polynomials, which is respectively 288, 286, 322 in three parameter sets. A valid parameter set has to satisfy $2p \geq 3w$ and $q \geq 16w + 1$. Reusing the notation of [BCLV19], we abbreviate the ring $\mathbb{Z}[x]/P$, the ring $(\mathbb{Z}/3)[x]/P$, and the field $(\mathbb{Z}/q)[x]/P$ as $\mathcal{R}$, $\mathcal{R}/3$, and $\mathcal{R}/q$, respectively. An element of the ring $\mathcal{R}$ is *small* if all its coefficients are in $\{-1, 0, 1\}$. Short is defined as the set of small weight-w elements of $\mathcal{R}$, while Rounded is the set of polynomials $r(x) \in \mathcal{R}$ where each coefficient r_i lies is the range $[-(q-1)/2, (q-1)/2]$ and is rounded to the nearest multiple of 3.

Key generation. To generate a key pair for Streamlined NTRU Prime, the following operations have to be performed (note that, for brevity, we skip some operations such as the encoding of polynomials to strings).

1. Generate a uniform random *small* polynomial $g(x) \in \mathcal{R}$. Repeat this step until $g(x)$ is invertible in $\mathcal{R}/3$.
2. Compute $v(x) = 1/g(x)$ in $\mathcal{R}/3$.
3. Generate a uniform random polynomial $f(x) \in$ Short.
4. Compute $h(x) = g(x)/(3f(x))$ in $\mathcal{R}/q$.
5. Generate a uniform random polynomial $\rho(x) \in$ Short.
6. Output $h(x)$ as *public key* and $(f(x), v(x), h(x), \rho(x))$ as *private key*.

Encapsulation. The encapsulation operation gets a public key as input and produces a ciphertext and session key as output (again, for brevity, we skip all encoding and decoding operations).

1. Generate a uniform random polynomial $r(x) \in$ Short.
2. Compute $c(x) = h(x)r(x) \in$ Rounded.
3. Compute $C = (c(x), \textsc{Hash}(r(x), h(x)))$.
4. Output C as *ciphertext* and $\textsc{Hash}(1, r(x), C)$ as *session key*.

Decapsulation. The decapsulation gets a key pair and a ciphertext as input and produces a session key as output (encodings and decodings are skipped).

1. Compute $e(x) = 3f(x)c(x) \in \mathcal{R}/q$ and represent each coefficient e_i of $e(x)$ as an integer between $-(q-1)/2$ and $(q-1)/2$.
2. Compute $e(x) = e(x) \bmod 3 \in \mathcal{R}/3$ (i.e., reduce each e_i modulo 3).
3. Compute $r'(x) = e(x)v(x) \in \mathcal{R}/3$.
4. Lift $r'(x) \in \mathcal{R}/3$ to a small polynomial $r'(x) \in \mathcal{R}$.
5. If the weight of $r'(x)$ is not w then set $r'(x) = (1, 1, \ldots, 1, 0, 0, \ldots, 0)$.

6. Compute $c'(x) = h(x)r'(x) \in$ Rounded.
7. Compute $C' = (c'(x), \text{HASH}(r'(x), h(x)))$.
8. If C' equals C then output $\text{HASH}(1, r'(x), C)$ else output $\text{HASH}(0, \rho(x), C)$ as *session key*.

3.3 Implementation

Polynomial multiplications in streamlined NTRU Prime. Since Streamlined NTRU Prime is closely related to the classical NTRU scheme (i.e., NTRUEncrypt), it is not surprising that they share many implementation aspects; in particular, they have in common that their performance depends to a large extent on the polynomial arithmetic. However, the underlying algebraic structures are (slightly) different: NTRUEncrypt is based on the residue class ring $\mathcal{R} = (\mathbb{Z}/q)[x]/(x^N - 1)$ where q is a power of two, while NTRU Prime uses the extension field $(\mathbb{Z}/q)[x]/(x^p - x - 1)$ where q is a prime, e.g., $q = 4621$. The reduction modulo q is basically free in the former case, but relatively expensive for NTRU Prime, especially when constant execution time is required so as to foil timing attacks. Furthermore, the irreducible polynomial P of NTRU Prime contains an additional non-0 coefficient, which makes the reduction operation more costly. Finally, most performance-optimized implementations of classical NTRU for constrained IoT devices use a parameter set with so-called product-form polynomials [HS03] to minimize the execution time of the ring multiplication (see, e.g., [BCE+01, CGRR21]). However, product-form parameter sets were not included in the NTRU Prime specification. For all these reasons, one can expect the arithmetic part of NTRU Prime, when implemented for an 8-bit AVR microcontroller, to be significantly slower than that of the classical NTRU cryptosystem.

The encapsulation operation of NTRU Prime includes a single polynomial multiplication where one operand is an element of $\mathcal{R}/q$ (i.e., its coefficients are bounded by q) and the other operand is an element of Short, which means it is a ternary polynomial with exactly w non-0 coefficients. Hence, the polynomial multiplication carried out in NTRU Prime encapsulation is very similar to the ring multiplication in the encryption operation of classical NTRU [HPS98]. On the other hand, the decapsulation of NTRU Prime involves three polynomial multiplications, which is one more than the number of multiplications that have to be executed in classical NTRU decryption. The first polynomial multiplication in the decapsulation gets an element of Rounded (i.e., an element of $\mathcal{R}/q$) and an element of Short as input. In contrast, the second polynomial multiplication (Step 3 of the decapsulation as presented in the previous section) is performed on two elements of $\mathcal{R}/3$, i.e., two ternary polynomials. The third multiplication of the decapsulation is exactly the same as the polynomial multiplication in the encapsulation, which means the operands are elements of $\mathcal{R}/q$ and Short.

Karatsuba-based polynomial multiplication. Most algorithms for high-speed polynomial multiplication have their origins in well-known algorithms for multiple-precision multiplication of integers, such as needed for common public-key cryptosystems like RSA and ECC [DHH+15, HMV04]. From a high-level perspective, polynomial multiplication algorithms can be split into two main categories, namely basic techniques that require n^2 coefficient multiplications to obtain the product of two polynomials consisting of n coefficients each, and advanced techniques with sub-quadratic complexity, e.g., Karatsuba's algorithm [KO63]. Examples of the former category are the *operand-scanning* and *product-scanning* method, which produce the coefficient-products in a row-wise or column-wise fashion and differ with respect of the number of load and store instructions they need to execute [HMV04]. The so-called *hybrid technique* proposed in [GPW+04] is beneficial on microcontrollers with a large number of general-purpose registers (e.g., AVR ATmega) and combines the individual strengths of operand scanning and product scanning. It has a "nested loop" structure and computes $d \geq 2$ coefficient-products in each iteration of the inner loop, which reduces the number of load instructions by a factor of d compared to product scanning.

Multiplication algorithms with sub-quadratic complexity have been known since the 1960s when Karatsuba published his seminal paper [KO63]. Karatsuba's method reduces a multiplication of two operands consisting of n coefficients to three multiplications of $(n/2)$-coefficient polynomials and a few additions. The half-size multiplications, in turn, can be implemented using any multiplication technique, including conventional operand and product scanning, as well as the hybrid method. Alternatively, it is possible to apply the Karatsuba algorithm recursively until the operands consist of just a single coefficient, in which case the asymptotic complexity becomes $\Theta(n^{\log_2 3})$. Yet another option is the so-called Arbitrary Degree Karatsuba (ADK) variant described and analyzed in detail in [Sco18]. Also a few multiplication algorithms with even better asymptotic complexity have been studied; an example is the Toom-Cook multiplication we mentioned in Section 3.1 in the context of Kannwischer et al.'s work on polynomial multiplication for ARM Cortex-M4 processors [KRS19]. An efficient implementation of a 4-way Toom-Cook algorithm for multiplication of degree-256 polynomials on a Cortex-M4 device is described in [KBSV18].

Multiplication strategy. Finding the optimal multiplication strategy for the two forms of polynomial multiplication mentioned at the beginning of this section (i.e., $\mathcal{R}/q \times$ Short and $\mathcal{R}/3 \times \mathcal{R}/3$) is a difficult task. Intuitively, one may assume that a combination of multiplication techniques with sub-quadratic and quadratic complexity will yield peak performance. Yet, the concrete implementation of such a combined strategy raises a few non-trivial questions. Asymptotic complexity bounds are not always meaningful in the real world, especially when the involved operands are relatively short. Therefore, it is necessary to find out which sub-quadratic algorithms are most efficient ones for the multiplications in NTRU Prime (this depends besides the lengths of the polynomials also on certain characteristics of the target architecture). For constrained platforms like 8-bit AVR, it makes sense to base this decision not solely on speed but also on RAM requirements and code size. A second important question is how many recursions of Karatsuba's and/or Toom-Cook's algorithm should be performed before switching to a multiplication method with quadratic complexity, i.e., what operand length is the "crossover" point? Finally, a third question is which of the basic algorithms should be used: operand scanning, product scanning, or the hybrid method? In order to answer all these questions, we conducted a multitude of experiments with different sub-quadratic algorithms[1] (e.g., Karatsuba algorithm and ADK), different numbers of recursions of the sub-quadratic algorithms (i.e., different "crossover" points), and different basic multiplication techniques with quadratic complexity (e.g., operand-scanning and product-scanning).

The results of these experiments show that for a polynomial multiplication of the form $\mathcal{R}/q \times$ Short (carried out in Step 2 of encapsulation as well as Step 1 and 6 of decapsulation), five recursions of Karatsuba's algorithm provide the best performance across all parameter sets specified in [BCLV19]. Below the five levels of Karatsuba, the normal product-scanning technique is used since, due to the bitlength of the coefficient-products and the limited register space, the hybrid multiplication is not efficient. Also alternative Karatsuba variants, such as the ADK algorithm from [Sco18], did not yield superior performance. The situation is different for the polynomial multiplication of the form $\mathcal{R}/3 \times \mathcal{R}/3$, which has to be carried out in Step 3 of the decapsulation. For this multiplication, a combination of the (recursive) Karatsuba algorithm and hybrid method achieves the best results. To be precise, we reached peak performance with four recursions of Karatsuba and using the hybrid method with $d = 4$ at the "lower level" (this is possible because the coefficient-products are relatively small and, thus, more free registers are available). We implemented Karatsuba's algorithm in C and the hybrid multiplication method in both C and AVR assembler, whereby the latter is very similar to the implementations described in [GPW+04, DHH+15].

[1] As stated in Section 3.1, we do not consider the Toom-Cook multiplication algorithm due to its high RAM consumption. The AVR device we use for benchmarking, an ATmega1284 microcontroller, has only 16 kB SRAM, which makes a strong case to take memory requirements into account in the algorithm exploration.

Algorithm 3.1: Table-based constant-time coefficient-reduction modular q.

Input: Integer s of a length of (up to) 29 bits, modulus q of a fixed length of 13 bits.
Output: $r = s \bmod q$.

```
1 b ← (s_28, ..., s_24)      /* extract the five bits b = (s_28, ..., s_24) from s */
2 r ← RT1[b]                 /* reduce b2^24 modulo q via look-up table RT1 */
3 b ← (s_23, ..., s_16)      /* extract the eight bits b = (s_23, ..., s_16) from s */
4 r ← r + RT2[b]             /* reduce b2^16 modulo q via look-up table RT2 */
5 r ← r + s & 0xffff         /* add 16 least-significant bits of s to r */
6 b ← (r_16, ..., r_12)      /* extract the five bits b = (r_16, ..., r_12) from r */
7 r ← (r & 0xfff) + RT3[b]   /* reduce b2^12 modulo q via look-up table RT3 */
8 r ← r − q · (r ⩾ q)        /* conditionally subtract q from r */
9 return r
```

A multiplication of two polynomials of degree $p-1$ through a combination of Karatsuba's algorithm and the hybrid method (or any other multiplication technique) yields a product-polynomial $r(x)$ of degree $2p-2$, which has to be reduced modulo the irreducible polynomial $P = x^p - x - 1$ to get a polynomial of degree $p-1$. Thanks to the relation $x^p \equiv x + 1 \bmod P$, this reduction can be performed by simply substituting each term $r_i x^i$ with $i \geq p$ in $r(x)$ by the sum $r_i x^{i-p+1} + r_i x^{i-p}$ [BCLV19]. These substitutions are nothing else than additions of the $p-1$ higher coefficients r_i to r_{i-p+1} and r_{i-p}, which reduces the degree of $r(x)$ to (at most) p so that two further coefficient additions suffice to obtain a result of degree $p-1$. Thus, the cost of the reduction modulo P amounts to $2p$ additions of (unreduced) coefficients. The final step of the multiplication is the reduction of the $p-1$ remaining coefficients modulo q or modulo 3.

Coefficient-reduction modulo q. As explained above, we implemented the multiplication of the form $\mathcal{R}/q \times \text{Short}$ using five recursions of Karatsuba as "higher level" algorithm and product scanning at the "lower level." Taking the parameter set `sntrup653` as example, we have $p = 653$, which means the hybrid method is executed with operands of degree $\lceil 653/2^5 \rceil = 21$. Furthermore, since $q = 4621$ and we represent the -1 coefficients of a ternary polynomial (i.e., an element of Short) as $q - 1 = 4620$, a single coefficient-product has a maximum length of 24 bits. The column sum to which the 24-bit coefficient-products are accumulated can become up to 29 bits long, i.e., we need an efficient algorithm for reducing a 29-bit integer modulo a 13-bit integer.

Algorithm 3.1 shows a generic technique for reducing a 29-bit integer modulo an arbitrary 13-bit integer q using three look-up tables, which we call reduction tables. It is assumed that the input s (representing a column sum of the hybrid method described above) is held in four 8-bit registers, i.e., the individual bytes of s can be conveniently accessed. At first, the five most-significant bits of s are assigned to b and then $b2^{24} \bmod q$ is computed with the help of reduction table RT1, which contains 32 entries. Next, the second-most significant byte of s is processed in a similar way, whereby the 256-entry table RT2 is used to obtain its residue modulo q. The two residues are added up and form the intermediate result r. Then, we extract the 16 least-significant bits from s and add them to r, which has now a length of at most 17 bits. Similar as before, we assign the five most-significant bits of r to b, reduce it using RT3, and add the residue to the 12 least-significant bits of r. Because r is now always less than $2q$, a single subtraction of q is sufficient to have a fully reduced result. However, to ensure constant execution time, we first compare r with the modulus q, which returns 1 if $r \geq q$ and 0 otherwise. This comparison-result is multiplied by q and the product (either q or 0) is then subtracted from r. Note that Algorithm 3.1 works for any 13-bit modulus q, though each q requires its own set of tables.

Table 3.1: Execution time (in cycles) of the `__udivmodhi4` function for all 2^{16} possible 16-bit unsigned integers. Columns labeled with "Frequency" and "%" give the frequency (in absolute numbers) and probability (in percentage) of the occurrence of the cycle count.

Cycles	Frequency	%	Cycles	Frequency	%	Cycles	Frequency	%
193	3	0.005	198	7956	12.140	203	3825	5.836
194	45	0.069	199	12243	18.681	204	1323	2.019
195	312	0.476	200	14121	21.547	205	312	0.476
196	1323	2.019	201	12244	18.683	206	45	0.069
197	3825	5.836	202	7956	12.140	207	3	0.005

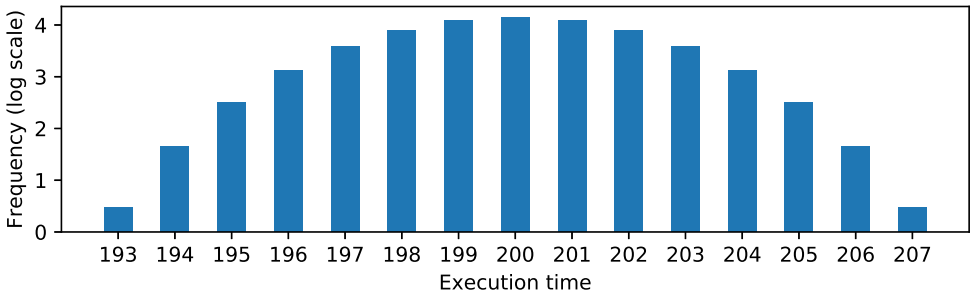


Figure 3.1: Frequency of the occurrence (in absolute numbers) of a certain execution time (in cycles) of the `__udivmodhi4` function for all 2^{16} possible 16-bit unsigned integers.

Coefficient-reduction modulo 3. The reduction modulo 3 can exploit the fact that some multiples of 3 (e.g., 15, 255) have the form $2^k \pm 1$, which allows for a particularly efficient implementation. Thus, the reduction modulo 3 is less costly (in terms of look-up tables) than the modulo-q case, but requires special attention regarding timing attacks. Namely, as described in Section 3.2, one of the operands of the $\mathcal{R}/3 \times \mathcal{R}/3$ multiplication in the decapsulation is $v(x)$, which is a part of the private key. Therefore, an implementer has to take care that this multiplication, including the reduction of all coefficient-products modulo 3, has constant execution time. When using C or C++, a modulo-3 reduction can be implemented by an operation of the form `y = x % 3`, whereby in our case `x` is a 16-bit integer. However, in the course of our work we found out that one can not take it for granted that a C compiler generates constant-time code for this operation. Concretely, we discovered that certain versions of `avr-gcc` generate code with operand-dependent execution time for some AVR models, which can leak information about the secret polynomial $v(x)$.

For example, we determined the execution time of the modulo-3 reduction compiled with `avr-gcc` 4.8.2 for an ATtiny45 microcontroller with help of the cycle-accurate simulator Avrora [TLP05]. For target devices that have no hardware multiplier, e.g., ATtiny MCUs, `avr-gcc` uses the `__udivmodhi4` function from the runtime library `libgcc` to perform the reduction modulo 3. The same function was also used for devices with hardware multiplier, including the ATmega1284 (our benchmarking device, see Section 3.4), until version 4.7.0 of the `avr-gcc` compiler; thereafter it was replaced with `__umulhisi3` [GCC17]. While the latter function has a constant execution time (i.e., 54 cycles) for all 2^{16} possible inputs, the time required by the former depends on the value of the operand to be reduced. Concretely, the execution time of `__udivmodhi4` varies between 193 clock cycles (for input values 0, 1, and 2) and 207 cycles (for 49149, 49150, and 49151). Thus, the time difference between the longest and shortest execution is 14 cycles. Further details are provided in Table 3.1 and Figure 3.1. In order to ensure that the resistance against timing attacks does not depend on the compiler, we implemented the modulo-3 reduction in assembly language following the approach described in [CGRR21].

Table 3.2: Execution time (in clock cycles) and code size (in bytes) of the main arithmetic operations and full encapsulation and decapsulation of NTRU Prime using the parameter set `sntrup653` on an ATmega1284 microcontroller.

Operation	Time	Size
$\mathcal{R}/q \times$ Short multiplication	5,604,929	2,230
$\mathcal{R}/3 \times \mathcal{R}/3$ multiplication	1,277,675	1,510
Full encapsulation	8,160,665	8,694
Full decapsulation	15,602,748	11,478

Table 3.3: Comparison of our NTRU Prime implementation with AVR implementations of other algorithms (all of which target 128 bits of security).

Implementation	Algorithm	Platform	Encaps	Decaps
This work	NTRU Prime	ATmega1284	8,160,665	15,602,748
Cheng et al. [CGRR21]	NTRU (PF)	ATmega1281	847,973	1,051,871
Düll et al. [DHH+15]	X25519	ATmega2560	13,900,397	13,900,397

3.4 Evaluation

The 8-bit AVR device we used to test and benchmark our NTRU Prime implementation is an ATmega1284 microcontroller, which features 16 kB SRAM and 128 kB flash memory for storing program code. Our software consists of a mix of C and assembly language; we implement the main arithmetic operations in assembly to achieve fast and operand-independent execution time, whereas all functions that are neither performance-critical nor security-critical are written in C to maximize portability. We use the optimized Assembler implementation of the SHA-512 hash function introduced in [CDG18] to minimize the execution time of certain auxiliary functions that are performance-critical. When executed on our target device, the compression function of SHA-512 takes slightly less than 60 k clock cycles, which corresponds to a compression rate of about 467 cycles per byte. Our implementation of (Streamlined) NTRU Prime can be compiled with Atmel Studio v7.0 under the `-O2` optimization option, which produces an executable that, according to our experiments, does not leak secret information through execution time and can, therefore, withstand timing attacks.

NTRU Prime. Table 3.2 summarizes the execution time and code size of the core arithmetic operations (i.e., polynomial multiplications) as well as a full encapsulation and decapsulation of our NTRU Prime software. The table shows the results of implementation of the polynomial multiplication of the form $\mathcal{R}/q \times$ Short and the form $\mathcal{R}/3 \times \mathcal{R}/3$. The latter is much faster (compared to the former) due to the smaller coefficients (enabling faster coefficient multiplication), smaller intermediate results (requiring fewer registers) and faster reduction (modulo 3 vs. modulo q). Also given in Table 3.2 are the execution times of encapsulation and decapsulation, which are primarily dominated by the polynomial arithmetic. The encapsulation includes just a single multiplication, namely a multiplication of an element of $\mathcal{R}/q$ by an element of Short (i.e., $\mathcal{R}/q \times$ Short) that accounts for roughly two thirds of the overall execution time. On the other hand, the decapsulation operation has to perform three polynomial multiplications (two of the form $\mathcal{R}/q \times$ Short and one of the form $\mathcal{R}/3 \times \mathcal{R}/3$); together they contribute 80% to the overall execution time.

Comparison with AVR implementations of other key-establishment. Our software is, to the best of our knowledge, the first optimized implementation of Streamlined NTRU Prime for AVR devices. The AVR assembler implementation of classical NTRU (i.e., NTRUEncrypt with `ees443ep1` parame-

ters) introduced in [CGRR21] uses a highly efficient product-form (PF) convolution and outperforms our NTRU Prime software by roughly an order of magnitude. On the other hand, our NTRU Prime encapsulation is much faster than a variable-base scalar multiplication on Curve25519, while the decapsulation is a bit slower.

3.5 Conclusion

We presented the first highly-optimized implementation of NTRU Prime for an 8-bit microcontroller that is capable to resist timing attacks. When executed on an ATmega1284 device, the encapsulation takes about 8.2 million cycles, while the decapsulation has an execution time of 15.6 million cycles (both results are based on the parameter set `sntrup653`). To achieve these results, we implemented all expensive operations in AVR assembly language, most notably the polynomial arithmetic, whereby we strived for a balance between execution time and code size. Furthermore, we showed that one cannot count on a C compiler to generate constant-time code for the modulo-3 reduction, which generally raises concerns about the security (i.e., resistance against timing attacks) of C implementations of NTRU Prime. In summary, our results show that NTRU Prime can be well optimized to run efficiently on small microcontrollers, which makes it an interesting candidate for securing the post-quantum IoT. Finally, although not discussed and verified in this Chapter, using signed integer arithmetic (instead of the unsigned version) might have the potential to further improve the performance.

CHAPTER

4

LIGHTWEIGHT THREEBEARS

This Chapter is based on our paper [CGRR20]. While we were conducting the research work described in this Chapter, the NIST PQC standardization process was in its second round.

4.1 Introduction

Lattice-based candidates. Lattice-based cryptosystems are considered the most promising candidates for deployment in constrained devices due to their relatively low computational cost and reasonably small keys and ciphertexts (resp. signatures). Indeed, the benchmarking results collected in the course of the `pqm4` project [PQM4], which uses a 32-bit ARM Cortex-M4 as target device, show that most of the lattice-based Key-Encapsulation Mechanisms (KEMs) in the second round of the evaluation process are faster than ECDH key exchange based on Curve25519 [Ber06], i.e. X25519, and some candidates are even notably faster than X25519 [PQM4]. However, the results of `pqm4` also indicate that lattice-based cryptosystems generally require a large amount of run-time memory since most of the benchmarked lattice KEMs have a RAM footprint of between 5 kB and 30 kB. For comparison, a variable-base scalar multiplication on Curve25519 can have a RAM footprint of less than 500 bytes [DHH+15]. One could argue that the `pqm4` implementations have been optimized to reach high speed rather than low memory consumption, but this argument is not convincing since even a conventional implementation of X25519 (i.e., an implementation without any specific measures for RAM reduction) still needs only little more than 500 bytes RAM. Therefore, the existing implementation results in the literature lead to the conclusion that lattice-based KEMs require an order of magnitude more RAM than ECDH key exchange.

ThreeBears. The high RAM requirements of lattice-based cryptosystems (in relation to X25519) pose a serious problem for the emerging Internet of Things (IoT) since many IoT devices feature only a few kB of RAM. For example, a typical wireless sensor node like the MICAz mote [Cro06] is equipped with an 8-bit micro- controller (e.g., ATmega128L) and comes with only 4 kB internal SRAM. These 4 kB are easily sufficient for X25519 (since there would still be 7/8 of the RAM available for system and application software), but not for lattice-based KEMs. Thus, there is a clear need to research how lattice-based cryptosystems can be optimized to reduce their memory consumption and what performance such low-memory implementations can reach. The present paper addresses this research need

and introduces various software optimization techniques for the ThreeBears KEM [Ham19], a lattice-based cryptosystem that was selected for the second round of NIST's standardization project. The security of ThreeBears is based on a special version of the Learning With Errors (LWE) problem, the so-called Integer Module Learning with Errors (I-MLWE) problem [Gu17]. ThreeBears is unique among the lattice-based second-round candidates since it uses an integer ring instead of a polynomial ring as algebraic structure. Hence, the major operation of ThreeBears is integer arithmetic (namely multiplication modulo a 3120-bit prime) and not polynomial arithmetic.

Optimization techniques for polynomial multiplication. The conventional way to speed up the polynomial multiplication that forms part of lattice-based cryptosystems is to use a multiplication technique with sub-quadratic complexity, e.g., Karatsuba's method [KO63] or the so-called Toom-Cook algorithm [Too63, Coo66] or the Number Theoretic Transform (NTT). However, the performance gain due to these techniques comes at the expense of a massive increase of the RAM requirements. For integer multiplication, on the other hand, there exists a highly effective approach for performance optimization that does not increase the memory footprint, namely the so-called hybrid multiplication method from CHES 2004 [GPW+04] or one of its variants like the Reverse Product Scanning (RPS) method [LSGK14]. In essence, the hybrid technique can be viewed as a combination of classical operand scanning and product scanning with the goal to reduce the number of load instructions by processing several bytes of the two operands in each iteration of the inner loop. Even though the hybrid technique can also be applied to polynomial multiplication, it is, in general, less effective because the bit-length of the polynomial coefficients of most lattice-based cryptosystems is not a multiple of eight.

Contributions. This work analyzes the performance of ThreeBears on an 8-bit AVR microcontroller and studies its flexibility to achieve different trade-offs between execution time and RAM footprint. Furthermore, we describe (to the best of our knowledge) the first highly-optimized software implementations of BabyBear (an instance of ThreeBears with parameters to reach NIST's security category 2) for the AVR platform. We developed four implementations of BabyBear, two of which are optimized for low RAM consumption, and the other two for fast execution times. Our two low-RAM BabyBear versions are the most memory-efficient software implementations of a NIST second-round candidate ever reported in the literature.

Our work is based on the optimized C code contained in the ThreeBears submission package [Ham19], which adopts a "reduced-radix" representation for the ring elements, i.e., the number of bits per limb is less than the word-size of the target architecture. On a 32-bit platform, a 3120-bit integer can be stored in an array of 120 limbs, each consisting of 26 bits. However, our AVR software uses a radix of 2^{32} (i.e., 32 bits of the operands are processed at a time) since this representation enables the RPS method to reach peak performance and it also reduces the RAM footprint. We present two optimizations for the performance-critical Multiply-ACcumulate (MAC) operation of ThreeBears; one aims to minimize the RAM requirements, while the goal of the second is to maximize performance. Our low-memory implementation of the MAC combines one level of Karatsuba with the RPS method [LSGK14] to accelerate the so-called *tripleMAC* operation of the optimized C source code from [Ham19], which is (relatively) light in terms of stack memory. On the other hand, the speed-optimized MAC consists of three recursive levels of Karatsuba multiplication and uses the RPS method underneath. We implemented both MAC variants in AVR Assembly language to ensure they have constant execution time and can resist timing attacks.

As already mentioned, our software contains four different implementations of the ThreeBears family: two versions of CCA-secure BabyBear, and two versions of CPA-secure BabyBearEphem. For both BabyBear and BabyBearEphem, we developed both a Memory-Efficient (ME) and a High-Speed (HS) implementation, which internally use the corresponding MAC variant. We abbreviate these four versions as ME-BBear, ME-BBear-Eph, HS-BBear, and HS-BBear-Eph. Our results show that Three-

BEARS provides the flexibility to optimize for low memory footprint *and* still achieves very good execution times compared to the other second-round candidates. In particular, the CCA-secure BABYBEAR can be optimized to run with only 2.4 kB RAM on AVR, and the CPA-secure version requires even less memory, namely just 1.7 kB.

4.2 Background

THREEBEARS has three parameter sets called BABYBEAR, MAMABEAR, and PAPABEAR, matching NIST security categories 2, 4, and 5, respectively. Each parameter set comes with two instances, one providing CPA security and the other CCA security. Taking BABYBEAR as example, the CPA-secure instance is named BABYBEAREPHEM (with the meaning of ephemeral BABYBEAR), while the CCA-secure one is simply called BABYBEAR. In the following, we only give a short summary of the CCA-secure instance of THREEBEARS. In contrast to encryption schemes with CCA-security, CPA-secure ones, roughly speaking, do not repeat and verify the key generation and encryption (i.e., encapsulation) as part of the decryption (i.e., decapsulation) procedure, see [Ham19] for details.

Notation and parameters. THREEBEARS operates in the field $\mathbb{Z}/N$, where the prime modulus $N = 2^{3120} - 2^{1560} - 1$ is a so-called "golden-ratio" Solinas trinomial prime [Ham15]. N is commonly written as $N = \phi(x) = x^D - x^{D/2} - 1$. The addition and multiplication $(+, *)$ in $\mathbb{Z}/N$ will be explained in Section 4.3.1. An additional parameter d determines the the module dimension; this dimension is 2 for BABYBEAR, 3 for MAMABEAR, and 4 for PAPABEAR, respectively.

Key generation. To generate a key pair for THREEBEARS, the following operations have to be performed:

1. Generate a uniform and random string sk with a fixed length.
2. Generate two noise vectors $(a_0, \ldots, a_{d-1})$ and $(b_0, \ldots, b_{d-1})$, where $a_i, b_i \in \mathbb{Z}/N$ is sampled from a noise sampler using sk.
3. Compute $r = \text{HASH}(sk)$.
4. Generate a $d \times d$ matrix $\boldsymbol{M}$, where each element $M_{i,j} \in \mathbb{Z}/N$ is sampled from a uniform sampler using r.
5. Obtain vector $z = (z_0, \ldots, z_{d-1})$ by computing each $z_i = b_i + \Sigma_{j=0}^{d-1} M_{i,j} * a_j \bmod N$
6. Output sk as *private key* and (r, z) as *public key*.

Encapsulation. The encapsulation operation gets a public key (r, z) as input and produces a ciphertext and shared secret as output:

1. Generate a uniform and random string *seed* with a fixed-length.
2. Generate two noise vectors $(\hat{a}_0, \ldots, \hat{a}_{d-1})$, $(\hat{b}_0, \ldots, \hat{b}_{d-1})$ and a noise c, where $\hat{a}_i, \hat{b}_i, c \in \mathbb{Z}/N$ is sampled from noise sampler by given r and *seed*.
3. Generate a $d \times d$ matrix $\boldsymbol{M}$, where each element $M_{i,j} \in \mathbb{Z}/N$ is sampled from uniform sampler by given r.
4. Obtain vector $\boldsymbol{y} = (y_0, \ldots, y_{d-1})$ by computing each $y_i = \hat{b}_i + \Sigma_{j=0}^{d-1} M_{j,i} * \hat{a}_j \bmod N$, and compute $x = c + \Sigma_{j=0}^{d-1} z_j * \hat{a}_j \bmod N$.

5. Use Melas FEC encoder to encode *seed*, and use this encoded output together with x to extract a fixed-length string f.
6. Compute $ss = \text{HASH}(r, seed)$.
7. Output ss as *shared secret* and $(f, \boldsymbol{y})$ as *ciphertext*.

Decapsulation. The decapsulation gets a private key sk and ciphertext $(f, \boldsymbol{y})$ as input and produces a shared secret as output:

1. Generate a noise vector $(a_0, \ldots, a_{d-1})$, where $a_i \in \mathbb{Z}/N$ is sampled from a noise sampler by given sk.
2. Compute $x = \Sigma_{j=0}^{d-1} y_j * a_j \bmod N$.
3. Derive a string from f together with x, and use Melas FEC decoder to decode this string to obtain the string *seed*.
4. Generate the public key $(r', \boldsymbol{z'})$ through Key Generation by given sk.
5. Repeat Encapsulation to get ss' and $(f', \boldsymbol{y'})$ by using the obtained *seed* and key pair $(sk, (r', \boldsymbol{z'}))$.
6. Check whether $(f', \boldsymbol{y'})$ equals to $(f, \boldsymbol{y})$; if they are equal then output ss' as *shared secret*; if not then output $\text{HASH}(sk, f, \boldsymbol{y})$ as *shared secret*.

Auxiliary functions. The three operations described above use a few auxiliary functions such as samplers (noise sampler and uniform sampler), hash functions, and a function for the correction of errors. Both the samplers and hash functions are based on `cSHAKE256` [KCP16], which uses the Keccak permutation [BDPA13] at the lowest layer. In addition, THREEBEARS adopts Melas BCH code for Forward Error Correction (FEC) since it is very fast, has low RAM consumption and small code size, and can also be easily implemented to have constant execution time.

The impact of polynomial multiplication on performance. We determined the execution time of several implementations contained in the THREEBEARS NIST submission package on an AVR microcontroller. Like is the case with other lattice-based cryptosystems, the arithmetic computations of THREEBEARS determine the memory footprint and have a big impact on the overall execution time. Hence, our work primarily focuses on the optimization of the costly MAC operation of the form $r = r + a * b \bmod N$. Concerning the auxiliary functions, we were able to significantly improve their performance (in relation to the C code of the submission package) thanks to a highly-optimized AVR assembler implementation of the permutation of Keccak[1]. Further details about the auxiliary functions are outside of the scope of this work; we refer the reader to the specification of THREEBEARS [Ham19].

The specific AVR device used. The specific AVR microcontroller on which we simulated the execution time of our software is the ATmega1284; it features 16 kB SRAM and 128 kB flash memory for storing program code.

[1] https://github.com/XKCP/XKCP/tree/master/lib/low/KeccakP-1600/AVR8.

4.3 Implementation

The Multiply-ACcumulate (MAC) operation of ThreeBears, in particular the 3120-bit multiplication that is part of it, is very costly on 8-bit microcontrollers and requires special attention. This section deals with optimization techniques for the MAC operation on the AVR platform. As already stated in Section 4.1, we follow two strategies to optimize the MAC, one whose goal is to minimize the RAM footprint, whereas the other aims to maximize performance. The result is a memory-optimized MAC and a speed-optimized MAC, which are described in Section 4.3.3 and Section 4.3.4, respectively.

4.3.1 The MAC operation of ThreeBears

ThreeBears defines its field operations $(+, *)$ as

$$a + b := a + b \bmod N \quad \text{and} \quad a * b := a \cdot b \cdot x^{-D/2} \bmod N$$

where $+$ and $\cdot$ (at the right hand side of the equations) are the conventional integer addition and multiplication, respectively. Note that a so-called clarifier $x^{-D/2}$ is multiplied with the factors in the field multiplication, which serves to reduce the distortion of the noise. As shown in [Ham15], the Solinas prime N enables very fast Karatsuba multiplication [KO63]. We can write the field multiplication in the following way, where $\lambda = x^{D/2}$ and the subscripts H and L are used to denote the higher/lower half of an integer:

$$\begin{aligned} z &= a * b = a \cdot b \cdot \lambda^{-1} = (a_L + a_H\lambda)(b_L + b_H\lambda) \cdot \lambda^{-1} \\ &= a_L b_L \lambda^{-1} + (a_L b_H + a_H b_L) + a_H b_H \lambda \\ &= a_L b_L (\lambda - 1) + (a_L b_H + a_H b_L) + a_H b_H \lambda \\ &= (a_L b_H + a_H b_L - a_L b_L) + (a_L b_L + a_H b_H)\lambda \\ &= (a_H b_H - (a_L - a_H)(b_L - b_H)) + (a_L b_L + a_H b_H)\lambda \bmod N \end{aligned} \tag{4.1}$$

Compared to a conventional Karatsuba multiplication, which requires three half-size multiplications and six additions/subtractions, the Karatsuba method for multiplication in $\mathbb{Z}/N$ saves one addition or subtraction. Consequently, the MAC operation can be performed as specified by Equation (4.2) and Equation (4.3):

$$\begin{aligned} r &= r + a * b \bmod N \\ &= (r_L + a_H b_H - (a_L - a_H)(b_L - b_H)) + (r_H + a_L b_L + a_H b_H)\lambda \bmod N \qquad (4.2) \\ &= (r_L + a_H b_L - a_L(b_L - b_H)) + (r_H + (a_L + a_H)b_H + a_L(b_L - b_H))\lambda \bmod N \qquad (4.3) \end{aligned}$$

4.3.2 Full-radix representation for field elements

Implementations provided in the NIST package. The NIST submission package of ThreeBears consists of different implementations, including a reference implementation, optimized implementations, and additional implementations (e.g., a low-memory implementation). As mentioned in Section 4.1, they all use a *reduced-radix* representation for the 3120-bit integers (e.g., on a 32-bit platform, each limb is 26 bits long, and an element of the field consists of 120 limbs). Since this representation implies that there are six free bits in a 32-bit word, it is possible to store the carry (or borrow) bits that are generated during a field operation instead of immediately propagating them to the next-higher word, which reduces dependencies and enables instruction-level parallelism. Modern super-scalar processors can execute several instructions in parallel and, in this way, improve the running time of ThreeBears.

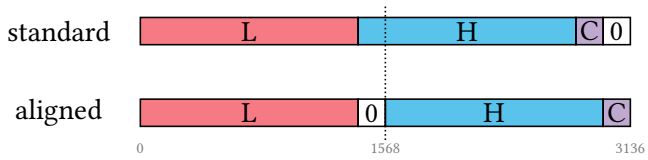


Figure 4.1: Standard and aligned form of a field element (AVR uses little-endian).

Our full-radix representation. Our implementations of BabyBear for AVR use a *full-radix* representation for the field elements for a number of reasons. First, small 8-bit microcontrollers have a single-issue pipeline and can not execute instructions in parallel, even when there are no dependencies among instructions. Furthermore, leaving six bits of "headroom" in a 32-bit word increases the number of limbs (in relation to the full-radix case) and, hence, the number of (32×32)-bit multiplications to be carried out. This is a bigger problem on AVR than on high-end processors where multiplications are a lot faster. Finally, the reduced-radix representation requires more space for a field-element (i.e., larger memory footprint) and more load/store instructions. In our full-radix representation, an element of the field consists of 98 words of a length of 32 bits and consumes $98 \times 4 = 392$ bytes in RAM, while the original representation requires $120 \times 4 = 480$ bytes. The full-radix representation with 32-bit words has also arithmetic advantages since (as mentioned in Section 4.1) it allows one to accelerate the MAC operation using the RPS method [LSGK14]. Thus, we fix the number representation radix to 2^{32}, despite the fact that we are working on an 8-bit microcontroller.

Re-alignment. We define two forms of storage for a full-radix field element: *standard* and *aligned*. Both forms are visually sketched in Figure 4.1, where "L" and "H" stands respectively for the lower and higher 1560 bits of a 3120-bit field element. The standard form is basically the straightforward way of storing a multi-precision integer. Since a 3120-bit integer occupies 98 32-bit words, there are 16 unused bits (i.e., two empty bytes) in the most significant word. In our optimized MAC operations, the result is not always strictly in the range $[0, N)$ but can also be in $[0, 2N)$, which means the second most significant byte is either 0 or 1. We call this byte "carry byte" and mark it with a "C" in Figure 4.1. Furthermore, we use "0" to indicate the most significant byte because it is 0 all the time. The reason why we convert a standard integer into aligned form is because it allows us to perform the Karatsuba multiplication more efficiently. From an implementer's viewpoint, the standard form is suboptimal for Karatsuba since it does not place the lower ("L") and upper ("H") 1560 bits into the lower and upper half of the operand space (see Figure 4.1). Concretely, the lowest byte of the upper 1560 bits is located at the most significant byte of the lower half in the space, which introduces some extra effort for alignment and addressing. The aligned form splits the lower and upper 1560 bits in such a way that they are ideally located for Karatsuba multiplication.

4.3.3 Memory-optimized MAC operation

The NIST submission package of ThreeBears includes a low-memory implementation for each instance, which aims to minimize stack consumption. These low-memory variants are based on a special memory-efficient MAC operation that uses one level of Karatsuba's technique [KO63], which follows a modification of Equation (4.3), namely Equation (4.4) shown below:

$$\begin{aligned} r &= r + a * b \bmod N \\ &= (r_L + a_H b_L - 2a_L(b_L - b_H)) + (r_H + (a_L + a_H)b_H)\lambda + a_L(b_L - b_H)\lambda^2 \bmod N \end{aligned} \tag{4.4}$$

This MAC implements the multiplications using the product-scanning method and operates on reduced-radix words. Our memory-optimized MAC operation was developed on basis of this original low-

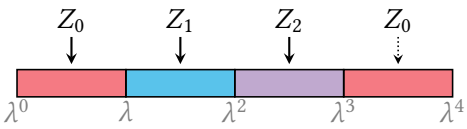


Figure 4.2: Three accumulators for coefficients of $\lambda^0 = 1$, λ, and λ^2 of a product R.

memory MAC, but performs all its computations on aligned full-radix words (after some alignment operations).

Algorithm 4.1 shows our low-RAM one-level Karatsuba MAC, which consists of two major parts: a main MAC loop interleaved with the modular reduction (from line 3 to 23) and a final reduction modulo N (from line 24 to 38). The designer of ThreeBears coined the term "tripleMAC" to refer to the three word-level MACs in the inner loops (line 8 to 10 and 15 to 17). Certainly, this tripleMAC is the most frequent computation carried out by Algorithm 4.1 and dominates the overall execution time. In order to reach peak performance on AVR, we replace the conventional product-scanning technique by an optimized variant of the hybrid multiplication method [GPW+04], namely the so-called Reverse Product-Scanning (RPS) method [LSGK14], which processes four bytes (i.e., 32 bits) of the operands per loop-iteration. In addition, we split each of the inner loops containing a tripleMAC up into three separate loops, and each of these three loops computes one word-level MAC. Due to the relatively small register space of AVR, it is not possible to keep all three accumulators (i.e., Z_0, Z_1, and Z_2) in registers, which means executing three word-level MACs in the same inner loop would require a large number of `LD` and `ST` instructions to load and store the accumulators in each iteration. Thanks to our modification, an accumulator has to be loaded/stored only before/after the whole inner loop.

The three inputs and the output of Algorithm 4.1 are aligned integers, where s is 98 and ω is 32. The parameter β specifies the shift-distance (in bits) when converting an ordinary integer to aligned form ($\beta = 8$ in our case). Each of the three accumulators Z_0, Z_1, and Z_2 in Algorithm 4.1 is 80 bits long and occupies 10 registers. Figure 4.2 illustrates the relation between the accumulators and the coefficients of λ^0, λ, and λ^2 in an aligned output R. Referring to Equation (4.4), we suppose each coefficient can be 3120 bits long, but Z_0, Z_1, and Z_2 accumulate only the lower 1560 bits of the coefficients of λ^0, λ, and λ^2, respectively, in the first tripleMAC (line 8 to 10). After the first inner loop, double of Z_2 must be subtracted from Z_0 (line 11), which corresponds to the operation of the form $a_H b_L - 2a_L(b_L - b_H)$ in Equation (4.4). The second inner loop (beginning at line 13) computes the higher half of each coefficient, but this time the word-products are added to different accumulators compared to the first tripleMAC (e.g., the 64-bit word-products of the form $A_{j+l} \cdot B_k$ are added to Z_1 instead of Z_0). In the second tripleMAC, the factor 2^β needs to be considered in order to ensure proper alignment. The third word-level MAC (at line 17) can be regarded as computing (the lower half of) the coefficient of λ^3. Normally, we should use an additional accumulator Z_3 for this third MAC, but it is more efficient to re-use Z_0. This is possible since, after the second inner loop, we would normally have to compute $Z_1 \leftarrow Z_1 - 2 \cdot Z_3$, a similar operation as at line 11. But because

$$\lambda^3 = \lambda^2 \cdot \lambda = (\lambda + 1) \cdot \lambda = \lambda^2 + \lambda = (\lambda + 1) + \lambda = 2\lambda + 1 \bmod N,$$

we also have to compute $Z_0 \leftarrow Z_0 + Z_3$ and $Z_1 \leftarrow Z_1 + 2 \cdot Z_3$. Combining these two computations with $Z_1 \leftarrow Z_1 - 2 \cdot Z_3$ implies that Z_1 can simply keep its present value and only Z_0 accumulates the value of Z_3. Thus, Algorithm 4.1 does not compute $Z_1 \leftarrow Z_1 - 2 \cdot Z_3$, but instead directly accumulates the sum of the word-products $A_j \cdot (B_k - B_{k+l})$ into Z_0 (which also saves a few load and store instructions). This "shortcut" is indicated in Figure 4.2 with a dashed arrow from Z_0 to the coefficient of λ^3. Line 18 to 23 add the lower 32-bit words of Z_0 and Z_1 to the corresponding words of the result R and right-shift Z_0 and Z_1. The part from line 24 to 27 brings the output of the MAC into a properly aligned form. Thereafter (line 28 to 38), a modulo-N reduction (based on the relation $\lambda^2 \equiv \lambda + 1 \bmod N$) along with a

Algorithm 4.1: Memory-optimized MAC operation.

Input: Aligned s-word integers $A = (A_{s-1}, \ldots, A_1, A_0)$, $B = (B_{s-1}, \ldots, B_1, B_0)$, and $R = (R_{s-1}, \ldots, R_1, R_0)$, each word contains ω bits; β is a parameter of alignment.

Output: Aligned s-word product $R = R + A \cdot B \cdot x^{-D/2} \bmod N = (R_{s-1}, \ldots, R_1, R_0)$.

$Z_0 \leftarrow 0, Z_1 \leftarrow 0$
$l \leftarrow s/2$
for i **from** 0 **to** $l-1$ **by** 1 **do**
    $Z_2 \leftarrow 0$
    $k \leftarrow i+1$
    for j **from** 0 **to** i **by** 1 **do**
        $k \leftarrow k-1$
        $Z_0 \leftarrow Z_0 + A_{j+l} \cdot B_k$
        $Z_1 \leftarrow Z_1 + (A_j + A_{j+l}) \cdot B_{k+l}$
        $Z_2 \leftarrow Z_2 + A_j \cdot (B_k - B_{k+l})$
    $Z_0 \leftarrow Z_0 - 2 \cdot Z_2$
    $k \leftarrow l$
    for j **from** $i+1$ **to** $l-1$ **by** 1 **do**
        $k \leftarrow k-1$
        $Z_1 \leftarrow Z_1 + 2^\beta \cdot A_{j+l} \cdot B_k$
        $Z_2 \leftarrow Z_2 + 2^\beta \cdot (A_j + A_{j+l}) \cdot B_{k+l}$
        $Z_0 \leftarrow Z_0 + 2^\beta \cdot A_j \cdot (B_k - B_{k+l})$
    $Z_0 \leftarrow Z_0 + Z_2 + R_i$
    $Z_1 \leftarrow Z_1 + Z_2 + R_{i+l}$
    $R_i \leftarrow Z_0 \bmod 2^\omega$
    $Z_0 \leftarrow Z_0/2^\omega$
    $R_{i+l} \leftarrow Z_1 \bmod 2^\omega$
    $Z_1 \leftarrow Z_1/2^\omega$
$Z_0 \leftarrow 2^\beta \cdot Z_0 + R_{l-1}/2^{\omega-\beta}$
$Z_1 \leftarrow 2^\beta \cdot Z_1 + R_{s-1}/2^{\omega-\beta}$
$R_{l-1} \leftarrow R_{l-1} \bmod 2^{\omega-\beta}$
$R_{s-1} \leftarrow R_{s-1} \bmod 2^{\omega-\beta}$
$Z_0 \leftarrow Z_0 + Z_1$
for i **from** 0 **to** $l-1$ **by** 1 **do**
    $Z_1 \leftarrow Z_1 + R_i$
    $R_i \leftarrow Z_1 \bmod 2^\omega$
    $Z_1 \leftarrow Z_1/2^\omega$
$Z_0 \leftarrow 2^\beta \cdot Z_0 + R_{l-1}/2^{\omega-\beta}$
$R_{l-1} \leftarrow R_{l-1} \bmod 2^{\omega-\beta}$
for i **from** l **to** $s-1$ **by** 1 **do**
    $Z_0 \leftarrow Z_0 + R_i$
    $R_i \leftarrow Z_0 \bmod 2^\omega$
    $Z_0 \leftarrow Z_0/2^\omega$
return $(R_{s-1}, \ldots, R_1, R_0)$

Algorithm 4.2: Speed-optimized MAC operation.

Input: Aligned field elements $A = (A_H, A_L)$, $B = (B_H, B_L)$ and $R = (R_H, R_L)$
Output: Aligned product $R = R + A \cdot B \cdot x^{-D/2} \bmod N = (R_H, R_L)$

1 $(Z_H, Z_L) \leftarrow (0, 0)$, $(T_H, T_L) \leftarrow (0, 0)$
2 $T_L \leftarrow |A_L - A_H|$
3 **if** $A_L - A_H < 0$ **then**
4 $\quad s_a \leftarrow 1$
5 **else**
6 $\quad s_a \leftarrow 0$
7 $T_H \leftarrow |B_L - B_H|$
8 **if** $B_L - B_H < 0$ **then**
9 $\quad s_b \leftarrow 1$
10 **else**
11 $\quad s_b \leftarrow 0$
12 $(Z_H, Z_L) \leftarrow T_L \cdot T_H \cdot (-1)^{1-(s_a \oplus s_b)}$ /* 1st half-size multiplication */
13 $(R_H, R_L) \leftarrow (R_H, R_L) + (Z_H, Z_L)$
14 $T_L \leftarrow A_H, T_H \leftarrow B_H$
15 $(Z_H, Z_L) \leftarrow T_L \cdot T_H$ /* 2nd half-size multiplication */
16 $R_H \leftarrow R_H + Z_H$
17 $T_L \leftarrow Z_H + Z_L$
18 $R_L \leftarrow R_L + T_L$
19 $R_H \leftarrow R_H + T_L$
20 $T_L \leftarrow A_L, T_H \leftarrow B_L$
21 $(Z_H, Z_L) \leftarrow T_L \cdot T_H$ /* 3rd half-size multiplication */
22 $R_H \leftarrow R_H + Z_L$
23 $R_L \leftarrow R_L + Z_H$
24 $R_H \leftarrow R_H + Z_H$
25 $(R_H, R_L) \leftarrow (R_H, R_L) \bmod N$ /* modular-N reduction and carry propagation */
26 **return** (R_H, R_L)

conversion to 32-bit words is performed. The output of Algorithm 4.1 is an aligned integer in the range of $[0, 2N)$.

We implemented Algorithm 4.1 completely in AVR Assembly language. Even though each accumulator Z_i consists of 80 bits (ten bytes), we only load and store nine bytes of Z_i in each inner loop. A simple analysis shows that that the accumulator values in the first inner loop can never exceed 2^{72}, which allows us to only load and store the nine least-significant bytes. In the second tripleMAC loop, each word-product is multiplied by 2^{β} (i.e., shifted left by eight bits) and so it is not necessary to load-/store the least-significant accumulator byte.

4.3.4 Speed-optimized MAC operation

The MAC operations of the implementations in the NIST submission package of ThreeBears are not suitable to reach high speed on AVR. Therefore, we developed our speed-optimized MAC operation from scratch and implemented it according to a variant of Equation (4.2), namely Equation (4.5) specified below. We divide the three full-size 3120-bit products (e.g., $a_L b_L$) of Equation (4.2) into two halves, and

Table 4.1: Execution time (in clock cycles) of our AVR implementations of BabyBear.

Implementation	Security	MAC	KeyGen	Encaps	Decaps
ME-BBear	CCA-secure	1,033,728	8,746,418	12,289,744	18,578,335
ME-BBear-Eph	CPA-secure	1,033,728	8,746,418	12,435,165	3,444,154
HS-BBear	CCA-secure	604,703	6,123,527	7,901,873	12,476,447
HS-BBear-Eph	CPA-secure	604,703	6,123,527	8,047,835	2,586,202

use l for representing $a_L b_L$, m for $-(a_L - a_H)(b_L - b_H)$ and h for $a_H b_H$:

$$\begin{aligned} r &= (r_L + h + m) + (r_H + l + h)\lambda \mod N \\ &= (r_L + (h_L + h_H\lambda) + (m_L + m_H\lambda)) + (r_H + (l_L + l_H\lambda) + (h_L + h_H\lambda))\lambda \\ &= (r_L + h_L + m_L) + (r_H + l_L + h_L + m_H + h_H)\lambda + (l_H + h_H)\lambda^2 \\ &= (r_L + m_L + \underline{h_L + h_H + l_H}) + (r_H + m_H + h_H + l_L + \underline{h_L + h_H + l_H})\lambda \end{aligned} \tag{4.5}$$

The underlined parts in Equation (4.5) are common parts of the coefficients of λ^0 and λ. Algorithm 4.2 specifies our speed-optimized MAC, which operates on half-size (i.e., 1560-bit) parts of the operands A, B, and R. We omitted the details of the final step (line 19) in Algorithm 4.2, i.e., the modulo-N reduction, because it is very similar to line 27 to 43 in Algorithm 4.1. Compared with Algorithm 4.1, the speed-efficient MAC operation is designed in a more straightforward way since it computes each entire half-size multiplication separately to obtain a full-size intermediate product (line 6, 9, and 15). However, it consumes more memory to store the intermediate products (e.g., Z_H, Z_L and T_H, T_L).

We still take advantage of RPS technique to speed up the inner-loop operation, but combine it Karatsuba's method. Our experiments with different levels of Karatsuba multiplication showed that the 2-level Karatsuba approach with the RPS technique underneath (i.e., 2-level KRPS) yields the best performance for a multiplication of 1560-bit operands. Consequently, we execute three levels of Karatsuba (i.e., 3-level KRPS) altogether for the MAC operation. Each level uses the so-called subtractive Karatsuba algorithm described in [HS15] to achieve fast and constant execution time. All half-size multiplications performed at the second and third level use space that was initially occupied by input operands to store intermediate values, i.e., we do not allocate additional RAM inside the half-size multiplications. This is also the reason for the two operations at line 14 and 20, where we move the operands to T_H and T_L before the multiplication so that we do not modify the inputs A and B.

4.4 Evaluation

Except for the MAC operations, all components of our ME and HS software are taken from the low-memory and speed-optimized implementation contained in the NIST package of ThreeBears (with minor optimizations). Our software consists of a mix of C and AVR assembly language, i.e., the performance-critical MAC operation and Keccak permutation are written in AVR assembly, and all other functions in C. Atmel Studio v7.0, our development environment, comes with the 8-bit AVR GNU toolchain including avr-gcc version 5.4.0. We used the cycle-accurate instruction set simulator of Atmel Studio to precisely determine the execution times. The source codes were compiled with optimization option `-O2` using the ATmega1284 microcontroller as target device.

Exectuin time. Table 4.1 shows the execution time of a MAC operation, key generation, encapsulation, and decapsulation of our software. A speed-optimized MAC takes only 605 k clock cycles, while the memory-optimized version requires 70% more cycles. The speed difference between these two types of MAC directly impacts the overall running time of ME-BBear(-Eph) versus HS-BBear(-Eph), because

Table 4.2: RAM usage and code size (both in bytes) of our AVR implementations.

Implementation	MAC		KeyGen		Encaps		Decaps		Total	
	RAM	Size	RAM	Size	RAM	Size	RAM	Size	RAM	Size
ME-BBear	82	2,760	1,715	6,432	1,735	7,554	2,368	10,110	2,368	12,264
ME-BBear-Eph	82	2,760	1,715	6,432	1,735	7,640	1,731	8,270	1,735	10,998
HS-BBear	934	3,332	2,733	7,000	2,752	8,140	4,559	10,684	4,559	11,568
HS-BBear-Eph	934	3,332	2,733	7,000	2,752	8,226	2,356	8,846	2,752	10,296

Table 4.3: Comparison of our implementation with other key-establishment schemes (all of which target 128-bit security) on the 8-bit AVR platform (the execution times of Encaps and Decaps are in clock cycles; RAM and code size are in bytes).

Implementation	Algorithm	Encaps	Decaps	RAM	Size
This work (ME-CCA)	ThreeBears	12,289,744	18,578,335	2,368	12,264
This work (ME-CPA)	ThreeBears	12,435,165	3,444,154	1,735	10,998
This work (HS-CCA)	ThreeBears	7,901,873	12,476,447	4,559	11,568
This work (HS-CPA)	ThreeBears	8,047,835	2,586,202	2,752	10,296
Cheng et al. [CDG+19]	NTRU Prime	8,160,665	15,602,748	-	11,478
Düll et al. [DHH+15] (ME)	X25519	14,146,844	14,146,844	510	9,912
Düll et al. [DHH+15] (HS)	X25519	13,900,397	13,900,397	494	17,710

there are several MACs in KeyGen, Encaps and Decaps. Taking HS-BBear as example, KeyGen, Encaps, and Decaps needs 6.12 M, 7.90 M, and 12.48 M clock cycles, respectively. Their ME counterparts are roughly 1.5 times slower.

RAM usage and code size. Table 4.2 specifies both the memory footprint and code size of the four basic operations (MAC, KeyGen, Encaps, and Decaps). The speed-optimized MAC consumes 934 bytes of memory, while the memory-optimized MAC requires as little as 82 bytes, which is just 9% of the former. Due to the memory-optimized MAC operation and full-radix representation of field elements, ME-BBear has a RAM footprint of 1.7 kB for each KeyGen and Encaps, while Decaps is more memory-demanding and needs 2.4 kB RAM. However, ME-BBear-Eph requires only 1.7 kB of RAM altogether. On the other hand, the HS implementations need over 1.5 times more RAM than their ME counterparts. In terms of code size, each of the four implementations requires roughly 11 kB.

Comparison with other AVR implementations. Table 4.3 compares implementations of both pre and post-quantum schemes (targeting 128-bit security) on 8-bit AVR microcontrollers. Compared to the CCA-secure version of the second-round NIST candidate NTRU Prime [CDG+19], HS-BBear is slightly faster for both Encapsulation and Decapsulation. On the other hand, when compared with the optimized implementation of X25519 in [DHH+15], the Encapsulation operation of each BabyBear variant in Table 4.3 is faster than a variable-base scalar multiplication, while the Decaps of ME-BBear is slower, but that of the HS variant still a bit faster. Notably, the Decaps operation of our CPA-secure implementations is respectively 4.0 times (ME) and 5.4 times (HS) faster than X25519.

Comparison with other MCU implementation: RAM usage. One of the most significant advantages of the ThreeBears cryptosystem is its relatively low RAM consumption, which is important for deployment on constrained devices. Table 4.4 compares the RAM footprint of implementations of ThreeBears and a few other NIST candidates on microcontrollers. Due to the very small number of state-of-the-art implementations of NIST candidates for the 8-bit AVR platform, we include in

Table 4.4: Comparison of RAM consumption (in bytes) of NIST PQC implementations (all of which target NIST security category 1 or 2) on 8-bit AVR and on 32-bit ARM Cortex-M4 microcontrollers.

Implementation	Algorithm	Platform	KeyGen	Encaps	Decaps
CCA-secure schemes					
This work (ME)	ThreeBears	AVR	1,715	1,735	2,368
Hamburg [Ham19]	ThreeBears	Cortex-M4	2,288	2,352	3,024
pqm4	ThreeBears	Cortex-M4	3,076	2,964	5,092
pqm4	NewHope	Cortex-M4	3,876	5,044	5,044
pqm4	Round5	Cortex-M4	4,148	4,596	5,220
pqm4	Kyber	Cortex-M4	2,388	2,476	2,492
pqm4	NTRU	Cortex-M4	11,848	6,864	5,144
pqm4	Saber	Cortex-M4	9,652	11,388	12,132
CPA-secure schemes					
This work (ME)	ThreeBears	AVR	1,715	1,735	1,731
Hamburg [Ham19]	ThreeBears	Cortex-M4	2,288	2,352	2,080
pqm4	ThreeBears	Cortex-M4	3,076	2,980	2,420
pqm4	NewHope	Cortex-M4	3,836	4,940	3,200
pqm4	Round5	Cortex-M4	4,052	4,500	2,308

Table 4.4 also some recent results from the pqm4 library, which targets 32-bit ARM Cortex-M4. In addition, we list the results of the original low-RAM implementations of BabyBear (for both the CCA and CPA variant) from the NIST package. Our memory-optimized BabyBear is the most RAM-efficient implementation among all CCA-secure PQC schemes and needs 5% less RAM than the second most RAM-efficient scheme Kyber. Furthermore, ME-BBear-Eph requires the least amount of RAM of all (CPA-secure) second-round NIST PQC candidates, and improves the original low-memory implementation of the designer by roughly 26.2%.

4.5 Conclusion

We presented the first highly-optimized Assembler implementation of ThreeBears for the 8-bit AVR architecture. Our simulation results show that, even with a fixed parameter set like BabyBear, many trade-offs between execution time and RAM consumption are possible. The memory-optimized CPA-secure version of BabyBear requires only slightly more than 1.7 kB RAM, which sets a new record for memory efficiency among all known software implementations of second-round candidates. Due to this low memory footprint, BabyBear fits easily into the SRAM of 8-bit AVR ATmega microcontrollers and will even run on severely constrained devices like an ATmega128L with 4 kB SRAM. While a RAM footprint of 1.7 kB is still clearly above the 500 B of X25519, the execution times are in favor of BabyBear since a CPA-secure decapsulation is four times faster than a scalar multiplication. ThreeBears is also very well suited to be part of a hybrid pre/post-quantum key agreement protocol since the multiple-precision integer arithmetic can (potentially) be shared with the low-level field arithmetic of X25519, thereby reducing the overall code size when implemented in software or the total silicon area in the case of hardware implementation. For all these reasons, ThreeBears is an excellent candidate for a post-quantum cryptosystem to secure the IoT.

Part III

Vectorized Implementation of Isogeny-Based Cryptography

CHAPTER

5

VECTORIZED CSIDH

This Chapter is based on our paper [CFG+21]. While we were conducting the research work described in this Chapter, CTIDH [BBC+21a], a constant-time and faster CSIDH variant (using a new key space), had not yet been published.

5.1 Introduction

Post-quantum cryptography. Quantum computing exploits quantum-mechanical effects and phenomena, such as state superposition and entanglement, to efficiently solve certain computational problems, in particular optimization and search problems [KLM07]. However, quantum computing has also a destructive side since it is assumed that a quantum computer with a few thousand logical qubits would be capable to break essentially any public-key cryptosystem in use today [RNSL17]. The dawning era of quantum computing has spurred much research on Post-Quantum Cryptography (PQC), a sub-domain of cryptography concerned with the design, analysis and implementation of cryptosystems that are expected to resist attacks executed on both conventional and quantum computers [SL21]. Almost all of the to-date existing post-quantum key establishment and signature algorithms fall into one of five categories, which are lattice-based cryptography, multivariate cryptography, hash-based cryptography, code-based cryptography, and supersingular isogeny cryptography. These categories differ with respect to the hard mathematical problems their security is based on, but also in terms of computational cost, key lengths, and the length of ciphertexts (resp. signatures) [SL21]. The security of isogeny-based cryptosystems rests upon the intractability of the problem of finding an explicit isogeny between two (supersingular) elliptic curves over a finite field that are known to be isogenous [DeF17]. While isogeny-based schemes are computation-intensive, their key sizes are among the smallest of the five categories and come even close to that of pre-quantum elliptic curve schemes.

Isogeny-based cryptography. Various isogeny-based cryptosystems have appeared in the literature in the past ten years. SIKE (short for Supersingular Isogeny Key Encapsulation) is a key encapsulation mechanism whose security relies upon the supersingular isogeny walk problem between two elliptic curves in the same isogeny class, which asks to find a path made of isogenies of small degree [Cos19]. A variant of SIKE is an alternative candidate in the third round of the PQC standardization project of the NIST [JAC+22]. CSIDH (an abbreviation of Commutative Supersingular Isogeny Diffie-Hellman)

is an "ECDH-like" key-exchange scheme based on a commutative group action of an ideal class group [CLM+18]. Given an initial elliptic curve E, a secret key in CSIDH is an ideal class $\mathfrak{a}$ in a class group (represented by its list of exponents), and the corresponding public key can be obtained by computing the group action $E' = \mathfrak{a} \star E$. The security of CSIDH is based on the hard problem of finding an isogeny path from the isogenous curves E and E'. CSIDH has received a lot of attention in recent years since it comes with highly attractive features like efficient validation of public keys, making it suitable for non-interactive (i.e., static) key exchange protocols. In fact, CSIDH can serve as "drop-in" replacement for classical ECDH key exchange and does even comply with the requirements of "0-RTT" protocols such as QUIC. Furthermore, class group actions provide a rich foundation for the design of various other cryptosystems, e.g., signature schemes [BKV19, DG19]. However, the downside of CSIDH is that the computation of group actions is very costly, which makes CSIDH extremely slow, not only in relation to X25519 [Ber06] and other pre-quantum ECDH variants, but also when compared to SIKE. For example, while an Intel Skylake processor can execute a variable-base scalar multiplication on Curve25519 in less than 100 k cycles [NS21] and a SIKEp434 encapsulation or decapsulation in about 10 M cycles [JAC+22], the to-date best constant-time implementation of a CSIDH-512 group action evaluation and key validation requires close to 240 M clock cycles [HLKA20].

Motivation. The lengthy computation time of CSIDH poses a major obstacle for its application in security protocols like TLS or HTTPS when taking into account that, for example, the web servers of large enterprises like Google or Facebook are confronted with thousands of HTTPS requests per second. In order to be able to cope with such extreme volumes of traffic, the server infrastructure of such enterprises often includes a so-called TLS termination proxy or TLS reverse proxy, which transparently translates HTTPS sessions to TCP sessions for back-end servers (e.g., web or database servers), see [JHH+11]. This offloading of the TLS termination to a dedicated proxy frees the web server from having to execute computation-intensive TLS handshakes that involve public-key operations to authenticate the server to the client and establish a shared secret key using e.g., X25519 key exchange [Ber06]. A TLS termination proxy equipped with a high-end 64-bit Intel processor clocked at 4 GHz is (in theory) able to perform 40,000 X25519 key exchanges per second per core since, as mentioned before, a variable-base scalar multiplication on Curve25519 costs below 100 k cycles[1]. Replacing X25519 by SIKEp434 would decrease the (theoretical) upper bound of the number of key exchanges per second on one core to around 400. Even worse, when X25519 gets replaced by CSIDH-512, the number of key exchanges per core would go down to a mere 17 per second, which is more than three orders of magnitude below the (theoretical) throughput of X25519. Therefore, it is little surprising that techniques to speed up CSIDH are eagerly sought.

Contributions. The straightforward way of maximizing the throughput of CSIDH is to minimize the latency of the underlying group action. However, we demonstrate in this work that the usual approach of maximizing throughput by minimizing latency leads to sub-optimal results on Intel processors that are equipped with recent vector (i.e., SIMD) extensions such as AVX-512. To be more concrete, we show that, when using AVX-512 instructions, minimizing the latency of one group action requires different optimization strategies than maximizing the throughput of several group actions that are executed in SIMD fashion. We explain how the "limb-slicing" method presented in [CGT+20] can be applied to compute eight independent CSIDH group actions in parallel using AVX-512 instructions, whereby each group action uses a 64-bit element of a 512-bit vector. Limb-slicing is somewhat related to the well-known "bit-slicing" technique used in symmetric cryptography since it increases throughput at the expense of latency. We discuss in detail the obstacles we had to overcome to efficiently batch group ac-

[1] These 40,000 key exchanges per second are a *theoretical upper bound* for the throughput of a single processor core, which can only be reached under the assumption that the core executes nothing else than scalar multiplications (i.e., all other operations, such as the transfer of public keys, are ignored).

tions and execute them in a SIMD-parallel way. Further, we describe software optimization techniques that enable a highly-efficient (8×1)-way parallel execution of the prime-field arithmetic operations using AVX-512F and AVX-512IFMA instructions. We also present a latency-optimized implementation of the group action for AVX-512IFMA, which can be used to speed up client-side TLS processing (while our throughput-optimized implementation targets the server side[2] and can be used for TLS termination as described in [JHH+11]). Our results for CSIDH-512 show that batch processing and limb-slicing achieve a throughput gain by a factor of 3.64 compared to an optimized (but non-vectorized) x64 implementation. In light of the recent debate about the post-quantum security of CSIDH-512, we emphasize that our optimizations can also be applied to parameter sets with larger primes, and we expect similar improvements in performance over non-vectorized implementations.

Source code. The source code of the presented software library is publicly available at https://gitlab.uni.lu/apsia/avx-csidh.

5.2 Background

In this section, we give a brief overview of the CSIDH protocol and the CSIDH class group action of Castryck, Lange, Martindale, Panny, and Renes [CLM+18]. Further, we summarize the existing constant-time implementations of the CSIDH class group action. For a detailed analysis of the theory of elliptic curves that is relevant for isogeny-based cryptography, we refer the reader to the lecture notes of De Feo [DeF17].

5.2.1 CSIDH

Underlying arithmetic. The CSIDH protocol works over a finite field $\mathbb{F}_p$, where p is a large prime of the special form $p = 4 \cdot \ell_1 \cdots \ell_n - 1$ and $\ell_1 < \ldots < \ell_n$ are small odd primes. In addition, it uses supersingular elliptic curves[3] E_A, defined over $\mathbb{F}_p$ and represented in Montgomery form $E_A : y^2 = x^3 + Ax^2 + x$, with $A^2 \neq 4$, where the $\mathbb{F}_p$-endomorphism ring[4] of such curves is isomorphic to an order in the imaginary quadratic field $\mathbb{Q}(\sqrt{-p})$. Specifically, the authors in [CLM+18] choose a supersingular Montgomery curve E_0 (i.e., $A = 0$) with $p \equiv 3 \pmod 4$, where in this case $\mathrm{End}_{\mathbb{F}_p}(E_0) \cong \mathbb{Z}[\sqrt{-p}]$. Further, we define $\mathcal{E}\ell\ell_{\mathbb{F}_p}(\mathbb{Z}[\sqrt{-p}])$ as the set of all supersingular elliptic curves with the same $\mathbb{F}_p$-endomorphism ring $\mathbb{Z}[\sqrt{-p}]$.

Class group action. The ideal class group $\mathrm{Cl}(\mathbb{Z}[\sqrt{-p}])$ acts freely and transitively on $\mathcal{E}\ell\ell_{\mathbb{F}_p}(\mathbb{Z}[\sqrt{-p}])$, via isogenies[5] ([CLM+18, Theorem 7]). Every principal ideal $(\ell_i) \subset \mathbb{Z}[\sqrt{-p}]$ splits as a product of prime ideals $(\ell_i) = \mathfrak{l}_i\bar{\mathfrak{l}}_i = \langle \ell_i, \pi - 1\rangle\langle \ell_i, \pi + 1\rangle$, where $\pi = \sqrt{-p}$ is the Frobenius endomorphism[6] and since (ℓ_i) is principal, we get $\bar{\mathfrak{l}}_i = \mathfrak{l}_i^{-1} \in \mathrm{Cl}(\mathbb{Z}[\sqrt{-p}])$. In CSIDH we are interested in computing the action of an ideal $\mathfrak{a} = \mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n} \in \mathrm{Cl}(\mathbb{Z}[\sqrt{-p}])$, where $e_1, \ldots, e_n$ are small exponents, chosen uniformly from some interval $[-b, b]$. This is done by computing in sequence the action of the ideal $\mathfrak{l}_i$, if $e_i \geq 0$, or $\bar{\mathfrak{l}}_i$, if $e_i < 0$,

[2] A TLS server under heavy load may have to serve thousands of connections per second, which means it may have to compute eight or more key exchanges every few milliseconds. On the other hand, if the load is low, it makes more sense to use a latency-optimized implementation. But when the load increases and the server gets confronted with (at least) eight connections in a short period of time, switching from the latency-optimized to the throughput-optimized implementation will lead to better performance. To date, OpenSSL and other TLS stacks do not support the batching of public-key cryptosystems, but an integration of batch processing is possible as demonstrated by SSLShader (see [JHH+11] for details).

[3] An elliptic curve E defined over $\mathbb{F}_p$ (where $p > 3$) is called *supersingular*, iff $\#E(\mathbb{F}_p) = p + 1$, otherwise it is *ordinary*.

[4] For an elliptic curve E, the $\mathbb{F}_p$-endomorphism ring $\mathrm{End}_{\mathbb{F}_p}(E)$ contains all endomorphisms from E to itself, that are defined over $\mathbb{F}_p$.

[5] An isogeny $\phi : E \to E'$ defined over $\mathbb{F}_p$ is a non-constant rational map defined over $\mathbb{F}_p$, which is also a group homomorphism from $E(\mathbb{F}_p)$ to $E'(\mathbb{F}_p)$.

[6] The Frobenius endomorphism π maps a point $P = (x, y)$ on an elliptic curve E defined over $\mathbb{F}_p$ to (x^p, y^p).

Algorithm 5.1: Original-style CSIDH class group action computation [CLM+18].

Input: $A \in \mathbb{F}_p$ and a list of integers $(e_1, \ldots, e_n)$.
Output: $B \in \mathbb{F}_p$, such that $(\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}) \star E_A = E_B$, where $E_B : y^2 = x^3 + Bx^2 + x$.

while some $e_i \neq 0$ **do**
  Sample a random $x \in \mathbb{F}_p$
  $s \leftarrow +1$ if $x^3 + Ax^2 + x$ is a square in $\mathbb{F}_p$, else $s \leftarrow -1$
  Let $S = \{i \mid e_i \neq 0, \text{sign}(e_i) = \text{sign}(s)\}$. **If** $S = \emptyset$ **then** start over with a new x.
  Let $P = (x : 1)$, $q \leftarrow \prod_{i \in S} \ell_i$ and compute $T \leftarrow [(p+1)/q]P$
  for each $i \in S$ **do**
    $R \leftarrow [q/\ell_i]T$ /* R is the kernel generator */
    if $R \neq \infty$ **then**
      Compute $\phi : E_A \rightarrow E_B = \mathfrak{l}_i \star E_A$ with $\ker(\phi) = \langle R \rangle$
      $A \leftarrow B$, $T \leftarrow \phi(T)$, $q \leftarrow q/\ell_i$, $e_i \leftarrow e_i - s$
return B

exactly $|e_i|$ times for every $i \in \{1, \ldots, n\}$. For the action of the ideal $\mathfrak{l}_i$ we choose a point $R \in E(\mathbb{F}_p)$ of order ℓ_i that lies in the kernel of $\pi - 1$ and compute the isogeny $\phi_{\mathfrak{l}_i} : E \rightarrow E/\langle R \rangle = \mathfrak{l}_i \star E$, with $\ker(\phi_{\mathfrak{l}_i}) = \langle R \rangle$ and $\deg(\phi_{\mathfrak{l}_i}) = \ell_i$. For the action of the ideal $\bar{\mathfrak{l}}_i$ we choose a random point $R \in E(\mathbb{F}_{p^2})$ (i.e., the quadratic twist of E), of order ℓ_i in the kernel of $\pi + 1$. Note that in this case, $R = (x, iy)$, where $x, y \in \mathbb{F}_p$ and $i = \sqrt{-1}$. Then we compute the isogeny $\phi_{\bar{\mathfrak{l}}_i} : E \rightarrow E/\langle R \rangle = \bar{\mathfrak{l}}_i \star E$, with $\ker(\phi_{\bar{\mathfrak{l}}_i}) = \langle R \rangle$ and $\deg(\phi_{\bar{\mathfrak{l}}_i}) = \ell_i$. Both isogenies are computed using the Vélu formulæ [Vél71], which require $O(\ell_i \log p^2)$ bit operations, hence they are efficiently computed for relatively small primes ℓ_i. Iterating each isogeny computation $|e_i|$ times, depending on the sign of e_i and composing the resulting isogenies in each step, yields the final codomain curve $\mathfrak{a} \star E = (\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}) \star E$ (see Algorithm 5.1 [CLM+18]).

Key exchange. The public parameters are the prime $p = 4 \cdot \ell_1 \cdots \ell_n - 1$, the starting curve $E_0 : y^2 = x^3 + x$ and a positive integer b, such that $(2b+1) \geq \sqrt[n]{\#\text{Cl}(\mathbb{Z}[\sqrt{-p}])}$ in order to maintain security. Alice's secret key is a list of exponents $sk_A = (e_1, \ldots, e_n) \in [-b, b]^n$, while her public key is derived from the action of the ideal $\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}$ on the curve E_0, using Algorithm 5.1, i.e., $pk_A = E_A = (\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}) \star E_0$, which is sent to Bob. In the same vein, Bob's secret key is $sk_B = (d_1, \ldots, d_n) \in [-b, b]^n$, and his public key $pk_B = E_B = (\mathfrak{l}_1^{d_1} \cdots \mathfrak{l}_n^{d_n}) \star E_0$ is sent to Alice. For the shared secret, Alice and Bob compute the codomain curves $k_A = (\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}) \star E_B$ and $k_B = (\mathfrak{l}_1^{d_1} \cdots \mathfrak{l}_n^{d_n}) \star E_A$ respectively, using Algorithm 5.1. The two curves are $\mathbb{F}_p$-isomorphic, because they are derived from the action of the ideal $\mathfrak{l}_1^{e_1+d_1} \cdots \mathfrak{l}_n^{e_n+d_n}$ on the initial curve E_0, as a result of the commutativity property of the ideal class group $\text{Cl}(\mathbb{Z}[\sqrt{-p}])$. Note that the public keys and the shared secret, are elliptic curves in Montgomery form, hence they are represented as a single coefficient in $\mathbb{F}_p$. CSIDH features an efficient public-key validation process, which corresponds to testing whether the public key is a supersingular Montgomery curve, and it is accomplished with a series of scalar multiplications [CLM+18]. Castryck, Lange, Martindale, Panny, and Renes presented a concrete instantiation for CSIDH. They choose a 511-bit prime $p = 4 \cdot \ell_1 \cdots \ell_{74} - 1$, where $\ell_1, \ldots, \ell_{73}$ are the first odd primes starting from $\ell_1 = 3$, and $\ell_{74} = 587$. The secret exponents $(e_1, \ldots, e_{74})$ are sampled from $[-5, 5]^{74}$ (hence $b = 5$), in which case $74 \log_2(2 \cdot 5 + 1) \approx 256$. This instantiation is referred to as CSIDH-512.

Security. The security of CSIDH is based on the Group Action Inverse Problem (GAIP). That is, given two supersingular elliptic curves E and E', defined over $\mathbb{F}_p$, with the same $\mathbb{F}_p$-endomorphism ring $\mathbb{Z}[\sqrt{-p}]$, to find an ideal $\mathfrak{a} = \mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}$ such that $\mathfrak{a} \star E = E'$. The best known classical attack for solving GAIP is the meet-in-the-middle attack with fully exponential complexity $2^{O(\sqrt{N})}$, where

$N = \#\mathrm{Cl}(BZ[\sqrt{-p}]) \approx \sqrt{p}$. In the quantum setting, Childs, Jao, and Soukharev [CJS14] have shown that solving the GAIP problem can be reduced to the abelian hidden-shift problem, for which the subexponential quantum algorithms of Regev [Reg04] and Kuperberg [Kup05] can be applied, where the latter has complexity $2^{O(\sqrt{\log N})}$ and the quantum space complexity $O(\log N)$. Based on the above, Castryck, Lange, Martindale, Panny, and Renes [CLM+18] conjectured that CSIDH-512 would achieve NIST's post-quantum security level 1 based on the asymptotic complexity of Kuperberg's algorithm [Kup05, Kup13]. However, the concrete security of CSIDH-512 is under debate since the works of Peikert [Pei20], Bonnetain and Schrottenloher [BS20], and more recently Chávez-Saab, Chi-Domínguez, Jaques, and Rodríguez-Henríquez [CCJR20], estimate that the prime p should be significantly larger in order to meet NIST's security level 1. In particular, [CCJR20] suggests that p should be updated to 4096 bits.

5.2.2 Optimization techniques for constant-time CSIDH

In practice, we require a constant-time implementation of Algorithm 5.1 to resist against side-channel attacks. Given a secret exponent list $(e_1, \ldots, e_n)$, Algorithm 5.1 computes $|e_1| + \ldots + |e_n|$ isogenies, and thus its execution time fully depends on the secret key. Meyer, Campos, and Reith [MCR19] presented three leakage scenarios that appear when implementing Algorithm 5.1. Timing leakage occurs since different secret keys lead to different execution times of evaluating the class group action. Power analysis leaks information on the sign distribution of the secret key since unbalanced, in terms of the sign, secret exponents lead to scalar multiplications with larger scalars. Cache timing attacks are also possible and leak information based on branch conditions or array indices. The authors in [CLM+18] argued that a constant-time implementation can be obtained when adding certain "dummy" operations, which will not be considered nor affect the final output of the group action. The first constant-time implementations of Algorithm 5.1 are due to Bernstein, Lange, Martindale, Panny [BLMP19] and Jalali, Azarderakhsh, Kermani, Jao [JAKJ19], which add a large amount of dummy operations and have a probability of failure in the class group action computation.

Meyer, Campos, and Reith. A constant-time implementation of the CSIDH class group action with significant optimizations is presented in [MCR19], and it is known as the *MCR-style*. The algorithm uses only positive secret exponents e_i, each sampled from its own space $[0, b_i]$ where all b_i are chosen such that security is maintained. This mitigates power attacks, while the different intervals allow to reduce the number of large degree isogenies. Meyer et al. use dummy isogenies so that the same number of isogenies is computed in each class group action. For each i their algorithm computes e_i "real" and $b_i - e_i$ "dummy" ℓ_i-isogenies. Further, they use the Elligator 2 map [BHKL13] for sampling points on the curve. As observed in [MR18], they compute the class group action in descending order in terms of the primes ℓ_i, which results in a speedup over the ascending order. The most significant optimization is the SIMBA-m-μ (Splitting Isogeny computations into Multiple BAtches) technique, where the idea is to partition the indices $\{1, \ldots, n\}$ into m disjoint subsets and evaluate the group action on each subset individually, which results in smaller scalars in step 7 of Algorithm 5.1. However, this should be done for a specific number of rounds μ, and the subsets should be merged back after this threshold. The authors argue that finding optimal values for m and μ, as well as for the upper bounds b_i is a hard task. They present various choices, based on the CSIDH-512 instance, where the best example is SIMBA-5-11.

Onuki, Aikawa, Yamazaki, and Takagi. In [OAYT19] the authors present a constant-time implementation of Algorithm 5.1, known as the *OAYT-style*, that improves on the MCR-style by 29.03%. In their algorithm each e_i is also sampled from its own space, but in contrast to the MCR-style, each e_i is allowed to be negative as well. The algorithm mitigates timing attacks by keeping track of two points $P_0 \in E[\pi - 1]$ and $P_1 \in E[\pi + 1]$ and picking the appropriate point, depending on the sign of e_i, in

Table 5.1: The latency (in clock cycles) and throughput (CPI) of relevant AVX-512 instructions on Intel Ice Lake Core processor.

Type	Mnemonic	Instruction	Latency	CPI
Arithmetic	ADD/SUB	vpaddq/vpsubq	1	0.5
	MUL	vpmuludq	–	1
Logic	SHL/SHR	vpsllq/vpsrlq	1	1
	AND	vpandd	1	0.5
Permutation	PERM	vpermq	3	1
	BCAST	vpbroadcastq	3	1
	ZERO	vpxorq †	1	0.5
IFMA	MACLO	vpmadd52luq	4	1
	MACHI	vpmadd52huq	4	1

† XOR a ZMM register with itself to set it to zero.

order to create the kernel generator. Both points P_0 and P_1 are mapped through the isogeny, and the point not used to derive the kernel is multiplied by ℓ_i. The algorithm also uses the Elligator 2 map for generating points on the curve and dummy isogenies as in the MCR-style.

Cervantes-Vázquez, Chenu, Chi-Domínguez, De Feo, Rodríguez-Henríquez, and Smith. The work in [CCC+19] provides significant improvements on both the MCR- and OAYT-style algorithms. The authors present efficient formulas in the twisted Edwards model for performing isogeny computations, isogeny evaluations and curve operations (point addition/doubling). They also use differential addition chains which provide a 25% improvement compared to the classical Montgomery ladder, for computing scalar multiplications. Besides the optimizations for the MCR- and OAYT-style algorithms, the authors present a constant-time implementation that excludes the dummy operations, known as *dummy-free-style*. Although this is less efficient compared to the MCR- and OAYT-style, it is resistant against fault-injection attacks, i.e., stronger attackers who can interfere in computations and determine whether a modification happened on a "real" or a "dummy" isogeny. Their optimized OAYT-style with SIMBA-3-8 is 39% faster than the MCR-style presented in [MCR19], and their dummy-free-style group action is two times slower compared to their OAYT-style implementation.

Optimal strategies. In [HLKA20], Hutchinson, LeGrow, Koziel, and Azarderakhsh further studied problems of choosing the optimal bounds b_i for sampling secret exponents, the optimal ordering of primes ℓ_i, and the optimal partition m for SIMBA technique. Such selections are referred to as *optimal strategies*. Their optimal strategies offer 5.06% improvement for the OAYT-style implementation in [CCC+19]. In [CR22], Chi-Domínguez and Rodríguez-Henríquez generalized the computational strategies that are used in the SIKE implementation [JAC+22] for the case of CSIDH. Their new algorithms do not rely on the SIMBA approach and provide an improvement of 12.09%, 3.36%, and 10.58% compared to the MCR-, OAYT-, and dummy-free-style implementations of [CCC+19], respectively. The OAYT-style algorithm of [HLKA20], the MCR- and dummy-free-style algorithms of [CR22] are to date the most efficient constant-time implementations of CSIDH in the literature. These algorithms are further optimized by Adj, Chi-Domínguez, and Rodríguez-Henríquez [ACR22] when applying certain tricks that reduce the computational cost of new Vélu formulæ of [BDLS20].

5.2.3 Target platform

In this work, we target the Intel Ice Lake processor which supports IFMA extension. The specific processor we used in our experiments is Intel Core i3-1005G1 CPU clocked at 1.2 GHz.

Relevant instructions. Table 5.1 lists the most relevant AVX-512 instructions used in this work, along with their latency and throughput data[7] on the Ice Lake processor which we obtained from Intel official document [Int20]; the throughput data is shown in Clock Per Instructions (CPI) ratio [Int18b]; the instruction mnemonic is used to facilitate the description of algorithms.

5.3 Implementation: high-throughput batched software

Starting point. Recall from Section 5.2.2 that the to-date fastest constant-time CSIDH implementations are the two OAYT-style variants of [HLKA20] and [CR22]. According to our measurements of their group action evaluation (see Table 5.3), the former is 1% faster than the latter. The optimization techniques used in both variants improve the OAYT-style implementation of [CCC+19] by 5%, and they are all considered in terms of x64 implementation. When the same optimization techniques are applied to AVX-512 software, the resulting effects may be different. In this work we focus on the OAYT-style implementation of [CCC+19]. However, we argue that the optimization techniques of [HLKA20, CR22] as well as [ACR22] can also be applied in our implementation.

The instantiation of OAYT-style group aciton. The OAYT-style class group action algorithm of [CCC+19] is described in Algorithm 5.2, which was originally presented in [OAYT19]. Algorithm 5.2 takes advantage of the SIMBA-m-μ technique [MCR19], in which the set of indices $S = \{1, \ldots, n\}$ is partitioned into m subsets $S_1, \ldots, S_m$, where $S_j = \{j, m+j, 2m+j, \ldots\}$ for each $j \in \{1, \ldots m\}$. The OAYT-style class group action algorithm of [CCC+19] is described in Algorithm 5.2, which was originally presented in [OAYT19]. Algorithm 5.2 takes advantage of the SIMBA-m-μ technique [MCR19], in which the set of indices $S = \{1, \ldots, n\}$ is partitioned into m subsets $S_1, \ldots, S_m$, where $S_j = \{j, m+j, 2m+j, \ldots\}$ for each $j \in \{1, \ldots m\}$. For the CSIDH-512 instantiation, the best choices according to [OAYT19] are $m = 3$ and $\mu = 8$. In this case, Algorithm 5.2 computes 404 "real" and "dummy" isogenies, i.e., the variable $t_{\max} = \sum_{i=1}^{n} b_i = 404$. The ordering of the small primes $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_{74})$ is:

$$\begin{aligned}\boldsymbol{\ell} = (&349, 347, 337, 331, 317, 313, 311, 307, 293, 283, 281, 277, 271,\\ &269, 263, 257, 251, 241, 239, 233, 229, 227, 223, 211, 199, 197,\\ &193, 191, 181, 179, 173, 167, 163, 157, 151, 149, 139, 137, 131,\\ &127, 113, 109, 107, 103, 101, 97, 89, 83, 79, 73, 71, 67,\\ &61, 59, 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13,\\ &11, 7, 5, 3, 587, 373, 367, 359, 353)\end{aligned}$$

The secret key space or equivalently, the list of bounds for the secret exponents is:

$$\begin{aligned}\boldsymbol{b} = (&2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4,\\ &4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6,\\ &7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 8, 9, 9, 9, 10, 10, 10,\\ &10, 9, 8, 8, 8, 7, 7, 7, 7, 7, 6, 5, 1, 2, 2, 2, 2)\end{aligned}$$

Following [CCC+19], we define the constant-time equality test and constant-time conditional swap functions `isequal` and `cswap`, respectively, as:

$$\texttt{isequal}(x, y) = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{if } x \neq y \end{cases}, \qquad \texttt{cswap}(x, y, a) = \begin{cases} x \leftrightarrow y, & \text{if } a = 1 \\ x \nleftrightarrow y, & \text{if } a = 0 \end{cases}$$

[7] Here we consider the case that these instructions work on 512-bit ZMM registers not 128-bit XMM or 256-bit YMM registers. The instruction CPI of the latter case is lower since Port 1 can handle AVX-512 instructions working on XMM or YMM but on ZMM registers (see [Int18b, Figure 2-1]).

Algorithm 5.2: OAYT-style CSIDH class group action computation (with SIMBA-m-μ).

Input: $A \in \mathbb{F}_p$, bound $\boldsymbol{b} = (b_1, \ldots, b_n)$, exponents $\boldsymbol{e} = (e_1, \ldots, e_n)$ where $e_i \in [-b_i, b_i]$.
Output: $B \in \mathbb{F}_p$, such that $(\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}) \star E_A = E_B$, where $E_B : y^2 = x^3 + Bx^2 + x$.

1 $(e'_1, \ldots, e'_n) \leftarrow (e_1, \ldots, e_n)$, $(b'_1, \ldots, b'_n) \leftarrow (b_1, \ldots, b_n)$
2 $E_B \leftarrow E_A$, $t_{\max} \leftarrow \sum_{i=1}^{n} b_i$, $t_{\text{isog}} \leftarrow 0$, $r \leftarrow 0$, $j \leftarrow 0$
3 **while** $t_{\text{isog}} < t_{\max}$ **do**
4 $j \leftarrow (j \bmod m) + 1$ /* Subset index */
5 **if** $r = \mu \cdot m$ **then**
6 $S_1 \leftarrow \{i \mid b_i \neq 0\}$, $j \leftarrow 1$, $m \leftarrow 1$ /* Merge subsets */
7 $u \leftarrow \texttt{random}(\{2, \ldots, (p-1)/2\})$, $q \leftarrow \prod_{i \in S_j} \ell_i$
8 $(P_0, P_1) \leftarrow \texttt{Elligator}(E_B, u)$ /* $P_0 \in E[\pi - 1]$ and $P_1 \in E[\pi + 1]$ */
9 $(T_0, T_1) \leftarrow ([(p+1)/q]P_0, [(p+1)/q]P_1)$
10 **for each** $i \in S_j$ **do**
11 $s \leftarrow \texttt{sign}(e'_i)$
12 $\texttt{cswap}(T_0, T_1, 1 - s)$
13 $R \leftarrow [q/\ell_i]T_0$ /* R is the kernel generator */
14 **if** $R \neq \infty$ **then**
15 $w \leftarrow 1 - \texttt{isequal}(e'_i, 0)$
16 Compute isogeny $\phi : E_B \to E_C = \mathfrak{l}_i \star E_B$ s.t. $\ker(\phi) = \langle R \rangle$
17 $T_1 \leftarrow [\ell_i]T_1$
18 $(T_2, T_3) \leftarrow (\phi(T_0), \phi(T_1))$
19 $T_0 \leftarrow [\ell_i]T_0$
20 $\texttt{cswap}(T_0, T_2, w)$, $\texttt{cswap}(T_1, T_3, w)$, $\texttt{cswap}(B, C, w)$
21 $e'_i \leftarrow e'_i + (-1)^s \cdot w$, $b'_i \leftarrow b'_i - 1$, $t_{\text{isog}} \leftarrow t_{\text{isog}} + 1$
22 **else**
23 $T_1 \leftarrow [\ell_i]T_1$
24 $\texttt{cswap}(T_0, T_1, 1 - s)$, $q \leftarrow q/\ell_i$
25 **if** $b'_i = 0$ **then**
26 Remove i from S_j
27 $r \leftarrow r + 1$
28 **return** B

Further, we consider a constant-time function $\texttt{sign}(x)$, which returns 0 if $x < 0$ while returns 1 if $x \geq 0$. More details on Algorithm 5.2 are given in [OAYT19, CCC+19].

5.3.1 Obstacles to batching CSIDH group actions

Requirements. We conceive our batched software where eight CSIDH group action instances in the fashion of Algorithm 5.2 are to be computed simultaneously by AVX-512 instructions. Besides, each instance is computed in a 64-bit lane, and instances are independent of each other. Since AVX-512 is in the paradigm of *Single Instruction Multiple Data*, from another perspective, this requires that the same instruction must be executed at the same time in eight instances. In other words, each of the eight instances in the execution of our batched software must process *the same instruction sequence* or, we say, *the same operation sequence* at a higher layer. This is a crucial requirement, in addition to having a constant-time implementation which is already accomplished by Algorithm 5.2. The sequence of operations in Algorithm 5.2, relies on specific conditional statements, which are (indirectly) affected by the value of the random curve points generated internally at line 7 to 8.

Mismatch problem. Specifically, a closer look at Algorithm 5.2 reveals that the sequence of operations (and instructions) that are carried out depends on whether the kernel generator R at line 13 is infinity or not. If $R \neq \infty$, the algorithm computes either a "real" or a"dummy" isogeny (depending on whether e_i is non-zero or not) in the "if"-branch, whereas it performs a scalar multiplication in the "else"-branch if $R = \infty$. In the scenario of eight parallel class group actions that we are considering, this is *problematic*, and especially, it is very likely that at some iterations the point R will be infinity at least in one of the parallel instances. In particular, the probability for a point of order ℓ_i to be infinity is $1/\ell_i$, which is considerably high when ℓ_i is small (e.g., $3, 5, 7, \ldots$ in CSIDH). This will cause a *mismatch* between the simultaneous instances and will affect other variables as well, such as the update of b'_i and the isogeny counter t_{isog} at line 21, leading to different instruction sequences for the different instances.

Solutions. Clearly, in order to obtain a constant-time batched software where eight running instances follow the same instruction sequence, we need to mitigate the mismatch caused by the infinity check at line 14. We explore three methodologies that achieve a batching-friendly CSIDH group action and deal with this specific if-else statement. In our first method, the idea is to rewrite Algorithm 5.2 so that this if-else clause is no longer needed. This requires the computation of additional dummy isogenies in the case where the kernel point is infinity, and hence we refer to this method as *extra-dummy* method (Section 5.3.2). In the second method, we still keep this if-else statement but we force all eight instances to always agree on the same branch. That is, if at least one kernel point in the eight instances is the point at infinity, then all instances will enter the "else"-branch at line 22. We refer to this methodology as *extra-infinity* method (Section 5.3.3). The third idea is based on the combination of the extra-dummy and extra-infinity methods, therefore we call it the *combined* method (Section 5.3.4).

Hybrid mode. Notably, all of our three methods are constructed in a *hybrid mode* which significantly improves the performance of the batched CSIDH implementation. In the context of this work, hybrid mode means that the entire batched software is composed of two different types of class group action implementations, namely the *batched component* and the *unbatched component*. The batched component is an *incomplete* implementation that performs eight class group action evaluations simultaneously. The unbatched component is a latency-optimized implementation accelerating a single class group action evaluation (such as the implementation of Algorithm 5.2, presented in [CCC+19]). The key idea in the three methods is to first take advantage of the batched component to compute the main bulk of the CSIDH group action (including almost all isogeny computations) for all instances, and then use eight times in sequence the unbatched component to handle the remaining computations needed in each instance. The three methods are based on the OAYT-style implementation of Algorithm 5.2 with SIMBA-m-μ. However, in Section 5.3.5, we show that all three methods can also be applied to batch the dummy-free-style algorithm of [CCC+19], which is considered to be secure against fault-injection attacks.

5.3.2 Extra-dummy method

Our first batching method initially aims at making Algorithm 5.2 independent of all inputs as well as *all randomness*. In brief, we remove the if-else clause (line 14 and line 22) that checks whether R is infinity, at a cost of *extra* dummy isogeny computations. This idea was first presented in [BLMP19] and also adopted in [JAKJ19] for a constant-time CSIDH implementation on 64-bit ARM processors. For both implementations there exists a probability of failure in computing the correct codomain curve, and a large number of dummy isogeny computations are required to make this probability negligible (e.g., 2^{-32}). In detail, according to [CCC+19], given a point $P = (Y : T)$ represented in twisted Edwards

YT-coordinates[8], we define a constant-time function for checking whether the point P is infinity as:

$$\texttt{isinfinity}(P) = \begin{cases} 1, & \text{if} \quad Y = T \Leftrightarrow P = \infty \\ 0, & \text{if} \quad Y \neq T \Leftrightarrow P \neq \infty \end{cases}$$

This time we compute a "real" isogeny from the kernel point R, if $R \neq \infty$ and $e'_i \neq 0$; whereas we compute a "dummy" isogeny if either $R = \infty$ or $e'_i = 0$. Similarly, these dummy isogenies will not be considered in the final result, but they will cause the counter t_{isog} to increment. Consequently, there is a possibility that for some indices i, the number of the computed "dummy" isogenies exceeds the value $b_i - |e_i|$ in which case we lose "real" isogenies that should be computed. This implies that although the algorithm will terminate, the resulting codomain curve will be incorrect since it will not correspond to the secret exponents $(e_1, \ldots, e_n)$. This probability of failure can be reduced by fixing the number of dummy isogenies to be computed, as done in [BLMP19, JAKJ19]. In other words, except for the dummy isogenies originally needed by Algorithm 5.2 to make the group action independent of the secret exponents (we call them *initial dummy isogenies*), we import extra dummy isogenies to make the group action independent of the randomness (we call them *extra dummy isogenies*). The modified group action has now the same operation sequence in each execution, which meets the requirement for batching. However, according to our calculation, it requires to import more than $\sum_{i=1}^{n} b_i$ extra dummy isogenies to make the failure probability below 2^{-32}. Hence, this idea needs to compute more than two times the number of isogenies needed in Algorithm 5.2, which significantly reduces the efficiency of the algorithm, while the probability of failure still exists.

Based on the above discussion, we are looking for a way to drastically reduce the number of extra dummy isogenies and eliminate the probability of failure, but meanwhile retain this batching-friendly fashion of group action. This can be done using the hybrid mode. As introduced in the previous subsection, the hybrid mode consists of the batched component and the unbatched component. In the batched component, we still compute $\sum_{i=1}^{n} b_i$ ("real" and "dummy") isogenies, where in this case, dummy isogenies appear whenever a secret exponent is zero, or the kernel point is infinity. In addition, we keep track of all the dummy isogenies that occurred from infinity kernel points. After the batched component terminates, we take advantage of the unbatched component to compute "compensatory" isogenies based on the previously recorded infinity cases. Since this method adds extra dummy isogenies for each instance that occurred from the infinity cases, we refer to it as the extra-dummy method.

Algorithm 5.3 explains the batched component of our extra-dummy method. We first apply this batched component for computing eight group action instances in parallel. Hence, in our batched software, the input is composed of a secret exponent list and a starting curve:

$$\langle (e_1^{(1)}, \ldots, e_n^{(1)}),\ (e_1^{(2)}, \ldots, e_n^{(2)}),\ \ldots,\ (e_1^{(8)}, \ldots, e_n^{(8)}) \rangle \text{ and } \langle A^{(1)},\ A^{(2)},\ \ldots,\ A^{(8)} \rangle.$$

In Algorithm 5.3, apart from the bound list $(b'_1, \ldots, b'_n)$, we also create an additional bound list for each instance to record the infinity cases:

$$\langle (\hat{b}_1^{(1)}, \ldots, \hat{b}_n^{(1)}),\ (\hat{b}_1^{(2)}, \ldots, \hat{b}_n^{(2)}),\ \ldots,\ (\hat{b}_1^{(8)}, \ldots, \hat{b}_n^{(8)}) \rangle,$$

which only decreases when an isogeny is computed from a non-infinity kernel point (line 22 in Algorithm 5.3). At the beginning of the batched component, each list $(\hat{b}_1^{(k)}, \ldots, \hat{b}_n^{(k)})$ is initialized to $\boldsymbol{b}$ (same as $(b'_1, \ldots, b'_n)$).

When the batched component terminates, it outputs for each instance, a curve coefficient $\hat{B}^{(k)}$, the list $(\hat{b}_1^{(k)}, \ldots, \hat{b}_n^{(k)})$, and the list of exponents $(e'^{(k)}_1, \ldots, e'^{(k)}_n)$, where most of the $e'^{(k)}_i$ (as well as current $\hat{b}_i^{(k)}$) are 0, for $i \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, 8\}$. As a result, there are only a few "real" (and

[8] A point P on a projective twisted Edwards curve in YT-coordinate representation is expressed as $P = (Y : T)$, where Y/T is the affine y-coordinate of the point P.

Algorithm 5.3: The batched component of our extra-dummy method for OAYT-style CSIDH class group action computation.

Input: $A \in \mathbb{F}_p$, bound $\boldsymbol{b} = (b_1, \ldots, b_n)$, exponents $\boldsymbol{e} = (e_1, \ldots, e_n)$ where $e_i \in [-b_i, b_i]$.
Output: $\hat{B} \in \mathbb{F}_p$, the lists $(\hat{b}_1, \ldots, \hat{b}_n)$ and $(e'_1, \ldots, e'_n)$.

$(e'_1, \ldots, e'_n) \leftarrow (e_1, \ldots, e_n)$, $(b'_1, \ldots, b'_n) \leftarrow (b_1, \ldots, b_n)$, $(\hat{b}_1, \ldots, \hat{b}_n) \leftarrow (b_1, \ldots, b_n)$
$E_{\hat{B}} \leftarrow E_A$, $t_{\max} \leftarrow \sum_{i=1}^{n} b_i$, $t_{\text{isog}} \leftarrow 0$, $r \leftarrow 0$, $j \leftarrow 0$
while $t_{\text{isog}} < t_{\max}$ **do**
  $j \leftarrow (j \bmod m) + 1$ /* Subset index */
  if $r = \mu \cdot m$ **then**
    $S_1 \leftarrow \{i \mid b_i \neq 0\}$, $j \leftarrow 1$, $m \leftarrow 1$ /* Merge subsets */
  $u \leftarrow \texttt{random}(\{2, \ldots, (p-1)/2\})$, $q \leftarrow \prod_{i \in S_j} \ell_i$
  $(P_0, P_1) \leftarrow \texttt{Elligator}(E_{\hat{B}}, u)$ /* $P_0 \in E[\pi - 1]$ and $P_1 \in E[\pi + 1]$ */
  $(T_0, T_1) \leftarrow ([(p+1)/q]P_0, [(p+1)/q]P_1)$
  for each $i \in S_j$ **do**
    $s \leftarrow \texttt{sign}(e'_i)$
    $\texttt{cswap}(T_0, T_1, 1 - s)$
    $R \leftarrow [q/\ell_i]T_0$ /* R is the kernel generator */
    $f \leftarrow 1 - \texttt{isinfinity}(R)$
    $w \leftarrow 1 - \texttt{isequal}(e'_i, 0)$
    Compute isogeny $\phi : E_{\hat{B}} \to E_C = \mathfrak{l}_i \star E_{\hat{B}}$ with $\ker(\phi) = \langle R \rangle$
    $T_1 \leftarrow [\ell_i]T_1$
    $(T_2, T_3) \leftarrow (\phi(T_0), \phi(T_1))$
    $T_0 \leftarrow [\ell_i]T_0$
    $\texttt{cswap}(T_0, T_2, f\&w)$, $\texttt{cswap}(T_1, T_3, f\&w)$, $\texttt{cswap}(\hat{B}, C, f\&w)$
    $e'_i \leftarrow e'_i + (-1)^s \cdot (f\&w)$, $b'_i \leftarrow b'_i - 1$, $t_{\text{isog}} \leftarrow t_{\text{isog}} + 1$
    $\hat{b}_i \leftarrow \hat{b}_i - f$
    $\texttt{cswap}(T_0, T_1, 1 - s)$, $q \leftarrow q/\ell_i$
    if $b'_i = 0$ **then**
      Remove i from S_j
  $r \leftarrow r + 1$
return $\hat{B}$, $(\hat{b}_1, \ldots, \hat{b}_n)$, $(e'_1, \ldots, e'_n)$

"dummy") remaining isogenies that need to be computed for each instance, based on $({e'}_1^{(k)}, \ldots, {e'}_n^{(k)})$ and $(\hat{b}_1^{(k)}, \ldots, \hat{b}_n^{(k)})$. Our experiments indicate that for each instance there are often around 10 ("real" and "dummy") isogenies remaining to be computed, i.e., current $\sum_{i=1}^{n} \hat{b}_i^{(k)} \approx 10$. We compute the remaining isogenies by executing the unbatched component for each instance in sequence:

$$B^{(k)} \leftarrow \texttt{unbatched}(\hat{B}^{(k)}, (\hat{b}_1^{(k)}, \ldots, \hat{b}_n^{(k)}), ({e'}_1^{(k)}, \ldots, {e'}_n^{(k)}))$$

for $k \in \{1, \ldots, 8\}$. Following the concrete CSIDH-512 parameters of [CCC+19], for each instance, there are exactly 404 isogeny computations ("real" and "dummy") in the batched component while a few isogeny computations corresponding to non-zero $\hat{b}_i^{(k)}$ in the unbatched component. Thus, for each instance, the extra-dummy method computes only a few more extra isogenies, compared to the conventional OAYT-style implementation (Algorithm 5.2). Moreover, since the unbatched component has no failure probability, we conclude that the extra-dummy method has no failure probability either.

5.3.3 Extra-infinity method

We assume that eight different instances of Algorithm 5.2 are computed in parallel. In brief, the idea in our second method is that for each iteration of the inner loop (line 10 to line 26 of Algorithm 5.2), if the kernel generator is infinity in at least one of the eight instances, then we force all instances to execute the "else"-branch at line 22. In particular, we define the variable inf as:

$$inf = \texttt{isinfinity}(R^{(1)}) \mid \texttt{isinfinity}(R^{(2)}) \mid \ldots \mid \texttt{isinfinity}(R^{(8)}),$$

where $R^{(k)}$ denotes the kernel generator in the k^{th} simultaneous instance. If $inf = 0$, then $R^{(k)} \neq \infty$ for all $k \in \{1, \ldots, 8\}$ and all eight instances will enter the "if"-branch at line 14 in Algorithm 5.2, in order to compute a "real" or a "dummy" ℓ_i-isogeny. On the other hand, if $inf = 1$, then $R^{(k)} = \infty$ for at least one $k \in \{1, \ldots, 8\}$ and all instances will proceed to the "else"-branch. In this case, we need to execute the scalar multiplication $T_0^{(k)} = [\ell_i]T_0^{(k)}$, in addition to $T_1^{(k)} = [\ell_i]T_1^{(k)}$. This is not needed in Algorithm 5.2, because the ℓ_i-torsion part of the point T_0 has already vanished (since $R = \infty$). In our approach, when $inf = 1$, we are forcing all instances to proceed as if all $R^{(k)}$ were infinity, however the ℓ_i-torsion parts of some points $T_0^{(k)}$ have not vanished.

However, when $inf = 1$, the above idea imports some extra infinity-related computations, which in principle are not needed by every instance. In addition to the two scalar multiplications for $T_0^{(k)}$ and $T_1^{(k)}$ in the "else"-branch, these infinity-related computations may include more point generation operations (using the Elligator map at line 8), which will result in more scalar multiplications for the full order points $P_0^{(k)}, P_1^{(k)}$ (line 9) and the kernel generator $R^{(k)}$ (line 13). For this reason, we refer to this method as the extra-infinity method. Note also that the probability of $inf = 1$ is $1-(1-1/\ell_i)^8$, which is considerably higher when ℓ_i is small (e.g., 3, 5, and 7). As a result, an increased number of $inf = 1$ cases (and hence, an increased number of infinity-related computations) is expected, which affects the efficiency of the extra-infinity method.

We mitigate this problem by considering again the hybrid mode. More precisely, we divide the primes $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_n)$ into two subsets, $\boldsymbol{\ell}_{\text{batch}}$ for the batched component and $\boldsymbol{\ell}_{\text{unbatch}}$ for the unbatched component. $\boldsymbol{\ell}_{\text{unbatch}}$ contains only the smaller primes, specifically 3, 5, 7, 11, 13, 17 and 19 in our implementation, whereas $\boldsymbol{\ell}_{\text{batch}}$ includes the remaining primes in $\boldsymbol{\ell}$. In the same way, the bound list $\boldsymbol{b} = (b_1, \ldots, b_n)$ and the secret exponent list $\boldsymbol{e^{(k)}} = (e_1^{(k)}, \ldots, e_n^{(k)})$ of each instance are split in two subsets, i.e. $\{\boldsymbol{b}_{\text{batch}}, \boldsymbol{b}_{\text{unbatch}}\}$ and $\{\boldsymbol{e}_{\text{batch}}^{(\boldsymbol{k})}, \boldsymbol{e}_{\text{unbatch}}^{(\boldsymbol{k})}\}$ for $k \in \{1, \ldots, 8\}$.

In the extra-infinity method, we first execute the batched component for eight parallel group action instances, to compute the isogenies for the larger primes with the corresponding subsets. The batched component outputs the resulting curve $\hat{B}^{(k)}$ for each instance:

$$\langle \hat{B}^{(1)}, \hat{B}^{(2)}, \ldots, \hat{B}^{(8)} \rangle \leftarrow \texttt{batched}(\langle A^{(1)}, A^{(2)}, \ldots, A^{(8)} \rangle, \boldsymbol{b}_{\text{batch}}, \langle \boldsymbol{e}_{\text{batch}}^{(\boldsymbol{1})}, \boldsymbol{e}_{\text{batch}}^{(\boldsymbol{2})}, \ldots, \boldsymbol{e}_{\text{batch}}^{(\boldsymbol{8})} \rangle)$$

Then we execute the unbatched component (such as Algorithm 5.2) sequentially in order to obtain the correct codomain curve for each instance:

$$B^{(k)} \leftarrow \texttt{unbatched}(\hat{B}^{(k)}, \boldsymbol{b}_{\text{unbatch}}, \boldsymbol{e}_{\text{unbatch}}^{(\boldsymbol{k})})$$

for $k \in \{1, \ldots, 8\}$. The number of total isogenies ("real" and "dummy") that are computed in the batched component for each instance is the sum of all b_i in $\boldsymbol{b}_{\text{batch}}$, which is 358 when considering the CSIDH-512 parameters of [CCC+19]. In the unbatched component, the number of total isogenies ("real" and "dummy") is 46, i.e. the sum of all b_i in $\boldsymbol{b}_{\text{unbatch}}$.

5.3.4 The combination of extra-dummy and extra-infinity methods

Before we introduce the combined method, we give a few more details on the extra-dummy and extra-infinity methods. We consider an example where in an iteration of the inner loop, n_{inf} of the eight kernel points $R^{(k)} = \infty$, in the batched component of both methods. The extra-dummy method will complete the computations of this iteration (from line 14 to 25 in Algorithm 5.3), and later it will compute n_{inf} "compensatory" isogenies with the unbatched component. On the other hand, the extra-infinity method will enter its "else"-branch to compute two scalar multiplications, for all eight instances, and it may later perform the other infinity-related computations, which are in theory needed by n_{inf} instances. Based on the operations that are carried out in each method, we observe that the extra-dummy method handles the infinity cases more efficiently than the extra-infinity method when only few $R^{(k)} = \infty$, hence when n_{inf} is small. On the other hand, the extra-infinity method seems to be more efficient in handling the infinity cases, when most of the eight $R^{(k)} = \infty$ (i.e. when n_{inf} is close to 8).

Based on the above observation, our idea is to combine the two approaches, aiming at obtaining a more efficient method. In order to do this, we set the variable n_{inf} as:

$$n_{\text{inf}} = \texttt{isinfinity}(R^{(1)}) + \texttt{isinfinity}(R^{(2)}) + \ldots + \texttt{isinfinity}(R^{(8)}),$$

so that $n_{\text{inf}} \in [0, 8]$. Taking Algorithm 5.3 to describe this combined method, after the computation at line 13 (where the kernel generator $R^{(k)}$ is computed), we add an if-else statement to check if the variable n_{inf} is within a predefined threshold n_{thld}. If $n_{\text{inf}} \leq n_{\text{thld}}$ which means there are few $R^{(k)} = \infty$, we do the same computations as in the extra-dummy method (line 14 to 22 of Algorithm 5.3). On the other hand, if $n_{\text{inf}} > n_{\text{thld}}$ which means there are more $R^{(k)} = \infty$, we proceed to the "else"-branch of the extra-infinity method and perform the two scalar multiplications $T_1^{(k)} = [\ell_i]T_1^{(k)}$ and $T_0^{(k)} = [\ell_i]T_0^{(k)}$. After this if-else statement, the operations at line 23 to 25 will be performed. Additionally, the unbatched component of the combined method is the same as the one in the extra-dummy method. From our experiments, when the threshold $n_{\text{thld}} = 3$, this combined method provides the best performance, and particularly, it is slightly faster than the extra-dummy method and quite faster than the extra-infinity method.

5.3.5 Batching dummy-free-style group actions

The methods that we have considered in Section 5.3.2, Section 5.3.3, and Section 5.3.4 for batching CSIDH group actions are all based on the OAYT-style algorithm of [OAYT19] and require the computation of dummy isogenies. More precisely, recall that Algorithm 5.2 computes $|e_i|$ "real" and $b_i - |e_i|$ "dummy" isogenies. Such implementations are vulnerable to fault injection attacks. As observed in [CCC+19], an attacker can modify the codomain curve or the images of the points T_0, T_1 under the isogeny ϕ in Algorithm 5.2 (fault injections), and if the result is correct, he knows that a dummy isogeny is computed and thus $e_i = 0$. This is also true in Algorithm 5.3. If the same modification produces the correct result, then the attacker knows that either $e_i = 0$, or the current kernel generator $R = \infty$.

In [CCC+19], Cervantes-Vázquez, Chenu, Chi-Domínguez, De Feo, Rodríguez-Henríquez, and Smith presented a constant-time evaluation of the CSIDH class group action, without the need of dummy isogeny computations [CCC+19, Algorithm 5]. In their dummy-free approach, the secret exponents are chosen such that $e_i \in [-b_i, b_i]$ and $e_i \equiv b_i \pmod 2$. This choice allows the algorithm to compute the required $|e_i|$ isogenies, while for the remaining $b_i - |e_i|$, it alternates between the actions of the ideals $\mathfrak{l}_i$ and $\mathfrak{l}_i^{-1}$ and hence these $b_i - |e_i|$ isogenies cancel out (see [CCC+19, Section 5] for details). The dummy-free approach is based on the SIMBA-m-μ technique, where the implementation of [CCC+19] uses $m = 5$ and $\mu = 11$. The secret key space or equivalently, the list of bounds for the secret exponents

is:

$$\begin{aligned}\boldsymbol{b} = (&7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,\\ &8, 8, 8, 8, 8, 8, 8, 11, 11, 11, 11, 11, 11, 11, 11, 11,\\ &11, 11, 11, 11, 11, 11, 11, 11, 11, 13, 13, 13, 13, 13, 13, 13,\\ &13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,\\ &13, 13, 13, 13, 13, 5, 7, 7, 7, 7)\end{aligned}$$

We argue that the three methods that we have introduced in Section 5.3.2, Section 5.3.3, and Section 5.3.4 can also be used to batch the dummy-free-style algorithm of [CCC+19]. In particular, for the extra-dummy and the combined methods, we are still using dummy isogenies for the case where the kernel generator $R = \infty$, however, these dummy isogenies do not reveal any information about the secret exponent, since they depend only on the random kernel generator. For the extra-infinity method, we follow the same strategy as in Section 5.3.3, with the only difference being that the small prime list in the unbatched component is $\boldsymbol{\ell}_{\text{unbatch}} = (3, 5, 7, 11, 13, 17, 19, 23, 29)$. For the combined method in the dummy-free-style, the optimal threshold to achieve the best performance is $n_{\text{thld}} = 5$.

5.3.6 (8×1)-way prime-field arithmetic

In the batched components of all batching methods, we use the same curve and isogeny arithmetic implementation that we developed, based on [CCC+19], with minor optimizations to better fit the batched software. At a lower layer, we developed all the needed (8×1)-way[9] prime-field operations from scratch, using respectively AVX-512F and IFMA, by taking advantage of "limb-slicing" technique [CGT+20]. This section only studies our IFMA vectorized implementation of prime-field operations. Compared to the IFMA version, the AVX-512F implementation has two fundamental differences; 1) it uses `vpmuludq` instead of IFMA instructions to perform vector multiplication; 2) the field element is represented in radix-2^{29} (with 18 limbs) due to the 32-bit multiplier.

Radix-2^{52} (8×1)-way limb vector set IFMA naturally provides a reduced radix representation, namely radix-2^{52}, for large integers. Fortunately, radix-2^{52} is well-suited for CSIDH-512. Specifically, there are ten limbs for a 511-bit field element under radix-2^{52}. When considering a smaller radix, such as radix-2^{51}, the representation of an element will require at least eleven limbs, which leads to a higher consumption than radix-2^{52}. Formally, a field element f represented in radix-2^{52} is shown as:

$$f = f_0 + 2^{52} f_1 + 2^{104} f_2 + 2^{156} f_3 + 2^{208} f_4 + 2^{260} f_5 + 2^{312} f_6 + 2^{364} f_7 + 2^{416} f_8 + 2^{468} f_9,$$

where $0 \leq f_i < 2^{52}$ for $0 \leq i \leq 9$. This representation allows field elements to be up to 520-bit during computations.

The main data structure of our parallel software is a (8×1)-way *limb vector set*, which is composed of eight radix-2^{52} elements. Given eight integers $a, b, c, d, e, f, g, h \in \mathbb{F}_p$, a (8×1)-way limb vector set $\boldsymbol{V}$ is defined as:

$$\boldsymbol{V} = \langle a, b, c, d, e, f, g, h \rangle = \left\{ \begin{matrix} [a_0, b_0, c_0, d_0, e_0, f_0, g_0, h_0] \\ [a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1] \\ \vdots \\ [a_9, b_9, c_9, d_9, e_9, f_9, g_9, h_9] \end{matrix} \right\} = (V_0, V_1, \ldots, V_9)$$

where each $V_i = [a_i, b_i, c_i, d_i, e_i, f_i, g_i, h_i]$ is called a limb vector. All the inputs and outputs of our (8×1)-way field operations are limb vector sets of which each limb is precisely 52 bits long. In terms of our field

[9] The $(n \times m)$-way implementation performs n field operations in parallel, where each field operation is executed in a m-way parallel fashion and, thus, uses m elements of a vector.

operations, we saved the final subtraction in Montgomery reduction, and our addition and subtraction perform reduction modulo $2p$ instead of p. This means all the integers inputted to or outputted from our field operations are in the range $[0, 2p-1]$. We use $\boldsymbol{P} = \langle p, p, p, p, p, p, p, p \rangle$ to denote an (8×1)-way limb vector set of prime p, and $\boldsymbol{Q} = 2 \times \boldsymbol{P} = \langle 2p, 2p, 2p, 2p, 2p, 2p, 2p, 2p \rangle$.

Addition and subtraction. Field addition $\boldsymbol{Z} \leftarrow \boldsymbol{X} + \boldsymbol{Y} \bmod \boldsymbol{Q}$ is performed in three steps. At first, we add $\boldsymbol{X}$ with $\boldsymbol{Y}$ and store their sum in $\boldsymbol{Z}$. We then subtract $\boldsymbol{Q}$ from $\boldsymbol{Z}$, so there might be negative results in some lanes of $\boldsymbol{Z}$. Finally, we create a 512-bit mask vector where the 64-bit element is set to all-1 if the corresponding lane's integer in $\boldsymbol{Z}$ is negative, or to all-0 if non-negative. Through the bitwise AND between this mask vector and $\boldsymbol{Q}$, we add $2p$ to the negative integers in $\boldsymbol{Z}$ whereas add 0 to the non-negative integers. There are only two steps in the field subtraction $\boldsymbol{Z} \leftarrow \boldsymbol{X} - \boldsymbol{Y} \bmod \boldsymbol{Q}$, which first subtracts $\boldsymbol{Y}$ from $\boldsymbol{X}$ and then carries out a same final step of field addition.

Multiplication. Field multiplication has a significant impact on the performance of any isogeny-based cryptosystem and deserves special care. The field multiplication used in CSIDH is Montgomery multiplication [Mon85] which consists of two phases, namely integer multiplication and Montgomery reduction. There exist some different variants of Montgomery multiplication, often termed with their implementation fashion, such as Separated Operand Scanning (SOS) [KAK96], Finely Integrated Product Scanning (FIPS) [KAK96], Karatsuba-Comba-Montgomery (KCM) [GAST05] and etc. The number of instructions (including addition, multiplication, load/store) and memory consumption required for different variants are different. Taking these two factors into account, implementers can choose a proper variant when they develop software especially on resource-constrained devices like AVR or ARM microcontrollers. For the cost comparison of different variants we refer to [KAK96, Table 1] and [GAST05, Table 4]. However, things become more complicated when developing on a processor with more computing power. Considering our case, Ice Lake processor is equipped with ten execution ports (and various execution units), so the processor can execute several instructions simultaneously. Excluding the number of needed instructions and instruction latency/throughput, we are supposed to take instruction-level parallelism into account. The memory consumption receives less attention in this case since an Ice Lake machine usually possesses a GB-level memory.

Currently, most of the related AVX-512 implementations are designed for accelerating 1-, 2- or 4-way Montgomery multiplication. However, these optimization techniques are not ideally applicable to our 8-way software. We discuss these implementations in more detail in Section 5.4.1. Due to the "limb-slicing" pattern, our 8-way Montgomery multiplication essentially "duplicates" 1-way implementation to eight lanes by AVX-512 instructions. To the best of our knowledge, there are only two AVX-512 implementations of this type in the literature. Takahashi proposed both AVX-512F and IFMA implementation of 8-way Montgomery multiplication in [Tak20], but this software works on 62-bit and 52-bit operands, respectively, and not in the case of large integers. Buhrow, Gilbert, and Haider in [BGH22] presented a Block Product Scanning (BPS) variant of Montgomery multiplication, which is based on radix-2^{32} representation. An 8-way 512-bit BPS variant implemented with AVX-512F takes 189 clock cycles for each instance, which translates to 1512 clock cycles for a whole 8-way implementation.

In order to find an optimal field multiplication for our software, the best way is to develop the corresponding 8-way AVX-512 implementation of various Montgomery multiplication variants and select the fastest one among them. From an algorithmic viewpoint, all the variants differ in three aspects: 1) different methods to implement integer multiplication, e.g., operand-scanning, product-scanning, or the advanced technique such as Karatsuba algorithm [KO63]; 2) different methods to implement Montgomery reduction, e.g., operand-scanning or product-scanning; 3) whether Montgomery reduction is *separated* from or *interleaved* with integer multiplication (and how it is interleaved in the latter case). For our IFMA version, we conducted experiments in which we developed a dozen of implementation candidates of 8-way field multiplication based on various combinations from above three aspects. No-

Algorithm 5.4: (8×1)-way Montgomery multiplication using IFMA.

Input: Operands $\boldsymbol{X}$ and $\boldsymbol{Y}$, prime $\boldsymbol{P}$, $w = -p^{-1} \bmod 2^{52}$
Output: Product $\boldsymbol{Z} = \boldsymbol{X} \times \boldsymbol{Y} \times 2^{-520} \bmod \boldsymbol{Q}$

1 $Z_i \leftarrow \mathsf{ZERO}$ for $i \in \{0, 1, \ldots, 19\}$
2 $W \leftarrow \mathsf{BCAST}(w)$, $M \leftarrow \mathsf{BCAST}(2^{52} - 1)$
3 **for** i **from** 0 **to** 9 **by** 1 **do**
4 **for** j **from** 0 **to** i **by** 1 **do**
5 $Z_i \leftarrow \mathsf{MACLO}(Z_i, X_j, Y_{i-j})$
6 $Z_{i+1} \leftarrow \mathsf{MACHI}(Z_{i+1}, X_j, Y_{i-j})$
7 **for** i **from** 0 **to** 9 **by** 1 **do**
8 **for** j **from** $i + 1$ **to** 9 **by** 1 **do** /* Skip this loop when $i = 9$ */
9 $Z_{i+10} \leftarrow \mathsf{MACLO}(Z_{i+10}, X_j, Y_{i-j+10})$
10 $Z_{i+11} \leftarrow \mathsf{MACHI}(Z_{i+11}, X_j, Y_{i-j+10})$
11 $T \leftarrow \mathsf{MACLO}(\mathsf{ZERO}, Z_i, W)$
12 **for** j **from** 0 **to** 9 **do**
13 $Z_{i+j} \leftarrow \mathsf{MACLO}(Z_{i+j}, T, P_j)$
14 $Z_{i+j+1} \leftarrow \mathsf{MACHI}(Z_{i+j+1}, T, P_j)$
15 $Z_{i+1} \leftarrow \mathsf{ADD}(Z_{i+1}, \mathsf{SHR}(Z_i, 52))$
16 **for** i **from** 10 **to** 18 **by** 1 **do**
17 $Z_{i+1} \leftarrow \mathsf{ADD}(Z_{i+1}, \mathsf{SHR}(Z_i, 52))$
18 $Z_{i-10} \leftarrow \mathsf{AND}(Z_i, M)$
19 $Z_9 \leftarrow Z_{19}$
20 **return** $\boldsymbol{Z} = (Z_0, Z_1, \ldots, Z_9)$

Table 5.2: Information of our (8×1)-way implementation of field multiplication and squaring.

Field operation	ISA/ISE	Integer Mul/Sqr	Reduction	Structure	Latency[†]
Multiplication	IFMA	Product-Scanning	Operand-Scanning	Interleaved	436
Squaring	IFMA	Product-Scanning	Operand-Scanning	Interleaved	344
Multiplication	AVX-512F	Karatsuba	Product-Scanning	Separated	848
Squaring	AVX-512F	Product-Scanning	Product-Scanning	Interleaved	723

[†] Latency (in clock cycles) is the execution time of eight parallel Montgomery multiplication/squaring instances, and it was measured on Ice Lake i3-1005G1 processor with turbo boost disabled.

tably, our 8-way implementation candidates are not straightforwardly "duplicating" the ordinary 1-way x64 implementation of different variants (or we say combinations). We rather concentrated on improving instruction-level parallelism in each implementation candidate. In order to achieve this purpose, we tried to improve the ports utilization by optimizing dependency chains in the code. From our benchmarking results on Ice Lake processor, the implementation candidate with the lowest latency is shown in Algorithm 5.4, which possesses a similar structure as Coarsely Integrated Hybrid Scanning (CIHS) [KAK96], and it serves as field multiplication in our 8-way IFMA software. Our field multiplication uses product-scanning for integer multiplication and utilizes operand-scanning to handle Montgomery reduction, and reduction is interleaved with the second outer loop of product-scanning integer multiplication (line 7 to 15 in Algorithm 5.4). As for our AVX-512F version, we carried out a similar procedure to evaluate also a dozen of AVX-512F implementation candidates. The optimal field multiplication in AVX-512F version switches to Karatsuba algorithm for integer multiplication since there are 18 limbs of each element, and uses a product-scanning Montgomery reduction that is separated from integer multiplication. The information and latency of field multiplication in both versions are shown in Table 5.2,

which indicates that our Karatsuba-based AVX-512F implementation outperforms the BPS variant in [BGH22]. We herein emphasize on the importance of using an optimal field multiplication in such parallel AVX-512 software of an isogeny-based cryptosystem. In our experiments for IFMA version, the 8-way Separated Product Scanning (SPS) variant [LG14] or 8-way original FIPS variant [KAK96] (i.e., the one that has not been optimized for improving instruction-level parallelism) costs more than 700 clock cycles, i.e., taking 60% more CPU-cycles than Algorithm 5.4. From our experiments, using such unsuitable field multiplication and squaring implementation will finally result in up to 30% more CPU-cycles for CSIDH group action evaluation compared to the one using optimal variants.

Squaring. Most of the existing CSIDH implementations, e.g. [MCR19, OAYT19, CCC+19, CR22, HLKA20], take advantage of a same x64 assembly implementation of field operations originally from [CLM+18]. In this assembly implementation, a field squaring just invokes a field multiplication in which two operands are same. In other words, field squaring possesses the same latency as field multiplication. In this work, we developed a dedicated Montgomery squaring instead of directly using field multiplication. Specifically, compared to field multiplication, our field squaring utilizes a dedicated integer squaring instead of integer multiplication.

In essence, integer squaring is a special instance of multiplication, where all partial products in the form of $f_i \cdot f_j$ with $i \neq j$ appear twice due to $f_i \cdot f_j = f_j \cdot f_i$. A classic technique for optimizing squaring is to just compute these partial products once and double them, thereby saving numerous multiplication instructions. We develop our integer squaring by this classic technique, and again we developed many squaring candidates to obtain an optimal implementation. The information of our field squaring implementation is also listed in Table 5.2 where it proves a dedicated field squaring saves at least 15% CPU-cycles than a field multiplication. The algorithmic description of our IFMA 8-way Montgomery squaring is shown in Algorithm 5.5.

5.4 Implementation: low-latency unbatched software

In our hybrid mode which is introduced in Section 5.3.1, the low-latency implementation of a single group action evaluation serves as the unbatched component and is needed by each instance. More importantly, this low-latency implementation can also be used in more applications e.g., accelerating the CSIDH key exchange protocol on the client side.In this section we describe our (2×4)-way IFMA implementation, which is developed for accelerating a single group action evaluation and used as the unbatched component in our IFMA throughput-optimized software. In the case of AVX-512F, our experiments showed that the (2×4)-way AVX-512F implementation is slower than the x64 implementation of [CCC+19] (on target Ice Lake Core processor). Hence, for our AVX-512F throughput-optimized software, we use the [CCC+19] implementation as the unbatched component.

5.4.1 (2×4)-way prime-field arithmetic

Radix-2^{43} (2×4)-way limb vector set. Neither the structure nor the radix of the (2×4)-way limb vector set is the same compared to the (8×1)-way set. To be specific, we take advantage of (2×4)-way interleaved vectors combined with radix-2^{43} this time. The (2×4)-way limb vector set $\boldsymbol{V} = \langle a, b \rangle$ is defined as follows:

$$\boldsymbol{V} = \langle a, b \rangle = \left\{ \begin{matrix} [a_0, a_3, a_6, a_9\ , b_0, b_3, b_6, b_9\ ] \\ [a_1, a_4, a_7, a_{10}, b_1, b_4, b_7, b_{10}] \\ [a_2, a_5, a_8, a_{11}, b_2, b_5, b_8, b_{11}] \end{matrix} \right\} = (V_0, V_1, V_2)$$

Each limb vector $V_i = [a_i, a_{i+3}, a_{i+6}, a_{i+9}, b_i, b_{i+3}, b_{i+6}, b_{i+9}]$ contains four limbs from each integer a and b, and limbs are arranged in an interleaved pattern. The reason for using radix-2^{43} instead of radix-2^{52}

Algorithm 5.5: (8×1)-way Montgomery squaring using IFMA.

Input: Operand $\boldsymbol{X}$, prime $\boldsymbol{P}$, $w = -p^{-1} \bmod 2^{52}$
Output: Product $\boldsymbol{Z} = \boldsymbol{X}^2 \times 2^{-520} \bmod \boldsymbol{Q}$

$Z_i \leftarrow \mathsf{ZERO}$ for $i \in \{0, 1, \ldots, 19\}$
$W \leftarrow \mathsf{BCAST}(w)$, $M \leftarrow \mathsf{BCAST}(2^{52} - 1)$
for i **from** 1 **to** 9 **by** 1 **do**
    for j **from** 0 **to** $\lfloor (i-1)/2 \rfloor$ **by** 1 **do**
        $Z_i \leftarrow \mathsf{MACLO}(Z_i, X_j, X_{i-j})$
        $Z_{i+1} \leftarrow \mathsf{MACHI}(Z_{i+1}, X_j, X_{i-j})$
    $Z_i \leftarrow \mathsf{ADD}(Z_i, Z_i)$
    if *i is odd* **then**
        $k \leftarrow (i-1)/2$
        $Z_{i-1} \leftarrow \mathsf{MACLO}(Z_{i-1}, X_k, X_k)$
        $Z_i \leftarrow \mathsf{MACHI}(Z_i, X_k, X_k)$
for i **from** 0 **to** 9 **by** 1 **do**
    for j **from** $\lfloor i/2 \rfloor + 6$ **to** 9 **by** 1 **do** /* Skip this loop when $i = 8$ or 9 */
        $Z_{i+10} \leftarrow \mathsf{MACLO}(Z_{i+10}, X_j, Y_{i-j+10})$
        $Z_{i+11} \leftarrow \mathsf{MACHI}(Z_{i+11}, X_j, Y_{i-j+10})$
    $Z_{i+10} \leftarrow \mathsf{ADD}(Z_{i+10}, Z_{i+10})$
    if *i is odd* **then**
        $k \leftarrow (i+9)/2$
        $Z_{i+9} \leftarrow \mathsf{MACLO}(Z_{i+9}, X_k, X_k)$
        $Z_{i+10} \leftarrow \mathsf{MACHI}(Z_{i+10}, X_k, X_k)$
    $T \leftarrow \mathsf{MACLO}(\mathsf{ZERO}, Z_i, W)$
    for j **from** 0 **to** 9 **do**
        $Z_{i+j} \leftarrow \mathsf{MACLO}(Z_{i+j}, T, P_j)$
        $Z_{i+j+1} \leftarrow \mathsf{MACHI}(Z_{i+j+1}, T, P_j)$
    $Z_{i+1} \leftarrow \mathsf{ADD}(Z_{i+1}, \mathsf{SHR}(Z_i, 52))$
for i **from** 10 **to** 18 **by** 1 **do**
    $Z_{i+1} \leftarrow \mathsf{ADD}(Z_{i+1}, \mathsf{SHR}(Z_i, 52))$
    $Z_{i-10} \leftarrow \mathsf{AND}(Z_i, M)$
$Z_9 \leftarrow Z_{19}$
return $\boldsymbol{Z} = (Z_0, Z_1, \ldots, Z_9)$

Algorithm 5.6: (2×4)-way Montgomery multiplication using IFMA.

Input: Operands $\boldsymbol{X} = \langle a, b \rangle$ and $\boldsymbol{Y} = \langle c, d \rangle$, prime $\boldsymbol{P} = \langle p, p \rangle$, $w = -p^{-1} \bmod 2^{43}$.
Output: Product $\boldsymbol{Z} = \langle e, f \rangle$ where $e = a \cdot c \cdot 2^{-516} \bmod 2p$ and $f = b \cdot d \cdot 2^{-516} \bmod 2p$.

$L_i \leftarrow \mathsf{ZERO}, H_i \leftarrow \mathsf{ZERO}$ for $i \in \{0, 1, \ldots, 14\}$
$W \leftarrow \mathsf{BCAST}(w), M \leftarrow \mathsf{BCAST}(2^{43} - 1)$
for i **from** 0 **to** 11 **by** 1 **do**
  $T \leftarrow \mathsf{PERM}(Y_{i \bmod 3}, 0\text{x}55 \cdot \lfloor i/3 \rfloor)$
  for j **from** 0 **to** 2 **by** 1 **do**
    $L_{i+j} \leftarrow \mathsf{MACLO}(L_{i+j}, T, X_j)$
    $H_{i+j} \leftarrow \mathsf{MACHI}(H_{i+j}, T, X_j)$
  $U \leftarrow \mathsf{AND}(\mathsf{MACLO}(\mathsf{ZERO}, W, \mathsf{PERM}(L_i, 0\text{x}00)), M)$
  for j **from** 0 **to** 2 **by** 1 **do**
    $L_{i+j} \leftarrow \mathsf{MACLO}(L_{i+j}, U, P_j)$
    $H_{i+j} \leftarrow \mathsf{MACHI}(H_{i+j}, U, P_j)$
  $L_{i+3} \leftarrow \mathsf{ADD}(L_{i+3}, 0\text{x}77, L_{i+3}, \mathsf{PERM}(L_i, 0\text{x}39))$
  $L_{i+1} \leftarrow \mathsf{ADD}(L_{i+1}, 0\text{x}11, L_{i+1}, \mathsf{SHR}(L_i, 43))$
  $L_{i+1} \leftarrow \mathsf{ADD}(L_{i+1}, \mathsf{SHL}(H_i, 9))$
$L_{13} \leftarrow \mathsf{ADD}(L_{13}, \mathsf{SHL}(H_{12}, 9))$
$L_{14} \leftarrow \mathsf{ADD}(L_{14}, \mathsf{SHL}(H_{13}, 9))$
$Z_0 \leftarrow L_{12}, Z_1 \leftarrow L_{13}, Z_2 \leftarrow L_{14}$
for i **from** 0 **to** 1 **by** 1 **do**
  $C \leftarrow \mathsf{SHR}(Z_i, 43)$
  $Z_i \leftarrow \mathsf{AND}(Z_i, M)$
  $Z_{i+1} \leftarrow \mathsf{ADD}(Z_{i+1}, C)$
$C \leftarrow \mathsf{SHR}(Z_2, 43)$
$Z_2 \leftarrow \mathsf{AND}(Z_2, M)$
$Z_0 \leftarrow \mathsf{ADD}(Z_0, 0\text{xEE}, Z_0, \mathsf{PERM}(C, 0\text{x}93))$
return $\boldsymbol{Z} = (Z_0, Z_1, Z_2)$

is now easily inferred from the equation above. It is because there will still remain three limb vectors if using radix-2^{52} (ten limbs for each integer), whereas radix-2^{43} offers more headroom in each limb (a_i or b_i) and is thus friendly for delaying the carry propagation. Similar to the (8×1)-way implementation, our (8×1)-way implementation also saves a final subtraction in Montgomery reduction and performs modulo $2p$ instead of p reduction in field addition and subtraction.

Mixed addition and subtraction. In curve and isogeny arithmetic, we can generally perform a pair of field addition and subtraction simultaneously, but not two additions or two subtractions due to sequential dependency. Therefore, it makes more sense to develop a parallel and mixed operation of addition and subtraction. We denote this mixed operation as "$\pm$". Formally, it works as $\langle r, s \rangle \leftarrow \langle a, b \rangle \pm \langle c, d \rangle$ where $r = a + c \bmod 2p$ and $s = b - d \bmod 2p$. In essence, this mixed operation executes the similar steps described in Section 5.3.6. At first, we construct two (2×4)-way limb vector sets $\langle c, 0 \rangle$ and $\langle 2p, d \rangle$. We add $\langle a, b \rangle$ with $\langle c, 0 \rangle$, and then subtract $\langle 2p, d \rangle$ from the sum to reach $\langle a + c - 2p, b - d \rangle$. The final step is similar to Section 5.3.6 in order to ensure the results of this mixed operation are in $[0, 2p - 1]$ by a mask vector.

Multiplication. As we mentioned in Section 5.3.6, some work has already been done for accelerating 1-, 2-, or 4-way Montgomery multiplication or squaring with AVX-512. Several papers have been published which focus on using IFMA to accelerate 1-way large integer arithmetic such as integer multipli-

Algorithm 5.7: (2×4)-way integer squaring using IFMA.

Input: Operand $\boldsymbol{X} = \langle a, b \rangle$.
Output: $\boldsymbol{L} = \langle lo(e), lo(f) \rangle$ and $\boldsymbol{H} = \langle hi(e), hi(f) \rangle$, where $e = a^2$ and $f = b^2$.

1 $L_i \leftarrow$ ZERO, $H_i \leftarrow$ ZERO for $i \in \{0, 1, \ldots, 14\}$
2 **for** i **from** 0 **to** 11 **do**
3 $k \leftarrow i \bmod 3$
4 $T \leftarrow \mathtt{PERM}(X_k, 0\text{x}55 \cdot \lfloor i/3 \rfloor)$
5 $L_{i+k} \leftarrow \mathtt{MACLO}(L_{i+k}, T, X_k)$
6 $H_{i+k} \leftarrow \mathtt{MACHI}(H_{i+k}, T, X_k)$
7 $D \leftarrow \mathtt{ADD}(T, T)$ /* Skip this addition when $k = 2$ */
8 **for** j **from** $k + 1$ **to** 2 **by** 1 **do** /* Skip this loop when $k = 2$ */
9 $L_{i+j} \leftarrow \mathtt{MACLO}(L_{i+j}, D, X_j)$
10 $H_{i+j} \leftarrow \mathtt{MACHI}(H_{i+j}, D, X_j)$
11 **return** $\boldsymbol{L} = (L_0, L_1, \ldots, L_{14})$, $\boldsymbol{H} = (H_0, H_1, \ldots, H_{14})$

cation [GK16, KG19] and Montgomery squaring [DG18]. Edamatsu and Takahashi in [ET19] presented an IFMA implementation of single large integer multiplication, which takes advantage of Karatsuba algorithm. Apart from the work on 1-way implementation acceleration, Orisaka, Aranha, and López presented a well-designed and fast (4×2)-way Montgomery multiplication for SIDH in [OAL18] by AVX-512F, and their approach is working on the 4-way interleaved vectors. We designed our (4×2)-way IFMA Montgomery multiplication based on the approach of [OAL18] with several modifications: 1) we use IFMA instructions to replace `vpmuldq` and save `vpaddq`; 2) we apply our (2×4)-way limb vector set; 3) we implement integer multiplication and reduction in interleaved fashion instead of separated one which is originally-used, because the interleaved fashion is measured to be faster than the separated one from our experiments. Our (2×4)-way field multiplication is described in Algorithm 5.6. Vector sets $\boldsymbol{L}$ and $\boldsymbol{H}$ respectively accumulate the partial products produced by `vpmadd52lo` and `vpmadd52hi`. Notably, excluding the computation at line 14, there is no dependency between $\boldsymbol{L}$ and $\boldsymbol{H}$ in the main loop (line 3 to 14), which benefits the efficient utilization of ports.

Squaring. Orisaka et al. did not present a dedicated integer squaring in [OAL18] but planned it as a future work. We herein propose a fast integer squaring by using the classic optimization technique that we described in Section 5.3.6. Our integer squaring can be slightly modified to fit any (2×4)-way or (2×4)-way AVX-512 Montgomery squaring that uses interleaved vectors, e.g. the integer squaring needed in [OAL18]. Our method is described in Algorithm 5.7, which saves 24 IFMA instructions compared to an integer multiplication (corresponding to line 5 to 7 in Algorithm 5.6) which requires 72 IFMA instructions in total. We keep the output of Algorithm 5.7 in two sets $\boldsymbol{L}$ and $\boldsymbol{H}$, since our Montgomery reduction is designed to directly work on them. Our complete Montgomery squaring just replaces the integer multiplication part of Algorithm 5.6 by Algorithm 5.7.

5.4.2 Curve and isogeny arithmetic

Following [CCC+19], the curve arithmetic mainly includes y-coordinate point doubling, point addition and scalar multiplication (using addition chains) on twisted Edwards curve, whereas the isogeny operations contains y-coordinate isogeny computation and isogeny evaluation. Fortunately, all of the above five operations can be internally parallelized in 2-way, where the cost[10] switches from $i\boldsymbol{M} + j\boldsymbol{S}$ to $\frac{i}{2}\boldsymbol{M^2} + \frac{j}{2}\boldsymbol{S^2}$.

[10] $\boldsymbol{M}$, $\boldsymbol{S}$, $\boldsymbol{A}$ denote a 1-way field multiplication, squaring, addition/subtraction operation, respectively; $\boldsymbol{M^2}$, $\boldsymbol{S^2}$, $\boldsymbol{A^2}$ denote a 2-way field multiplication, squaring, mixed addition and subtraction operation, respectively.

Algorithm 5.8: 2-way implementation of Elligator 2 map.

Input: The values $(A_0 : A_1) = (a : a - d)$, $u \leftarrow \texttt{random}(\{2, \ldots, (p-1)/2\})$ and Montgomery constant $R = 2^{516} \bmod p$.

Output: A pair of points $P_0 \in E_{a,d}[\pi - 1]$ and $P_1 \in E_{a,d}[\pi + 1]$.

1	$\alpha \leftarrow 0$	
2	$t_0 \leftarrow A_0 - A_1$	
3	$t_0 \leftarrow A_0 + t_0$	
4	$t_0 \leftarrow t_0 + t_0 = A'$	
5	$t_1 \leftarrow u \times R^2$	$s_1 \leftarrow t_0 \times A_1$
6	$t_2 \leftarrow t_1^2$	$s_2 \leftarrow t_0^2$
7	$t_3 \leftarrow t_2 + R$	$s_3 \leftarrow t_2 - R$
8	$t_4 \leftarrow A_1 \times s_3$	$s_4 \leftarrow s_1 \times s_3$
9	$t_5 \leftarrow t_4 \times t_4$	$s_5 \leftarrow s_2 \times t_2$
10	$t_5 \leftarrow s_5 + t_5$	$s_0 \leftarrow \alpha - t_0 = -A'$
11	$t_6 \leftarrow s_4 \times t_5$	$s_6 \leftarrow s_0 \times t_2$
12	$\texttt{cswap}(\alpha, t_1, \texttt{isequal}(t_6, 0))$	/* $\alpha \leftarrow u$ if $t_6 = 0$; else $\alpha \leftarrow 0$ */
13	$t_3 \leftarrow \alpha \times t_3$	$s_3 \leftarrow \alpha \times t_4$
14	$t_5 \leftarrow t_0 + s_3$	$s_5 \leftarrow s_6 - t_3$
15	$t_3 \leftarrow t_3 + t_6$	
16	$m \leftarrow \texttt{issquare}(t_3)$	/* $m \leftarrow 1$ if t_3 is a square in $\mathbb{F}_p$; else $m \leftarrow 0$ */
17	$Y_0 \leftarrow t_5 - t_4$	$T_0 \leftarrow t_5 + t_4$
18	$Y_1 \leftarrow s_5 - t_4$	$T_1 \leftarrow s_5 + t_4$
19	$\texttt{cswap}(Y_0, Y_1, 1 - m)$	$\texttt{cswap}(T_0, T_1, 1 - m)$
20	**return** $P_0 = (Y_0 : T_0)$ and $P_1 = (Y_1 : T_1)$	

Elligator 2. The Elligator 2 map was originally introduced in [BHKL13] for generating random points on Montgomery curves and was modified in [CCC+19] for the twisted Edwards case. The latter takes as input the values $A_0 = a$ and $A_1 = a - d$ where $a, d \in \mathbb{F}_p$ are the coefficients of the curve $E_{a,d}$ in twisted Edwards form, a random $u \in \{2, \ldots, (p-1)/2\}$ which is used to derive the random curve points. Then it outputs two points $P_0 \in E_{a,d}[\pi - 1]$ and $P_1 \in E_{a,d}[\pi + 1]$. The method of [CCC+19] requires $8\boldsymbol{M} + 3\boldsymbol{S} + 16\boldsymbol{A}$ plus one square test for the Legendre symbol, which we slightly improved by saving $2\boldsymbol{A}$. Our 2-way implementation of the Elligator 2 map is based on [CCC+19], and it is presented in Algorithm 5.8, with total cost $5\boldsymbol{M}^2 + 1\boldsymbol{S}^2 + 9\boldsymbol{A}^2$ plus the square test for the Legendre symbol. Specifically, the value u will be used to derive the random curve points, and the Montgomery constant R is used to map the random value u to the Montgomery domain. The algorithm first generates the two points using XZ-coordinate representation, namely $P_0 = (X_0 : Z_0)$ and $P_1 = (X_1 : Z_1)$ on the birationally equivalent Montgomery curve, $C'Y^2Z^2 = C'X^3Z + A'X^2Z^2 + C'XZ^3$, where $A' = 2(a + d)$ and $C' = a - d$. More precisely, the two points are defined as:

$$P_0 = (A' + \alpha C'(u^2 - 1) : C'(u^2 - 1)) \quad \text{and} \quad P_1 = (-A'u^2 - \alpha C'(u^2 - 1) : C'(u^2 - 1)),$$

where $\alpha = 0$, if $A' \neq 0$; and $\alpha = u$, if $A' = 0$[11]. Then, the algorithm converts the two points in twisted Edwards form, using YT-coordinate representation, at line 17 and 18. This is relatively cheap, since it requires only $2\boldsymbol{A}^2$ operations, namely

$$P_0 = (Y_0 : T_0) = (X_0 - Z_0 : X_0 + Z_0) \quad \text{and} \quad P_1 = (Y_1 : T_1) = (X_1 - Z_1 : X_1 + Z_1).$$

[11] Given a point $P = (X : Y : Z)$ on a Montgomery curve in projective form, the XZ-coordinate representation of P is $P = (X : Z)$, where $x = X/Z$ is the x coordinate of P in affine form.

Algorithm 5.9: 2-way implementation for YT-coordinate point doubling.

Input: A point $P = (Y_P : T_P)$ on the curve $E_{a,d}$ and the values $(A_0 : A_1) = (a : a - d)$.
Output: The point $R = [2]P = (Y_R : T_R)$.

1	$t_0 \leftarrow Y_P^2$	$s_0 \leftarrow T_P^2$
2	$t_1 \leftarrow s_0 - t_0$	
3	$t_2 \leftarrow A_0 \times t_1$	$s_2 \leftarrow A_1 \times t_0$
4	$t_3 \leftarrow s_2 + t_2$	
5	$t_0 \leftarrow t_1 \times t_3$	$s_0 \leftarrow s_2 \times s_0$
6	$Y_R \leftarrow s_0 - t_0$	$T_R \leftarrow s_0 + t_0$
7	**return** $R = (Y_R : T_R)$	

Algorithm 5.10: 2-way implementation for YT-coordinate (differential) addition.

Input: Points $P = (Y_P : T_P)$, $Q = (Y_Q : T_Q)$ and $PQ = (Y_{P-Q} : T_{P-Q})$ on $E_{a,d}$.
Output: The point $R = P + Q = (Y_R : T_R)$.

1	$t_0 \leftarrow T_{P-Q} + Y_{P-Q}$	$s_0 \leftarrow T_{P-Q} - Y_{P-Q}$
2	$t_1 \leftarrow Y_P \times T_Q$	$s_1 \leftarrow Y_Q \times T_P$
3	$t_2 \leftarrow t_1 - s_1$	$s_2 \leftarrow t_1 + s_1$
4	$t_2 \leftarrow t_2^2$	$s_2 \leftarrow s_2^2$
5	$t_1 \leftarrow t_0 \times t_2$	$s_1 \leftarrow s_0 \times s_2$
6	$Y_R \leftarrow s_1 - t_1$	$T_R \leftarrow s_1 + t_1$
7	**return** $R = (Y_R : T_R)$	

At line 16, we use the constant time function `issquare` to check whether the value

$$t_3 = \alpha(u^2 + 1) + A'C'(u^2 - 1)((A'u)^2 + (C'(u^2 - 1))^2)$$

is a square in $\mathbb{F}_p$. If it is a square ($m = 1$), then the generated point lies on $E_{a,d}$, otherwise ($m = 0$), the point lies on the quadratic twist of $E_{a,d}$. At the final step (line 19), the two points are swapped according to m, so that the point $P_0 \in E_{a,d}[\pi - 1]$ and $P_1 \in E_{a,d}[\pi + 1]$.

Point doubling. For a point $P = (Y_P : T_P)$ on the curve $E_{a,d}$, the point $R = [2]P$ is defined as:

$$Y_R = eY_P^2T_P^2 - (T_P^2 - Y_P^2)(eY_P^2 + a(T_P^2 - Y_P^2))$$
$$T_R = eY_P^2T_P^2 + (T_P^2 - Y_P^2)(eY_P^2 + a(T_P^2 - Y_P^2)),$$

where $e = a - d$ [CCC+19]. Algorithm 5.9 describes our 2-way doubling process using YT-coordinate arithmetic, with cost $2\boldsymbol{M}^2 + 1\boldsymbol{S}^2 + 3\boldsymbol{A}^2$.

Point addition. For point addition, we use the formulas that are presented in [CCC+19]. These formulas correspond to the differential addition using YT-coordinates on twisted Edwards curves. In particular, let $P = (Y_P : T_P)$ and $Q = (Y_Q : T_Q)$ be two points on the curve and let $PQ = P - Q = (Y_{P-Q} : T_{P-Q})$. The point $R = P + Q$ is derived from the coordinates of the points P, Q and PQ, using the formulas:

$$Y_R = (T_{P-Q} - Y_{P-Q})(Y_PT_Q + Y_QT_P)^2 - (T_{P-Q} + Y_{P-Q})(Y_PT_Q - Y_QT_P)^2$$
$$T_R = (T_{P-Q} - Y_{P-Q})(Y_PT_Q + Y_QT_P)^2 + (T_{P-Q} + Y_{P-Q})(Y_PT_Q - Y_QT_P)^2,$$

Algorithm 5.10 is the 2-way (differential) addition process using YT-coordinate arithmetic with cost $2\boldsymbol{M}^2 + 1\boldsymbol{S}^2 + 3\boldsymbol{A}^2$.

Algorithm 5.11: 2-way ℓ-isogeny computation, with $\ell = 2k + 1$.

Input: Point $P = (Y_P : T_P)$, $(A_0 : A_1) = (a : a - d)$, $\ell = 2k + 1$.
Output: Curve $(A'_0 : A'_1) = (a' : a' - d')$, list $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$.

$(\ell)_2 \leftarrow (b_n, \ldots, b_0)_2$ binary representation of ℓ
$t_1 \leftarrow A_0 - A_1$
$t_0 \leftarrow A_0$, $Y_1 \leftarrow Y_P$, $Y_Q \leftarrow Y_P$ $\quad$ $s_0 \leftarrow t_1$, $T_1 \leftarrow T_P$, $T_Q \leftarrow T_P$
$P_2 \leftarrow [2]P = (Y_2 : T_2)$ $\quad$ /* point doubling */
for i **from** 3 **to** k **by** 1 **do**
  $Y_Q \leftarrow Y_Q \times Y_{i-1}$ $\quad$ $T_Q \leftarrow T_Q \times T_{i-1}$
  $P_i \leftarrow P_{i-1} + P = (Y_i : T_i)$ $\quad$ /* point addition */
end
$t_2 \leftarrow Y_Q \times Y_k$ $\quad$ $s_2 \leftarrow T_Q \times T_k$
$m \leftarrow \texttt{isequal}(\ell, 3)$
$\texttt{cswap}(Y_Q, t_2, 1 - m)$ $\quad$ $\texttt{cswap}(T_Q, s_2, 1 - m)$
for i **from** $n - 1$ **to** 0 **by** 1 **do**
  $t_0 \leftarrow t_0^2$ $\quad$ $s_0 \leftarrow s_0^2$
  if $b_i = 1$ **then**
    $t_0 \leftarrow t_0 \times A_0$ $\quad$ $s_0 \leftarrow s_0 \times t_1$
  end
end
for i **from** 0 **to** 2 **by** 1 **do**
  $Y_Q \leftarrow Y_Q^2$ $\quad$ $T_Q \leftarrow T_Q^2$
end
$A'_0 \leftarrow t_0 \times T_Q$ $\quad$ $A'_1 \leftarrow s_0 \times Y_Q$
$A'_1 \leftarrow A'_0 - A'_1$
return $(A'_0 : A'_1)$, $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$

ℓ-isogeny computation. Algorithm 5.11 describes the procedure for computing an isogeny of some odd degree ℓ, using YT-coordinate representation on twisted Edwards curves. The algorithm takes as input the values $A_0 = a, A_1 = a - d$, where $E_{a,d}$ is an elliptic curve in twisted Edwards form, a point $P = (Y_P : T_P)$ and the degree of the isogeny $\ell = 2k+1$. Then the algorithm computes the codomain curve $E_{a',d'}$ and the list of points $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$, where $P_i = [i]P$, for each $i \in \{1, \ldots, k\}$. Based on the work of Moody and Shumow [MS16], the coefficients of the codomain curve are defined as:

$$a' = a^\ell \left(\prod_{i=1}^{k} T_i \right)^8 \quad \text{and} \quad d' = d^\ell \left(\prod_{i=1}^{k} Y_i \right)^8 .$$

Algorithm 5.11 outputs the values $A'_0 = a'$ and $A'_1 = a' - d'$, as well as the list of points $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$.

ℓ-isogeny evaluation. Algorithm 5.12 computes the image $R = (Y_R : T_R)$ of a point $Q = (Y_Q : T_Q)$ under an isogeny of odd degree $\ell = 2k + 1$, that is computed with Algorithm 5.11. In particular, the algorithm takes as input the point Q and the list of points $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$ that was computed in Algorithm 5.12, where P is the kernel point and $P_i = [i]P$, for each $i \in \{1, \ldots, k\}$. Based

Algorithm 5.12: 2-way ℓ-isogeny evaluation, with $\ell = 2k + 1$.

Input: Point $Q = (Y_Q : T_Q)$ and list $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$.
Output: The point $R = \phi(Q) = (Y_R : T_R)$.

1	$t_0 \leftarrow Y_Q \times T_1$	$s_0 \leftarrow T_Q \times Y_1$
2	$Y_R \leftarrow t_0 + s_0$	$T_R \leftarrow t_0 - s_0$
3	**for** i **from** 2 **to** k **by** 1 **do**	
4	$t_0 \leftarrow Y_Q \times T_i$	$s_0 \leftarrow T_Q \times Y_i$
5	$t_1 \leftarrow t_0 + s_0$	$s_1 \leftarrow t_0 - s_0$
6	$Y_R \leftarrow Y_R \times t_1$	$T_R \leftarrow T_R \times s_1$
7	**end**	
8	$Y_R \leftarrow Y_R^2$	$T_R \leftarrow T_R^2$
9	$t_0 \leftarrow T_Q + Y_Q$	$s_0 \leftarrow T_Q - Y_Q$
10	$t_0 \leftarrow t_0 \times Y_R$	$s_0 \leftarrow s_0 \times T_R$
11	$Y_R \leftarrow t_0 - s_0$	$T_R \leftarrow t_0 + s_0$
12	**return** $R = (Y_R : T_R)$	

on the formulas provided in [CCC+19], the image R of Q has coefficients:

$$Y_R = (T_Q + Y_Q)\left(\prod_{i=1}^{k}(Y_Q T_i + T_Q Y_i)\right)^2 - (T_Q - Y_Q)\left(\prod_{i=1}^{k}(Y_Q T_i - T_Q Y_i)\right)^2$$

$$T_R = (T_Q + Y_Q)\left(\prod_{i=1}^{k}(Y_Q T_i + T_Q Y_i)\right)^2 + (T_Q - Y_Q)\left(\prod_{i=1}^{k}(Y_Q T_i - T_Q Y_i)\right)^2$$

The total cost for our 2-way isogeny evaluation, that is described in Algorithm 5.12 is $2k\boldsymbol{M}^2 + 1\boldsymbol{S}^2 + (k+2)\boldsymbol{A}^2$.

Higher-layer arithmetic. At the top layer, we respectively implemented an OAYT-style group action and a dummy-free-style group action according to [CCC+19].

5.5 Evaluation

We downloaded the original CSIDH software [CLM+18], all the OAYT-style and dummy-free-style constant-time CSIDH software including [OAYT19], [CCC+19], [HLKA20] and [CR22]. All the source codes are publicly available. In particular, the source code of [CLM+18] is available at CSIDH website[12], while the authors of [CCC+19], [HLKA20] and [CR22] provided their source code links in their articles. In addition, although the authors of [OAYT19] did not give the link of their source code in the article, we found the source code repository of the implementation in [OAYT19] on GitHub[13].

In order to figure out the real improvement of our work, we benchmarked our software and the CSIDH group action evaluation of all the above implementations on an Intel Core i3-1005G1 Ice Lake CPU clocked at 1.2 GHz. All source codes were compiled with GCC version 9.3.0 and Turbo boost was disabled during the performance measurements. The results of the OAYT-style implementations are shown in Table 5.3, where the speedup ratio is defined by comparing the "CPU-cycles/#instances" between the baseline and the specific implementation, i.e., the normalized throughput. We use [CCC+19] as baseline, because in this way we know precisely how much our vector processing techniques improve the results (note that [CCC+19] also served as baseline in other papers, e.g., [HLKA20, CR22]).

[12] https://csidh.isogeny.org/software.html.
[13] https://github.com/hiroshi-onuki/constant-time-csidh.

Table 5.3: Benchmark of OAYT-style CSIDH-512 group action implementations on the Ice Lake Core processor.

Implementation		ISA/ISE	Vectorization	#Inst.	CPU-Cycles	Speedup†
	[CLM+18]‡	x64	1-way	1	133.7 M	1.52×
	[OAYT19]	x64	1-way	1	248.4 M	0.82×
	[CCC+19]	x64	1-way	1	203.6 M	1.00×
	[HLKA20]	x64	1-way	1	194.7 M	1.05×
	[CR22]	x64	1-way	1	195.0 M	1.04×
This work	Low-latency	AVX-512F	(2×4)-way	1	232.2 M	0.88×
	Extra-dummy	AVX-512F	(8×1)-way	8	858.0 M	1.90×
	Extra-infinity	AVX-512F	(8×1)-way	8	1003.9 M	1.62×
	Combined	AVX-512F	(8×1)-way	8	850.1 M	1.92×
This work	Low-latency	IFMA	(2×4)-way	1	132.1 M	1.54×
	Extra-dummy	IFMA	(8×1)-way	8	454.1 M	3.59×
	Extra-infinity	IFMA	(8×1)-way	8	550.5 M	2.96×
	Combined	IFMA	(8×1)-way	8	446.9 M	3.64×

† The speedup ratio is calculated with "CPU-cycles/#instances" and uses [CCC+19] as the baseline.
‡ This implementation is not constant-time.

Table 5.4: Benchmark of dummy-free-style CSIDH-512 group action implementations on the Ice Lake Core processor.

Implementation		ISA/ISE	Vectorization	#Inst.	CPU-cycles	Speedup†
	[CCC+19]	x64	1-way	1	433.3 M	1.00×
	[CR22]	x64	1-way	1	394.3 M	1.10×
This work	Low-latency	AVX-512F	(2×4)-way	1	447.0 M	0.97×
	Extra-dummy	AVX-512F	(8×1)-way	8	1811.0 M	1.91×
	Extra-infinity	AVX-512F	(8×1)-way	8	2172.3 M	1.60×
	Combined	AVX-512F	(8×1)-way	8	1801.4 M	1.92×
This work	Low-latency	IFMA	(2×4)-way	1	253.8 M	1.71×
	Extra-dummy	IFMA	(8×1)-way	8	967.0 M	3.58×
	Extra-infinity	IFMA	(8×1)-way	8	1220.5 M	2.84×
	Combined	IFMA	(8×1)-way	8	955.3 M	3.63×

† The speedup ratio is calculated with "CPU-cycles/#instances" and uses [CCC+19] as the baseline.

OATY-style implementation. As shown in Table 5.3, our 2-way low-latency IFMA implementation has roughly the same latency as the original non-constant-time implementation in [CLM+18], and it is about 1.5 times faster than the x64 implementation of [CCC+19]. Our (8×1)-way IFMA implementation, when applied with the combined batching method, takes 446.9 M clock cycles for eight parallel instances, which represents a 3.64 times higher throughput compared to the x64 implementation in [CCC+19]. An analysis of the execution times of our (8×1)-way software shows that all the IFMA implementations are nearly 1.9 times faster than the corresponding AVX-512F implementations, which confirms that the IFMA extension indeed significantly accelerates CSIDH compared to general AVX-512F.

Dummy-free-style implementation. The benchmarking results of dummy-free-style implementations are summarized in Table 5.4. These results show that our proposed batching methods still work efficiently when applied to the dummy-free-style CSIDH group action and can yield an up to 3.63 times higher throughput compared to the x64 implementation in [CCC+19].

Analysis: high-throughput implementation. Though AVX-512 can work on eight 64-bit elements simultaneously with a single instruction, the theoretical maximum speedup factor of an AVX-512 implementation (compared to x64 implementation) of isogeny-based crypto is actually far from eight. The main reason is the multiplier. An x64 implementation executed on an Ice Lake Core CPU has to use a single multiplier sequentially, but this multiplier can execute a full $(64 \times 64 \longrightarrow 128)$-bit multiplication, which is very beneficial for the field arithmetic. On the other hand, AVX-512F can execute eight $(64 \times 64 \longrightarrow 64)$-bit multiplications (`vpmullq`) or eight $(32 \times 32 \longrightarrow 64)$-bit multiplications (`vpmuludq/vpmuldq`) in parallel, whereby the latter is typically used in multi-precision arithmetic. An AVX-512IFMA instruction can perform eight multiplications on 52-bit operands, but the result is either the lower half or the upper half of the eight 104-bit products, i.e., two IFMA instructions are necessary. Taking the multiplication of 512-bit integers using the schoolbook method as example, an x64 implementation needs $8^2 = 64$ mul instructions for one instance, while AVX-512F needs at least $16^2 = 256$ vectorized mul instructions for eight instances (a radix-2^{29} representation would even need more instructions) and AVX-512IFMA requires $10^2 \cdot 2 = 200$ IFMA instructions for eight instances. Since the CPI of these mul instructions is same on Ice Lake Core CPU (see [Int20]), the approximate speed-up (compared to an x64 implementation) of AVX-512F and AVX-512IFMA is a factor of 2.0 and 2.56, respectively. This is also the case for the Montgomery reduction. As we mentioned before in Section 5.3.6, the field multiplication significantly affects the performance of CSIDH so that the theoretical maximum speedup factor of AVX-512 for CSIDH group action evaluation should be far from 8. Taking this analysis into account, our throughput-optimized AVX-512 implementations have the expected speed-ups.

Analysis: low-latency implementation. As for the latency-optimized implementation, a 2-way IFMA latency-optimized implementation of SIKEp503 was presented by Kostic and Gueron in [KG19], which is 1.72 times faster than the x64 assembly implementation of SIKEp503. We can thus conclude that our 2-way IFMA low-latency implementations (which achieve speed-up factors of 1.54 and 1.71, respectively) also correspond to the expected acceleration. There are several reasons that make the 2-way latency-optimized implementation less efficient than the throughput-optimized implementation, including 1) the overheads caused by aligning and blending AVX-512 vectors in 2-way curve and isogeny operations; 2) the fact that some point operations (e.g., y-coordinate doubling and Elligator 2) can not be parallelized in an ideal[14] 2-way fashion due to the dependencies of internal field operations; 3) some computations in the field operations (e.g., the complete carry propagation) cannot be parallelized in an ideal (2×4)-way fashion due to sequential dependencies of instructions; 4) the instruction-level parallelism of (2×4)-way is lower than (2×4)-way since four limbs are stored in one vector. For all these reasons and because of the 32-bit multiplier in AVX-512F, the 2-way AVX-512F implementation is actually slower than the x64 implementation, which is confirmed by our experimental results.

5.6 Conclusion

Summary. Vector engines like Intel's AVX have become steadily more powerful from one generation to the next, not only because of the addition of new functionality, but also through the extension of the supported vector lengths. The expectation of this trend to continue in the coming years makes AVX an important platform for the implementation of PQC, in particular for computation-intensive isogeny-based cryptosystems. Although CSIDH has a couple of highly-desirable and unique features, the massive computational cost of the underlying class group action hampers its deployment in security protocols like TLS. In this work we demonstrated how the enormous parallel processing power of AVX-512 can be exploited to, respectively, maximize the throughput of eight instances and minimize

[14] We define an ideal 2-way parallelized fashion of point or isogeny operation has the cost of $\frac{i}{2}\boldsymbol{M}^2 + \frac{j}{2}\boldsymbol{S}^2 + \frac{k}{2}\boldsymbol{A}^2$ when the corresponding 1-way implementation has the cost of $i\boldsymbol{M} + j\boldsymbol{S} + k\boldsymbol{A}$.

the latency of one instance of CSIDH-512 group action evaluation; the former alleviates the burden of server-side TLS processing, while the latter is beneficial on the client side. Our latency-optimized implementation makes CSIDH-512 group action evaluation roughly 1.5 times faster compared to a state-of-the-art non-vectorized x64 implementation that can resist timing attacks. On the other hand, by developing efficient batching methods for the class group action and combining them with highly-optimized (8×1)-way parallel field arithmetic based on the "limb-slicing" technique, we were able to achieve a 3.6-fold gain in throughput compared to a state-of-the-art x64 implementation of the CSIDH-512 group action evaluation. In light of this significant improvement, we expect that our batching methods are also highly beneficial for optimizing CSIDH-based digital signature schemes, such as CSI-FiSh [BKV19] and SeaSign [DG19], in which multiple independent class group actions are computed in the key generation, signing and verification processes.

Migration to a larger prime field. The correct parameterization of CSIDH (including the order of the underlying prime field) to achieve NIST's security level 1 is currently still a topic of debate. It was suggested that, for level-1 security, the prime p should be much longer than 512 bits, e.g., 4096 bits [CCJR20]. Our CSIDH software was developed in a modular and parameterized way so as to reduce the effort when adapting it for other parameter sets since the point arithmetic (e.g., point addition, point doubling, scalar multiplication) and also certain parts of the field arithmetic can be re-used.

CHAPTER

6

VECTORIZED SIKE

This Chapter is based on our paper [CFGR22]. While we were conducting the research work described in this Chapter, the NIST PQC standardization process was in its third round, and the destructive attacks on SIDH/SIKE, i.e., [CD23, MMP+23, Rob23], had not yet been published.

6.1 Introduction

SIKE. The Supersingular Isogeny Key Encapsulation (SIKE) protocol [JAC+22] is one of the alternate candidates for quantum-safe key encapsulation retained by the NIST. Its main attractions are relatively short secret and public keys, making it somewhat comparable with conventional ("pre-quantum") elliptic-curve key exchange protocols like ECDH and X25519 [HMV04]. Furthermore, since the low-level arithmetic of SIKE is basically long-integer arithmetic, implementers can (potentially) re-use existing hardware accelerators and software libraries for pre-quantum cryptosystems like RSA and ECC. SIKE is based on the Supersingular Isogeny Diffie-Hellman (SIDH) key exchange, which was proposed by Jao and De Feo in 2011 [JD11] as a post-quantum cryptosystem whose security rests on the difficulty of finding isogenies between supersingular curves. In short, SIKE applies a Fujisaki-Okamoto transformation [HHK17] on SIDH to obtain a Key-Encapsulation Mechanism (KEM) that is secure against Chosen Ciphertext Attacks (CCA). State-of-the-art parameter sets for SIKE use supersingular curves over quadratic extension fields of prime characteristic, where the length of the prime is between 434 and 751 bits. The main drawback of SIKE is high computing costs and long latency, caused mainly by the serial computation of these isogenies, which represents a serious bottleneck for practical applications. For example, the currently fastest software implementation of SIKE for the ARM Cortex-M4 platform [AAK21] is more than two orders of magnitude slower than the best lattice-based KEMs benchmarked in [PQM4]. Therefore, optimization techniques to accelerate SIDH and SIKE are an important topic in PQC research.

Performance optimizations for SIKE. An analysis of the academic literature on performance optimizations for SIDH and SIKE shows that past research can be broadly divided into two categories. Research in the first category is concerned with mathematical techniques and higher-level arithmetic optimizations to make the point arithmetic and isogeny computations more efficient, see e.g., [CLN16,

ABJK18, FLOR18, COR22]. The second category covers research on software optimizations for the underlying field arithmetic, whereby the modular reduction received particular attention [SLLH18, BF20, BI21, TWL+22]. Most of the highly-tuned implementations published in the literature adopt the Montgomery modular reduction method [Mon85] since it is extremely efficient in software. The very first implementation of SIDH was introduced roughly 10 years ago [JD11] and uses the GMP library for the low-level arithmetic. Since then, many implementations of SIDH or SIKE with dedicated field-arithmetic functions written in Assembly language have been developed. Microsoft's PQCrypto-SIDH library, which is available on GitHub under MIT license, contains the to-date fastest x64 Assembler implementation of SIKE. This library features most of the improvements and optimizations that were presented in the literature to accelerate the SIKE protocol and make it more practical. However, despite a large body of research on fast software implementation, SIKE is still significantly slower than other post-quantum KEMs, in particular the lattice-based third-round NIST candidates.

Intel x64 architecture and vector extensions. The 64-bit Intel architecture (i.e., x64) serves as the main benchmarking platform to analyze and compare the efficiency of the NIST PQC candidates. Besides the x64 base instruction set, 64-bit Intel processors also support different kinds of vector instructions for a SIMD-parallel execution of workloads. Vector extensions for the Intel architecture have a history that stretches back some 25 years and began with the introduction of the MMX extensions for the 32-bit x86 architecture. Thereafter came numerous generations of Streaming SIMD Extensions (SSE), which support vectors of a length of 128 bits, and *Advanced Vector eXtensions (AVX)*. The most recent new member of the AVX family is AVX-512, which augments the execution environment of x64 by 32 registers of a length of 512 bits and various new instructions. These instructions can operate on e.g., sixteen 32-bit elements or eight 64-bit elements in a SIMD-parallel fashion. AVX-512 comprises a set of core instructions called AVX-512F and multiple extensions that are optional and may be implemented independently. One of these optional extensions provides the so-called "Integer Fused Multiply and Add" (IFMA) instructions, which were designed to speed up big-integer arithmetic [GK16]. The IFMA extension is supported by all mobile and workstation/server processors code-named "Ice Lake" and their successors[1].

Contributions. In this work, we study how the massive parallel processing capabilities of the latest generation of the AVX vector engines, in particular AVX-512IFMA, can be used to improve the efficiency of SIKE-based key encapsulation. Since AVX-512IFMA is a relatively recent extension of the AVX-512 architecture, it is still (widely) unexplored how its new instructions can be used to speed up SIKE. To our knowledge, there exists currently only one publication dealing with AVX-512 optimizations for SIKE, namely the ARITH 2019 paper of Kostic and Gueron [KG19], but their work focuses solely on the low-level field arithmetic, i.e., they did not explore avenues for parallel processing at the higher levels of SIKE. Hence, it is still unknown how AVX-512IFMA can be exploited to unleash the full potential of modern Intel processors for executing SIKE and what latency (resp. throughput) a carefully optimized implementation could achieve.

The present paper aims to fill this gap by introducing novel techniques to parallelize (and speed up) the field arithmetic, point arithmetic, and isogeny computations. At the lowest level, we present a carefully-optimized library for arithmetic operations in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ that uses a radix-2^{51} representation for the operands (i.e., 51 bits/limb) and adopts Montgomery's algorithm for modular reduction. We developed different variants of this arithmetic library, including one that minimizes the latency of two (resp. four) instances of an arithmetic operation, and one that maximizes the throughput of eight

[1] According to Intel there exist currently 13 "Ice Lake" processors for the mobile segment and 43 "Ice Lake" processors for the workstation/server segment, see https://ark.intel.com/content/www/us/en/ark/products/codename/74979/products-formerly-ice-lake.html.

instances using the so-called limb-slicing technique[2]. At the medium level, we describe techniques for parallel point arithmetic operations on Montgomery curves, whereby we paid special attention to find viable trade-offs between the number of parallel instances of point and field operations, respectively. Finally, at the highest layer, we discuss various approaches for vectorized isogeny computation and key encapsulation. All these parallel processing techniques are combined in AvxSike, an optimized implementation of SIKE using Intel's AVX-512IFMA instructions. AvxSike supports all four (uncompressed) parameter sets given in [JAC+22] and comes with a low-latency version and a high-throughput version of SIKE, which we call AvxSike-LL and AvxSike-HT, respectively. Both versions are resistant against timing-based side-channel attacks in the sense that they do not contain any secret-dependent conditional statements or memory accesses. Our latency-optimized AvxSike instantiated with the SIKEp503 parameters is about 1.5 times faster than the AVX-512IFMA-based SIKE software presented in [KG19]. It also outperforms Microsoft's x64 Assembler implementation[3] of SIKE by a factor of about 2.5 for both key generation and decapsulation, and even 3.2 for encapsulation, when benchmarked on an Intel Core i3-1005G1 processor. Furthermore, our throughput-optimized AvxSike reaches an up to 4.6-fold higher throughput than Microsoft's SIKE library.

Source code. The source code of our AvxSike software is available online at https://gitlab.uni.lu/apsia/avxsike. This repository contains both the low-latency version AvxSike-LL and the high-throughput version AvxSike-HT.

Organization. In Section 6.2, we review the SIKE key encapsulation mechanism and describe our target platform (focussing on the AVX-512IFMA vector instructions) as well as the experimental environment for collecting benchmarking results. Then, from Section 6.3 to Section 6.6, we introduce our AvxSike software layer by layer. Section 6.3 explains two different types of vectorized integer multiplication and Montgomery reduction. Various vectorized implementations of quadratic extension-field operations are described in detail in Section 6.4. Later, in Section 6.5, we focus on vectorized implementations of arithmetic operations on Montgomery curves. In Section 6.6, we present a low-latency version and a high-throughput version of AvxSike, our vectorized SIKE software. We compare the performance of AvxSike, Microsoft's SIDHv3.4 assembly library, and the IFMA-based SIKEp503 implementation of Kostic and Gueron in Section 6.7. Finally, in Section 6.8, we draw conclusions and discuss avenues for future work.

6.2 Background

We start with a summary of the mathematical background of isogenies of elliptic curves and proceed with a concise description of the SIKE mechanism [JAC+22]. Later, we give an overview of the AVX-512 instruction set architecture and introduce our experimental environment for performance measurements.

Isogeny. Let E and E' be two elliptic curves over a finite field $\mathbb{F}_q$ of prime characteristic p. An *isogeny* $\phi : E \to E'$ defined over $\mathbb{F}_q$ is a non-constant rational map defined over $\mathbb{F}_q$, which is also a group homomorphism from $E(\mathbb{F}_p)$ to $E'(\mathbb{F}_p)$. And we say E, E' are *isogenous* if and only if $\#E(\mathbb{F}_q) = \#E'(\mathbb{F}_q)$ [Tat66]. In isogeny-based cryptography, we are interested in *separable* isogenies [JD11]. Such isogenies

[2]Limb-slicing uses a "reduced-radix" representation for the operands and is somewhat similar to the bit-slicing technique used in symmetric cryptography, i.e., it allows one to compute a batch of arithmetic operations in a SIMD-parallel way, which increases throughput at the expense of latency [CGT+20].

[3]We used version 3.4 of Microsoft's PQCrypto-SIDH library (i.e., SIDHv3.4), which is available on GitHub at https://github.com/Microsoft/PQCrypto-SIDH, as starting point for our work and the x64 assembly implementation of SIKE contained in this library as baseline for performance comparisons.

have finite kernel and their degree is defined as $\deg \phi = \#\ker \phi$. In addition, given an elliptic curve E over $\mathbb{F}_q$ and a finite subgroup $G \subseteq E(\mathbb{F}_q)$, there exists a unique isogeny $\phi : E \to E' = E/G$, with $\ker(\phi) = G$ and $\deg(\phi) = \#G$. Isogeny-based cryptosystems generally use supersingular elliptic curves of smooth order since they facilitate the computation of isogenies of exponentially large degree by composing lower-degree isogenies that can be efficiently computed with Vélu's formulæ [Vél71]. Furthermore, every supersingular elliptic curve E defined over $\mathbb{F}_q$ can also be defined over $\mathbb{F}_{p^2}$, in which case $\#E(\mathbb{F}_{p^2}) = (p+1)^2$.

Montgomery curve. In the SIKE protocol, supersingular elliptic curves are represented using the Montgomery model [JAC+22]. A Montgomery curve in affine form is given by the equation $E_{(a,b)} : by^2 = x^3 + ax^2 + x$, where $a, b \in \mathbb{F}_{p^2}$ and $b(a^2 - 4) \neq 0$. It is often beneficial to work in the projective form, both in terms of curve points and curve coefficients. In this case, we write $E_{(A:B:C)} : BY^2Z = CX^3 + AX^2Z + CXZ^2$ with $a = A/C$, $b = B/C$, and $(x, y) = (X/Z, Y/Z)$. Montgomery curves are well known for efficient point arithmetic on their Kummer line, originally proposed in [Mon87], which entirely ignores the projective Y coordinate. In addition, they allow one to ignore the coefficient B in point operations and isogeny computations. Consequently, we will denote by $E_{(A:C)}$ a Montgomery curve with $B = 1$ and by $P = (X_P : Z_P)$ a point on the curve.

6.2.1 Supersingular Isogeny Key Encapsulation (SIKE)

SIKE is a key encapsulation mechanism from the family of isogeny-based schemes. It was inspired by the SIDH protocol of Jao and De Feo [JD11] and is currently evaluated as an alternate candidate in the NIST PQC standardization process [AAC+22].

Public parameters. We fix two positive integers e_2 and e_3 such that $p = 2^{e_2}3^{e_3} - 1$ is prime. Primes of this form are Montgomery-friendly, meaning that they allow for some optimizations of the modular arithmetic, see e.g., [CLN16]. We define the two key spaces $\mathcal{K}_2 = \{0, \ldots, 2^{e_2} - 1\}$ and $\mathcal{K}_3 = \{0, \ldots, 3^{e_3} - 1\}$ for sampling secret keys. Further, we also fix a starting supersingular elliptic curve $E_0 : y^2 = x^3 + 6x^2 + x$ over $\mathbb{F}_{p^2}$, where $\#E_0(\mathbb{F}_{p^2}) = (2^{e_2}3^{e_3})^2$, and two bases $\{P_2, Q_2\}$ and $\{P_3, Q_3\}$, which generate the torsion subgroups $E_0[2^{e_2}]$ and $E_0[3^{e_3}]$, respectively. The public parameters consist of the curve E_0 and the 3-tuples $\{x_{P_2}, x_{Q_2}, x_{PQ_2}\}$ and $\{x_{P_3}, x_{Q_3}, x_{PQ_3}\}$, where $x_{PQ_2} = x_{P_2} - x_{Q_2}$ and $x_{PQ_3} = x_{P_3} - x_{Q_3}$[4]. For each $\ell \in \{2, 3\}$, we denote by $(E', \phi_\ell) \leftarrow \texttt{isogeny}_\ell(E, x_{R_\ell})$ the computation of an isogeny $\phi_\ell : E \to E'$ of degree ℓ^{e_ℓ} and $\ker \phi_\ell = \langle x_{R_\ell} \rangle$, where $x_{R_\ell} = x_{P_\ell} + [\mathrm{sk}_\ell] x_{Q_\ell}$. Each secret key sk_ℓ is chosen randomly from $\mathcal{K}_\ell$ and the corresponding public key is obtained as $\mathrm{pk}_\ell = (\phi_\ell(x_{P_m}), \phi_\ell(x_{Q_m}), \phi_\ell(x_{PQ_m}))$, where $m \in \{2, 3\}$ such that $m \neq \ell$.

SIKE and SIPKE. The SIKE submission comes with four parameter sets that provide different levels of security and are named according to the size of the underlying prime p: SIKEp434, SIKEp503, SIKEp610, and SIKEp751. In all versions, the public parameters e_2 and e_3 are chosen so that $2^{e_2} \approx 3^{e_3}$. At the core of SIKE is the Supersingular Isogeny Public Key Encryption scheme (SIPKE) [DJP14], which offers the usual three functions (Gen, Enc, Dec) for key generation, encryption, and decryption (see Algorithm 6.1). Note that the ciphertext consists of two components $c_1 = (\phi_2(x_{P_3}), \phi_2(x_{Q_3}), \phi_2(x_{PQ_3}))$ and $c_2 = h \oplus m$, where h is the hash of $j(E_{32})$, the j-invariant of curve E_{32}. The first component is essential in the decryption process, particularly for the computation of the kernel generator $\phi_2(x_{P_3}) + [\mathrm{sk}_3]\phi_2(x_{Q_3})$, which defines the isogeny $\phi'_3 : E_2 \to E_{23}$. Decryption works because $E_{32} \cong E_{23}$ and hence $j(E_{32}) = j(E_{23})$. Like any other key encapsulation mechanism, SIKE consists of the usual three functions (KeyGen, Encaps, Decaps) for the key generation, encapsulation, and decapsulation, as described in Algorithm 6.2. In the original SIKE submission, the hash function used by both SIPKE and SIKE is actually

[4] The x coordinates of $PQ_2 = P_2 - Q_2$ and $PQ_3 = P_3 - Q_3$ are used in the differential addition.

Algorithm 6.1: Public key encryption: SIPKE = (Gen, Enc, Dec)

function Gen()
  $\text{sk}_3 \leftarrow_\$ \mathcal{K}_3$ /* choose random secret key from $\mathcal{K}_3$ */
  $x_{R_3} \leftarrow x_{P_3} + [\text{sk}_3]x_{Q_3}$ /* construct kernel generator $R_3 \in E_0[3^{e_3}]$ */
  $(\phi_3, E_3) \leftarrow \texttt{isogeny}_3(E_0, x_{R_3})$ /* $\phi_3 : E_0 \to E_3$ with $\deg \phi_3 = 3^{e_3}$, $\ker \phi_3 = \langle x_{R_3} \rangle$ */
  $\text{pk}_3 \leftarrow (\phi_3(x_{P_2}), \phi_3(x_{Q_2}), \phi_3(x_{PQ_2}))$ /* evaluate ϕ_3 at points $P_2, Q_2, PQ_2 \in E_0[2^{e_2}]$ */
  return $(\text{sk}_3, \text{pk}_3)$

function Enc($\text{pk}_3, m \in \mathcal{M}, \text{sk}_2 \in \mathcal{K}_2$) /* message m from message space $\mathcal{M}$ */
  $x_{R_2} \leftarrow x_{P_2} + [\text{sk}_2]x_{Q_2}$ /* construct kernel generator $R_2 \in E_0[2^{e_2}]$ */
  $(\phi_2, E_2) \leftarrow \texttt{isogeny}_2(E_0, x_{R_2})$ /* $\phi_2 : E_0 \to E_2$ with $\deg \phi_2 = 2^{e_2}$, $\ker \phi_2 = \langle x_{R_2} \rangle$ */
  $c_1 \leftarrow (\phi_2(x_{P_3}), \phi_2(x_{Q_3}), \phi_2(x_{PQ_3}))$ /* evaluate ϕ_2 at points $P_3, Q_3, PQ_3 \in E_0[3^{e_3}]$ */
  $x'_{R_2} \leftarrow \phi_3(x_{P_2}) + [\text{sk}_2]\phi_3(x_{Q_2})$ /* construct kernel generator $R'_2 \in E_3[2^{e_2}]$ */
  $(\phi'_2, E_{32}) \leftarrow \texttt{isogeny}_2(E_3, x'_{R_2})$ /* $\phi'_2 : E_3 \to E_{32}$, $\deg \phi'_2 = 2^{e_2}$, $\ker \phi'_2 = \langle x'_{R_2} \rangle$ */
  $h \leftarrow \texttt{SHAKE256}(j(E_{32}))$ /* compute the hash of the j-invariant of E_{32} */
  $c_2 \leftarrow h \oplus m$ /* XOR hash h with message m */
  return (c_1, c_2) /* the ciphertext is the pair (c_1, c_2) */

function Dec($\text{sk}_3, (c_1, c_2)$)
  $x'_{R_3} \leftarrow \phi_2(x_{P_3}) + [\text{sk}_3]\phi_2(x_{Q_3})$ /* construct kernel generator $R'_3 \in E_2[3^{e_3}]$ from c_1 */
  $(\phi'_3, E_{23}) \leftarrow \texttt{isogeny}_3(E_2, x'_{R_3})$ /* $\phi'_3 : E_2 \to E_{23}$, $\deg \phi'_3 = 3^{e_3}$, $\ker \phi'_3 = \langle x'_{R_3} \rangle$ */
  $h \leftarrow \texttt{SHAKE256}(j(E_{23}))$ /* compute the hash of the j-invariant of E_{23} */
  $m \leftarrow h \oplus c_2$ /* XOR hash h with message c_2 */
  return m

Algorithm 6.2: Key encapsulation: SIKE = (KeyGen, Encaps, Decaps)

function KeyGen()
  $(\text{sk}_3, \text{pk}_3) \leftarrow \text{Gen}()$ /* generate key pair with Gen function */
  $s \leftarrow_\$ \{0,1\}^n$ /* generate a secret random bitstring of length n */
  return $(s, \text{sk}_3, \text{pk}_3)$

function Encaps(pk_3)
  $m \leftarrow_\$ \{0,1\}^n$
  $\text{sk}_2 \leftarrow \texttt{SHAKE256}(m \,\|\, \text{pk}_3)$ /* random message and secret key sk_2 */
  $(c_1, c_2) \leftarrow \text{Enc}(\text{pk}_3, m, \text{sk}_2)$ /* encrypt the message m with public key pk_3 */
  $k \leftarrow \texttt{SHAKE256}(m \,\|\, (c_1, c_2))$ /* compute shared key k */
  return $(k, (c_1, c_2))$

function Decaps($s, \text{sk}_3, \text{pk}_3, (c_1, c_2)$)
  $m' \leftarrow \text{Dec}(\text{sk}_3, (c_1, c_2))$ /* decrypt ciphertext to obtain message m' */
  $\text{sk}'_2 \leftarrow \texttt{SHAKE256}(m' \,\|\, \text{pk}_3)$ /* reconstruct secret key sk_2 */
  $x_{R_2} \leftarrow x_{P_2} + [\text{sk}'_2]x_{Q_2}$ /* construct kernel generator $R_2 \in E_0[2^{e_2}]$ */
  $(\phi_2, E_2) \leftarrow \texttt{isogeny}_2(E_0, x_{R_2})$ /* $\phi_2 : E_0 \to E_2$ with $\deg \phi_2 = 2^{e_2}$, $\ker \phi_2 = \langle x_{R_2} \rangle$ */
  $c'_1 \leftarrow (\phi_2(x_{P_3}), \phi_2(x_{Q_3}), \phi_2(x_{PQ_3}))$ /* reconstruct first compoenent of ciphertext */
  if $c'_1 = c_1$ **then**
    $k \leftarrow \texttt{SHAKE256}(m' \,\|\, (c_1, c_2))$ /* compute shared key k */
  else
    $k \leftarrow \texttt{SHAKE256}(s \,\|\, (c_1, c_2))$
  return k

an eXtendable Output Function (XOF), namely `SHAKE256` [Dwo15], which belongs to the SHA-3 family and has been approved by the NIST and other standardization bodies.

Security of SIKE. The security of SIKE relies on the SIDH problem, which is defined as follows: given the curves E_0, E_2, E_3 and points $\phi_2(P_3), \phi_2(Q_3), \phi_3(P_2), \phi_3(Q_2)$, determine the j-invariant of the curve $E_2/\langle \phi_2(P_3) + [\mathrm{sk}_3]\phi_3(Q_3)\rangle$ or $E_3/\langle \phi_3(P_2) + [\mathrm{sk}_2]\phi_3(Q_2)\rangle$. To date, the best classical algorithm for attacking SIKE is due to Galbraith [Gal99] and has a complexity of $O(\sqrt[4]{p})$, while the best quantum attack is Tani's claw finding algorithm [Tan09] with a complexity of $O(\sqrt[6]{p})$. In addition, Proposition 1 in the SIKE specification [JAC+22] proves that the SIPKE scheme described in Algorithm 6.1 is IND-CPA secure in the random oracle model, if the SIDH problem is hard and the SIKE mechanism is also proven to be IND-CCA secure. The parameter sets SIKEp434, SIKEp503, SIKEp610, and SIKEp751 correspond to NIST security level 1, 2, 3, and 5, respectively [JAC+22].

6.2.2 Optimized isogeny computations

Multiplication-oriented approach. The most demanding part of the SIKE protocol is the isogeny computations. Namely, in all three functions the SIKE suite consists of, an isogeny $\phi : E_0 \longrightarrow E_{e_\ell}$ of degree ℓ^{e_ℓ} has to be computed, with kernel generated by a point $R_0 = P_0 + [\mathrm{sk}_\ell]Q_0 \in E[\ell^{e_\ell}]$, where $\ell \in \{2, 3\}$ and $\mathrm{sk}_\ell \in \{0, \ldots, \ell^{e_\ell} - 1\}$. Instead of directly computing this isogeny ϕ, the best practice is to break the computation in smaller parts where we iteratively compute e_ℓ isogenies of degree ℓ using the Vélu formulæ [Vél71] and compose them to obtain the desired ℓ^{e_ℓ}-isogeny. The straightforward approach is to carry out an iterative procedure for $0 \leq i < e_\ell$, whereby in each iteration we fix the kernel point $S_i = [\ell^{\ell_i - i - 1}]R_i$ to be of order ℓ and compute an isogeny $\phi_i : E_i \longrightarrow E_{i+1} = E_i/\langle S_i\rangle$ of degree ℓ. Thereafter, we compute the image $R_{i+1} = \phi_i(R_i)$ in order to be able to obtain the kernel point on the new curve E_{i+1} for the next isogeny. The desired ℓ^{e_ℓ}-isogeny $\phi : E_0 \longrightarrow E_\ell$ is represented as the composition $\phi = \phi_{e_\ell - 1} \circ \ldots \circ \phi_0$. This approach is *multiplication-oriented* since, in each iteration, a scalar multiplication $[\ell^{\ell_i - i - 1}]S_i$ has to be performed [JD11].

Isogeny-oriented approach. An alternative method, known as *isogeny-oriented*, was proposed by Jao and De Feo [JD11]. The aim in this method is to reduce the number of scalar multiplications at the cost of extra isogeny computations. This is done by computing an initial list of points $([\ell^j]R_0)_{j<e_\ell}$ on the starting curve and then, in each iteration for $0 \leq i < e_\ell$, update this list as the image of the points under the isogeny ϕ_i, i.e. $[\ell^j]R_{i+1} = \phi_i([\ell^j]R_i)$, for each $j = i, \ldots, e_\ell - 1$. In [DJP14], De Feo, Jao, and Plût showed that it is possible to speed up the isogeny-oriented approach by introducing the notion of *optimal strategies*, which allow for less scalar multiplications compared to the multiplication-based approach and less isogeny computations compared to the isogeny-based approach. These strategies are presented and implemented in the SIKE specification document [JAC+22] for the small primes $\ell \in \{2, 3\}$. Instead of point additions, the authors of the SIKE specification use only point doubling for the case $\ell = 2$ and point tripling for the case $\ell = 3$. Moreover, in the case $\ell = 2$, the authors compute iteratively isogenies of degree 4 instead of 2.

6.2.3 Target platform

AVX-512IFMA started to become commonly available with the Intel x64 processor family codenamed "Ice Lake" and its successors, e.g., "Tiger Lake", "Rocket Lake", and "Sapphire Rapids". The "Ice Lake" family comprises 10th generation Intel Core mobile and 3rd generation Xeon scalable server processors based on the "Sunny Cove" microarchitecture. We developed our AvxSike software on (and optimized it for) a 10th generation Core, namely the i3-1005G1. On an "Ice Lake" Core CPU, both `vpmadd52luq` and `vpmadd52luq` have a throughput of one instruction/cycle and a latency of four cycles (see details in Table 5.1). AvxSike was written in C and uses compiler intrinsics to perform AVX-512 vector operations.

We compiled the source code of AvxSike and the Microsoft SIDHv3.4 library with GCC version 9.3.0 and measured their execution times on our Core CPU, whereby turbo boost was disabled. However, we could not measure the execution time of the AVX-512IFMA implementation of Kostic and Gueron because the source code is not publicly available. Therefore, we resort to the timings reported in [KG19] and include these in our performance comparisons.

6.3 Implementation: prime-field arithmetic

In this section, we describe vectorized implementations of big-integer multiplication and Montgomery reduction at the $\mathbb{F}_p$-arithmetic layer, which are highly performance-critical operations of our AvxSike software. Note that AvxSike uses only IFMA instructions (i.e., `vpmadd52luq` and `vpmadd52huq`) for all vector-parallel multiplications, i.e. the basic AVX-512F multiply instructions like `vpmuludq` and `vpmuldq` are not executed at all. We adopt the term "$(y \times z)$-way parallelism" to describe an implementation that performs y prime-field (or integer-arithmetic) operations simultaneously, whereby each operation is executed in a z-way parallel fashion and, thus, uses z elements of a vector. For example, Algorithm 2 in [KG19] contains pseudo-code of a (1×8)-way-way integer multiplication for SIKEp503, which means it is a single multiplication of 512-bit integers that uses all eight 64-bit elements of an AVX-512 vector. Due to better instruction-level parallelism and the possibility of taking advantage of Karatsuba's method, the theoretically-optimal (8×1)-way approach is more efficient than other parallel processing techniques such as (4×2)-way, (2×4)-way, and (1×8)-way. Taking into account options for higher-level parallelism offered by the $\mathbb{F}_{p^2}$-arithmetic layer and the curve-arithmetic layer, we found that all prime-field operations can be executed in either an (8×1)-way or a (4×2)-way fashion, i.e., AvxSike always performs eight or four $\mathbb{F}_p$-operations simultaneously.

In this section (and also the four subsequent sections), we focus on SIKEp503 as case study to explain our vectorization techniques since it is the only parameter set that was considered in essentially every previous paper on fast SIKE software, including [KG19].

6.3.1 Radix-2^{51} representation

Due to the 52-bit wide vector multiplier, most AVX-512IFMA implementations, such as [KG19, CFG+21], directly adopt the natural radix-2^{52} (i.e., 52 bits/limb) representation for the operands. However, in this work, we take advantage of a radix-2^{51} representation based on two main considerations. First, although a radix of 2^{52} is a reduced radix with respect to the 64-bit length of an element of an AVX-512 vector (there are still 12 bits of "headroom" for storing carry bits to delay carry propagation), it is saturating for the 52-bit multiplier because all limbs must be reduced to 52 bits before IFMA instructions can be executed on them. This is not ideal for operations like our (8×1)-way parallel version of Karatsuba multiplication [KO63], where the sums of two half-length additions are operands of the last half-length multiplication[5]. In such a situation, a representation based on radix 2^{52} would make it necessary to instantly propagate the carries produced by the two additions, which does not only require extra instructions, but also generates one more limb. In contrast, a radix of 2^{51} allows one to simply keep the carry bits and delay the carry propagation, thereby increasing the length of limbs to 52 bits. A second reason to favor a radix of 2^{51} is the efficient (4×2)-way carry propagation introduced in [OAL18], which we use in our (4×2)-way implementation of the prime-field arithmetic operations. This efficient carry propagation is "incomplete" in the sense that, after the propagation, two limbs are allowed to exceed the nominal limb-length by one bit. But in our case, when using a radix of 2^{52}, it is not possible to tolerate two over-length limbs (i.e., all limbs strictly have to fit into 52 bits, which costs

[5] An $2n$-bit integer multiplication according to Karatsuba's algorithm is computed via the equation $r = a \cdot b = (a_0 + a_1 \cdot 2^n) \cdot (b_0 + b_1 \cdot 2^n) = a_0 b_0 + [(a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1] \cdot 2^n + a_1 b_1 \cdot 2^{2n}$. To obtain $(a_0 + a_1)(b_0 + b_1)$, two n-bit additions have to be carried out before the n-bit multiplication.

some extra instructions). On the other hand, with a radix of 2^{51}, this problem does not arise since we can allow two limbs to have a length of 52 bits, while the other limbs are still 51 bits long. Finally, we remark that using a radix-2^{51} representation does not increase the number of limbs (in relation to a radix of 2^{52}) for any of the four parameter sets of SIKE.

(8×1)-way limb vector set. The main data structure of our (8×1)-way prime-field operations is the (8×1)-way *limb vector set* composed of eight radix-2^{51} integers. Given eight integers $a, b, c, d, e, f, g, h \in \mathbb{F}_p$, an (8×1)-way limb vector set $\boldsymbol{U}$ for SIKEp503 is defined as:

$$\boldsymbol{U} = \langle a, b, c, d, e, f, g, h\rangle = \left\{ \begin{array}{c} [a_0, b_0, c_0, d_0, e_0, f_0, g_0, h_0] \\ [a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1] \\ \vdots \\ [a_9, b_9, c_9, d_9, e_9, f_9, g_9, h_9] \end{array} \right\} = (U_0, U_1, \ldots, U_9)$$

where each $U_i = [a_i, b_i, c_i, d_i, e_i, f_i, g_i, h_i]$ is called a *limb vector*.

(4×2)-way limb vector set. Our (4×2)-way field-operations use (4×2)-way limb vector sets, which also based on the radix 2^{51}, but contain only four integers. In the case of SIKEp503, a (4×2)-way limb vector set $\boldsymbol{V} = \langle a, b, c, d\rangle$ has the following form:

$$\boldsymbol{V} = \langle a, b, c, d\rangle = \left\{ \begin{array}{c} [a_0, a_5, b_0, b_5, c_0, c_5, d_0, d_5] \\ [a_1, a_6, b_1, b_6, c_1, c_6, d_1, d_6] \\ \vdots \\ [a_4, a_9, b_4, b_9, c_4, c_9, d_4, d_9] \end{array} \right\} = (V_0, V_1, \ldots, V_4)$$

Each limb vector $V_i = [a_i, a_{i+5}, b_i, b_{i+5}, c_i, c_{i+5}, d_i, d_{i+5}]$ contains two 51-bit limbs from each integer, whereby the limbs are arranged in an interleaved pattern.

Reduction modulo $2p$. Similar to SIDHv3.4, both our (8×1)-way and (4×2)-way implementation omit the final subtraction in the Montgomery reduction. All prime-field operations of AvxSike actually perform the reduction modulo $2p$ instead of p. To give a concrete example, the modular addition operation first computes $t \leftarrow a + b$ and then performs a subtraction $r \leftarrow t - 2p$. If $r < 0$ we add $2p$ to r, otherwise we add 0 (to have operand-independent execution time). The modular subtraction just directly computes $r \leftarrow a - b$ and then executes the same correction step as the modular addition.

6.3.2 Integer multiplication

(8×1)-way implementation. All (8×1)-way prime-field arithmetic functions operate on (8×1)-way limb vector sets and were developed based on the "limb-slicing" approach [CGT+20], which essentially duplicates a 1-way implementation to eight 64-bit elements using AVX-512 instructions. The multiplication consists of one level of Karatsuba with product scanning underneath because, according to our experiments, this combination is faster than other techniques (e.g., basic operand scanning and product scanning) for all four parameter sets of SIKE. Note that our implementation of the integer multiplication does not involve a carry propagation, which reduces sequential dependencies among the instructions (though carries are always propagated during Montgomery reduction).

(4×2)-way Implementation. Orisaka, Aranha, and López introduced in [OAL18] an efficient (4×2)-way AVX-512F implementation of Montgomery multiplication, which is composed of integer multiplication, Montgomery reduction, and carry propagation. This implementation operates on the (4×2)-way

Table 6.1: Experimental results of $\mathbb{F}_p$-arithmetic operations for SIKEp503.

Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
Integer multiplication	SIDHv3.4	x64 asm	1-way	1	100	100	1.00×
	AvxSike	AVX-512	(8×1)-way	8	165	21	4.85×
	AvxSike	AVX-512	(4×2)-way	4	102	26	3.92×
Montgomery reduction	SIDHv3.4	x64 asm	1-way	1	75	75	1.00×
	AvxSike	AVX-512	(8×1)-way	8	144	18	4.17×
	AvxSike	AVX-512	(4×2)-way	4	140	35	2.14×
Montgomery multiplication	SIDHv3.4	x64 asm	1-way	1	201	201	1.00×
	[KG19]	AVX-512	hybrid	1	195	195	1.03×
	AvxSike	AVX-512	(8×1)-way	8	302	38	5.32×
	AvxSike	AVX-512	(4×2)-way	4	264	66	3.05×

limb vector sets we defined in the previous subsection and uses the normal `vpmuldq` multiply instruction of AVX-512F but not the two IFMA instructions. The integer multiplication part of this implementation is based on the operand-scanning method. We developed our (4×2)-way integer multiplication following the approach of [OAL18] but replaced `vpmuldq` by IFMA instructions.

6.3.3 Montgomery reduction

(8×1)-way implementation. SIKE uses so-called "SIDH-friendly" primes, which are a special form of Montgomery-friendly primes and allow one to speed up the modular reduction operation compared to general primes [CLN16]. The optimized Montgomery reduction for a SIDH-friendly prime $p = 2^{e_2}3^{e_3} - 1$ is given in [CLN16, Algorithm 1] and exploits the fact that the e_2 least significant bits of $p + 1$ are all 0 (i.e., $p + 1$ is nothing else than 3^{e_3} left-shifted by e_2 bits). In other words, $p + 1$ consists of many limbs that are 0 and this makes it possible to save a large number of instructions compared to a conventional Montgomery reduction. As already mentioned above, the reduction operation involves a carry propagation to get a final result represented by 51-bit limbs.

(4×2)-way implementation. The modular reduction part of the implementation from [OAL18] is a (4×2)-way version of conventional Montgomery reduction. We optimized this implementation by applying the approach of [CLN16] and using IFMA instructions to obtain our (4×2)-way Montgomery reduction. In addition, a final carry propagation is always performed after the Montgomery reduction to get a final result whose limbs are sufficiently short. As already mentioned earlier, there is one limb vector (containing two limbs of each field-element) in this final result that is one bit longer than the other limb vectors, i.e., in our case 52 bits instead of 51 (see [OAL18, Algorithm 4] for details).

6.3.4 Results and comparison

Table 6.1 shows the execution times of integer multiplication, Montgomery reduction, and Montgomery multiplication of different implementations. As mentioned in Section 6.1, we use the SIDHv3.4 x64 assembly library as baseline for comparisons. Since our software is vectorized, the "cycles/instance" is a useful metric for us, and the speed-up ratio relates a specific implementation with the baseline under this metric. The (8×1)-way and the (4×2)-way parallel integer multiplication is respectively 4.9 and 3.9 times faster than SIDHv3.4. Regarding our parallel Montgomery reduction, the (8×1)-way version has almost the same latency as the (4×2)-way implementation, which means it is twice as fast from the viewpoint of a single instance. This massive difference of speed-up factors (in relation to the integer multiplication) can be explained with sequential dependencies in the (4×2)-way Montgomery reduction and, as a consequence, lower instruction-level parallelism compared to the (8×1)-way parallel

Algorithm 6.3: $\mathbb{F}_{p^2}$-multiplication with 1-way parallelization at $\mathbb{F}_p$-level.

Input: $\mathbb{F}_{p^2}$ elements $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$.
Output: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 b_0 - a_1 b_1) + [(a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1]i$.

1 $t_1 \leftarrow a_0 + a_1$
2 $t_2 \leftarrow b_0 + b_1$
3 $tt_1 \leftarrow a_0 \times b_0$
4 $tt_2 \leftarrow a_1 \times b_1$
5 $tt_3 \leftarrow t_1 \times t_2$
6 $tt_3 \leftarrow tt_3 - tt_1 - tt_2$
7 $tt_1 \leftarrow tt_1 - tt_2$
8 $r_0 \leftarrow tt_1 \bmod p$
9 $r_1 \leftarrow tt_3 \bmod p$
10 **return** $r = r_0 + r_1 i$

version. Finally, when looking at the results for the Montgomery multiplication, it is striking that the IFMA implementation of Kostic and Gueron [KG19] is merely a few cycles faster than SIDHv3.4. According to [KG19], their Montgomery multiplication combines a (1×8)-way integer multiplication using IFMA with an x64 assembly implementation of Montgomery reduction. Thus, it is necessary to convert between radix-2^{52} vectors and radix-2^{64} large integers, which is an obvious bottleneck of their software. Since the multiplication is vectorized, but not the reduction, we can say that their software follows a *hybrid* implementation approach.

6.4 Implementation: quadratic extension-field arithmetic

We first define an "$(x \times y \times z)$-way" parallel $\mathbb{F}_{p^2}$-arithmetic implementation: it performs x $\mathbb{F}_{p^2}$-operations in parallel, whereby each of them executes y prime-field operations in parallel, and each of the y prime-field operation uses z 64-bit elements of a vector. In all algorithms of this section, the variable t denotes an integer having a similar length as an element of $\mathbb{F}_p$, whereas tt usually represents an integer of roughly twice the length of an $\mathbb{F}_p$-element; mod p stands for a Montgomery reduction modulo p, but as mentioned in Section 6.3.1, the result is in the range of $[0, 2p - 1]$ and not always fully reduced; $\times$ denotes an *integer* multiplication and + is an *integer* addition (except of Algorithm 6.5, where it is a *modular* addition). Note that, for reasons of brevity and to have succinct pseudo-code descriptions of algorithms, we do not distinguish between various (*integer* and *modular*) subtractions, which are uniformly denoted as $-$, but the result of a subtraction is always non-negative. We refer readers who are interested in the full details of the algorithms to the source code of Microsoft's SIDHv3.4 or our AvxSike.

6.4.1 $\mathbb{F}_{p^2}$-multiplication

1-way parallelization at $\mathbb{F}_p$-level. In this implementation of a parallel $\mathbb{F}_{p^2}$-multiplication, each $\mathbb{F}_{p^2}$-multiplication instance performs only one integer-arithmetic or prime-field operation at a time. Formally, based on the above-defined $(x \times y \times z)$-way notation, we have $y = 1$. For such an $(x \times 1 \times z)$-way $\mathbb{F}_{p^2}$-multiplication, we use the same technique as the SIDHv3.4 library, namely Karatsuba's method. We explain this variant with a single $\mathbb{F}_{p^2}$-multiplication instance in Algorithm 6.3. Unlike to SIDHv3.4, which can perform the carry propagation through instructions like `ADC`, `ADCX`, and `ADOX`, our AVX-512 software has to carefully handle the carry propagation. Since carry propagation normally causes strong instruction dependencies, it always requires more clock cycles than a basic limb addition or subtraction. Algorithm 6.3 was designed to perform as few carry propagations as possible; the integer multiplication does not involve a propagation of carries and the subtractions at line 6 and 7 can "postpone" the carry propagations and integrate them into the subsequent Montgomery reductions. Using the (8×1)-way $\mathbb{F}_p$-arithmetic, we developed an $(8 \times 1 \times 1)$-way $\mathbb{F}_{p^2}$-multiplication via Algorithm 6.3, while the $(4 \times 1 \times 2)$-way implementation is based on the (4×2)-way $\mathbb{F}_p$-arithmetic.

Algorithm 6.4: $\mathbb{F}_{p^2}$-multiplication with 2-way parallelization at $\mathbb{F}_p$-level.

Input: $\mathbb{F}_{p^2}$ elements $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$.
Output: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0)i$.

1 $tt_1 \leftarrow a_0 \times b_0$ $\quad$ $ss_1 \leftarrow a_0 \times b_1$
2 $tt_2 \leftarrow a_1 \times b_0$ $\quad$ $ss_2 \leftarrow a_1 \times b_1$
3 $tt_3 \leftarrow tt_1 - ss_2$ $\quad$ $ss_3 \leftarrow tt_2 + ss_1$
4 $r_0 \leftarrow tt_3 \bmod p$ $\quad$ $r_1 \leftarrow ss_3 \bmod p$
5 **return** $r = r_0 + r_1 i$

Algorithm 6.5: $\mathbb{F}_{p^2}$-multiplication with 4-way parallelization at $\mathbb{F}_p$-level.

Input: $\mathbb{F}_{p^2}$ elements $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$.
Output: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0)i$.

1 $tt_1 \leftarrow a_0 \times b_0$ $\quad$ $ss_1 \leftarrow a_0 \times b_1$ $\quad$ $tt_2 \leftarrow a_1 \times b_0$ $\quad$ $ss_2 \leftarrow a_1 \times b_1$
2 $t_1 \leftarrow tt_1 \bmod p$ $\quad$ $s_1 \leftarrow ss_1 \bmod p$ $\quad$ $t_2 \leftarrow tt_2 \bmod p$ $\quad$ $s_2 \leftarrow ss_2 \bmod p$
3 $r_0 \leftarrow t_1 - s_2$ $\quad$ $r_1 \leftarrow t_2 + s_1$
4 **return** $r = r_0 + r_1 i$

2-way parallelization at $\mathbb{F}_p$-level. In this variant (Algorithm 6.4), each $\mathbb{F}_{p^2}$-multiplication instance is internally parallelized in a 2-way fashion, namely each instance executes two prime-field (or integer-arithmetic) operations simultaneously, i.e., $y = 2$ according to the $(x \times y \times z)$-way notation from above. This variant uses the schoolbook method instead of Karatsuba's algorithm because it turned out that the latter is less efficient for 2-way parallelization at the $\mathbb{F}_p$-level. Compared to the $(x \times 1 \times z)$-way $\mathbb{F}_{p^2}$-multiplication, this 2-way variant has fewer additions, subtractions, and carry propagations, but needs one more multiplication at the $\mathbb{F}_p$-level. Using (8×1)-way and (4×2)-way $\mathbb{F}_p$ arithmetic, we developed a $(4 \times 2 \times 1)$-way and a $(2 \times 2 \times 2)$-way $\mathbb{F}_{p^2}$-multiplication, respectively. In terms of a parallel $\mathbb{F}_{p^2}$-multiplication performing four instances, there are currently the $(4 \times 2 \times 1)$-way and $(4 \times 1 \times 2)$-way options. Note that, although the former option does not use Karatsuba at the $\mathbb{F}_{p^2}$-layer, its underlying (8×1)-way integer multiplication is implemented with Karatsuba's algorithm. On the other hand, the $(4 \times 1 \times 2)$-way option uses Karatsuba at the $\mathbb{F}_{p^2}$-layer, whereas the (4×2)-way integer multiplication is simply a vectorized schoolbook multiplication.

4-way parallelization at $\mathbb{F}_p$-level. Algorithm 6.5 illustrates our 4-way $\mathbb{F}_{p^2}$-multiplication variant. Since the $\mathbb{F}_{p^2}$-multiplication using schoolbook involves four multiplications, it is possible to execute them all in parallel. Performing the Montgomery reductions before the operations at line 3 halves the length of the operands, which means the addition and subtraction at line 3 are single-length operations and can use the fast reduction modulo $2p$ we briefly described in Section 6.3.1. For a parallel $\mathbb{F}_{p^2}$-multiplication performing two instances, this 4-way variant could be used to develop a $(2 \times 4 \times 1)$-way implementation based on the optimal (8×1)-way prime-field operations.

6.4.2 $\mathbb{F}_{p^2}$-Squaring

A conventional squaring operation in $\mathbb{F}_{p^2}$ with the operand $a = a_0 + a_1 i$ is computed as $r = a^2 = (a_0^2 - a_1^2) + 2a_0 a_1 i = r_0 + r_1 i$, which means two conventional integer squarings and an integer multiplication are required. A classic optimization, which is also used in SIDHv3.4, is to replace $a_0^2 - a_1^2$ by $(a_0 + a_1)(a_0 - a_1)$, i.e., two squaring operations are substituted by one multiplication. Since the conventional $\mathbb{F}_{p^2}$-squaring consists of three integer multiplication (or squaring) operations, and the optimized version still involves two integer multiplications, the 4-way variant for $\mathbb{F}_{p^2}$-squaring is not efficient. Thus, we only present a

Algorithm 6.6: $\mathbb{F}_{p^2}$-squaring with 1-way parallelization at $\mathbb{F}_p$-level.

Input: $\mathbb{F}_{p^2}$ element $a = a_0 + a_1 i$.
Output: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 + a_1)(a_0 - a_1) + 2a_0 a_1 i$.

1 $t_1 \leftarrow a_0 + a_1$
2 $t_2 \leftarrow a_0 - a_1$
3 $t_3 \leftarrow a_0 + a_0$
4 $tt_1 \leftarrow t_1 \times t_2$
5 $tt_2 \leftarrow t_3 \times a_1$
6 $r_0 \leftarrow tt_1 \bmod p$
7 $r_1 \leftarrow tt_2 \bmod p$
8 **return** $r = r_0 + r_1 i$

Algorithm 6.7: $\mathbb{F}_{p^2}$-squaring with 2-way parallelization at $\mathbb{F}_p$-level.

Input: $\mathbb{F}_{p^2}$ element $a = a_0 + a_1 i$.
Output: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 + a_1)(a_0 - a_1) + 2a_0 a_1 i$.

1 $t_1 \leftarrow a_0 + a_1$ | $s_1 \leftarrow a_0 + a_0$
2 $t_2 \leftarrow a_0 - a_1$
3 $tt_1 \leftarrow t_1 \times t_2$ | $ss_1 \leftarrow s_1 \times a_1$
4 $r_0 \leftarrow tt_1 \bmod p$ | $r_1 \leftarrow ss_1 \bmod p$
5 **return** $r = r_0 + r_1 i$

1-way and a 2-way variant for $\mathbb{F}_{p^2}$-squaring, both of which take advantage of the optimization described above.

1-way parallelization at $\mathbb{F}_p$-level. The $(x \times 1 \times z)$-way parallel $\mathbb{F}_{p^2}$ squaring is implemented according to Algorithm 6.6. Using (8×1)-way and (4×2)-way $\mathbb{F}_p$-arithmetic, we developed a $(8 \times 1 \times 1)$-way and a $(4 \times 1 \times 2)$-way version of $\mathbb{F}_{p^2}$-squaring, respectively.

2-way parallelization at $\mathbb{F}_p$-level. The 2-way variant for squaring in $\mathbb{F}_{p^2}$ is specified in Algorithm 6.7, whereby a "perfect" parallelization is not possible at line 2. Based on this algorithm, we developed $(4 \times 2 \times 1)$-way and $(2 \times 2 \times 2)$-way $\mathbb{F}_{p^2}$-squaring with our two different implementations of the prime-field operations.

6.4.3 $\mathbb{F}_{p^2}$-addition and subtraction

A vectorized implementation of $\mathbb{F}_{p^2}$-addition/subtraction is fairly straightforward. The addition $r = a + b = (a_0 + a_1 i) + (b_0 + b_1 i) = (a_0 + b_0) + (a_1 + b_1)i = r_0 + r_1 i$ in $\mathbb{F}_{p^2}$ is composed of two additions in $\mathbb{F}_p$. Our $(x \times 2 \times z)$-way implementation executes just the two additions in parallel, while the $(x \times 1 \times z)$-way version performs them sequentially one after the other. The $\mathbb{F}_{p^2}$-subtraction is vectorized in the same way.

6.4.4 Results and comparison

The execution times of $\mathbb{F}_{p^2}$-multiplication and $\mathbb{F}_{p^2}$-squaring for SIKEp503 are shown in Table 6.2. We used our "Ice Lake" CPU to measure the execution times of AvxSike and the Microsoft SIDHv3.4 x64 assembly library, while the timings of Kostic and Gueron's IFMA implementation were taken from [KG19]. As explained in [KG19], they used the schoolbook method instead of Karatsuba's algorithm to develop their $\mathbb{F}_{p^2}$-multiplication and $\mathbb{F}_{p^2}$-squaring in order to mitigate the overhead caused by conversions between the radix-2^{52} vector representation and the radix-2^{64} big-integer representation. The results in Table 6.2 show that our vectorized implementations are more efficient than [KG19] and SIDHv3.4 when considering the "cycles/instance" metric. Furthermore, according to the measured timings, the $(4 \times 2 \times 1)$-way version is faster than the $(4 \times 1 \times 2)$-way version for both $\mathbb{F}_{p^2}$-multiplication and

Table 6.2: Experimental results of $\mathbb{F}_{p^2}$-arithmetic implementations for SIKEp503.

Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
$\mathbb{F}_{p^2}$ Multiplication	SIDHv3.4	x64 asm	1-way	1	503	503	1.00×
	[KG19]	AVX-512	hybrid	1	282	282	1.78×
	AvxSike	AVX-512	$(8 \times 1 \times 1)$-way	8	900	113	4.47×
	AvxSike	AVX-512	$(4 \times 2 \times 1)$-way	4	570	143	3.53×
	AvxSike	AVX-512	$(4 \times 1 \times 2)$-way	4	684	171	2.94×
	AvxSike	AVX-512	$(2 \times 2 \times 2)$-way	2	439	220	2.29×
	AvxSike	AVX-512	$(2 \times 4 \times 1)$-way	2	395	198	2.55×
$\mathbb{F}_{p^2}$ Squaring	SIDHv3.4	x64 asm	1-way	1	427	427	1.00×
	[KG19]	AVX-512	hybrid	1	287	287	1.49×
	AvxSike	AVX-512	$(8 \times 1 \times 1)$-way	8	666	83	5.13×
	AvxSike	AVX-512	$(4 \times 2 \times 1)$-way	4	380	95	4.49×
	AvxSike	AVX-512	$(4 \times 1 \times 2)$-way	4	576	144	2.97×
	AvxSike	AVX-512	$(2 \times 2 \times 2)$-way	2	307	168	2.78×

squaring, which means vectorization at the $\mathbb{F}_p$-layer has more impact on the performance than vectorization at the $\mathbb{F}_{p^2}$-layer. Furthermore, the $(2 \times 4 \times 1)$-way $\mathbb{F}_{p^2}$-multiplication requires fewer cycles than the $(2 \times 2 \times 2)$-way version because (8×1)-way integer multiplication is much more efficient than (4×2)-way, while the two vectorized reduction variants have similar latency (see Table 6.1). However, there is no $(2 \times 4 \times 1)$-way $\mathbb{F}_{p^2}$-squaring, which means we have to use the $(2 \times 4 \times 1)$-way $\mathbb{F}_{p^2}$-multiplication also for squaring. As a result, when implementing curve arithmetic with $(2 \times y \times z)$-way $\mathbb{F}_{p^2}$-operations, the $(2 \times 4 \times 1)$-way version has faster $\mathbb{F}_{p^2}$-multiplication (by 22 cycles) but slower $\mathbb{F}_{p^2}$-squaring (by 30 cycles) than the $(2 \times 2 \times 2)$-way version. In light of these results, we can not decide yet whether $(2 \times 4 \times 1)$-way or $(2 \times 2 \times 2)$-way is the more efficient option for the higher layers (see Section 6.5 for further discussions).

6.5 Implementation: Montgomery elliptic curve arithmetic

In this section, we define a "$(w \times x \times y \times z)$-way" parallel Montgomery-curve arithmetic implementation: it performs w curve operations (e.g., point doublings, point triplings) in parallel, whereby each of them executes x $\mathbb{F}_{p^2}$-operations simultaneously, and each of the $\mathbb{F}_{p^2}$-operations performs y prime-field operations in parallel, and each of the prime-field operations uses z 64-bit elements of a vector. Further, given an elliptic curve $E_{(A:C)}$ in Montgomery form, we define the three constants $A_{24}^{+} = A + 2C$, $A_{24}^{-} = A - 2C$, as well as $C_{24} = 4C$, which are used in the isogeny computations and the point arithmetic.

6.5.1 Three-point ladder

SIKE uses the *three-point ladder* algorithm from [FLOR18] as standard way to compute the kernel generator $R \leftarrow P + [k]Q$. For each bit of the scalar k, this algorithm performs a so-called *Montgomery ladder step* (`xDBLADD`), which essentially consists of a differential point addition and a point doubling; both operations are carried out using (projective) X and Z coordinates only, i.e., the Y coordinate is not needed. The ladder step executes a fixed operation (resp. instruction) sequence, which means the three-point ladder has constant run-time. Various papers in the literature describe vectorized implementations of the Montgomery ladder-step, focusing particularly on reducing the latency of X25519 key exchange [Ber06]. For example, [Cho15, FL15, FLD19] discuss how to vectorize the Montgomery ladder-step in a 2-way fashion, while 4-way parallel implementations were presented in [CS09, HEY20, NS21]. The benchmarking results in [NS21, Table 2] and in [HEY20, Table 1] clearly indicate that the 4-way vectorized Montgomery ladder is more efficient than the 2-way variant on both AVX2 and AVX-512, and

hence we decided to focus on the vectorization of the ladder-step for AvxSike-LL in 4-way fashion, or more precisely, by adopting $(1 \times 4 \times y \times z)$-way parallelization. In addition, Table 6.2 suggests that the $(4 \times 2 \times 1)$-way $\mathbb{F}_{p^2}$-operations are faster than the $(4 \times 1 \times 2)$-way versions, and so we finally chose the $(1 \times 4 \times 2 \times 1)$-way implementation for the ladder-step.

In [NS21], Nath and Sarkar analyze in detail four different methods to vectorize the Montgomery ladder step in 4-way fashion (one from [CS09], one from [HEY20], and two developed by themselves) and compared in [NS21, Table 1] the operations that all these methods have to perform. The methods presented in [CS09] and in [NS21] require three 4-way vectorized field-multiplications (resp. field-squarings), plus a special multiplication by a small constant in each step of the ladder. This special multiplication computes the product of a field-element and the coefficient of Curve25519, which is much faster than a conventional field multiplication. However, for some scalar multiplications of SIKE, the coefficient of the Montgomery curve is not a small constant but an element of $\mathbb{F}_{p^2}$. As a consequence, the vectorization of [CS09] and [NS21] will, in the case of SIKE, require four vectorized $\mathbb{F}_{p^2}$-multiplication (or $\mathbb{F}_{p^2}$-squaring) operations. On the other hand, the vectorization technique presented by Hisil, Egrice, and Yassi in [HEY20] needs only two vectorized field multiplications and one vectorized field squaring in each ladder step (no special multiplication by a small curve-constant is carried out). Thus, the vectorization proposed in [HEY20] is the better choice for a low-latency SIKE implementation.

Based on the 4-way vectorization from [HEY20] and our $(4 \times 2 \times 1)$-way $\mathbb{F}_{p^2}$-operations, we developed a $(1{\times}4{\times}2{\times}1)$-way parallel ladder step for the latency-optimized AvxSike-LL. In addition, to speed up the SIPKE encryption operation (Algorithm 6.15) of AvxSike-LL, we implemented a $(2 \times 4 \times 1 \times 1)$-way ladder step for two simultaneous scalar multiplications, which uses the efficient $(8 \times 1 \times 1)$-way $\mathbb{F}_{p^2}$-operations. For the throughput-oriented AvxSike-HT software, we developed an $(8{\times}1{\times}1{\times}1)$-way parallel ladder step according to the batched X25519 implementation described in [CFG+21].

6.5.2 Point doubling and tripling

Given a point $P = (X_P : Z_P)$ on the Montgomery curve $E_{(A:C)}$, we define the double $[2]P = (X_{[2]P} : Z_{[2]P})$ as:

$$X_{[2]P} = C_{24}(X_P^2 - Z_P^2)^2 \quad \text{and} \quad Z_{[2]P} = 4X_PZ_P\left(4A_{24}^+X_PZ_P + C_{24}(X_P - Z_P)^2\right).$$

Algorithm 6.8 shows our 2-way parallel implementation of the doubling operation. For the tripling, we took advantage of a new formula that uses the value C_{24} instead of A_{24}^- as in the original xTPL function from the SIKE specification (Algorithm 6 in [JAC+22]) since the latter is less efficient for 2-way vectorization. Given $P = (X_P : Z_P)$ on the curve $E_{(A:C)}$, we compute the point $[3]P = (X_{[3]P} : Z_{[3]P})$ via the formulae:

$$X_{[3]P} = X_P\left[C_{24}\left(X_P^2 - Z_P^2\right)^2 - 4Z_P^2\left(4A_{24}^+X_PZ_P + C_{24}(X_P - Z_P)^2\right)\right]^2$$
$$Z_{[3]P} = Z_P\left[C_{24}\left(X_P^2 - Z_P^2\right)^2 - 4X_P^2\left(4A_{24}^+X_PZ_P + C_{24}(X_P - Z_P)^2\right)\right]^2.$$

The 2-way implementation of our point-tripling function is described in Algorithm 6.9.

6.5.3 Isogeny generation

Recall that in SIKE the 2^{e_2}-isogeny and the 3^{e_3}-isogeny are computed as a composition of 4-isogenies and 3-isogenies, respectively. Given a point $R_4 = (X_4 : Z_4)$ of order 4 on $E_{(A:C)}$, a 4-isogeny $\phi_4 : E_{(A:C)} \to E_{(A':C')}$ is constructed with $\ker \phi_4 = \langle R_4 \rangle$. Algorithm 6.10 describes our 2-way implementation of such a 4-isogeny computation. This algorithm outputs the two parameters $A_{24}^+ = A' + 2C'$ and $C_{24} = 4C'$ that define the target curve $E_{(A':C')}$ (where $A_{24}^+ = 4X_4^4$, $C_{24} = 4Z_4^4$) and the values $K_0 = 4Z_4^2$, $K_1 = X_4 - Z_4$, and $K_2 = X_4 + Z_4$, which are used when evaluating the 4-isogeny ϕ_4 at a point. When the point

Algorithm 6.8: XZ-coordinate point doubling with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P = (X_P : Z_P)$, $(A_{24}^+ : C_{24})$.
Output: $Q = [2]P = (X_Q : Z_Q)$.

1	$t_0 \leftarrow X_P + Z_P$	$s_0 \leftarrow X_P - Z_P$
2	$t_1 \leftarrow t_0^2$	$s_1 \leftarrow s_0^2$
3	$t_0 \leftarrow t_1 - s_1$	
4	$t_2 \leftarrow C_{24} \times s_1$	$s_2 \leftarrow A_{24}^+ \times t_0$
5	$t_3 \leftarrow t_2 + s_2$	
6	$X_Q \leftarrow t_2 \times t_1$	$Z_Q \leftarrow t_3 \times t_0$
7	**return** $Q = (X_Q : Z_Q)$	

Algorithm 6.9: XZ-coordinate point tripling with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P = (X_P : Z_P)$, $(A_{24}^+ : C_{24})$.
Output: $Q = [3]P = (X_Q : Z_Q)$.

1	$t_0 \leftarrow X_P + Z_P$	$s_0 \leftarrow X_P - Z_P$
2	$t_1 \leftarrow t_0^2$	$s_1 \leftarrow s_0^2$
3	$t_0 \leftarrow X_P + X_P$	$s_0 \leftarrow t_1 - s_1$
4	$t_2 \leftarrow t_1 + t_1$	$s_2 \leftarrow s_0 + s_0$
5	$t_3 \leftarrow C_{24} \times s_1$	$s_3 \leftarrow A_{24}^+ \times s_0$
6	$t_0 \leftarrow t_3 \times t_1$	$s_0 \leftarrow t_0 \times t_0$
7	$t_1 \leftarrow t_2 + t_2$	$s_1 \leftarrow s_0 + s_2$
8	$t_3 \leftarrow t_3 + s_3$	$s_3 \leftarrow t_1 - s_1$
9	$t_4 \leftarrow s_3 \times t_3$	$s_4 \leftarrow s_0 \times t_3$
10	$t_4 \leftarrow t_0 - t_4$	$s_4 \leftarrow t_0 - s_4$
11	$t_4 \leftarrow t_4^2$	$s_4 \leftarrow s_4^2$
12	$X_Q \leftarrow X_P \times t_4$	$Z_Q \leftarrow Z_P \times s_4$
13	**return** $Q = (X_Q : Z_Q)$	

Algorithm 6.10: 4-isogeny computation with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P_4 = (X_4 : Z_4)$.
Output: $(A_{24}^+ : C_{24})$, $(K_0 : K_1 : K_2)$.

1	$K_2 \leftarrow X_4 + Z_4$	$K_1 \leftarrow X_4 - Z_4$
2	$t_0 \leftarrow Z_4^2$	$s_0 \leftarrow X_4^2$
3	$t_0 \leftarrow t_0 + t_0$	$s_0 \leftarrow s_0 + s_0$
4	$C_{24} \leftarrow t_0^2$	$A_{24}^+ \leftarrow s_0^2$
5	$K_0 \leftarrow t_0 + t_0$	
6	**return** $(A_{24}^+ : C_{24}), (K_0 : K_1 : K_2)$	

Algorithm 6.11: 3-isogeny computation with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P_3 = (X_3 : Z_3)$.
Output: $(A_{24}^+ : C_{24})$, $(K_1 : K_2)$.

$K_2 \leftarrow X_3 + Z_3$	$K_1 \leftarrow X_3 - Z_3$
$t_0 \leftarrow K_2^2$	$s_0 \leftarrow K_1^2$
$t_1 \leftarrow X_3 + X_3$	$s_1 \leftarrow s_0 - t_0$
$t_2 \leftarrow t_1^2$	$s_2 \leftarrow Z_3^2$
$t_1 \leftarrow t_2 + t_2$	$s_1 \leftarrow s_1 + s_1$
$t_3 \leftarrow t_1 - s_1$	$s_3 \leftarrow s_2 + s_2$
$t_4 \leftarrow t_3 + s_0$	$s_4 \leftarrow t_2 - t_0$
$C_{24} \leftarrow s_1 \times s_3$	$A_{24}^+ \leftarrow t_4 \times s_4$
return $(A_{24}^+ : C_{24}), (K_1 : K_2)$	

Algorithm 6.12: 4-isogeny evaluation with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P = (X_P : Z_P)$, $(K_0 : K_1 : K_2)$.
Output: $P' = \phi_4(P) = (X_{P'} : Z_{P'})$.

$t_0 \leftarrow X_P + Z_P$	$s_0 \leftarrow X_P - Z_P$
$t_1 \leftarrow K_1 \times t_0$	$s_1 \leftarrow t_0 \times s_0$
$t_2 \leftarrow K_2 \times s_0$	$s_2 \leftarrow K_0 \times s_1$
$t_0 \leftarrow t_2 + t_1$	$s_0 \leftarrow t_2 - t_1$
$t_0 \leftarrow t_0^2$	$s_0 \leftarrow s_0^2$
$t_1 \leftarrow t_0 + s_2$	$s_1 \leftarrow s_0 - s_2$
$X_{P'} \leftarrow t_1 \times t_0$	$Z_{P'} \leftarrow s_1 \times s_0$
return $(X_{P'} : Z_{P'})$	

Algorithm 6.13: 3-isogeny evaluation with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P = (X_P : Z_P)$, $(K_1 : K_2)$.
Output: $P' = \phi_3(P) = (X_{P'} : Z_{P'})$.

$t_0 \leftarrow X_P + Z_P$	$s_0 \leftarrow X_P - Z_P$
$t_0 \leftarrow K_1 \times t_0$	$s_0 \leftarrow K_2 \times s_0$
$t_1 \leftarrow t_0 + s_0$	$s_1 \leftarrow t_0 - s_0$
$t_1 \leftarrow t_1^2$	$s_1 \leftarrow s_1^2$
$X_{P'} \leftarrow X_P \times t_1$	$Z_{P'} \leftarrow Z_P \times s_1$
return $(X_{P'} : Z_{P'})$	

$R_3 = (X_3 : Z_3)$ on $E_{(A:C)}$ is of order 3, a 3-isogeny $\phi_3 : E_{(A:C)} \rightarrow E_{(A':C')}$ has to be constructed with $\ker \phi_3 = \langle R_3 \rangle$. Algorithm 6.11 shows our 2-way implementation of the function to compute an isogeny of degree 3. The 3-isogeny generation algorithm outputs the values $A^+_{24} = A' + 2C'$, $C_{24} = 4C'$[6] that define the target curve $E_{(A':C')}$, namely:

$$A^+_{24} = (3X_3^2 - 2X_3Z_3 - Z_3^2)(3X_3 + Z_3)^2 \quad \text{and} \quad C_{24} = -16X_3Z_3^3.$$

The algorithm also outputs the constants $K_1 = X_3 - Z_3$ and $K_2 = X_3 + Z_3$, which will be used in the evaluation of a 3-isogeny at a point.

6.5.4 Isogeny evaluation

Let $\phi_4 : E_{(A:C)} \rightarrow E_{(A':C')}$ be a 4-isogeny with kernel $\ker \phi_4 = \langle (X_4 : Z_4) \rangle$ and let the point $P = (X_P : Z_P)$ be on curve $E_{(A:C)}$. Then, the point $P' = \phi_4(P) = (X_{P'} : Z_{P'})$ is derived after the evaluation of the 4-isogeny ϕ_4 at P and defined as:

$$X_{P'} = 16\left((X_4X_P - Z_4Z_P)^2 + Z_4^2(X_P^2 - Z_P^2)\right)(X_4X_P - Z_4Z_P)^2$$
$$Z_{P'} = 16\left((X_4Z_P - Z_4X_P)^2 - Z_4^2(X_P^2 - Z_P^2)\right)(X_4Z_P - Z_4X_P)^2$$

Our 2-way implementation for the 4-isogeny evaluation is specified in Algorithm 6.12. In the case of 3-isogeny, let $\phi_3 : E_{(A:C)} \rightarrow E_{(A':C')}$ with kernel $\ker \phi_3 = \langle (X_3 : Z_3) \rangle$ and $P = (X_P : Z_P)$ be a point on curve $E_{(A:C)}$. Then, the image of P under the 3-isogeny ϕ_3 is a point $P' = \phi_3(P) = (X_{P'} : Z_{P'})$ such that:

$$X_{P'} = 4X_P\,(X_3X_P - Z_3Z_P)^2 \quad \text{and} \quad Z_{P'} = 4Z_P\,(X_3Z_P - Z_3X_P)^2\,.$$

The 2-way implementation of the 3-isogeny evaluation is presented in Algorithm 6.13.

6.5.5 Results and comparison

Because of dependencies among the involved operations, our AvxSike-LL only requires a $(1\times x\times y\times z)$-way implementation of the point tripling and 3-isogeny generation. On the other hand, for the doubling operation and 4-isogeny generation, we can besides the $(1 \times x \times y \times z)$-way implementation also use a $(2 \times x \times y \times z)$-way implementation in an optimized version of SIPKE encryption (see Algorithm 6.15). Table 6.3 indicates that the $(1 \times 2 \times 2 \times 2)$-way point-operations outperform their $(1 \times 2 \times 4 \times 1)$-way counterparts by a few clock cycles. In addition, the 4-isogeny computation (Algorithm 6.10) just uses $\mathbb{F}_{p^2}$-squaring but not $\mathbb{F}_{p^2}$-multiplication and, therefore, the $(1 \times 2 \times 4 \times 1)$-way version is clearly not efficient in this case. As a result, we chose $(1 \times 2 \times 2 \times 2)$-way parallelism to implement all the $(1\times x\times y\times z)$-way operations for the curve arithmetic. According to Table 6.4, the difference (in terms of cycles/instance) between the $(8 \times 1 \times 1 \times 1)$-way and the $(4 \times 2 \times 1 \times 1)$-way parallelism is rather small. Both of these parallelization options are much more efficient than the $(2 \times 2 \times 2 \times 1)$-way and $(1 \times 2 \times 2 \times 2)$-way versions.

6.6 Implementation: higher-layer arithmetic

6.6.1 Low-latency implementation

All functions in Algorithm 6.1 and Algorithm 6.2 contain two types of costly operations, namely the computation of the kernel generator $R \leftarrow P + [k]Q$ and the generation/evaluation of the ℓ^{e_ℓ}-isogeny

[6] The 3-isogeny generation algorithm in the SIKE specification (Algorithm 15 in [JAC+22]) originally outputs A^-_{24}. Our formula outputs C_{24} instead of A^-_{24} since our point tripling takes C_{24} as input.

Table 6.3: Experimental results of point-operation implementations for SIKEp503.

Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
Ladder step (xDBLADD)	SIDHv3.4	x64 asm	1-way	1	5056	5056	1.00×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	9417	1177	4.30×
	AvxSike	AVX-512	$(2 \times 4 \times 1 \times 1)$-way	2	2880	1440	3.51×
	AvxSike	AVX-512	$(1 \times 4 \times 2 \times 1)$-way	1	1757	1757	2.88×
Point doubling (xDBL)	SIDHv3.4	x64 asm	1-way	1	2873	2873	1.00×
	[KG19]	AVX-512	hybrid	1	1782	1782	1.61×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	5052	632	4.55×
	AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	1660	830	3.46×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1273	1273	2.26×
	AvxSike	AVX-512	$(1 \times 2 \times 4 \times 1)$-way	1	1319	1319	2.18×
Point tripling (xTPL)	SIDHv3.4	x64 asm	1-way	1	5794	5794	1.00×
	[KG19]	AVX-512	hybrid	1	3527	3527	1.64×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	10063	1258	4.61×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	2730	2730	2.12×
	AvxSike	AVX-512	$(1 \times 2 \times 4 \times 1)$-way	1	2745	2745	2.11×

Table 6.4: Experimental results of isogeny-operation implementations for SIKEp503.

Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
4-isogeny generation (get_4_isog)	SIDHv3.4	x64 asm	1-way	1	1729	1729	1.00×
	[KG19]	AVX-512	hybrid	1	1379	1379	1.25×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	3113	389	4.44×
	AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	843	422	4.10×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	673	673	2.57×
4-isogeny evaluation (eval_4_isog)	SIDHv3.4	x64 asm	1-way	1	3852	3852	1.00×
	[KG19]	AVX-512	hybrid	1	2292	2292	1.68×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	6925	866	4.45×
	AvxSike	AVX-512	$(4 \times 2 \times 1 \times 1)$-way	4	3569	892	4.32×
	AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	2289	1145	3.36×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1858	1858	2.07×
3-isogeny generation (get_3_isog)	SIDHv3.4	x64 asm	1-way	1	2783	2783	1.00×
	[KG19]	AVX-512	hybrid	1	2011	2011	1.38×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	4508	564	4.93×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1350	1350	2.06×
3-isogeny evaluation (eval_3_isog)	SIDHv3.4	x64 asm	1-way	1	2893	2893	1.00×
	[KG19]	AVX-512	hybrid	1	1628	1628	1.78×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	5130	641	4.51×
	AvxSike	AVX-512	$(4 \times 2 \times 1 \times 1)$-way	4	2651	663	4.36×
	AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	1521	761	3.80×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1235	1235	2.34×

Algorithm 6.14: Vectorized $\texttt{isogeny}_2$ in SIKEp503 using the optimal strategies.

Input: Curve E_A, public parameter e_2, point (X_R, Z_R), and a *strategy* $(s_1, \ldots, s_{e_2/2-1})$.
Output: Curve E_B such that $\phi_2 : E_A \rightarrow E_B$ with $\deg \phi_2 = 2^{e_2}$, $\ker \phi_2 = \langle (X_R : Z_R) \rangle$.

```
1  pts ← [ ], i ← 0, E_B ← E_A, k ← 1
2  for j from 1 to e_2/2 − 1 by 1 do
3      while i < e_2/2 − j do
4          push (X_R, Z_R, i) to pts                          /* Append it to the end of pts */
5          e ← s_k
6          (X_R, Z_R) ← xDBLe_1x2x2x2w(X_R, Z_R, E_B, 2e)     /* [2^2e]R */
7          i ← i + e, k ← k + 1
8      E_B, φ ← get_4_isog_1x2x2x2w(X_R, Z_R)                 /* 4-isogeny generation */
9      pts ← eval_4_isog_parallel(φ, pts)                     /* Parallel 4-isogeny evaluation */
10     pop (X_S, Z_S, i_S) from pts                           /* Remove it from the end of pts */
11     X_R ← X_S, Z_R ← Z_S, i ← i_S
12 E_B, φ ← get_4_isog_1x2x2x2w(X_R, Z_R)                     /* 4-isogeny generation */
13 return E_B
```

for $\ell \in \{2, 3\}$. The former is performed using the three-point ladder, whose efficient four-way vectorization has been explained in Section 6.5.1. On the other hand, as discussed in Section 6.2.2, SIKE takes advantage of the optimal strategies from [DJP14] to reduce the execution time of the ℓ^{e_ℓ}-isogeny generation and evaluation (these algorithms for $\ell \in \{2, 3\}$ are analyzed in detail in the SIKE specification, see [JAC+22, Algorithm 19 and 20]).

Vectorized ℓ^{e_ℓ}-isogeny computation and evaluation. We pick the 2^{e_2}-isogeny case as example to demonstrate our vectorized implementation (note that the 3^{e_3}-isogeny can be vectorized in a very similar fashion). It is common practice that, when e_2 is even, the 2^{e_2}-isogeny is computed as the composition of $e_2/2$ isogenies of degree 4, while an extra isogeny of degree 2 needs to be computed when e_2 is odd. The 2^{e_2}-isogeny generation for even e_2 is described in Algorithm 6.14. Because of dependencies among the operations in each iteration, we choose the $(1 \times 2 \times 2 \times 2)$-way parallel point doubling[7] to compute the kernel $[2^{2e}]R$ of the 4-isogeny, and the $(1 \times 2 \times 2 \times 2)$-way 4-isogeny generation to obtain the coefficients of the target curve. The vector $(s_1, \ldots, s_{e_2/2-1})$ denotes the tree traversal strategy used for fast isogeny computations. These strategies are described in [JAC+22, Appendix D] for the four parameter sets of our SIKE implementation. On the other hand, for the 4-isogeny evaluation at different points in the *pts* queue, we decided to develop a dedicated `eval_4_isog_parallel` function to achieve a fast simultaneous isogeny evaluation, which uses the more efficient $(8 \times 1 \times 1 \times 1)$-way and $(4 \times 2 \times 1 \times 1)$-way parallel implementations. This function checks at first the number of points in the *pts* queue and then uses the different vectorized implementations of 4-isogeny evaluation (i.e., `eval_4_isog`) with corresponding points in *pts* to handle the computation[8]:

$$
\begin{array}{ll}
\#pts = 1 : (1 \times 2 \times 2 \times 2)\text{-way} & \#pts = 5 : (4 \times 2 \times 1 \times 1)\text{-way} + (1 \times 2 \times 2 \times 2)\text{-way} \\
\#pts = 2 : (2 \times 2 \times 2 \times 1)\text{-way} & \#pts = 6 : (4 \times 2 \times 1 \times 1)\text{-way} + (2 \times 2 \times 2 \times 1)\text{-way} \\
\#pts = 3 : (4 \times 2 \times 1 \times 1)\text{-way} & \#pts = 7 : (8 \times 1 \times 1 \times 1)\text{-way} \\
\#pts = 4 : (4 \times 2 \times 1 \times 1)\text{-way} & \#pts = 8 : (8 \times 1 \times 1 \times 1)\text{-way}
\end{array}
$$

[7] $(X_Q, Z_Q) \leftarrow \texttt{xDBLe}\ (X_P, Z_P, E, n)$ denotes computing $Q \leftarrow [2^n]P$ on curve E by using n times the point-doubling operation `xDBL`.

[8] Note that the number of points in the *pts* queue (i.e., $\#pts$) is public information. Hence, using the different vectorized 4-isogeny evaluation implementations does not leak any secrets.

Algorithm 6.15: Optimized SIPKE encryption operation.

1 **function** EncOpt($\text{pk}_3, m \in \mathcal{M}, \text{sk}_2 \in \mathcal{K}_2$)
2 $x_{R_2} \leftarrow x_{P_2} + [\text{sk}_2]x_{Q_2}$ $\quad$ $x'_{R_2} \leftarrow \phi_3(x_{P_2}) + [\text{sk}_2]\phi_3(x_{Q_2})$
3 $(\phi_2, E_2) \leftarrow \texttt{isogeny}_2(E_0, x_{R_2})$ $\quad$ $(\phi'_2, E_{32}) \leftarrow \texttt{isogeny}_2(E_3, x'_{R_2})$
4 $c_1 \leftarrow (\phi_2(x_{P_3}), \phi_2(x_{Q_3}), \phi_2(x_{PQ_3}))$
5 $h \leftarrow \texttt{SHAKE256}(j(E_{32}))$
6 $c_2 \leftarrow h \oplus m$
7 **return** (c_1, c_2)

Optimized SIKE encapsulation. Even at the highest layer of SIKE, there are options to parallelize some internal operations. For example, we managed to further optimize the Encaps operation by parallelizing two scalar multiplications (computed with the same scalar sk_2) and parallelizing two 2^{e_2}-isogeny generation and evaluation operations, which are denoted as $\texttt{isogeny}_2$ in our optimized Enc function for SIPKE that is described in Algorithm 6.15. As stated in Section 6.5.1, the kernel generator in AvxSike-LL is obtained with help of the $(1 \times 4 \times 2 \times 1)$-way three-point ladder. However, Algorithm 6.15 computes in line 2 two kernel generators simultaneously, and therefore it is possible to use a more efficient ladder, namely the $(2 \times 4 \times 1 \times 1)$-way vectorized version. More importantly, in line 3, the algorithm performs two 2^{e_2}-isogeny generation and evaluation operations in parallel, which makes it possible to use the $(2 \times 2 \times 2 \times 1)$-way implementation for the point doubling and 4-isogeny generation instead of the $(1 \times 2 \times 2 \times 2)$-way version. This will lead to a significant difference in performance because the underlying $\mathbb{F}_p$-arithmetic implementation changes from (4×2)-way to (8×1)-way. According to our results, the optimized SIPKE encryption in Algorithm 6.15 improves the speed of a SIKE encapsulation by around 27% compared to the straightforward version of Enc (i.e., Algorithm 6.1).

6.6.2 High-throughput implementation

Since constant-time SIKE executes a fixed operation sequence, a batched implementation using AVX-512 is fairly easy to develop. We modified the SIDHv3.4 x64 implementation by using our $(8 \times 1 \times 1 \times 1)$-way curve arithmetic, $(8 \times 1 \times 1)$-way $\mathbb{F}_{p^2}$-operations, and (8×1)-way $\mathbb{F}_p$-operations at the different layers. We also developed a $(8 \times 1 \times 1 \times 1)$-way version of some other subroutines of AvxSike-HT, most notably the computation of the j-invariant (`j_inv`) and the curve coefficient (`get_A`), as well as a 3-way simultaneous inversion (`inv_3_way`), which takes advantage of the (8×1)-way $\mathbb{F}_p$-inversion.

Parallel SHAKE256. As explained in Section 6.2, the hash function used by SIKE is an XOF, namely `SHAKE256`, which employs the Keccak permutation. For AvxSike-HT, we developed a batched `SHAKE256` implementation based on AVX-512 instructions that can process eight inputs independently and in parallel. The eXtended Keccak Code Package (XKCP) contains various Keccak-related software artifacts, including highly-optimized implementations of the Keccak permutation for two generations of AVX, namely AVX2 and AVX-512[9]. Sinha Roy showed in [Sin19] that the AVX2-based Keccak source code from XKCP can serve as a starting point to build a batched `SHAKE256` implementation capable to process four inputs simultaneously, each using a 64-bit slot of a 256-bit AVX2 vector. We followed Sinha Roy's idea and used the AVX-512 implementation of Keccak from the XKCP to batch `SHAKE256` with AVX-512 instructions.

Table 6.5: Execution times (in cycles) of implementations of SIKEp503 on an Intel Core i3-1005G1 processor. The cycle-counts for SIDHv3.4, Kostic-Gueron's work [KG19], and AvxSike-LL are for the execution of one instance of an operation and "Speed-up" is the speed-up factor compared to SIDHv3.4. The cycle-counts for AvxSike-HT are for the execution of *eight* instances of an operation and "Throughput" is the throughput gain compared to SIDHv3.4 when it executes eight instances.

Operation	SIDHv3.4 (1 instance)	Kostic [KG19] (1 instance)		AvxSike-LL (1 instance)		AvxSike-HT (8 instances)	
	Cycles	Cycles	Speed-up	Cycles	Speed-up	Cycles	Throughput
KeyGen	8,078,669	4,842,909	1.67×	3,215,375	2.51×	14,179,026	4.56×
Encaps	13,188,788	7,923,514	1.66×	4,111,650	3.21×	22,992,807	4.59×
Decaps	14,026,750	8,513,409	1.65×	5,715,005	2.45×	24,619,263	4.56×

6.7 Evaluation

SIKEp503. Table 6.5 shows the cycle counts for the different SIKEp503 implementations. The timings for SIDHv3.4 and AvxSike are measured on our target Ice Lake CPU, while the results for the implementation reported in [KG19] are taken from the paper. As the first three implementations in Table 6.5 are designed to reduce the latency, we can compare them in terms of speed, while AvxSike-HT aims at increasing throughput and, thus, it makes sense to compare it with SIDHv3.4 in terms of throughput. Regarding key generation and decapsulation, AvxSike-LL is about 2.5 times faster than SIDHv3.4 and outperforms the implementation of [KG19] by a factor of 1.5.Furthermore, due to our optimizations for encapsulation (see Section 6.6.1), AvxSike-LL reaches a 3.2-fold higher encapsulation speed compared to SIDHv3.4, which can be beneficial for, e.g., server-side TLS processing since, when SIKE is integrated into TLS, the server has to perform encapsulations. TLS servers could profit even more from our high-throughput AvxSike-HT implementation because it outperforms SIDHv3.4 throughput-wise by a factor of almost 4.6.

Analysis. An x64 implementation of SIKE executed on an Ice Lake Core has to use one single $(64 \times 64 \rightarrow 128)$-bit multiplier sequentially, whereas AVX-512IFMA is able to perform eight parallel $(52 \times 52 + 64 \rightarrow 64)$-bit multiply-add operations. But this does not mean that IFMA instructions can lead to an (almost) eight-fold performance gain, not even in theory. Though the IFMA engine can carry out eight element-wise multiplications simultaneously, various other architectural and micro-architectural features and effects have to be considered, e.g., different multiplier widths (52 vs. 64 bits), different carry chains and other sequential dependencies, different instruction latencies and throughputs, as well as differences in the register space and occupation[10]. For all these reasons, the theoretical speed-up factor of an IFMA implementation compared to an x64 implementation like SIDHv3.4 is far from eight and very difficult to estimate. Kostic and Gueron focused in [KG19] on optimizing the $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ layer of SIKE, especially the multiplication of field elements, using AVX-512IFMA and achieved a reduction of the overall execution time by a factor of roughly 1.7 (these results still represent the speed record for SIKE on an Intel CPU). On the other hand, AvxSike takes advantage of sophisticated vectorization of the higher layers of SIKE, in addition to notably more efficient vectorized prime-field arithmetic. Thanks to careful optimizations at the higher layers, AvxSike-LL reaches 1.5 times faster execution times for both key generation and decapsulation compared to [KG19] (see Table 6.5). Furthermore, the encapsulation is almost two times faster.

[9] https://github.com/XKCP/XKCP/tree/master/lib/low/KeccakP-1600-times8/AVX512.

[10] The SIDHv3.4 library uses a full-radix representation (64 bits/limb) for field elements, which enables a 100% occupation of the registers (except for the register with the highest limb), while our 51 bits/limb representation implies that around 20% of each 64-bit element of an AVX-512 register is empty.

Table 6.6: Execution times (in cycles) of implementations of SIKEp434, SIKEp610, and SIKEp751 on an Intel Core i3-1005G1 processor. The cycle-counts for SIDHv3.4 and AvxSike-LL are for the execution of one instance of an operation and "Speed-up" is the speed-up factor compared to SIDHv3.4. The cycle-counts for AvxSike-HT are for the execution of *eight* instances of an operation and "Throughput" is the throughput gain compared to SIDHv3.4 when it executes eight instances.

Scheme	Operation	SIDHv3.4 (1 instance) Cycles	AvxSike-LL (1 instance) Cycles	AvxSike-LL (1 instance) Speed-up	AvxSike-HT (8 instances) Cycles	AvxSike-HT (8 instances) Throughput
SIKEp434	KeyGen	5,976,700	2,474,187	2.42×	10,442,609	4.58×
	Encaps	9,690,764	3,062,491	3.16×	16,801,041	4.61×
	Decaps	10,357,218	4,341,099	2.39×	18,053,398	4.59×
SIKEp610	KeyGen	14,096,085	6,918,618	2.04×	32,172,538	3.51×
	Encaps	25,875,968	10,001,282	2.59×	58,747,976	3.52×
	Decaps	26,040,095	13,124,052	1.98×	59,103,361	3.52×
SIKEp751	KeyGen	23,843,419	10,212,410	2.33×	46,662,723	4.09×
	Encaps	38,446,643	12,804,923	3.00×	74,885,499	4.11×
	Decaps	41,368,995	17,834,974	2.32×	80,684,214	4.10×

SIKEp434, SIKEp610, and SIKEp751. The benchmarking results of Microsoft's SIDHv3.4 and AvxSike for the other three parameter sets are given in Table 6.6. When analyzing the cycle counts achieved by the low-latency version AvxSike-LL, it is apparent that the speed-up factors (in relation to SIDHv3.4) for SIKEp434 and SIKEp751 are similar to the speed-up for SIKEp503, though a bit smaller. This reduced performance gain can be explained by the number of limbs needed for the elements of a 434-bit and 751-bit prime field, respectively. Namely, due to the radix-2^{51} representation, the number of limbs is *odd* in both the 434-bit case (nine limbs) and the 751-bit case (15 limbs), whereas it is *even* for the 503-bit field. An odd number of limbs is not ideal for the structure of (4×2)-way limb vector set (since there are four elements unused in the last limb vector) and causes an "underutilization" of the parallelism in many of the executed AVX-512 vector instructions. Furthermore, also the register allocation (i.e., how many operands and results can be kept in registers) impacts the overall performance, which, in turn, depends on the length of the field elements. In general, the larger the order of the underlying prime field, the more difficult it becomes to keep operands in the register file and the more register spills will happen in both the low-latency and the high-throughput version, respectively. While the speed-up factors for SIKEp434 and SIKEp751 are only marginally smaller than that for SIKEp503, it turns out that the gap between SIKEp610 and SIKEp503 is bigger. Even though the elements of a 610-bit field can be represented by an even number of limbs ($51 \times 12 = 612$), there are only two bits of "headroom" in this representation, which is not ideal with respect to lazy reduction. Concretely, when using the SIKEp610 parameters, a number of additional modular reductions have to be carried out to prevent overflows, which impacts both the low-latency version and the high-throughput version of AvxSike.

6.8 Conclusion

Summary. Vector processing engines like Intel's AVX offer a great potential to reduce the execution time (or increase the throughput) of public-key cryptosystems, and this is also the case for post-quantum KEMs such as SIKE. The AVX-512IFMA instructions deserve special attention because they allow one to execute eight multiplications of 52-bit operands in parallel, followed by a parallel addition of the upper or lower halves of the products to eight 64-bit operands. By developing sophisticated vector processing techniques for field arithmetic, point arithmetic, and isogeny computations, all of which are integrated into our AvxSike software, we were able to significantly improve both the la-

tency and the throughput of SIKE on modern Intel processors. For example, AvxSike-LL instantiated with the SIKEp503 parameters is about 1.5 times faster than the AVX-512IFMA-based SIKE implementation described in [KG19] and outperforms Microsoft's highly-optimized SIDHv3.4 library by a factor of roughly 2.5 for key generation and decapsulation, while the speed-up factor for the encapsulation reaches even 3.2. In summary, AvxSike does not only set new software speed and throughput records for SIKE on Intel CPUs, but also narrows the gap between SIKE and lattice-based post-quantum KEMs, mainly because the IFMA instructions are more beneficial for the multiplication in prime fields than the multiplication in the polynomial rings of e.g., NTRU, Kyber, or Saber.

Future work. We envision that follow-up work in two directions can yield interesting results. The first direction concerns the integration of AvxSike into an existing SSL/TLS protocol stack like OpenSSL to evaluate the impact of the latency-optimized implementation on the side of the client and the impact of the throughput-optimized implementation on the server side. In particular, it would be interesting to figure out to what extent the speed and throughput improvements of AvxSike propagate up to the protocol layer. A second research direction could target the question of how beneficial the presented vectorization techniques can be for compressed SIKE. For example, the public-key compression process described in [JAC+22] requires the execution of some very expensive operations, such as pairing and discrete logarithm computations over $\mathbb{F}_{p^2}$. Since these operations constitute the main bottleneck in the compression algorithm, it can be expected that utilizing the parallel processing power of AVX-512IFMA has the potential to significantly accelerate the overall execution of compressed SIKE.

Part IV

Efficient Cryptographic Instruction Set Extension Design

CHAPTER

7

RISC-V ISES FOR LIGHTWEIGHT SYMMETRIC CRYPTOGRAPHY

This Chapter is based on our paper [CGM+23]. While we were conducting the research work described in this Chapter, the NIST LWC standardization process was in its third and final round.

7.1 Introduction

The LWC selection process. In a detailed survey of various examples, Bernstein [Ber20] notes that modern, open cryptographic selection processes (or contests) are not without their issues. Set within the broader context of standardized cryptographic functionality, however, they represent an undeniably important and influential mechanism: modulo imperfections stemming from the non-trivial technical *and* non-technical challenges involved, they act to motivate and organize collaborative effort, and, at best, produce more robust outcomes as a result.

The (ongoing) final round of LWC standardization is expected to last approximately 12 months, implying a conclusion to the process in 2022. Beyond application of the minimum acceptability requirements [NIS18, Section 3], a range of factors mean that objective comparison between and then selection of submissions in each round, the final round perhaps most importantly, is a significant challenge. First, even in the final round, there are a large number of submissions and variants thereof. Second, there are a large number of relevant implementation technologies: these include hardware-oriented (e.g., FPGA, ASIC) and software-oriented (e.g., micro-controller) instances. Third, there are a large number of relevant evaluation criteria [NIS18, Section 4]: focusing on implementation-related examples, and so ignoring the complex, stand-alone challenge of cryptanalytic evaluation, these span at least cost [NIS18, Section 4.3] (e.g., area and/or memory footprint[1]), efficiency [NIS18, Section 4.3] (e.g., latency, throughput), and resilience to implementation (e.g., side-channel and fault) attack [NIS18, Section 4.2]. The product of these and other factors demands significant effort be invested, in part due to the design space of implementation techniques (spanning representation of data, and computation with it) and technologies which must be explored.

[1] Whereas the term memory footprint includes, e.g., static and dynamic data, we focus on instruction (or code) throughout: instruction footprint is the amount of memory required to house the instructions for a given implementation, which is a proxy for the number of instructions required.

ISE-supported software implementation. Instruction Set Extensions (ISEs) attempt to add domain-specific support (e.g., state, instructions) to an otherwise general-purpose base Instruction Set Architecture (ISA). Although applicable to many domains, the study of cryptographic ISEs [BGM09, HV11, RI16] spans at least a 25 year period; work by Nahum et al. [NOOS95] is among the first identifiable instances.

As a fundamental and long-lived computer systems interface, the design and extension of an ISA demands careful consideration (cf. [Gue09, Section 4]) and must deliver quantified improvement for the workload of interest to be viable. ISEs often *are* viable, however, because, for example, they represent a hybrid between use of hardware or software alone. This is particularly true with respect to the constrained platforms and evaluation metrics of relevance to the LWC selection process: a well designed ISE *can* result in lower memory footprint and latency than a software-only implementation, *and* greater flexible and efficiency (with respect to improvement per additional logic gate) than a hardware-only implementation.

ISEs were not (explicitly) considered *during* the AES selection process, but, *after* it concluded in 2002, were added to almost every major ISA; at the time of writing, these include (at least) x86 [Int18a, Section 12.13] (see also [Gue09, DGK19]), POWER [POW18, Section 6.11.1], ARMv8-A [ARM20, Section A2.3], SPARC [SPA16, Sections 7.3+7.4], and RISC-V [RVK22, Sections 2.4+2.5] (see also [MNP+21]). Using this fact as motivation, we argue that considering ISEs during the LWC selection process is important because doing so offers 1) improved understanding and concrete evidence which can inform the LWC process itself, and 2) preparatory analysis which can inform ISA designers seeking to support the LWC process outcome.

Contributions. As such, this work makes two central contributions:

1. Based on careful analysis, we present the design, implementation, and evaluation of one separate ISE for each of the 10 LWC final round submissions; for most submissions, our work represents the first exploration of implementations supported by domain-specific ISEs.

2. We present a number of novel software-only (i.e., without requiring an ISE) techniques and implementations. For most submissions, our work represents the first exploration of implementations supported by special-purpose, cryptographic Zbkb and Zbkx bit manipulation extensions, an approach which is particularly effective for Elephant (Section 7.3.3) and GIFT-COFB (Section 7.3.4); in the latter case, for example, we demonstrate how to optimize bit-sliced implementations of GIFT-128 (as used in GIFT-COFB) using Zbkb, rendering it more efficient than either standard or fix-sliced alternatives for short plaintexts/ciphertexts.

Source code. Note that all material associated with this work, e.g., documentation and source code relating to all hardware and software implementations, are openly available at `https://github.com/scarv/lwise` under an open source license: we expect this material to evolve throughout the remainder of the LWC process and beyond.

7.2 Background

Scope. In part to cope with the large design space considered, and thus engineering effort required, we fix the scope of our work in the following ways:

1. For each submission, we only consider the primary algorithm; each such algorithm is based on a "building block" component or kernel which dominates computation. We only consider intra-kernel ISEs, i.e., ISEs for use *within* a given kernel: the definition of a kernel implies that any extra-kernel opportunities for ISEs have at best a marginal impact, so are not considered viable.

Furthermore, we only consider partial implementation of a given kernel where appropriate. Romulus is based on the Skinny-128-384+ kernel, for example, but only uses it to encrypt data; we do not consider support for decryption, therefore, although it would clearly be possible to do so *if* it were more generally useful.

2. We do not consider the hash function API: focusing on the the AEAD API alone seems sufficient, because, for each submission, use of the same kernel is evident across the algorithms which support both APIs.

3. We only consider a 32-bit ISA (and also ISEs for it therefore). Although consideration of a wider set of ISAs is more generally useful, we rationalize this decision by noting it aligns with the (implied) scope of the LWC process: the NIST call outlines a requirement to consider "8-*bit,* 16-*bit and* 32-*bit microcontroller architectures*" [NIS18, Section 3.4], for example, meaning a 64-bit ISA is deemed out of scope.

4. Although some discussion of the topic is included for completeness in Section 7.5, we do not consider support in the ISA nor ISEs for countermeasures against implementation attacks (other than their ability to deliver data-independent execution latency). We rationalize this decision by noting it aligns with the (implied) scope of the RISC-V scalar cryptographic extensions [RVK22]: for example, the Zkne and Zknd extensions [RVK22, Sections 2.4+2.5] for AES do not consider interaction with masking-based countermeasures against DPA-like attacks [KJJ99, MOP07].

5. For most submissions, we considered *multiple* ISE design variants. However, we only present results for the *single* ISE design variant we deem most effective, i.e., that which offers the greatest improvement in execution latency per additional logic gate. We stress that the results are therefore a "snapshot", rather than exhaustive exploration of the (large) design space.

RISC-V. In line with our scope, we focus on use of the **unextended ISA** RV32GC [RV19, Chapter 2] as a starting point. We define the **base ISA**, i.e., a baseline for our work, as the unextended ISA *plus* Zbkb and Zbkx: it therefore includes the 32-bit integer ISA *plus* the general-purpose M (multiplication) [RV19, Chapter 7], A (atomic) [RV19, Chapter 8], F (single-precision floating-point) [RV19, Chapter 11], D (double-precision floating-point) [RV19, Chapter 12], and C (compressed) [RV19, Chapter 16] extensions, *plus* the special-purpose, cryptographic Zbkb (a subset of K for bit manipulation instructions) [RVK22, Section 2.1] and Zbkx (a subset of K for crossbar permutation instructions) [RVK22, Section 2.2] extensions. We define an **extended ISA** as then capturing our work, i.e., the base ISA extended with support for an LWC-specific ISE.

Notation. Let $x_{(b)}$ denote an x expressed in radix- or base-b; the base may be omitted, in which case it is safe to assume $b = 10$. Let $\mathsf{MEM}[i]^b$ denote a b-byte access to some byte-addressable memory, using the address i; note that where $b = 1$, the access granularity may be omitted. Let $\mathsf{GPR}[i]$, for $0 \leq i < 32$, denote the i-th entry of the general-purpose register file. Note that $\mathsf{GPR}[0]$ is fixed to 0, in the sense reads from it always yield 0 and writes to it are ignored. Let $x \ll y$ and $x \lll y$ (resp. $x \gg y$ and $x \ggg y$) denote left-shift and left-rotate (resp. right-shift and right-rotate) of x by y bits respectively. Let $x \parallel y$ denote concatenation of x and y, and $x_{h \ldots l}$ denote extraction of bits h (the high, or more-significant index) through l (the low, or less-significant index) inclusive from some x. RISC-V uses XLEN to denote the word size. We adopt same approach, meaning XLEN = 32 because the context is RV32GC.

7.3 Design

NIST are careful to use "*algorithm(s)*" throughout [NIS18, Section 5], presumably to at least allow selection of a suite of rather than a single algorithm. Although one could conclude that multi-algorithm ISEs, i.e., ISEs which support more than one algorithm, are attractive therefore, focusing on them is arguably premature until the outcome is clear. In this section, we therefore adopt a 2-step design process. First, we focus on independently developing an ISE design for each algorithm: each of the following subsections first summarizes such a design at a high level and then provides the associated lower-level technical detail (e.g., instruction encoding, semantics, etc.). We use a uniform structure in each such subsection by presenting, at a higher-level, 1) an overview of the submission, 2) an overview of the kernel within said submission that we focus on, 3) implementation options (including related work, e.g., implementation results), 4) a description of the ISE design, then, finally, at a lower-level, 5) the detailed material relating to the ISE design. Second, and based on the above, Section 7.3.12 concludes with a broader discussion of opportunities relating to design of ISAs, ISEs, and the algorithms themselves; by taking a broader perspective, this second step therefore highlights if and where multi-algorithm ISEs can be extracted from the single-algorithm ISE designs.

7.3.1 Constraints

In their study of support for AES in RISC-V, Marshall et al. [MNP+21, Section 3] codify a set of ISE requirements to guide their subsequent design process. We adopt the same requirements, which, for completeness, we reproduce here (numbered to match, noting we omit their AES-specific requirement 1):

Requirement 2. The ISE must align with the wider RISC-V design principles. This means it should favour simple building-block operations, and use instruction encodings with at most 2 source register addresses and 1 destination register address.

Requirement 3. The ISE must use the RISC-V general-purpose scalar register file to store operands.

Requirement 4. The ISE must not introduce special-purpose architectural state, nor rely on special-purpose micro-architectural state.

On one hand, we recognize that adopting these constraints means potential ISE designs might be ignored; this fact potentially renders our results sub-optimal, at least versus a more permissive alternative where the constraints are *not* adhered to. A pertinent example is the approach of Steinegger and Primas [SP21], which captures the 320-bit Ascon state within 10 general-purpose registers then used as input and output by a tightly-coupled accelerator for an entire round. This approach may be reasonable for a specific use-cases, and variants of it are in fact viable for *all* the LWC candidates. However, the approach violates Requirement 2: although a useful option in the overall design space, our approach (namely a focus on more traditional, RISC-like ISEs) is fundamentally different.

On the other hand, we argue that the same constraints maximize potential utility of our ISE designs. For example, within the context of RISC-V they 1) support multiple implementation options, including a more traditional integrated approach or via the in-development Custom Function Unit (CFU)[2] specification, and 2) offer an easier route to standardization and deployment as a result of limiting impact on other aspects of the ISA. Beyond this, the constraints also permit extrapolation to other ISAs, e.g., via the ARMv8-M custom instruction mechanism [CP20]; doing so would be more difficult otherwise.

[2] https://cfu.readthedocs.io.

7.3.2 Ascon

Submission overview. The Ascon [DEMS21] submission specifies the AEAD algorithms [DEMS21, Section 2.4] Ascon-128, Ascon-128a, and Ascon-80pq, and the hash function algorithms [DEMS21, Section 2.5] Ascon-Hash and Ascon-Hasha. We focus on the primary algorithm Ascon-128, and, more specifically therefore, a kernel represented by the p^a and p^b permutations [DEMS21, Section 2.6] (a single permutation p, often referred to as Ascon-p, with a and b rounds respectively).

Kernel overview. The Ascon-p permutation manipulates a 320-bit state, which is organized in five 64-bit words, by iteratively applying a round function p. This round function is essentially a Substitution-Permutation Network (SPN) and comprises three parts: 1) the addition of an 8-bit round constant c_r to a 64-bit state-word, 2) a substitution layer that operates across the five words of the state and implements an affine equivalent of the S-box in the χ mapping of Keccak, and 3) a permutation layer consisting of linear functions that are similar to the Σ functions in SHA2 and performed on each state-word individually. The S-box maps five input bits to five output bits and is applied to each column of the state, whereby the five state-words are arranged vertically.

Implementation options. The substitution layer is normally implemented in a bit-sliced fashion using logical ANDs, XORs, and NOTs. On the other hand, the permutation layer performs an operation of the form $x = x \oplus (x \ggg n) \oplus (x \ggg m)$ on each 64-bit word x of the state. On 32-bit architectures, the Ascon-p permutation is usually implemented in a Bit-Interleaved (BI) fashion, which means each 64-bit word of the state is split up into two 32-bit words, one containing the bits at even positions and the other the bits at odd positions. This representation has the advantage that one can perform a 64-bit rotation through two 32-bit rotations, which is particularly beneficial on 32-bit ARM Cortex-M microcontrollers due to their "free" rotations. Even though bit-interleaving has the potential to speed up the linear functions of Ascon-p on any 32-bit platform (including RV32), one has to take into account that this performance gain for the permutation comes at the expense of conversions between the BI representation and normal representation whenever data is injected into or extracted from the state.

ISE description. The substitution layer consists of logical operations on 64-bit words, which can be split up into two operations on 32-bit chunks. An optimized implementation of the S-box requires 17 native RV32GC instructions [CJL+20], which can be reduced to 15 with the help of two Zbkb instructions. The permutation layer can achieve a more significant speed-up since its operations of the form $x = x \oplus (x \ggg n) \oplus (x \ggg m)$ map naturally to two custom `sigma` instructions that use the upper and lower part of a 64-bit state-word as input and produce either the upper or lower part of the result. The rotation amounts can be specified through immediate values. In this way, the instruction-count of the full permutation layer can be reduced from 80 (i.e., 16 per-word) to only 10. This reduction of the number of instructions to 10 is independent of whether bit-interleaving is applied or not, which means that using the BI representation has actually an adverse impact on the overall performance due to the conversions between BI and normal representation.

ISE design. The additional, more detailed material relating to the ISE design is shown below.

31 30	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
00	imm	rs2	rs1	111	rd	0101011	ascon.sigma.lo
01	imm	rs2	rs1	111	rd	0101011	ascon.sigma.hi

- **additional notation**

```
/* Define the look-up tables */
ROT_0 = { 19, 61,  1, 10,  7 }
ROT_1 = { 28, 39,  6, 17, 41 }
```

- **ascon.sigma.lo rd, rs1, rs2, imm**

```
x_hi    ← GPR[rs2]
x_lo    ← GPR[rs1]
x       ← x_hi || x_lo
r       ← x ^ ( x >>> ROT_0[ imm ] ) ^ ( x >>> ROT_1[ imm ] )
GPR[rd] ← r_{31.. 0}
```

- **ascon.sigma.hi rd, rs1, rs2, imm**

```
x_hi    ← GPR[rs2]
x_lo    ← GPR[rs1]
x       ← x_hi || x_lo
r       ← x ^ ( x >>> ROT_0[ imm ] ) ^ ( x >>> ROT_1[ imm ] )
GPR[rd] ← r_{63..32}
```

7.3.3 Elephant

Submission overview. The Elephant [BCDM21] submission specifies the AEAD algorithms

$$\begin{aligned} \text{Dumbo} &= \text{Elephant-Spongent-}\pi[160] \\ \text{Jumbo} &= \text{Elephant-Spongent-}\pi[176] \\ \text{Delirium} &= \text{Elephant-Keccak-}f[200] \end{aligned}$$

We focus on the primary algorithm Dumbo, and, more specifically therefore, a kernel represented by the Spongent-$\pi[160]$ permutation (see also [BKL+13]).

Kernel overview. Spongent-$\pi[160]$ used in Dumbo is a 80-round Spongent permutation [BKL+13] (essentially a PRESENT-type permutation [BKL+07]). It operates on a 160-bit state and consists of three layers in each round: 1) XORing the state with two round constants, of which one is computed by a 7-bit LFSR ICounter_{160}, i.e., $0^{153} \parallel \text{ICounter}_{160}(i)$, while the other one is rev $(0^{153} \parallel \text{ICounter}_{160}(i))$, where i denotes the round index and rev is a function reversing the order of the bits of its input; 2) sBoxLayer_{160}, a 4-bit S-box applied 40 times in parallel; 3) pLayer_{160}, moving the bit j of state to bit position $40 \cdot j \bmod 159$ while the bit 159 keeps unmoved.

Implementation options. We developed the pure-software implementation of Spongent-$\pi[160]$ from scratch by ourselves, in which we presented several optimisation techniques based on our base ISA. The 160-bit state is stored in five 32-bit words S_0, S_1, S_2, S_3, and S_4, where each S_i stores bits $32i$ to $32i+31$ of the state. First, we precompute all the round constants so that the first layer is simplified to require only few instructions to load/prepare the constants plus then two XOR instructions. Second, Zbkx provides a dedicated instruction for the parallel 4-bit S-box, namely `xperm4`, which is very beneficial for sBoxLayer_{160}. Concretely, the `xperm`-style look-up table for sBoxLayer_{160} is construct with three registers before Spongent-$\pi[160]$ starts:

```
li rl, 0xF4120BDE       /* the lower  half of S-box look-up table */
li rh, 0x63C958A7       /* the higher half of S-box look-up table */
li rm, 0x88888888       /* the mask used in xperm-style S-box     */
```

Each 32-bit word S_i (stored in `rx`) can perform eight 4-bit S-boxes simultaneously with two `xperm4` and two XOR instructions via

```
xperm4 ry, rl, rx
xor    rx, rx, rm
```

```
xperm4  rx, rh, rx
xor     rx, rx, ry
```

so in each round the whole sBoxLayer_{160} needs 20 instructions in total. Last, we divide the pLayer_{160} into two steps: 1) for each word S_i, we firstly apply the `unzip` instruction (from Zbkb) twice and thus make S_i be a form shown in the third row of Figure 7.1; 2) we then take advantage of eight `SWAPMOVE` operations (`SWAPMOVE` will be explained in detail in Section 7.3.12) to swap the bits between different words, i.e.,

```
SWAPMOVE(S0, S1, 0x000000FF,  8);    SWAPMOVE(S0, S2, 0x000000FF, 16);
SWAPMOVE(S0, S3, 0x000000FF, 24);    SWAPMOVE(S1, S2, 0x0000FF00,  8);
SWAPMOVE(S1, S4, 0x000000FF, 24);    SWAPMOVE(S2, S3, 0x0000FF00,  8);
SWAPMOVE(S2, S4, 0x0000FF00, 16);    SWAPMOVE(S3, S4, 0x00FF0000,  8);
```

and, afterwards, we use three `rori` instructions (for right-rotation, also from Zbkb) to make S_1, S_2, and S_3 correctly-aligned.

ISE description. At first, we designed a custom instruction for the parallel 4-bit S-box, where we integrated the first step of pLayer_{160} (i.e., two `unzip` instructions) at the end. Moreover, we designed two instructions for the specific `SWAPMOVE` operations used in our second step of pLayer_{160}. Because each of our custom instruction has 1 destination register and each `SWAPMOVE` swaps bits between two different words, so 2 custom instructions are therefore required to perform one complete `SWAPMOVE` here. We also integrated the final three right-rotations into the custom instruction to further reduce the latency.

ISE design. The additional, more detailed material relating to the ISE design is shown below, where we let `SBOX` denote the 4-bit Spongent S-box per [BKL+13].

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0000	imm	rs2	rs1	111	rd	0001011	elephant.pstep.x
0001	imm	rs2	rs1	111	rd	0001011	elephant.pstep.y
0010	imm	00000	rs1	110	rd	0001011	elephant.sstep

- **additional notation**

```
/* Define the functions */
SWAPMOVE32(x,m,n) {
  t ← x ^ ( x >> n )
  t ← t & m
  t ← t ^ ( t << n )
  x ← t ^ x
  return x
}
SWAPMOVE32_X(x,y,m,n) {
  t ← y ^ ( x >> n )
  t ← t & m
  x ← x ^ ( t << n )
  return x
}
SWAPMOVE32_Y(x,y,m,n) {
  t ← y ^ ( x >> n )
  t ← t & m
  y ← y ^ ( t      )
  return y
}
```

- **elephant.pstep.x rd, rs1, rs2, imm**

```
x       ← GPR[rs1]
y       ← GPR[rs2]

if      ( imm == 0 ) {
  r ← SWAPMOVE32_X( x, y, 0x000000FF,  8 )
}
else if ( imm == 1 ) {
  r ← SWAPMOVE32_X( x, y, 0x000000FF, 16 )
}
else if ( imm == 2 ) {
  r ← SWAPMOVE32_X( x, y, 0x000000FF, 24 )
}
else if ( imm == 3 ) {
  r ← SWAPMOVE32_X( x, y, 0x0000FF00,  8 )
}
else if ( imm == 4 ) {
  r ← SWAPMOVE32_X( x, y, 0x000000FF, 24 ) >>> 24
}
else if ( imm == 5 ) {
  r ← SWAPMOVE32_X( x, y, 0x0000FF00, 16 ) >>> 16
}
else if ( imm == 6 ) {
  r ← SWAPMOVE32_X( x, y, 0x00FF0000,  8 ) >>>  8
}

GPR[rd] ← r
```

- **elephant.pstep.y rd, rs1, rs2, imm**

```
x       ← GPR[rs1]
y       ← GPR[rs2]

if      ( imm == 0 ) {
  r ← SWAPMOVE32_Y( x, y, 0x000000FF,  8 )
}
else if ( imm == 1 ) {
  r ← SWAPMOVE32_Y( x, y, 0x000000FF, 16 )
}
else if ( imm == 2 ) {
  r ← SWAPMOVE32_Y( x, y, 0x000000FF, 24 )
}
else if ( imm == 3 ) {
  r ← SWAPMOVE32_Y( x, y, 0x0000FF00,  8 )
}
else if ( imm == 4 ) {
  r ← SWAPMOVE32_Y( x, y, 0x000000FF, 24 )
}
else if ( imm == 5 ) {
  r ← SWAPMOVE32_Y( x, y, 0x0000FF00, 16 )
}
else if ( imm == 6 ) {
  r ← SWAPMOVE32_Y( x, y, 0x00FF0000,  8 )
}

GPR[rd] ← r
```

- **elephant.sstep rd, rs1**

```
x       ← GPR[rs1]

r       ← SBOX[ x_{31..28} ] || SBOX[ x_{27..24} ] ||
          SBOX[ x_{23..20} ] || SBOX[ x_{19..16} ] ||
          SBOX[ x_{15..12} ] || SBOX[ x_{11.. 8} ] ||
          SBOX[ x_{ 7.. 4} ] || SBOX[ x_{ 3.. 0} ]

r       ← SWAPMOVE32( r, 0x0A0A0A0A,  3 )
```

```
r       ← SWAPMOVE32( r, 0x00CC00CC,  6 )
r       ← SWAPMOVE32( r, 0x0000F0F0, 12 )
r       ← SWAPMOVE32( r, 0x0000FF00,  8 )

GPR[rd] ← r
```

7.3.4 GIFT-COFB

Submission overview. The GIFT-COFB [BCI+21] submission specifies an eponymous AEAD algorithm. We focus on this, the only and therefore primary algorithm, and, more specifically therefore, a kernel represented by the GIFT-128 block cipher (see also [BPP+17]).

Kernel overview. GIFT-128, belonging to GIFT block cipher family, is based on a SPN with a key length and a block size of both 128 bits. It is a 40-round block cipher with an identical round function that consists of three steps, namely `SubCells`, `PermBits`, and `AddRoundKey`. A typical technique to implement GIFT-128 is bit-slicing [BPP+17], where the 128-bit cipher state is expressed as four 32-bit slices S_0, S_1, S_2, and S_3. `SubCells` is essentially a 4-bit S-box, which needs 11 bitwise logical operations in bit-slicing. `PermBits` has a special property that bits in S_i remain in the same slice through the permutation. `AddRoundKey` includes three sub-steps: add round key (to S_1 and S_2), add round constant (to S_3), and key state update (with a main operation of two 16-bit word-wise rotations). We refer readers to [BPP+17] or the GIFT-COFB specification [BCI+21] for more details.

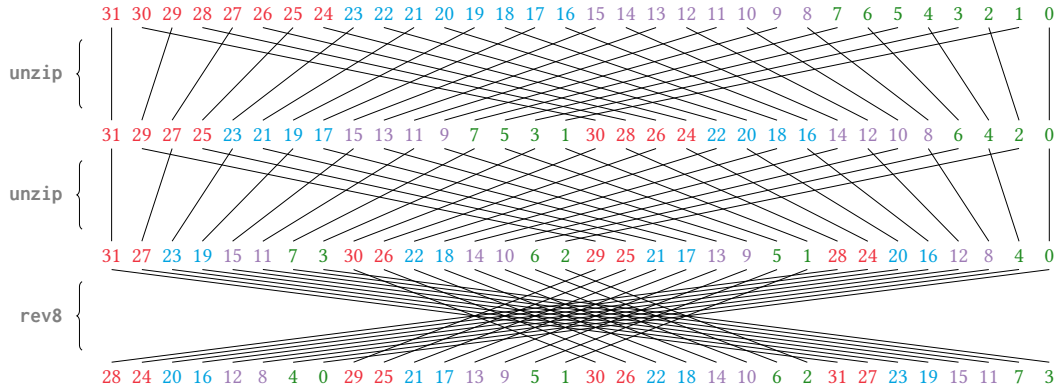


Figure 7.1: `PermBits` of GIFT-COFB using instructions from Zbkb. The numbers denote the bit indices of the input 32-bit state slice.

Implementation options. In addition to naive bit-slicing, a new representation for GIFT-128, namely the fix-slicing, is proposed in [ANP20]. In this work, we considered both different types of state representation for GIFT-128. According to [ANP20], fix-slicing is faster on 32-bit ARM Cortex-M microcontrollers in relation to the naive bit-slicing. However, thanks to Zbkb instructions, we are able to execute the `PermBits` very efficiently, which makes naive bit-slicing outperform fix-slicing on our base ISA. In detail, only three or four instructions are required in order to permute a 32-bit state slice S_i in each `PermBits` operation (we save the last `rori` for S_3):

```
unzip rx, rx
unzip rx, rx
rev8  rx, rx
rori  rx, rx, imm
```

Figure 7.1 illustrates how `unzip` and `rev8` permute bits (of a single S_i) during `PermBits`, from which we observe that the output of `rev8` is already the output for S_3 [BCI+21, Table 2.2]. For S_0, S_1, and S_2, we just further rotate the resulting state slice to the right (using `rori`) with the corresponding offset (i.e., 24, 16, and 8 respectively). Furthermore, Zbkb can also speed up the key state update operation. Concretely, we assume a 32-bit key state word $W_6 \parallel W_7$ (stored in `rx`). With the help of `pack` instruction, we can quickly obtain $W_6 \ggg 2 \parallel W_7 \ggg 12$ through

```
pack ry, rx, rx          /* ry = ( W7         ) || ( W7         ) */
rori rx, rx, 16          /* rx = ( W7         ) || ( W6         ) */
pack rx, rx, rx          /* rx = ( W6         ) || ( W6         ) */
rori ry, ry, 12          /* ry = ( W7 >>> 12 ) || ( W7 >>> 12 ) */
rori rx, rx,  2          /* rx = ( W6 >>>  2 ) || ( W6 >>>  2 ) */
pack rx, ry, rx          /* rx = ( W6 >>>  2 ) || ( W7 >>> 12 ) */
```

ISE description. We implemented both the fix-slicing and the naive bit-slicing implementation of GIFT-128 on the base ISA, and designed ISE for each of them. The fix-slicing implementation separates the computation of round key-update from the main GIFT-128 and uses an efficient round key pre-computation to align with the fix-slicing representation. On the other hand, the ISE for the bit-slicing implementation includes only two instructions to accelerate `PermBits` and the key state update, respectively. In essence, the ISE for fix-slicing include an instruction for the so-called `SWAPMOVE` operation (which will be discussed in detail in Section 7.3.12), three instructions for the rotation of nibbles, bytes, and halfwords in a 32-bit register, whereby the rotation amount is encoded as an immediate value, and three further instructions for the key-update function. The latter three instructions perform a sequence of `SWAPMOVE`s and operations that consist of rotations of 32-bit words, logical ANDs with a constant, and logical ORs. Each of the three key-update instructions operates on a single 32-bit word.

ISE design for fix-slicing implementation. The additional, more detailed material relating to the ISE design is shown below.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

00	imm	rs2	rs1	111	rd	0001011	gift.swapmove
01	imm	00000	rs1	110	rd	0001011	gift.rori.n
10	imm	00000	rs1	110	rd	0001011	gift.rori.b
11	imm	00000	rs1	110	rd	0001011	gift.rori.h
00	imm	00000	rs1	110	rd	0101011	gift.key.reorg
01	00000	00000	rs1	110	rd	0101011	gift.key.updstd
10	imm	00000	rs1	110	rd	0101011	gift.key.updfix

- **additional notation**

```
/* Define the function */
SWAPMOVE32(x,m,n) {
  t ← x ^ ( x >> n )
  t ← t & m
  t ← t ^ ( t << n )
  x ← t ^ x
  return x
}
```

- **gift.swapmove rd, rs1, rs2, imm**

```
x       ← GPR[rs1]
m       ← GPR[rs2]
```

```
r       ← SWAPMOVE32( x, m, imm )
GPR[rd] ← r
```

- **gift.rori.n rd, rs1, imm**

```
x_7     ← GPR[rs1]_{31..28}
x_6     ← GPR[rs1]_{27..24}
x_5     ← GPR[rs1]_{23..20}
x_4     ← GPR[rs1]_{19..16}
x_3     ← GPR[rs1]_{15..12}
x_2     ← GPR[rs1]_{11.. 8}
x_1     ← GPR[rs1]_{ 7.. 4}
x_0     ← GPR[rs1]_{ 3.. 0}
r       ← ( x_7 >>> imm ) || ( x_6 >>> imm ) ||
          ( x_5 >>> imm ) || ( x_4 >>> imm ) ||
          ( x_3 >>> imm ) || ( x_2 >>> imm ) ||
          ( x_1 >>> imm ) || ( x_0 >>> imm )
GPR[rd] ← r
```

- **gift.rori.b rd, rs1, imm**

```
x_3     ← GPR[rs1]_{31..24}
x_2     ← GPR[rs1]_{23..16}
x_1     ← GPR[rs1]_{15.. 8}
x_0     ← GPR[rs1]_{ 7.. 0}
r       ← ( x_3 >>> imm ) || ( x_2 >>> imm ) ||
          ( x_1 >>> imm ) || ( x_0 >>> imm )
GPR[rd] ← r
```

- **gift.rori.h rd, rs1, imm**

```
x_1     ← GPR[rs1]_{31..16}
x_0     ← GPR[rs1]_{15.. 0}
r       ← ( x_1 >>> imm ) || ( x_0 >>> imm )
GPR[rd] ← r
```

- **gift.key.reorg rd, rs1, imm**

```
x       ← GPR[rs1]

if      ( imm == 0 ) {
  r ← SWAPMOVE32( x, 0x00550055,  9 )
  r ← SWAPMOVE32( r, 0x00003333, 18 )
  r ← SWAPMOVE32( r, 0x000F000F, 12 )
  r ← SWAPMOVE32( r, 0x000000FF, 24 )
}
else if ( imm == 1 ) {
  r ← SWAPMOVE32( x, 0x11111111,  3 )
  r ← SWAPMOVE32( r, 0x03030303,  6 )
  r ← SWAPMOVE32( r, 0x000F000F, 12 )
  r ← SWAPMOVE32( r, 0x000000FF, 24 )
else if ( imm == 2 ) {
  r ← SWAPMOVE32( x, 0x0000AAAA, 15 )
  r ← SWAPMOVE32( r, 0x00003333, 18 )
  r ← SWAPMOVE32( r, 0x0000F0F0, 12 )
  r ← SWAPMOVE32( r, 0x000000FF, 24 )
else if ( imm == 3 ) {
  r ← SWAPMOVE32( x, 0x0A0A0A0A,  3 )
  r ← SWAPMOVE32( r, 0x00CC00CC,  6 )
  r ← SWAPMOVE32( r, 0x0000F0F0, 12 )
  r ← SWAPMOVE32( r, 0x000000FF, 24 )
}

GPR[rd] ← r
```

- **gift.key.updstd rd, rs1**

```
x       ← GPR[rs1]

r       ←     ( ( x >> 12 ) & 0x0000000F )
r       ← r | ( ( x & 0x00000FFF ) <<  4 )
r       ← r | ( ( x >>  2 ) & 0x3FFF0000 )
r       ← r | ( ( x & 0x00030000 ) << 14 )

GPR[rd] ← r
```

- **gift.key.updfix rd, rs1, imm**

```
x       ← GPR[rs1]

if      ( imm == 0 ) {
  r ← SWAPMOVE32( x, 0x00003333, 16 )
  r ← SWAPMOVE32( r, 0x55554444,  1 )
}
else if ( imm == 1 ) {
  r ←     ( ( x & 0x33333333 ) >>> 24 )
  r ← r | ( ( x & 0xCCCCCCCC ) >>> 16 )
  r ← SWAPMOVE32( r, 0x55551100,  1 )
}
else if ( imm == 2 ) {
  r ←     ( ( x >>  4 ) & 0x0F000F00 ) | ( ( x & 0x0F000F00 ) <<  4 )
  r ← r | ( ( x >>  6 ) & 0x00030003 ) | ( ( x & 0x003F003F ) <<  2 )
}
else if ( imm == 3 ) {
  r ←     ( ( x >>  6 ) & 0x03000300 ) | ( ( x & 0x3F003F00 ) <<  2 )
  r ← r | ( ( x >>  5 ) & 0x00070007 ) | ( ( x & 0x001F001F ) <<  3 )
}
else if ( imm == 4 ) {
  r ←     ( ( x & 0xAAAAAAAA ) >>> 24 )
  r ← r | ( ( x & 0x55555555 ) >>> 16 )
}
else if ( imm == 5 ) {
  r ←     ( ( x & 0x55555555 ) >>> 24 )
  r ← r | ( ( x & 0xAAAAAAAA ) >>> 20 )
}
else if ( imm == 6 ) {
  r ←     ( ( x >>  2 ) & 0x03030303 ) | ( ( x & 0x03030303 ) <<  2 )
  r ← r | ( ( x >>  1 ) & 0x70707070 ) | ( ( x & 0x10101010 ) <<  3 )
}
else if ( imm == 7 ) {
  r ←     ( ( x >> 18 ) & 0x00003030 ) | ( ( x & 0x01010101 ) <<  3 )
  r ← r | ( ( x >> 14 ) & 0x0000C0C0 ) | ( ( x & 0x0000E0E0 ) << 15 )
  r ← r | ( ( x >>  1 ) & 0x07070707 ) | ( ( x & 0x00001010 ) << 19 )
}
else if ( imm == 8 ) {
  r ←     ( ( x >>  4 ) & 0x0FFF0000 ) | ( ( x & 0x000F0000 ) << 12 )
  r ← r | ( ( x >>  8 ) & 0x000000FF ) | ( ( x & 0x000000FF ) <<  8 )
}
else if ( imm == 9 ) {
  r ←     ( ( x >>  6 ) & 0x03FF0000 ) | ( ( x & 0x003F0000 ) << 10 )
  r ← r | ( ( x >>  4 ) & 0x00000FFF ) | ( ( x & 0x0000000F ) << 12 )
}

GPR[rd] ← r
```

ISE design for bit-slicing implementation. The additional, more detailed material relating to the ISE design is shown below.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

01	00000	00000	rs1	110	rd	0101011	gift.key.updstd
11	imm	00000	rs1	110	rd	0101011	gift.permbits.step

- **additional notations**

```
/* Define the function */
SWAPMOVE32(x,m,n) {
  t ← x ^ ( x >> n )
  t ← t & m
  t ← t ^ ( t << n )
  x ← t ^ x
  return x
}
```

- **gift.key.updstd rd, rs1**

```
x       ← GPR[rs1]

r       ←     ( ( x >> 12 ) & 0x0000000F )
r       ← r | ( ( x & 0x00000FFF ) <<  4 )
r       ← r | ( ( x >>  2 ) & 0x3FFF0000 )
r       ← r | ( ( x & 0x00030000 ) << 14 )

GPR[rd] ← r
```

- **gift.permbits.step rd, rs1, imm**

```
x       ← GPR[rs1]

r       ← SWAPMOVE32( x, 0x0A0A0A0A,  3 )
r       ← SWAPMOVE32( r, 0x00CC00CC,  6 )
r       ← SWAPMOVE32( r, 0x0000F0F0, 12 )
r       ← SWAPMOVE32( r, 0x000000FF, 24 )
r       ← r >>> imm

GPR[rd] ← r
```

7.3.5 Grain-128AEADv2

Submission overview. The Grain-128AEADv2 [HJM+21] submission specifies an eponymous AEAD algorithm. We focus on this, the only and therefore primary algorithm, and, more specifically therefore, a kernel represented by the keystream-generation function of the underlying Grain-128a stream cipher (see also [HJM07, AHJM11]).

Kernel overview. Grain-128a is based on (a variant of) the "original" stream cipher Grain, which was a candidate of the eSTREAM competition and selected for the final eSTREAM portfolio. The kernel is a function that computes a 32-bit word of the keystream using an internal state of a size of 256 bits. This state consists of a 128-bit Linear Feedback Shift Register (LFSR) and a 128-bit Nonlinear Feedback Shift Register (NFSR). The kernel consists of three major sub-functions: one to update the LFSR (called f function), and one to update the NFSR (called g function) and one to compute the 32-bit output word (called h function).

Implementation options. A naive implementation of the sub-functions to update the LFSR and NFSR consists of a large number of bit-level operations. It is therefore more efficient to implement the sub-functions such that they operate on 32-bit words, in which case the kernel basically consists of shifts, ANDs, and XORs. The kernel of Grain-128AEADv2 is simpler (and, therefore, faster) than the

kernel of the other NIST finalists, but this simplicity comes at the expense that the kernel is executed more often. Another specific property of this kernel is that the instructions provided by Zbkb/x (e.g., rotations) are not capable to reduce the execution time significantly.

ISE description. The kernel can be accelerated through a set of ten custom instructions, the most important of which is an instruction to extract a 32-bit word that lies at a certain position within a 64-bit word (held in two source registers). Furthermore, the set includes two instructions for the f function, three instructions for the g function, and four for the h function. Each of these instruction gets two state-words as input and computes the contribution of these two state-words to the result of f, g, and h, respectively. Finally, all the contributions have to be XORed together.

ISE design. The additional, more detailed material relating to the ISE design is shown below.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

00	imm	rs2	rs1	111	rd	0001011	grain.extr
0000	000	rs2	rs1	111	rd	0101011	grain.fln0
0001	000	rs2	rs1	111	rd	0101011	grain.fln2
0010	000	rs2	rs1	111	rd	0101011	grain.gnn0
0011	000	rs2	rs1	111	rd	0101011	grain.gnn1
0100	000	rs2	rs1	111	rd	0101011	grain.gnn2
0101	000	rs2	rs1	111	rd	0101011	grain.hnn0
0110	000	rs2	rs1	111	rd	0101011	grain.hnn1
0111	000	rs2	rs1	111	rd	0101011	grain.hnn2
1000	000	rs2	rs1	111	rd	0101011	grain.hln0

- **grain.extr rd, rs1, rs2, imm**

```
x_hi    ← GPR[rs1]
x_lo    ← GPR[rs2]
x       ← x_hi || x_lo
r       ← x >> imm
GPR[rd] ← r_{31.. 0}
```

- **grain.fln0 rd, rs1, rs2**

```
x_hi    ← GPR[rs1]
x_lo    ← GPR[rs2]
x       ← x_hi || x_lo
r       ← ( x_lo ) ^ ( x >> 7 )
GPR[rd] ← r_{31.. 0}
```

- **grain.fln2 rd, rs1, rs2**

```
x_hi    ← GPR[rs1]
x_lo    ← GPR[rs2]
x       ← x_hi || x_lo
r       ← ( x_hi ) ^ ( x >> 6 ) ^ ( x >> 17 )
GPR[rd] ← r_{31.. 0}
```

- **grain.gnn0 rd, rs1, rs2**

```
x_hi    ← GPR[rs1]
x_lo    ← GPR[rs2]
x       ← x_hi || x_lo
r       ← ( x_lo ) ^ ( x >> 26 ) ^ ( ( x >> 11 ) & ( x >> 13 ) ) ^
```

```
          ( ( x >> 17 ) & ( x >> 18 ) ) ^
          ( ( x >> 22 ) & ( x >> 24 ) & ( x >> 25 ) )
GPR[rd] ← r_{31.. 0}
```

- **grain.gnn1 rd, rs1, rs2**

```
x_hi    ← GPR[rs1]
x_lo    ← GPR[rs2]
x       ← x_hi || x_lo
r       ← ( x >> 24 ) ^ ( ( x >> 8 ) & ( x >> 16 ) )
GPR[rd] ← r_{31.. 0}
```

- **grain.gnn2 rd, rs1, rs2**

```
x_hi    ← GPR[rs1]
x_lo    ← GPR[rs2]
x       ← x_hi || x_lo
r       ← ( x_hi ) ^ ( x >> 27 ) ^ ( ( x >> 4 ) & ( x >> 20 ) ) ^
          ( ( x >> 24 ) & ( x >> 28 ) & ( x >> 29 ) & ( x >> 31 ) ) ^
          ( ( x >> 6 ) & ( x >> 14 ) & ( x >> 18 ) )
GPR[rd] ← r_{31.. 0}
```

- **grain.hnn0 rd, rs1, rs2**

```
x_hi    ← GPR[rs1]
x_lo    ← GPR[rs2]
x       ← x_hi || x_lo
r       ← ( x >> 2 ) ^ ( x >> 15 )
GPR[rd] ← r_{31.. 0}
```

- **grain.hnn1 rd, rs1, rs2**

```
x_hi    ← GPR[rs1]
x_lo    ← GPR[rs2]
x       ← x_hi || x_lo
r       ← ( x >> 4 ) ^ ( x >> 13 )
GPR[rd] ← r_{31.. 0}
```

- **grain.hnn2 rd, rs1, rs2**

```
x_hi    ← GPR[rs1]
x_lo    ← GPR[rs2]
x       ← x_hi || x_lo
r       ← ( x_lo ) ^ ( x >> 9 ) ^ ( x >> 25 )
GPR[rd] ← r_{31.. 0}
```

- **grain.hln0 rd, rs1, rs2**

```
x_hi    ← GPR[rs1]
x_lo    ← GPR[rs2]
x       ← x_hi || x_lo
r       ← ( x >> 13 ) & ( x >> 20 )
GPR[rd] ← r_{31.. 0}
```

7.3.6 ISAP

Submission overview. The ISAP [DEM+21] submission specifies a family of permutation-based AEAD algorithms [DEMS21, Section 2.5] consisting of ISAP-A-128A, ISAP-K-128A, ISAP-A-128, and ISAP-K-128. We focus on the primary algorithm ISAP-A-128A, and, more specifically therefore, a kernel represented by the ASCON-p permutation (see [DEMS21] and Section 7.3.2).

Kernel overview. The main distinguishing feature of the ISAP family is their built-in mode-level countermeasures against passive side-channel attacks. However, from a kernel perspective, the main instance Isap-A-128a uses exactly the same Ascon-p permutation as the Ascon family of AEAD algorithms. Isap-A-128a evaluates this permutation over either 1, 6, or 12 rounds, depending on the concrete (sub-)operation the permutation is part of. As already explained in Section 7.3.2, Ascon-p operates over a 320-bit state and consists of 1) a round-constant addition, 2) a substitution layer based on a bit-sliced 5-bit S-box, and 3) a linear layer performing XORs and rotations of 64-bit words.

Implementation options. Similar to Ascon, optimized implementation of Isap-A-128a for 32-bit platforms can take advantage of bit-interleaving to speed up the linear layer of the permutation. However, as explained in Section 7.3.2, bit-interleaving has actually a negative effect on the overall performance when the linear layer is accelerated through a small set of custom instructions. This is because an ISE-supported implementation of the linear layer always consists of only 10 instructions, regardless of whether bit-interleaving is applied or not, which means the conversions between bit-interleaved and normal representation actually slow down the execution.

ISE description. The ISE described in Section 7.3.2 for Ascon-p can re-used for Isap-A-128a.

ISE design. The ISE for ISAP is the same as for Ascon, which can be found in Section 7.3.2.

7.3.7 PHOTON-Beetle

Submission overview. The PHOTON-Beetle [BCD+21] submission specifies the AEAD algorithm family PHOTON-Beetle-AEAD [BCD+21, Section 3.2] and the hash function algorithm family PHOTON-Beetle-Hash [BCD+21, Section 3.3]. We focus on the primary algorithm PHOTON-Beetle-AEAD[128], and, more specifically therefore, a kernel represented by the PHOTON_{256} permutation (see also [GPP11]).

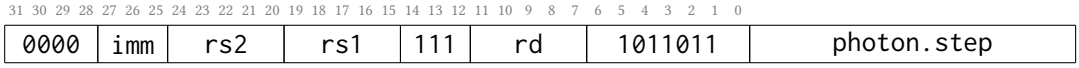
Kernel overview. The PHOTON_{256} permutation operates on an internal state of 256 bits, organized into an (8×8)-element matrix of 4-bit nibbles. The permutation is SPN-like, consisting of 12 rounds that each apply 4 round functions: these are `AddConstant`, `SubCells`, `ShiftRows`, and `MixColumnsSerial`. Per [GPP11, Section 2.2], the 4-bit PRESENT S-box is used in `SubCells`; in contrast to the AES `MixColumns` round function, `MixColumnsSerial` is specifically optimized to facilitate a serial application of operations in $\mathbb{F}_{2^4}$.

Implementation options. As reflected by the submission, 3 implementation techniques are applicable to PHOTON_{256}; in line with the similar SPN-like structure, and, at least to some extent, round functions, said techniques to analogous to those for AES. First, one can focus on online computation. Doing so mirrors the algorithmic description, whereby each round function is computed; this potentially includes arithmetic in $\mathbb{F}_{2^4}$, bar small look-up tables, e.g., for the S-box. Second, one can focus on offline pre-computation. Doing so mirrors the AES T-tables technique: the action of `SubCells` and `MixColumnsSerial` is pre-computed using a look-up table, careful indexing into which can also cater for `ShiftRows`. Third, and finally, one can use bit-slicing.

ISE description. The ISE design assumes a column-packed representation, and consists of 1 instruction: the second implementation strategy above is followed, but the look-up table that would normally be computed offline is instead computed online (in hardware). Given an input column, the instruction computes 1 nibble of the output column by applying `SubCells` and `MixColumnsSerial`. This allows 8 such instructions to compute an entire output column (including `AddConstant` and `ShiftRows`, the latter realized simply through indexing of the columns); 64 such instructions can be used to compute

an entire round. In a sense, this approach is similar to the design adopted by RISC-V [RVK22, Sections 2.4+2.5] for AES (as documented in [MNP+21], stemming from work by Nadehara et al. [NIK04] and Saarinen [Saa20]).

ISE design. The additional, more detailed material relating to the ISE design is shown below, where we let `SBOX` denote the 4-bit `PHOTON` S-box per [GPP11] and `GF2N_MUL` denote multiplication in the `PHOTON` finite field.



- **additional notation**

```
/* Define the look-up table */
M = { { 0x2, 0x4, 0x2, 0xB, 0x2, 0x8, 0x5, 0x6 },
      { 0xC, 0x9, 0x8, 0xD, 0x7, 0x7, 0x5, 0x2 },
      { 0x4, 0x4, 0xD, 0xD, 0x9, 0x4, 0xD, 0x9 },
      { 0x1, 0x6, 0x5, 0x1, 0xC, 0xD, 0xF, 0xE },
      { 0xF, 0xC, 0x9, 0xD, 0xE, 0x5, 0xE, 0xD },
      { 0x9, 0xE, 0x5, 0xF, 0x4, 0xC, 0x9, 0x6 },
      { 0xC, 0x2, 0x2, 0xA, 0x3, 0x1, 0x1, 0xE },
      { 0xF, 0x1, 0xD, 0xA, 0x5, 0xA, 0x2, 0x3 } }
```

- **photon.step rd, rs1, rs2, imm**

```
x       ← GPR[rs1]
y       ← GPR[rs2]

t       ← SBOX[ ( y >> ( 4 * imm ) ) & 0xF ]
r       ← 0

for( int i = 0; i < 8; i++ ) {
  r ← r | ( ( GF2N_MUL( M[ i ][ imm ], t ) ) << ( 4 * i ) )
}

r       ← r ^ x

GPR[rd] ← r
```

7.3.8 Romulus

Submission overview. The Romulus [GIK+21] submission specifies the AEAD algorithms Romulus-N [GIK+21, Section 2.4.3], Romulus-M [GIK+21, Section 2.4.4], and Romulus-T [GIK+21, Section 2.4.5], and the hash function algorithm Romulus-H [GIK+21, Section 2.4.6]. We focus on the primary algorithm Romulus-N, and, more specifically therefore, a kernel represented by the Skinny-128-384+ tweakable block cipher (which is a reduced round variant of Skinny-128-384; see also [BJK+16]).

Kernel overview. Skinny-128-384 is an SPN-based tweakable block cipher that uses a compact S-box, a very sparse diffusion layer, and a very light key schedule. Due to the high security margin of Skinny, the Romulus designers decided to use a Skinny variant with a reduced number of rounds, namely 40 instead of 56. Skinny-128-384 operates on an internal state of a size of 128 bits that can be viewed as a (4×4)-element matrix of bytes, similar to the AES. The round function is composed of five operations in the following order: `SubCells`, `AddConstants`, `AddRoundTweakey`, `ShiftRows`, and `MixColumns`. `SubCells` applies an 8-bit S-box, which can be efficiently implemented in hardware, to every byte of the state. The `AddConstants` operation XORs some round-dependent constants to the first column of the state. `AddRoundTweakey` extracts eight bytes from the tweakey state and XORs them to the state,

whereby the bytes are permuted and updated with simple LFSRs. ShiftRows rotates the bytes of the state row-wise to the right by 0, 1, 2, and 3 positions, similar to the ShiftRows transformation of the AES. Finally, MixColumns multiplies each byte-column of the state by a binary matrix.

Implementation options. The most efficient software implementations of Skinny-128-384 for 32-bit platforms are based on the fix-slicing technique, which can be seen as a special form of bit-slicing [AP20]. In this work, we considered both the straightforward implementation that uses a look-up table for S-box as well as the fix-slicing implementation.

ISE description. For the table-based implementation, the ISE design assumes a row-packed representation of the state matrix, and can be described as supporting 1) update and use of the round constant (which involves application of an LFSR), 2) update of the tweak key (which involves application of an LFSR), and 3) application of the round functions. Using a row-packed representation, MixColumns can be realized via a short sequence of XORs; this allows the latter aspect of the ISE to focus on the remaining, row-oriented round functions, i.e, SubCells, ShiftRows, and AddRoundTweakey. Application of SubCells across an entire packed row of the state matrix is rationalized by the low cost S-box design: even if 4 parallel S-box instances are used, the cost in terms of area is still low in relative terms. For the fix-slicing implementation, the ISE includes instructions for MixColumns, specific SWAPMOVE operations, and round key pre-computation (e.g., LFSR, key permutation, and key update).

ISE design for table-based implementation. The additional, more detailed material relating to the ISE design is shown below, where we let SBOX denote the 8-bit Skinny S-box per [BJK+16].

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0000	000	00000	rs1	110	rd	0001011	romulus.rc.upd.enc
0010	000	rs2	rs1	111	rd	0001011	romulus.rc.use.enc.0
0011	000	rs2	rs1	111	rd	0001011	romulus.rc.use.enc.1
0001	imm	rs2	rs1	111	rd	0101011	romulus.tk.upd.enc.0
0010	imm	rs2	rs1	111	rd	0101011	romulus.tk.upd.enc.1
0000	imm	rs2	rs1	111	rd	1011011	romulus.rstep.enc

- **additional notation**

```
/* Define the functions */
LFSR_RC( x ) {
  return x_4 || x_3 || x_2 || x_1 || x_0 || ( x_5 ^ x_4 ^ 1 )
}
LFSR_TK2( x ) {
  return x_6 || x_5 || x_4 || x_3 || x_2 || x_1 || x_0 || ( x_5 ^ x_7 )
}
LFSR_TK3( x ) {
  return ( x_6 ^ x_0 ) || x_7 || x_6 || x_5 || x_4 || x_3 || x_2 || x_1
}
```

- **romulus.rc.upd.enc rd, rs1**

```
x       ← GPR[rs1]
r       ← LFSR_RC( x )
GPR[rd] ← r
```

- **romulus.rc.use.enc.0 rd, rs1, rs2**

```
x       ← GPR[rs1]
```

```
y       ← GPR[rs2]
r       ← y ^ x_{3..0}
GPR[rd] ← r
```

- **romulus.rc.use.enc.1 rd, rs1, rs2**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
r       ← y ^ x_{6..4}
GPR[rd] ← r
```

- **romulus.tk.upd.enc.0 rd, rs1, rs2, imm**

```
x       ← GPR[rs1]
y       ← GPR[rs2]

if     ( imm == 1 ) {
  r ←           y_{15.. 8}   ||           x_{ 7.. 0}   ||
                y_{31..24}   ||           x_{15.. 8}
}
else if( imm == 2 ) {
  r ← LFSR_TK2( y_{15.. 8} ) || LFSR_TK2( x_{ 7.. 0} ) ||
      LFSR_TK2( y_{31..24} ) || LFSR_TK2( x_{15.. 8} )
}
else if( imm == 3 ) {
  r ← LFSR_TK3( y_{15.. 8} ) || LFSR_TK3( x_{ 7.. 0} ) ||
      LFSR_TK3( y_{31..24} ) || LFSR_TK3( x_{15.. 8} )
}

GPR[rd] ← r
```

- **romulus.tk.upd.enc.1 rd, rs1, rs2, imm**

```
x       ← GPR[rs1]
y       ← GPR[rs2]

if     ( imm == 1 ) {
  r ←           x_{31..24}   ||           y_{ 7.. 0}   ||
                y_{23..16}   ||           x_{23..16}
}
else if( imm == 2 ) {
  r ← LFSR_TK2( x_{31..24} ) || LFSR_TK2( y_{ 7.. 0} ) ||
      LFSR_TK2( y_{23..16} ) || LFSR_TK2( x_{23..16} )
}
else if( imm == 3 ) {
  r ← LFSR_TK3( x_{31..24} ) || LFSR_TK3( y_{ 7.. 0} ) ||
      LFSR_TK3( y_{23..16} ) || LFSR_TK3( x_{23..16} )
}

GPR[rd] ← r
```

- **romulus.rstep.enc rd, rs1, rs2, imm**

```
x       ← GPR[rs1]
y       ← GPR[rs2]

if     ( imm == 2 ) {
  y ← 2
}
else if( imm == 3 ) {
  y ← 0
}

t       ← SBOX[ x_{31..24} ] || SBOX[ x_{23..16} ] ||
          SBOX[ x_{15.. 8} ] || SBOX[ x_{ 7.. 0} ]

t       ← t ^ y
```

```
15
16 if     ( imm == 0 ) {
17   r ← t <<<  0
18 }
19 else if( imm == 1 ) {
20   r ← t <<<  8
21 }
22 else if( imm == 2 ) {
23   r ← t <<< 16
24 }
25 else if( imm == 3 ) {
26   r ← t <<< 24
27 }
28
29 GPR[rd] ← r
```

ISE design for fix-slicing implementation. The additional, more detailed material relating to the ISE design is shown below.

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17	16 15 14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
0000	imm	00000	rs1	110	rd	1111011	romulus.mixcolumns
0001	imm	rs2	rs1	111	rd	1111011	romulus.swapmove.x
0010	imm	rs2	rs1	111	rd	1111011	romulus.swapmove.y
0100	imm	00000	rs1	110	rd	0101011	romulus.permtk
0101	imm	00000	rs1	110	rd	0101011	romulus.tkupd.0
0110	imm	00000	rs1	110	rd	0101011	romulus.tkupd.1
0100	000	rs2	rs1	111	rd	0001011	romulus.lfsr2
0101	000	rs2	rs1	111	rd	0001011	romulus.lfsr3

- **additional notation**

```
1  /* Define the functions */
2  SWAPMOVE32_X(x,y,m,n) {
3    t ← y ^ ( x >> n )
4    t ← t & m
5    x ← x ^ ( t << n )
6
7    return x
8  }
9  SWAPMOVE32_Y(x,y,m,n) {
10   t ← y ^ ( x >> n )
11   t ← t & m
12   y ← y ^ ( t      )
13
14   return y
15 }
```

- **romulus.mixcolumns rd, rs1, imm**

```
1  x       ← GPR[rs1]
2
3  if      ( imm == 0 ) {
4    r ← x ^ ( ( ( x >>> 24 ) & 0x0C0C0C0C ) >>> 30 )
5    r ← r ^ ( ( ( r >>> 16 ) & 0xC0C0C0C0 ) >>>  4 )
6    r ← r ^ ( ( ( r >>>  8 ) & 0x0C0C0C0C ) >>>  2 )
7  }
8  else if ( imm == 1 ) {
9    r ← x ^ ( ( ( x >>> 16 ) & 0x30303030 ) >>> 30 )
10   r ← r ^ ( ( ( r        ) & 0x03030303 ) >>> 28 )
11   r ← r ^ ( ( ( r >>> 16 ) & 0x30303030 ) >>>  2 )
```

```
}
else if ( imm == 2 ) {
  r ← x ^ ( ( ( x >>>  8 ) & 0xC0C0C0C0 ) >>>  6 )
  r ← r ^ ( ( ( r >>> 16 ) & 0x0C0C0C0C ) >>> 28 )
  r ← r ^ ( ( ( r >>> 24 ) & 0xC0C0C0C0 ) >>>  2 )
}
else if ( imm == 3 ) {
  r ← x ^ ( ( ( x        ) & 0x03030303 ) >>> 30 )
  r ← r ^ ( ( ( r        ) & 0x30303030 ) >>>  4 )
  r ← r ^ ( ( ( r        ) & 0x03030303 ) >>> 26 )
}

GPR[rd] ← r
```

- **romulus.swapmove.x rd, rs1, rs2, imm**

```
x       ← GPR[rs1]
y       ← GPR[rs2]

if      ( imm == 0 ) {
  r ← SWAPMOVE32_X( x, y, 0x55555555, 1 )
}
else if ( imm == 1 ) {
  r ← SWAPMOVE32_X( x, y, 0x30303030, 2 )
}
else if ( imm == 2 ) {
  r ← SWAPMOVE32_X( x, y, 0x0C0C0C0C, 4 )
}
else if ( imm == 3 ) {
  r ← SWAPMOVE32_X( x, y, 0x03030303, 6 )
}
else if ( imm == 4 ) {
  r ← SWAPMOVE32_X( x, y, 0x0C0C0C0C, 2 )
}
else if ( imm == 5 ) {
  r ← SWAPMOVE32_X( x, y, 0x03030303, 4 )
}
else if ( imm == 6 ) {
  r ← SWAPMOVE32_X( x, y, 0x03030303, 2 )
}
else if ( imm == 7 ) {
  r ← SWAPMOVE32  ( x,    0x0A0A0A0A, 3 )
}

GPR[rd] ← r
```

- **romulus.swapmove.y rd, rs1, rs2, imm**

```
x       ← GPR[rs1]
y       ← GPR[rs2]

if      ( imm == 0 ) {
  r ← SWAPMOVE32_Y( x, y, 0x55555555, 1 )
}
else if ( imm == 1 ) {
  r ← SWAPMOVE32_Y( x, y, 0x30303030, 2 )
}
else if ( imm == 2 ) {
  r ← SWAPMOVE32_Y( x, y, 0x0C0C0C0C, 4 )
}
else if ( imm == 3 ) {
  r ← SWAPMOVE32_Y( x, y, 0x03030303, 6 )
}
else if ( imm == 4 ) {
  r ← SWAPMOVE32_Y( x, y, 0x0C0C0C0C, 2 )
}
else if ( imm == 5 ) {
  r ← SWAPMOVE32_Y( x, y, 0x03030303, 4 )
```

```
}
else if ( imm == 6 ) {
  r ← SWAPMOVE32_Y( x, y, 0x03030303, 2 )
}

GPR[rd] ← r
```

- **romulus.permtk rd, rs1, imm**

```
x       ← GPR[rs1]

if      ( imm == 0 ) {
  r ←     ( ( ( x >>> 14 ) & 0xCC00CC00 )        )
  r ← r | ( ( ( x         ) & 0x000000FF ) << 16 )
  r ← r | ( ( ( x         ) & 0xCC000000 ) >>  2 )
  r ← r | ( ( ( x         ) & 0x0033CC00 ) >>  8 )
  r ← r | ( ( ( x         ) & 0x00CC0000 ) >> 18 )
}
else if ( imm == 1 ) {
  r ←     ( ( ( x >>> 22 ) & 0xCC0000CC )        )
  r ← r | ( ( ( x >>> 16 ) & 0x3300CC00 )        )
  r ← r | ( ( ( x >>> 24 ) & 0x00CC3300 )        )
  r ← r | ( ( ( x         ) & 0x00CC00CC ) >>  2 )
}
else if ( imm == 2 ) {
  r ←     ( ( ( x >>>  6 ) & 0xCCCC0000 )        )
  r ← r | ( ( ( x >>> 24 ) & 0x330000CC )        )
  r ← r | ( ( ( x >>> 10 ) & 0x00003333 )        )
  r ← r | ( ( ( x          & 0x000000CC ) << 14 )
  r ← r | ( ( ( x          & 0x00003300 ) <<  2 )
}
else if ( imm == 3 ) {
  r ←     ( ( ( x >>> 24 ) & 0xCC000033 )        )
  r ← r | ( ( ( x >>>  8 ) & 0x33CC0000 )        )
  r ← r | ( ( ( x >>> 26 ) & 0x00333300 )        )
  r ← r | ( ( ( x         ) & 0x00333300 ) >>  6 )
}
else if ( imm == 4 ) {
  r ←     ( ( ( x >>>  8 ) & 0xCC330000 )        )
  r ← r | ( ( ( x >>> 26 ) & 0x33000033 )        )
  r ← r | ( ( ( x >>> 22 ) & 0x00CCCC00 )        )
  r ← r | ( ( ( x         ) & 0x00330000 ) >> 14 )
  r ← r | ( ( ( x         ) & 0x0000CC00 ) >>  2 )
}
else if ( imm == 5 ) {
  r ←     ( ( ( x >>>  8 ) & 0x0000CC33 )        )
  r ← r | ( ( ( x >>> 30 ) & 0x00CC00CC )        )
  r ← r | ( ( ( x >>> 10 ) & 0x33330000 )        )
  r ← r | ( ( ( x >>> 16 ) & 0xCC003300 )        )
}
else if ( imm == 6 ) {
  r ←     ( ( ( x >>> 24 ) & 0x0033CC00 )        )
  r ← r | ( ( ( x >>> 14 ) & 0x00CC0000 )        )
  r ← r | ( ( ( x >>> 30 ) & 0xCC000000 )        )
  r ← r | ( ( ( x >>> 16 ) & 0x000000FF )        )
  r ← r | ( ( ( x >>> 18 ) & 0x33003300 )        )
}

GPR[rd] ← r
```

- **romulus.tkupd.0 rd, rs1, imm**

```
x       ← GPR[rs1]

if      ( imm == 0 ) {
  r ←     ( ( x >>> 26 ) & 0xC3C3C3C3 )
}
else if ( imm == 1 ) {
```

```
  r ←     ( ( x >>> 16 ) & 0xF0F0F0F0 )
}
else if ( imm == 2 ) {
  r ←     ( ( x >>> 10 ) & 0xC3C3C3C3 )
}

GPR[rd] ← r
```

- **romulus.tkupd.1 rd, rs1, imm**

```
x       ← GPR[rs1]

if      ( imm == 0 ) {
  r ←     ( ( x >>> 28 ) & 0x03030303 )
  r ← r | ( ( x >>> 12 ) & 0x0C0C0C0C )
}
else if ( imm == 1 ) {
  r ←     ( ( x >>> 14 ) & 0x30303030 )
  r ← r | ( ( x >>>  6 ) & 0x0C0C0C0C )
}
else if ( imm == 2 ) {
  r ←     ( ( x >>> 12 ) & 0x03030303 )
  r ← r | ( ( x >>> 28 ) & 0x0C0C0C0C )
}
else if ( imm == 3 ) {
  r ←     ( ( x >>> 30 ) & 0x30303030 )
  r ← r | ( ( x >>> 22 ) & 0x0C0C0C0C )
}

GPR[rd] ← r
```

- **romulus.lfsr2 rd, rs1, rs2**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
r       ← x ^ ( ( y & 0xAAAAAAAA )      )
r       ← ( ( ( r      ) & 0xAAAAAAAA ) >> 1 ) |
          ( ( ( r << 1 ) & 0xAAAAAAAA )      )
GPR[rd] ← r
```

- **romulus.lfsr3 rd, rs1, rs2**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
r       ← x ^ ( ( y & 0xAAAAAAAA ) >> 1 )
r       ← ( ( ( r      ) & 0xAAAAAAAA ) >> 1 ) |
          ( ( ( r << 1 ) & 0xAAAAAAAA )      )
GPR[rd] ← r
```

7.3.9 Sparkle

Submission overview. The Sparkle [BBC+21b] submission specifies the AEAD algorithm family Schwaemm [BBC+21b, Section 2.3] and the hash function algorithm family Esch [BBC+21b, Section 2.2]. We focus on the primary algorithm Schwaemm256-128 and, more specifically therefore, a kernel represented by the Sparkle permutation [BBC+21b, Section 2.1] (see also [BBC+20b], noting underlying use of the Alzette [BBC+20a] ARX-box).

Kernel overview. The Sparkle permutation consists of three basic building blocks, namely 1) a non-linear layer that is composed of six parallel instances of the ARX-box Alzette, 2) a simple linear diffusion layer, 3) the addition of a step counter and round constant to the 384-bit state. Alzette can be seen as a small 64-bit block cipher that operates on two 32-bit words and performs three additions and four XORs whereby one of the operands is rotated by a fixed distance, as well as one ordinary addition and

four ordinary XORs. On the other hand, the linear layer is, in essence, a Feistel round with a linear Feistel function, followed by a swap of the left and right half of the state.

Implementation options. An ARM Cortex-M implementation of Alzette consists of only 12 instructions when exploiting the "free" rotation of the second operand. On the other hand, when Alzette is implemented using the base RV32GC instruction set, a total of 33 arithmetic/logical instruction are necessary, which can be reduced to 19 instructions when the bit-manipulation extension Zbkb is available. The linear layer consists of two rotations of 32-bit words (which are part of the so-called ℓ operation) and a number of `xor` and register-move (i.e., `mv`) instructions. Using the base-ISA, the linear layer consists of 32 instructions, among which are six `mv` instructions. However, these `mv` instructions can be avoided when the permutation is fully unrolled, thereby reducing the instruction count of the linear layer to 24. A further reduction by four instructions is possible when using the rotation instructions from Zbkb.

ISE description. There are two basic options for speeding up Alzette with the help of custom instructions. The first is to define instructions for operations of the form $x = x \oplus (y \ggg n)$ and $x = x + (y \ggg n)$, where x and y are two 32-bit words and n is a fixed rotation amount, which can be encoded as an immediate value. In this case, a single instance of Alzette consists of 12 instructions and is very similar to an ARM Cortex-M implementation. A more speed-optimized ISE would consist of two custom instructions, of which one computes the x word of the output and the other the y word. Each of these instructions can be encoded with two source register addresses, one destination register address, and an immediate value specifying one of six 32-bit constants. In this case, Alzette consists of only two instructions. The instruction count of the linear layer can be reduced from 24 to 16 with the help of a custom instruction for the ℓ operation.

ISE design. The additional, more detailed material relating to the ISE design is shown below.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0000010		rs2	rs1	111	rd	1111011	sparkle.ell
0000	imm	rs2	rs1	110	rd	1011011	sparkle.rcon
1000	imm	rs2	rs1	111	rd	1011011	sparkle.whole.enci.x
1001	imm	rs2	rs1	111	rd	1011011	sparkle.whole.enci.y

- **additional notation**

```
/* Define the look-up tables */
ROT_0 = { 31, 17,  0, 24 }
ROT_1 = { 24, 17, 31, 16 }
RCON  = { 0xB7E15162, 0xBF715880, 0x38B4DA56, 0x324E7738,
          0xBB1185EB, 0x4F7C7B57, 0xCFBFA1C8, 0xC2B3293D }

/* Define the function */
ELL( x ) {
  return ( x ^ ( x << 16 ) ) >>> 16
}
```

- **sparkle.ell rd, rs1, rs2**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
r       ← ELL( x ^ y )
GPR[rd] ← r
```

- **sparkle.rcon rd, rs1, imm**

```
x       ← GPR[rs1]
r       ← x   ^ RCON[imm]
GPR[rd] ← r
```

- **sparkle.whole.enci.x rd, rs1, rs2, imm**

```
xi      ← GPR[rs1]
yi      ← GPR[rs2]
ci      ← RCON[imm]
xi      ← xi + ( yi >>> 31 )
yi      ← yi ^ ( xi >>> 24 )
xi      ← xi ^   ci
xi      ← xi + ( yi >>> 17 )
yi      ← yi ^ ( xi >>> 17 )
xi      ← xi ^   ci
xi      ← xi + ( yi >>>  0 )
yi      ← yi ^ ( xi >>> 31 )
xi      ← xi ^   ci
xi      ← xi + ( yi >>> 24 )
yi      ← yi ^ ( xi >>> 16 )
xi      ← xi ^   ci
GPR[rd] ← xi
```

- **sparkle.whole.enci.y rd, rs1, rs2, imm**

```
xi      ← GPR[rs1]
yi      ← GPR[rs2]
ci      ← RCON[imm]
xi      ← xi + ( yi >>> 31 )
yi      ← yi ^ ( xi >>> 24 )
xi      ← xi ^   ci
xi      ← xi + ( yi >>> 17 )
yi      ← yi ^ ( xi >>> 17 )
xi      ← xi ^   ci
xi      ← xi + ( yi >>>  0 )
yi      ← yi ^ ( xi >>> 31 )
xi      ← xi ^   ci
xi      ← xi + ( yi >>> 24 )
yi      ← yi ^ ( xi >>> 16 )
xi      ← xi ^   ci
GPR[rd] ← yi
```

7.3.10 TinyJAMBU

Submission overview. The TinyJAMBU [WH21] submission specifies an eponymous AEAD algorithm family. We focus on the primary algorithm TinyJAMBU-128 [WH21, Section 3.3], and, more specifically, a kernel represented by the keyed permutation P_n, which is iterated either $n = 640$ times (P_{640}) or $n = 1024$ times (P_{1024}).

Kernel overview. The permutation P is based on a 128-bit non-linear feedback shift register whose feedback path consists of four bit-wise XORs and a bit-wise NAND, which is the only non-linear operation of TinyJAMBU. One can easily identify the state-update function as the most performance-critical operation; it gets besides the 128-bit state and the number of rounds also a key as input. However, TinyJAMBU does not involve a key-schedule. The permutation P_n distinguishes itself from the permutations of other finalists like Ascon, Sparkle, and Xoodyak by an extremely small state size the fact that it is keyed (i.e., P_n is a non-public permutation). Furthermore, the number of rounds is much higher, which is compensated by an extremely simple round function (basically just a shift of the 128-bit state along with five bit-operations).

Implementation options. On a 32-bit processor, it is possible to compute 32 instances of the permutation simultaneously, which means the XOR and NAND operations are performed on 32-bit words. One of them is a word of the state, one is a word from the key and the other four are extracted from the state at certain positions. The latter boils down to extracting a 32-bit word from two adjacent 32-bit state-words through an operation of the form $w = (S_i \gg n) \wedge (S_j \ll (32 - n))$.

ISE description. Extracting a 32-bit words from two state-words can be done with three native RV32GC instructions. However, this operation can be easily mapped to a custom instruction (which we call `fsri`) that reads two 32-bit words from registers and gets the position of the word to extract through an immediate value. Even though `fsri` saves only two instructions, it still improves the execution time of TinyJAMBU significantly since these word-extractions account for about 80% of the execution time of the state-update operation.

ISE design. The additional, more detailed material relating to the ISE design is shown below.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

00	00000	rs2	rs1	111	rd	1111011	jambu.fsr.15
00	00001	rs2	rs1	111	rd	1111011	jambu.fsr.6
00	00010	rs2	rs1	111	rd	1111011	jambu.fsr.21
00	00011	rs2	rs1	111	rd	1111011	jambu.fsr.27

- **jambu.fsr.15 rd, rs1, rs2**

```
x_hi    ← GPR[rs2]
x_lo    ← GPR[rs1]
r       ← ( x_hi || x_lo ) >>> 15
GPR[rd] ← r_{31.. 0}
```

- **jambu.fsr.6 rd, rs1, rs2**

```
x_hi    ← GPR[rs2]
x_lo    ← GPR[rs1]
r       ← ( x_hi || x_lo ) >>>  6
GPR[rd] ← r_{31.. 0}
```

- **jambu.fsr.21 rd, rs1, rs2**

```
x_hi    ← GPR[rs2]
x_lo    ← GPR[rs1]
r       ← ( x_hi || x_lo ) >>> 21
GPR[rd] ← r_{31.. 0}
```

- **jambu.fsr.27 rd, rs1, rs2**

```
x_hi    ← GPR[rs2]
x_lo    ← GPR[rs1]
r       ← ( x_hi || x_lo ) >>> 27
GPR[rd] ← r_{31.. 0}
```

7.3.11 Xoodyak

Submission overview. The Xoodyak [DHM+21] submission specifies an eponymous algorithm, which supports both AEAD and hash function modes. We focus on this, the only and therefore primary algorithm, and, more specifically therefore, a kernel represented by the Xoodoo[12] permutation (see also [DHAK18]).

Kernel overview. The state of the Xoodoo[12] permutation has the form of a (3×4)-element matrix of 32-bit words, which can be visualized via three horizontal 128-bit planes (one above the other), each consisting of four 32-bit lanes. It is also possible to view the 384-bit state as 128 columns of three bits lying upon another (i.e., each bit belongs to a different plane). As its name indicates, Xoodoo[12] executes 12 iterations of a round function consisting of five steps: a column-parity mixing layer θ, a non-linear layer χ, two plane-shifting layers (ρ_{west} and ρ_{east}) between them, and a round-constant addition. Both ρ layers move bits horizontally and perform lane-wise rotations of planes as well as rotations of lanes by 11, 1, and 8 bits to the left. On the other hand, in the parity-computation part of θ and in the χ layer, state-bits interact only vertically, i.e., within 3-bit columns. The θ layer mainly executes XORs and left-rotations by 5 and 14 bits. Finally, the non-linear layer χ applies a 3-bit S-box to each column of the state, which can be computed using logical ANDs, XORs, and bitwise complements.

Implementation options. An optimized implementation of Xoodoo[12] permutation on RV32IMAC was proposed in [CJL+20]. This implementation takes advantage of a technique known as lane complementing, which allows one to reduce the number of bitwise complements that have to be carried out in the χ transformation from 12 per round to three. However, this optimization is not necessary on our base ISA, due to the `andn` instruction provided by Zbkb. `andn` combines a logical AND with a bitwise complement of the second operand, which benefits the implementation of χ to be more straightforward and more efficient on our base ISA.

ISE description. When adhering to the requirements for custom instructions mentioned in Section 7.2, then the only opportunity to speed up Xoodoo[12] is the manipulation of the parity-plane (i.e., three 32-bit parity-lanes) through an operation of the form $e = (p \lll 5) \oplus (p \lll 14)$. We call the custom instruction implementing this operation `xorrol`.

ISE design. The additional, more detailed material relating to the ISE design is shown below.

31 30	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0
01	00000	rs2	rs1	111	rd	0101011

`xoodyak.xorrol`

- **`xoodyak.xorrol rd, rs1, rs2`**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
r       ← ( x <<< 5 ) ^ ( y <<< 14 )
GPR[rd] ← r
```

7.3.12 Discussion

Observations regarding ISA design.

- There are several algorithms (e.g., Sparkle) where operations of the form

$$\text{GPR}[rd] \leftarrow \text{GPR}[rs_1] \odot (\text{GPR}[rs_2] \boxdot imm)$$

for $\odot \in \{\oplus, +, -, \ldots\}$ and $\boxdot \in \{\ll, \gg, \lll, \ggg\}$ are useful. Consider, without loss of generality, an example operation where $\odot = \oplus$ and $\boxdot = \ll$ is realized using the base ISA by the 2-instruction sequence

```
slli rx, ry, imm
xor  rx, rx, rz
```

One could imagine two different approaches to improving this starting point. The arguably more CISC-like approach (see [CDPA16, Section V]) would be to add a dedicated "shift-then-XOR" instruction to the base ISA; more general-purpose instances of this same approach include the ARM Cortex-M "flexible second operand" mechanism. The arguably more RISC-like approach (see [CDPA16, Section VI]) would be to retain the original instructions (resp. micro-ops) only, but implement a mechanism by which they can be fused (or combined, into a macro-op). By using compressed instructions [RV19, Chapter 16], for example, one can express a similar operation as

```
1 c.slli ry, imm
2 c.xor  ry, rz
```

Celio et al. [CDPA16] argue that by fusing these 2 instructions in the micro-architecture front-end, the same (effective) instruction throughput is achieved as use of the 1 non-compressed, dedicated instruction, but, crucially, without "bloating" the base ISA. However, a micro-architecture which supports fusion is more complex as a result; for resource-constrained devices, support for *dynamic*, run-time fusion is potentially unattractive therefore. A conceptual alternative would be *static*, compile-time fusion. If there were a way to "merge" 2 compressed instructions into 1 non-compressed instruction, their fused semantics could be expressed at compile-time and executed by a less complex micro-architecture.

- There are several algorithms which use 32-bit (e.g., Sparkle) or 64-bit (e.g., Ascon) rotation. This fact relates to a more general challenge of selecting an n-bit natural word size for an algorithm: one could say that a larger n can be a positive for base ISAs with a large word size (e.g., allowing more effective use of the data-path) but a negative for base ISAs with a small word size (e.g., because n-bit operations need to be synthesized by a sequence of m-bit alternatives, for $m < n$), and vice versa. Put another way, choice of an n somewhat biases how efficient an implementation of the algorithm can be when using a given base ISA.

  The other dimension to this choice, however, is how well a particular ISA supports a particular n. There is precedent in RISC-V for supporting 32-bit operations when XLEN = 64 (e.g., `rorw` in Zbkb [RVK22, Section 3.26] and similar), but not 64-bit operations when XLEN = 32. Following a RISC-like design philosophy, the argument would likely be that the latter, e.g., 64-bit rotation, can and so therefore should be synthesized using a sequence of 32-bit instructions. That said, and although total orthogonality is clearly unrealistic, it seems there are some opportunities along similar lines. A pertinent example is a family of so-called funnel shift instructions, which appeared in drafts[3] of the B extension but not the ratified B (i.e., Zba, Zbb, Zbc, and Zbs) nor Zbkb extensions. Although counterarguments (e.g., their ternary, 4-address format) exist, one *could* view their omission as a missed opportunity: a general-purpose funnel shift eliminates the need for bit-interleaving (where relevant) without needing a further, special-purpose ISE.

Observations regarding ISE design.

- For some algorithms, an ISE design for RV32GC is harder to scale (or generalize) into one for RV64GC than for other algorithms. PHOTON-Beetle uses PHOTON_{256}, for example, which uses an (8×8)-element state matrix of 4-bit nibbles. Where XLEN = 32 it is possible to pack 1 column into each 32-bit word; where XLEN = 64, the natural generalization is to pack 2 columns into each 64-bit word. However, this natural generalization of the representation renders the associated implementation more difficult, e.g., with respect to the `ShiftRows` round function.

  On one hand, this does not seem a significant problem; it is *already* true of support for AES in RISC-V (cf. `aes32esi` versus `aes64es` in Zkne [RVK22, Section 2.5]), for example. On the other

[3] See, e.g., Section 2.9.3 of version 0.93 via https://github.com/riscv/riscv-bitmanip.

hand, however, one *could* also argue that scalability is an attractive property and so favour designs which facilitate it.

- There are several algorithms (e.g., Elephant and Romulus) where "small" n-bit LFSRs, for $n <$ XLEN, are used. Although the LFSR update is typically dominated by other components of a given algorithm, an associated ISE could plausibly offer incremental improvement over use of the base ISA alone; if it were parameterizable (e.g., with respect to the tap sequence), such an ISE could represent a somewhat general-purpose primitive.

- There are several algorithms (e.g., GIFT and Romulus) where the implementation technique of fix-slicing [ANP20, AP21] is applicable; this fact is specifically highlighted and explored by Adomnicai and Peyrin [AP20]. Where fix-slicing is applied, an implementation will often make use of a primitive termed `SWAPMOVE`. May et al. [MPC00, Section 3.1] are among the first[4] to define and make use of this primitive: the basic idea is that some bits in an operand x are swapped with some bits in another operand y, with n and m controlling *which* bits. As such, `SWAPMOVE` has 3 inputs of XLEN bits (x, y, and m), 1 input of $\lceil \log_2 \text{XLEN} \rceil$ bits (n), and 2 outputs of XLEN bits (x and y). In various ISE designs, we cope with the number and type of inputs and outputs through specialization, e.g., employing 1) a 1-operand variant that involves only x, and 2) a small, hard-coded set of n and m. Given a more general-purpose ISE for `SWAPMOVE` is more attractive, however, it seems useful to carefully explore the trade-off between general- and special-purpose. For example, through careful inter-algorithm analysis, it might be possible to identify a *somewhat* general-purpose set of n and m which afford a compact and so viable encoding.

Observations regarding algorithm design.

- For some algorithms, a change to the interface could plausibly yield more efficient implementations. PHOTON-Beetle uses PHOTON_{256} for example, which initializes an (8×8)-element state matrix of 4-bit nibbles from a 16-element array of 8-bit bytes using a row-major ordering. Use of a column-oriented representation of the state matrix can imply a significant conversation overhead therefore, which could be reduced by changing the interface to allow a column-major ordering (although doing so clearly then penalizes row-oriented representation in the same way).

- For some algorithms, a change to the parameterisation could plausibly yield more efficient implementations. PHOTON-Beetle, uses PHOTON_{256} for example, which, per [GPP11, Section 2.2], implies use of the 4-bit PRESENT S-box. A different parameterization is possible, however, which implies use of the 8-bit AES S-box: although reasonable counterarguments also exist, one could argue that opting for the latter will maximize overlap with existing ISEs and so minimize the additional hardware components required (e.g., by using an AES S-box shared with Zkne [RVK22, Section 2.5], if that extension were also supported).

7.4 Implementation

In the same way as the ISA, a given ISE design represents an interface between hardware and software. In this section we consider both sides of said interface, as defined in Section 7.3: Section 7.4.1 considers the hardware-oriented side, i.e., how the ISE is realized, then Section 7.4.2 considers the software-oriented side, i.e., how the ISE is utilized. Doing so shifts our focus from abstract design to concrete implementation, which then represents the basis for evaluation in Section 7.5.

[4] Their goal is efficient software implementation of permutations, such as those used by DES; they cite some prior art, e.g., noting "*[t]his technique is utilised in versions of DES available from the Internet (for example Eric Young's libdes)*".

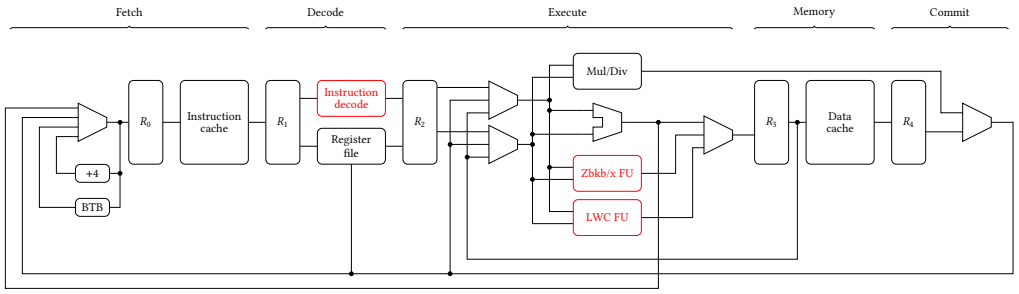


Figure 7.2: A block diagram highlighting features in our hardware implementation (e.g., integration of the Zbkb/x and LWC FUs) in red. Note that R_i denotes the i-th pipeline register, the component labelled Mul/Div supports multiplication and division, and a Branch Target Buffer (BTB) is shown toward the left-hand end of the pipeline.

7.4.1 Hardware

Host core. To realize each ISE design, we use the highly configurable, RISC-V compliant Rocket [AAB+16] host core. At a high level, the core executes instructions using a 5-stage, in-order pipeline; support is included within the core for a branch prediction mechanism, and in the wider system for a 16 kB instruction cache and a 16 kB data cache.

To support the execution of associated instructions, two modifications are made to the host core for each ISE design. First, an ISE-specific Functional Unit (FU) is integrated into the host core. At least two different approaches are possible, namely 1) an *internal* integration, where the FU is integrated directly into the pipeline, and 2) an *external* integration, which integrates the FU using the Rocket Custom Coprocessor (RoCC) [AAB+16, Section 4] interface. Although it requires less micro-architectural modification, using the RoCC interface locates the FU in the commit stage; this can degrade performance, due to inefficiency resulting from how forwarding is implemented. Our ISE designs are intended to permit single-cycle execution, which means the efficiency of forwarding is important. As such, we opt for the former approach, which allows location of the FU in the execute stage. Second, ISE-specific modifications are made to the instruction decoder, which, e.g., allow it to correctly provide input operands to the FU, control the FU so it performs the required computation, and accept output operands from the FU.

What we term the **unextended core**, i.e., Rocket as is, supports RV32GC only. In line with our definition of base ISA, we define the **base core**, i.e., a baseline for our work, as the unextended core *plus* additional[5] support for Zbkb and Zbkx. We then further extend this base core with support for an LWC-specific ISE, yielding what we term an **extended core**. Figure 7.2 illustrates the outcome, with our modifications highlighted in red. Note that the Zbkb/x FU realizes the Zbkb and Zbkx extensions, so is fixed across all ISE designs; the LWC FU realizes a given ISE design, so is different for each ISE design therefore. Also note that neither the Zbkb/x FU nor any of the LWC FU extend the existing critical path, so have no impact on the clock frequency. As such, and by design, the associated instructions have a 1 cycle execution latency.

LWC FU. The implementation of each LWC FU stems fairly directly from the associated ISE definition; each such definition uses pseudo-code which is intentionally similar to the openly available[6] Register Transfer Language (RTL) implementation used.

[5] Per Section 7.2, recall that although Zbkb and Zbkx represent extensions to RV32GC, for example, they form part of the defined base ISA. Viewed from the perspective of the unextended core, however, they represent unsupported extensions and thus need an associated implementation.

[6] See https://github.com/scarv/lwise.

Submission	Base implementation	Kernel implementation
Ascon	ascon128v12/ref	P[6|12]
Elephant	elephant160v2/ref	permutation
GIFT-COFB	giftcofb128v1/ref	giftb128
Grain-128AEADv2	grain128aeadv2/x64	grain_keystream32
ISAP	isapa128av20/ref	Ascon_Permute_Nrounds
PHOTON-Beetle	photonbeetleaead128rate128v1/ref	PHOTON_Permutation
Romulus	romulusn/[ref|fixslice_opt32]	Skinny[_128_384_plus_enc|128_384_plus]
Sparkle	schwaemm256128v2/opt	Sparkle_opt
TinyJAMBU	tinyjambu128v2/opt	state_update
Xoodyak	xoodyakround3/ref	Xoodoo_Permute_12rounds

Table 7.1: A per-algorithm summary of the base and kernel implementations.

Experimental platform. To produce an experimental platform which permits evaluation of, e.g., area and cycle-accurate execution latency, we make use of the SASEBO-GIII [HKSS12]: this includes two FPGAs, namely a Xilinx Kintex-7 (model `xc7k160tfbg676`) target FPGA, and a Xilinx Spartan-6 (model `xc6slx45`) support FPGA. We use the former exclusively, synthesizing stand-alone designs for it using Xilinx Vivado 2019.1; default synthesis settings are used, with no effort invested in synthesis or post-implementation optimization. The FPGA uses a 200 MHz external clock input, which is adjusted into a 50 MHz internal clock signal for use by the host core itself.

7.4.2 Software

High-level strategy. To utilize each ISE design, we developed a software implementation which can be executed by the associated extended core. For a given algorithm, we start with a base implementation. This is the source code[7] submitted for a given algorithm. The base implementation is used as is, with one exception: the submission for `Grain-128AEADv2` was ported from C++ to C, then adapted to cope with, e.g., assumptions around unaligned access to memory. Using appropriate C pre-processor directives, we make minor alterations to the base implementation so the kernel implementation is selectable between the original and a compatible replacement developed by us; Table 7.1 summarizes this information on a per-algorithm basis. We try to be consistent, using the most efficient parameterization of and implementation strategy for the base implementation which is compatible with our replacement kernel.

We view this approach as effective, in the sense it 1) allows focus on the kernel in question (so limits the volume of work involved), but, equally, 2) allows evaluation of the ISE design within a algorithm-wide rather than kernel-only context (so maximizes utility of the outcomes).

Low-level strategy. We use a RISC-V capable instance of the GNU tool-chain[8] to compile each software implementation. Each replacement kernel implementation is written in assembly language. Rather than modify the tool-chain, instances of the `.insn` directive are used to generate ISE-based instructions.

- Each replacement kernel implementation is captured in a single, leaf function; there is no further opportunity for, e.g., function inlining. We respect the ABI, in the sense that a function prologue and epilogue are careful to preserve and restore any callee-save registers by using the stack.
- Use of an ISE almost always reduces the number of instructions required to implement a replacement kernel, meaning loop overhead which stems from iteration, e.g., over rounds within it, can become more prominent.

[7] For submission `X`, use of a base implementation `Y` typically means use of source code located in `X/Implementations/crypto_aead/Y` within the submission archive `X.zip`.

[8] We use commit `b468107e701433e1caca3dbc8aef8d40e0c967ed` of `https://github.com/riscv/riscv-gnu-toolchain`, yielding, e.g., a working GCC whose version was 9.2.0.

To address this while providing at least some consistency, we support either partial, 2-fold unrolling or full, n-fold unrolling (for an appropriate n) of rounds within a replacement kernel. The former is often useful, for example, to avoid unnecessary copying of state output by an i-th round for use as input by the subsequent, $(i + 1)$-th round.

7.5 Evaluation

In this section, we present the result of evaluating our ISEs designs from both hardware and software perspectives. As a non-LWC comparison point, we consider an existing[9] ISE-supported implementation of AES-GCM [NIS07]. We attempt to align said implementation as closely as possible with the API used, by 1) "upgrading" it to support additional data, and 2) parameterising it using a 128-bit key. A set of results, limited to the relevant, extended ISA case only, are included for reference alongside those for LWC candidates.

Note throughout that, within the context of GIFT-COFB, we use FS and BS to refer to implementations based on fix-slicing and bit-slicing respectively; within the context of Romulus, we use FS and TB to refer to implementations based on fix-slicing and look-up tables respectively.

Hardware. Table 7.2 presents a summary of synthesis results for each ISE design. Reflecting the constraints in Section 7.3.1, note that all ISE design require combinational logic only, i.e., no state, so we report the number of FPGA Look-up Tables (LUTs) only. We measure (cumulative) overhead relative to the unextended core alone, and so exclude the wider system: doing so seems more representative, in that, e.g., the caches, would dominate otherwise. For example, the ISE for Sparkle (resp. TinyJAMBU) demands the most (resp. least) area: implementation of the Zbkb/x and LWC FUs produce a 14% and 10% (resp. 3%) overhead respectively, meaning 24% (resp. 17%) cumulative versus the unextended core.

For comparison, the ISE-supported implementation of AES-GCM makes use of Zbkc (for carryless multiplication) [RVK22, Section 2.2], Zbnd (for AES decryption) [RVK22, Section 2.4] and Zbne (for AES encryption) [RVK22, Section 2.5]. Our synthesis results show implementation of these extensions requires 567 additional LUTs, meaning an overhead of 31% cumulative versus the unextended core.

Software: kernel. Table 7.3 presents a summary of low-level results, focusing on the kernels in isolation. For each kernel, we report both absolute results i.e., execution latency (measured in clock cycles) and instruction footprint (measured in bytes), *and* relative results i.e., increase/decrease factor versus use of the base ISA alone. Note that for some kernels, e.g., GIFT and Romulus, we use auxiliary functions relating to pre-computation of round keys. For clarity, and because our ISEs can be used within them, we include these in addition to the kernel itself. For comparison, single-block encryption via `aes128_enc_ecb_rvk32` (resp. decryption via `aes128_dec_ecb_rvk32`) using the ISE-supported implementation of AES-GCM requires 324 (resp. 321) cycles; the encryption key schedule via `aes128_enc_key_rvk32` (resp. decryption key schedule via `aes128_dec_key_rvk32`) requires 264 (resp. 719) cycles; the GHASH function (dominated by a multiplication in $\mathbb{F}_{2^{128}}$) via `ghash_mul_rv32` requires 135 cycles.

Software: API. Table 7.4, Table 7.5, and Table 7.6 present a summary of high-level results, focusing on the kernels in context, i.e., as invoked via the API using the `aead_encrypt` and `aead_decrypt` functions for a 16, 128, and 1024 byte plaintext (resp. ciphertext) respectively. This is important, because one kernel may represent a different proportion of the associated algorithm than another, and thus yield different overall improvements. We consider a range of cases, constrained such that the associated data and plaintext/ciphertext lengths are equal: counterarguments clearly exist (e.g., one might

[9] https://github.com/rvkrypto/rvkrypto-fips.

Submission	Unextended core: RV32GC	Base core: RV32GC + Zbkb/x	Extended core: RV32GC + Zbkb/x + ISE
Ascon	3303 (1.000×)	3764 (1.140×)	4234 (1.282×)
Elephant			3938 (1.192×)
GIFT-COFB (BS)			3906 (1.183×)
GIFT-COFB (FS)			4370 (1.323×)
Grain-128AEADv2			4271 (1.293×)
ISAP			4234 (1.282×)
PHOTON-Beetle			3892 (1.178×)
Romulus (TB)			3998 (1.210×)
Romulus (FS)			4205 (1.273×)
Sparkle			4097 (1.240×)
TinyJAMBU			3863 (1.170×)
Xoodyak			3814 (1.155×)
AES-GCM			4331 (1.311×)

Table 7.2: Hardware-oriented evaluation, i.e., realization of each ISE design: area measured in FPGA LUTs (plus increase/decrease factor versus unextended core in parentheses).

Submission	Kernel	Metric	Replacement kernel implementation: Base ISA: RV32GC + Zbkb/x	Replacement kernel implementation: Extended ISA: RV32GC + Zbkb/x + ISE
Ascon	P6	latency	700 (1.00×)	280 (2.50×)
		footprint	2718 (1.00×)	1050 (2.59×)
Elephant	permutation	latency	15804 (1.00×)	1944 (8.13×)
		footprint	25662 (1.00×)	7702 (3.33×)
GIFT-COFB (BS)	giftb128	latency	1481 (1.00×)	842 (1.76×)
		footprint	5770 (1.00×)	3210 (1.80×)
GIFT-COFB (FS)	giftb128	latency	1386 (1.00×)	972 (1.43×)
		footprint	4888 (1.00×)	3412 (1.43×)
	precompute_rkeys	latency	1306 (1.00×)	251 (5.20×)
		footprint	4830 (1.00×)	768 (6.29×)
Grain-128AEADv2	grain_keystream32	latency	235 (1.00×)	86 (2.73×)
		footprint	858 (1.00×)	262 (3.27×)
ISAP	Ascon_Permute_Nrounds	latency	736 (1.00×)	316 (2.33×)
		footprint	5980 (1.00×)	2364 (2.53×)
PHOTON-Beetle	PHOTON_Permutation	latency	67035 (1.00×)	1473 (45.51×)
		footprint	82486 (1.00×)	3466 (23.80×)
Romulus (TB)	Skinny_128_384_plus_enc	latency	14268 (1.00×)	1502 (9.50×)
		footprint	23508 (1.00×)	4612 (5.10×)
Romulus (FS)	Skinny128_384_plus	latency	6208 (1.00×)	2156 (2.88×)
		footprint	17402 (1.00×)	7274 (2.39×)
	precompute_rtk1	latency	867 (1.00×)	200 (4.34×)
		footprint	2814 (1.00×)	610 (4.61×)
	precompute_rtk2_3	latency	3402 (1.00×)	1557 (2.18×)
		footprint	11290 (1.00×)	5186 (2.18×)
Sparkle	Sparkle_opt	latency	1647 (1.00×)	525 (3.14×)
		footprint	5908 (1.00×)	1816 (3.25×)
TinyJAMBU	state_update (P1024)	latency	575 (1.00×)	319 (1.80×)
		footprint	2208 (1.00×)	1184 (1.86×)
Xoodyak	Xoodoo_Permute_12rounds	latency	873 (1.00×)	777 (1.12×)
		footprint	3394 (1.00×)	3010 (1.13×)
AES-GCM	aes128_enc_ecb_rvk32	latency		324
		footprint		556
	aes128_dec_ecb_rvk32	latency		321
		footprint		570
	aes128_enc_key_rvk32	latency		264
		footprint		266
	aes128_dec_key_rvk32	latency		719
		footprint		348
	ghash_mul_rv32	latency		135
		footprint		252

Table 7.3: Software-oriented evaluation, i.e., utilization of each ISE design: latency measured in clock cycles and instruction footprint measured in bytes (plus increase/decrease factor versus base ISA in parentheses) for direct kernel use.

Submission	Functionality	Original kernel implementation	Replacement kernel implementation	
		Unextended ISA: RV32GC	Base ISA: RV32GC + Zbkb/x	Extended ISA: RV32GC + Zbkb/x + ISE
Ascon	aead_encrypt	14801 (1.00×)	7839 (1.89×)	4059 (3.65×)
	aead_decrypt	14523 (1.00×)	7862 (1.85×)	4083 (3.56×)
Elephant	aead_encrypt	3487575 (1.00×)	87596 (39.81×)	14209 (245.45×)
	aead_decrypt	3487689 (1.00×)	87608 (39.81×)	14262 (244.54×)
GIFT-COFB (BS)	aead_encrypt	118062 (1.00×)	6957 (16.97×)	5082 (23.23×)
	aead_decrypt	118058 (1.00×)	6926 (17.05×)	5050 (23.38×)
GIFT-COFB (FS)	aead_encrypt	118062 (1.00×)	7957 (14.84×)	5664 (20.84×)
	aead_decrypt	118058 (1.00×)	7919 (14.91×)	5626 (20.98×)
Grain-128AEADv2	aead_encrypt	15471 (1.00×)	15025 (1.03×)	9962 (1.55×)
	aead_decrypt	15389 (1.00×)	14988 (1.03×)	9917 (1.55×)
ISAP	aead_encrypt	374476 (1.00×)	74480 (5.03×)	45521 (8.23×)
	aead_decrypt	198129 (1.00×)	42721 (4.64×)	25305 (7.83×)
PHOTON-Beetle	aead_encrypt	1407143 (1.00×)	203088 (6.93×)	5224 (269.36×)
	aead_decrypt	1407742 (1.00×)	203254 (6.93×)	5227 (269.32×)
Romulus (TB)	aead_encrypt	161068 (1.00×)	33293 (4.84×)	5287 (30.46×)
	aead_decrypt	161103 (1.00×)	33453 (4.82×)	5318 (30.29×)
Romulus (FS)	aead_encrypt	29686 (1.00×)	36613 (0.81×)	7104 (4.18×)
	aead_decrypt	30093 (1.00×)	36458 (0.83×)	7186 (4.19×)
Sparkle	aead_encrypt	13141 (1.00×)	5829 (2.25×)	2424 (5.42×)
	aead_decrypt	13166 (1.00×)	5818 (2.26×)	2449 (5.38×)
TinyJAMBU	aead_encrypt	7908 (1.00×)	6690 (1.18×)	3891 (2.03×)
	aead_decrypt	7978 (1.00×)	6761 (1.18×)	3951 (2.02×)
Xoodyak	aead_encrypt	57766 (1.00×)	4191 (13.78×)	3921 (14.73×)
	aead_decrypt	57775 (1.00×)	4200 (13.76×)	3905 (14.80×)
AES-GCM	aes128_enc_gcm			2144
	aes128_dec_vfy_gcm			2309

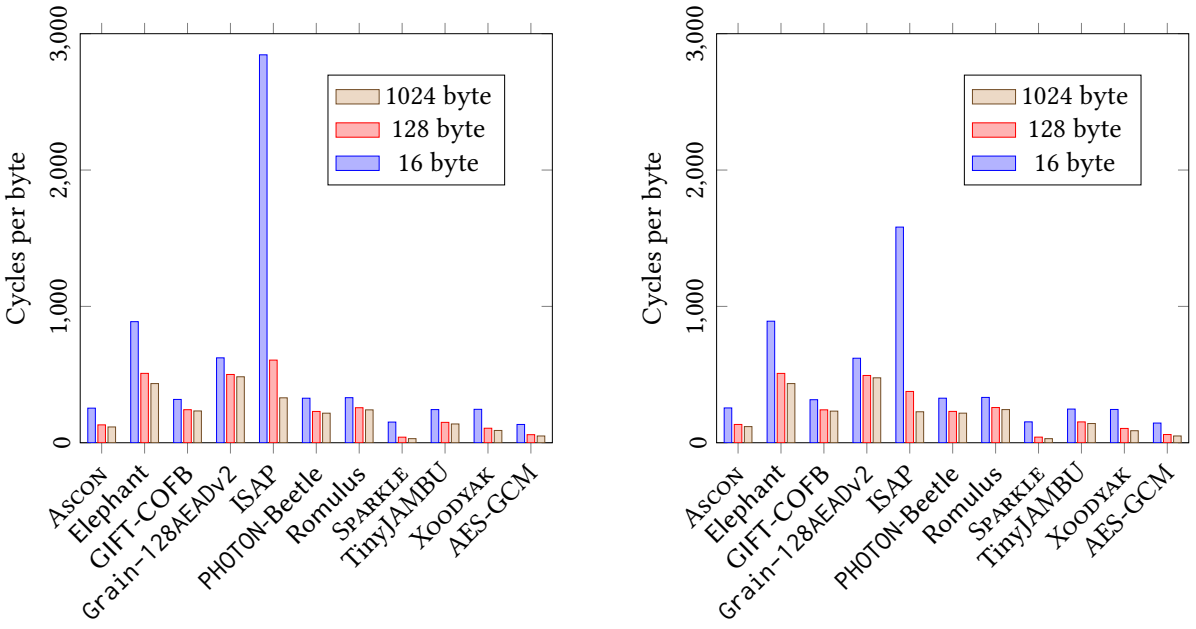
Table 7.4: Software-oriented evaluation, i.e., utilization of each ISE design: latency measured in clock cycles (plus increase/decrease factor versus unextended ISA in parentheses) for indirect kernel use via AEAD API (with 16 B plaintext, ciphertext, and associated data).

Submission	Functionality	Original kernel implementation	Replacement kernel implementation	
		Unextended ISA: RV32GC	Base ISA: RV32GC + Zbkb/x	Extended ISA: RV32GC + Zbkb/x + ISE
Ascon	aead_encrypt	43005 (1.00×)	32316 (1.33×)	16775 (2.56×)
	aead_decrypt	43414 (1.00×)	32694 (1.33×)	17159 (2.53×)
Elephant	aead_encrypt	16044010 (1.00×)	401543 (39.96×)	65118 (246.38×)
	aead_decrypt	16044075 (1.00×)	402787 (39.83×)	65079 (246.53×)
GIFT-COFB (BS)	aead_encrypt	687611 (1.00×)	42048 (16.35×)	31018 (22.17×)
	aead_decrypt	687543 (1.00×)	42093 (16.33×)	30887 (22.26×)
GIFT-COFB (FS)	aead_encrypt	687611 (1.00×)	41884 (16.42×)	33763 (20.36×)
	aead_decrypt	687543 (1.00×)	41749 (16.47×)	33642 (20.44×)
Grain-128AEADv2	aead_encrypt	87682 (1.00×)	85826 (1.02×)	64083 (1.37×)
	aead_decrypt	86656 (1.00×)	84897 (1.02×)	63148 (1.37×)
ISAP	aead_encrypt	489529 (1.00×)	135851 (3.60×)	77577 (6.31×)
	aead_decrypt	285242 (1.00×)	88894 (3.21×)	48138 (5.93×)
PHOTON-Beetle	aead_encrypt	8065027 (1.00×)	1149521 (7.02×)	29372 (274.58×)
	aead_decrypt	8063672 (1.00×)	1150013 (7.01×)	29407 (274.21×)
Romulus (TB)	aead_encrypt	1018364 (1.00×)	213180 (4.78×)	32880 (30.97×)
	aead_decrypt	1017990 (1.00×)	213444 (4.77×)	33049 (30.80×)
Romulus (FS)	aead_encrypt	177043 (1.00×)	203476 (0.87×)	40351 (4.39×)
	aead_decrypt	177326 (1.00×)	203444 (0.87×)	41257 (4.30×)
Sparkle	aead_encrypt	30033 (1.00×)	12883 (2.33×)	5218 (5.76×)
	aead_decrypt	30053 (1.00×)	12910 (2.33×)	5268 (5.70×)
TinyJAMBU	aead_encrypt	39851 (1.00×)	33574 (1.19×)	19118 (2.08×)
	aead_decrypt	40432 (1.00×)	34033 (1.19×)	19562 (2.07×)
Xoodyak	aead_encrypt	192338 (1.00×)	14579 (13.19×)	13616 (14.13×)
	aead_decrypt	192149 (1.00×)	14397 (13.35×)	13429 (14.31×)
AES-GCM	aes128_enc_gcm			7566
	aes128_dec_vfy_gcm			7716

Table 7.5: Software-oriented evaluation, i.e., utilization of each ISE design: latency measured in clock cycles (plus increase/decrease factor versus unextended ISA in parentheses) for indirect kernel use via AEAD API (with 128 B plaintext, ciphertext, and associated data).

Submission	Functionality	Original kernel implementation	Replacement kernel implementation	
		Unextended ISA: RV32GC	Base ISA: RV32GC + Zbkb/x	Extended ISA: RV32GC + Zbkb/x + ISE
Ascon	aead_encrypt	270239 (1.00×)	228119 (1.18×)	118500 (2.28×)
	aead_decrypt	271095 (1.00×)	230828 (1.17×)	121209 (2.24×)
Elephant	aead_encrypt	109520728 (1.00×)	2749081 (39.84×)	444374 (246.46×)
	aead_decrypt	109520760 (1.00×)	2746736 (39.87×)	444425 (246.43×)
GIFT-COFB (BS)	aead_encrypt	5221431 (1.00×)	322059 (16.21×)	238677 (21.88×)
	aead_decrypt	5220757 (1.00×)	322841 (16.17×)	237408 (21.99×)
GIFT-COFB (FS)	aead_encrypt	5221431 (1.00×)	312881 (16.69×)	258136 (20.23×)
	aead_decrypt	5220757 (1.00×)	312008 (16.73×)	257072 (20.31×)
Grain-128AEADv2	aead_encrypt	663938 (1.00×)	650688 (1.02×)	495442 (1.34×)
	aead_decrypt	655304 (1.00×)	642831 (1.02×)	487569 (1.34×)
ISAP	aead_encrypt	1504643 (1.00×)	631668 (2.38×)	337357 (4.46×)
	aead_decrypt	1029262 (1.00×)	461594 (2.23×)	232652 (4.42×)
PHOTON-Beetle	aead_encrypt	61215512 (1.00×)	8718676 (7.02×)	221919 (275.85×)
	aead_decrypt	61215428 (1.00×)	8717466 (7.02×)	222088 (275.64×)
Romulus (TB)	aead_encrypt	7587976 (1.00×)	1600969 (4.74×)	246905 (30.73×)
	aead_decrypt	7579477 (1.00×)	1605325 (4.72×)	249031 (30.44×)
Romulus (FS)	aead_encrypt	1282828 (1.00×)	1442806 (0.89×)	286404 (4.48×)
	aead_decrypt	1287633 (1.00×)	1441150 (0.89×)	294374 (4.37×)
Sparkle	aead_encrypt	185179 (1.00×)	78316 (2.36×)	30688 (6.03×)
	aead_decrypt	185202 (1.00×)	78346 (2.36×)	30720 (6.03×)
TinyJAMBU	aead_encrypt	295003 (1.00×)	248622 (1.19×)	140980 (2.09×)
	aead_decrypt	299601 (1.00×)	251993 (1.19×)	144338 (2.08×)
Xoodyak	aead_encrypt	1307532 (1.00×)	98574 (13.26×)	92139 (14.19×)
	aead_decrypt	1306193 (1.00×)	96744 (13.50×)	90319 (14.46×)
AES-GCM	aes128_enc_gcm			50742
	aes128_dec_vfy_gcm			50896

Table 7.6: Software-oriented evaluation, i.e., utilization of each ISE design: latency measured in clock cycles (plus increase/decrease factor versus unextended ISA in parentheses) for indirect kernel use via AEAD API (with 1024 B plaintext, ciphertext, and associated data).



(a) Encryption using `aead_encrypt`.

(b) Decryption using `aead_decrypt`.

Figure 7.3: Two graphs summarizing the data in Table 7.4, Table 7.5, and Table 7.6, with respect to encryption (left) and decryption (right): for each algorithm, we select the most efficient ISE variant (with respect to execution latency) and plot the normalized cycles per byte across all parameterizations considered (i.e., 16 B, 128 B, and 1024 B plaintext, ciphertext, and associated data).

expect common use-cases to require a short(er), fixed length associated data, and a longer, variable length plaintext/cipher), but adopting this approach aligns with the NIST micro-controller benchmarking framework[10] and so allows easier comparison of results. For comparison, multi-block encryption via `aes128_enc_gcm` (resp. decryption via `aes128_dec_vfy_gcm`) using the ISE-supported implementation of AES-GCM requires 2144, 7566, and 50742 (resp. 2309, 7716, and 50896) cycles for a 16, 128, and 1024 byte plaintext (resp. ciphertext).

Finally, Figure 7.3 presents a similar summary of the data to that used by NIST: for each algorithm, we select the most efficient ISE variant (with respect to execution latency) and plot the normalised cycles per byte across all parameterisations considered (i.e., 16 B, 128 B, and 1024 B plaintext, ciphertext, and associated data). As well as more clearly illustrating relative execution latency, including for the AES-GCM case, the graphs highlight cases where the overhead of initialization is more (resp. less) effectively amortized for large (resp. small) inputs.

The results for the ISE-supported kernels show that the more hardware-oriented designs (e.g., Elephant, PHOTON-Beetle, Romulus (TB)) are generally accelerated by a larger extent than the more software-oriented designs, such as Ascon, Sparkle, and Xoodyak, which were already relatively efficient with only the base-ISA. Among the latter three algorithms, Sparkle achieves a much higher speed-up than Xoodyak, which is mainly because the ARX-box Alzette can be implemented with only two custom instructions since it operates on 64-bit parts of the state (i.e., two 32-bit words). On the other hand, Xoodyak is not particularly well-suited for ISE because it does not contain many operations that can be mapped to custom instructions with two source registers and one destination register.

An additional benefit of the ISE-supported implementations is their significantly smaller code size, which is mainly due to the reduced footprint of the kernels. Such size reductions are often downplayed and only seen as a minor side benefit of ISE, but such a view neglects the fact that a size reduction can yield a non-negligible reduction of execution time on processors with a small instruction cache. For example, according to Table 7.3, the base-ISA implementations of the kernels of Elephant, PHOTON-Beetle, and Romulus (TB) have a footprint of more than 16 kB and exceed the instruction-cache size of our Rocket core, thereby slowing down the execution due to cache misses. On the other hand, all ISE-supported kernels fit conveniently into the instruction cache.

Comparison with related work: hardware. Strictly limited to cases based on RISC-V, and presented in chronological order, various elements of related work yield useful comparison points.

Tehrani et al. [TGSD20] describe an ISE for RV32 to support a range of lightweight, 64-bit block ciphers including GIFT-64-128 and Skinny-64-128, implementing and evaluating it using the VexRiscv core. First, they support computation of the substitution layer using a general-purpose instruction for nibble-wise table look-up; doing so is achieved by capturing the table (i.e., S-box) in 3 CSRs, and then applying it nibble-wise to a 32-bit input word supplied in GPR$[rs_1]$. Second, they support computation of the permutation layer. For GIFT-64-128 this takes the form of a special-purpose instruction, whereas for Skinny-64-128, a general-purpose instruction for nibble-wise matrix-vector multiplication is used; doing so is achieved by capturing a (constant) matrix in 8 CSRs, then applying it to a 64-bit input vector supplied in GPR$[rs_1]$ and GPR$[rs_2]$ (with two instructions required to compute the most- and least-significant 32-bit half of the result). We do not present a comparison with this work, because the ISE cannot be used[11] for either GIFT-128-128 or Skinny-128-384+ so is not applicable to GIFT-COFB or Romulus.

Altınay and Örs [AO21] describe an ISE for RV32 to support Ascon-p, implementing and evaluating it using the `spike` instruction set simulator. Their ISE includes two instructions. First, they

[10] See, e.g., https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking, and results in [TMC+21, Section 4 + Appendix A]: note that although the data format allows "x *bytes of associated data and* y *bytes of message*", the data itself has x = y in all cases.

[11] This is due to, e.g., the diffing substitution and permutation layers used, a fact which stems from the different block size (per [BPP+17, Section 2] and [BJK+16, Section 2]).

support general-purpose rotation; similar instructions are now available via the standard B (bit manipulation) [RVB21, Section 1.3] and K (cryptography) [RVK22, Section 2.1] extensions. Second, they support special-purpose computation of the S-box. Their instruction for doing so is CISC-like, in the sense it operates on data resident in memory: using an input register address rs_1, it loads five 32-bit inputs $x_i \leftarrow \mathsf{MEM}[\mathsf{GPR}[rs_1] + 4 \cdot i]^4$, applies the S-box to produce outputs r_i from the inputs x_i, then stores five 32-bit outputs $\mathsf{MEM}[\mathsf{GPR}[rs_1] + 4 \cdot i]^4 \leftarrow r_i$, where $0 \leq i < 5$ throughout. We do not present a comparison with this work, because 1) the ISE falls outside our constraints as outlined in Section 7.3.1, and, moreover, 2) no non-simulated evaluation results (i.e., area overhead, and cycle accurate execution latency) are available for it.

Steinegger and Primas [SP21] describe an ISE for RV32 to support Ascon-p, implementing and evaluating it using the RI5CY core. Their ISE includes one instruction, which essentially supports computation of an entire Ascon-p round in hardware. Implementation therefore demands tight integration with the core (e.g., using 10 hard-wired general-purpose registers to store the state), which, although delivering performance, arguably renders it more akin to a tightly-coupled accelerator than traditional ISE. Although the ISE falls outside our constraints as outlined in Section 7.3.1, it *does* represent a competitive trade-off: modulo differences with respect to the core used, [SP21, Table 1 + Section 4] demonstrate that a factor of 1.1 area overhead permits a significant, factor of 50 improvement in execution latency for Ascon. For certain use-cases, this trade-off can be argued as more attractive than one based on a more hardware-oriented (e.g., purely using an IP core) or more software-oriented (i.e., using a more tightly constrained ISE, as in our work) alternative.

Comparison with related work: software. Strictly limited to cases based on RISC-V, and presented in chronological order, various elements of related work yield useful comparison points.

Jellema [Jel19] presents an optimized implementation of Ascon, based on use of an E31 (supporting RV32IMAC) core; [Jel19, Figure 10] suggests a measured $6 \cdot 118 = 708$ cycle execution latency for the 6-round Ascon-p permutation. Modulo the different core, this can be compared with the base ISA and extended ISA columns of Table 7.3, where we measure 700 and 280 cycles respectively. At face value one might expect use of Zbkb/x to offer greater improvement, but in fact this result is expected: although we can use `andn` and `orn` within the substitution layer, we cannot use `rol` or `ror` within the diffusion layer (because XLEN = 32, so 64-bit rotation is not supported). Alternatively, [Jel19, Figure 11] suggests a measured 552076 cycle execution latency for the encryption of a 4096 byte plaintext and (inferred) 0 byte associated data; for this parameterization, our implementation takes 479764 cycles using the base ISA or 263043 cycles using the extended ISA, i.e., the LWC-specific ISE.

Lemmen [Lem20] presents an optimized implementation of Elephant, based on use of an E31 (supporting RV32IMAC) core. We do not present a comparison with this work, because it focuses on the non-primary parameterization Elephant-Keccak-f[200] so falls outside our scope.

Campos et al. [CJL+20] present a limited study of LWC algorithms, with the goal of assessing the impact of selecting assembly language versus C for their implementation. Per [CJL+20, Section 2], their work is based on use of an E31 (supporting RV32IMAC) or VexRiscv (supporting RV32IM) core; we ignore use of the riscvOVPsim simulator, because, as they explain, it may not produce representative results. Modulo the different core, can be compared with the base ISA and extended ISA columns of Table 7.3. For Ascon, [CJL+20, Table 7] suggests a measured 750 cycle execution latency for the 6-round Ascon-p permutation; per Table 7.3, use of Zbkb/x means our implementation takes 700 cycles, or 280 with an LWC-specific ISE. For Sparkle, [CJL+20, Section 3.2] suggests an approximated 1708 cycle execution latency for the Sparkle-384 permutation; per Table 7.3, use of Zbkb/x means our implementation takes 1647 cycles, or 525 with an LWC-specific ISE. For Xoodyak, [CJL+20, Section 3.2] suggests an approximated 1596 cycle execution latency for the 12-round Xoodoo permutation; per Table 7.3, use of Zbkb/x means our implementation takes 873 cycles, or 777 with an LWC-specific ISE.

Renner et al. [RPM20] present a hardware-in-the-loop benchmarking framework for the LWC pro-

cess; since their focus is the framework, they use the source code submitted for a given algorithm. Their work is based on use of a Kendryte K210 core. Modulo the different core, their results can be compared with the unextended ISA column of Table 7.4, Table 7.5, and Table 7.6.

Resilience against implementation attack. For constrained platforms of relevance to the LWC selection process, countermeasures against implementation attack are often classified as being either based on hiding [MOP07, Chapter 7] and/or masking [MOP07, Chapter 10]. Although we do not consider such countermeasures per se, some discussion of how our ISEs interact with them may still be useful:

- The principle of constant-time implementation (i.e., that which exhibits data-independent execution latency) is important; delivering it acts as a hiding-based countermeasure against certain forms of attack, and is generally easier for ISE-supported than software-only implementations. We note that *all* our replacement kernel implementations are constant-time, in certain cases[12] representing an improvement to the base implementation considered.
- Other hiding countermeasures instrumented at the ISA level, e.g., temporal skewing or shuffling, typically apply to ISE-supported implementation much like software-only implementations. That said, however, one can debate whether they are as effective. For example, an ISE-supported implementation will typically comprise fewer instructions, meaning less Instruction Level Parallelism (ILP) to harness through shuffling, and lower diversification. In turn, this acts to limit the security improvement possible.
- The situation for masking-based countermeasures is more involved. For a linear operation, our ISEs can be used on a share-wise basis. For a non-linear operation, this is not possible: one would need to redefine the ISE to accept masked inputs and outputs, and augment the associated FU so it is mask-aware. We note that our adherence to 3-address instruction formats means [GGM+21] would be one way to accommodate this for $n = 2$ shares, whereas [MP21] would be another way to do so more generally, i.e., for $n > 2$ shares.
- It is important to note that ISAP is a somewhat special case, in the sense it delivers inherent mitigation for selected side-channel and fault attacks; since this is achieved at the mode level and our ISE applies at the kernel level (i.e., the Ascon-p permutation), we do not expect any negative interaction between said ISE and any security argument for ISAP.

  That said, it is important to keep this functionality in mind when interpreting performance results. Although inefficient in relative terms, ISAP includes by-design mitigation that other candidates would have to deliver via post-design means: the resulting overhead is costed into ISAP already, complicating any direct comparison.

7.6 Conclusion

Summary. ISEs to support standard cryptographic algorithms, e.g., AES, have now been included in almost every major ISA. Anticipating the LWC process will yield an outcome that warrants similar support, this work investigated ISEs for each of the 10 LWC final round submissions. Through careful analysis of the constituent algorithms, and following a set of principled constraints (e.g., alignment with the wider RISC-V design principles, such as use of 3-address instructions), we first developed ISE designs for Ascon, Elephant, GIFT-COFB, Grain-128AEADv2, ISAP, PHOTON-Beetle, Romulus, Sparkle, Tiny-JAMBU, and Xoodyak, then implemented said designs using the RISC-V compliant Rocket core. Broadly

[12] It might be an unfair criticism given the overtly explanatory goal, but, for example, the reference implementation of PHOTON-Beetle involves multiplication in $\mathbb{F}_{2^4}$ whose execution latency is data-dependent.

speaking, comparison with software-only alternatives shows that 1) the ISEs overhead in hardware is low, 2) the ISEs allow a reduction in execution latency, the degree of which is algorithm-dependent but significant in some cases, and, at the same time, 3) the ISEs allow constant-time execution, *and* a reduction in instruction footprint. Put together, these features highlight the value of ISEs within the context of resource-constrained devices and therefore the LWC process.

Observations. Based on our work, several high-level observations seem important to stress. First, and particularly when carefully paired with implementation techniques such as fix-slicing, our results demonstrate software-only implementations using Zbkb/x can be significantly more efficient than using RV32GC alone. This fact paints Zbkb/x (and so also Zbb) in a positive light with respect to general-purpose support: implementations and benchmarking for RISC-V which do *not* consider Zbkb/x (or Zbb) disadvantage it versus, e.g., ARM. Second, our results highlight a difference in relative improvement between algorithms that are more hardware-oriented versus more software-oriented. Put simply, ISEs for the former (e.g., Elephant, PHOTON-Beetle, Romulus) typically offer a greater improvement than for the latter (e.g., Ascon, Sparkle, Xoodyak): although the most efficient software-only implementations remain so when ISE support is considered, the difference between most and least efficient algorithms is significantly smaller. Stemming from the hybrid nature of ISE-supported software, this fact could be read as complicating the classification of hardware- versus software-oriented algorithms; either way, it highlights the need to consider use of ISEs as part of their evaluation. Third, our results act as evidence that ISEs which target an implementation technique (e.g., fix-slicing) are typically more general-purpose but less efficient, whereas ISEs which target an algorithm are typically less general-purpose but more efficient. Although a somewhat obvious statement, this suggests that once an outcome from the LWC process is known, the latter approach is more sensible in the longer term.

CHAPTER

8

RISC-V ISES FOR MULTI-PRECISION INTEGER ARITHMETIC

This Chapter is based on our paper [CFG+23].

8.1 Introduction

Motivation. Public-key cryptosystems like RSA and ECC are highly computation-intensive and, therefore, relatively slow when implemented in software on general-purpose processors. The high cost of public-key cryptography has initiated a large body of research on hardware acceleration to speed up the underlying arithmetic operations, in particular the modular multiplication and squaring of long integers. Besides classical hardware accelerators in the form of loosely-coupled cryptographic co-processors also various forms of hardware-software co-design have been studied in the literature. A promising approach is to extend the Instruction Set Architecture (ISA) of a general-purpose processor by a small set of custom instructions tailored specifically to speed up the most performance-critical operations of long integer arithmetic, which are the operations carried out in the "inner loops" of algorithms for multiplication, squaring, and modular reduction. In some sense, such Instruction Set Extensions (ISEs) allow one to combine the flexibility of software with the performance of hardware and, in this way, get the best of both worlds. Flexibility in the context of public-key cryptography usually refers to algorithm agility and/or parameter agility. Namely, a set of custom instructions for long-integer arithmetic allows one to accelerate not only pre-quantum schemes like ECDH and ECDSA, but also public-key algorithms that will remain secure in the post-quantum world, e.g., isogeny-based schemes. However, since many isogeny-based cryptosystems are still in their infancy, it can be expected that basic parameters such as the order of the underlying finite field may need to be adjusted in response to new cryptanalytic techniques, which is much easier for ISE-supported software than for an algorithm cast in silicon.

Data-paths for n-bit integer arithmetic. The storage of and computation with n-bit integers offers a low-level basis for higher-level structures (e.g., $\mathbb{Z}_N$ or $\mathbb{F}_p$) of fundamental importance to public-key cryptography. Consider a w-bit data-path which is able to support w-bit integers, where w is termed the word size. The case $w = n$ implies a bit-parallel data-path which *directly* supports n-bit integers;

the resulting trade-off favours efficiency over area, implying a "wide", high-area data-path but low-latency (e.g., 1-cycle) computational steps. However, if $w < n$ then the data-path cannot offer such direct support. Instead, one must offer *indirect* support via data-structures and algorithms for multi-precision integer arithmetic which harness the w-bit data-path available. A standard approach, for example, would be to employ a radix-2^w representation [MOV96, Section 14.2.1] of some n-bit $x \in \mathbb{Z}$, thereby splitting it into $l = \lceil n/w \rceil$ digits (or limbs); since $0 \leq x_i < 2^w$ for $0 \leq i < l$, operations on digits can therefore be supported by the w-bit data-path. The case $w = 1$ implies a bit-serial data-path; the resulting trade-off favours area over efficiency, implying a "narrow", low-area data-path but high-latency (e.g., n-cycle) computational steps. Any intermediate case where $1 < w < n$ implies a digit-serial data-path, which allows a more nuanced balance between area and efficiency.

Constraints on software implementation of n-bit integer arithmetic. From a hardware perspective, design of a special-purpose data-path is possible; w typically represents a *selectable* parameter in such a case. From a software perspective, in contrast, w is a *specified* aspect of the ISA: one is reliant on the data-path provided by an associated micro-architecture, both of which are necessarily general-purpose. Typically examples include $w \in \{8, 16, 32, 64\}$. Beyond this, however, further aspects of the ISA constrain how implementation of multi-precision integer arithmetic is approached. For example, standard algorithms for multiple-precision integer addition [MOV96, Algorithm 14.7] and multiplication [MOV96, Algorithm 14.12] have $O(l)$ and $O(l^2)$ execution latencies respectively. Although l asymptotically dominates therefore, the constant factors are an important consideration; these are influenced, at least in part, by features of the ISA such as the instructions (and their semantics) plus general- and special-purpose registers available. ARMv7-M [ARM21, Section A4.4.3] offers a concrete example, in the sense it offers a rich set of instructions for unsigned integer multiplication that includes (using the same notation): `mul` (multiply: $32 = 32 \times 32$), `mla` (multiply accumulate: $32 = 32 + 32 \times 32$), `umull` (multiply long: $64 = 32 \times 32$), `umlal` (multiply accumulate long: $64 = 64 + 32 \times 32$), and `umaal` (multiply accumulate accumulate long: $64 = 32 + 32 + 32 \times 32$). These cases use up to 4 general-purpose register input and output operands (in some cases with inputs reused as outputs), to cater for 1) different computation (e.g., including or excluding accumulation), and 2) different types (e.g., a half- or full-width product).

Contributions. In this work, we start with the premise that a software-based implementation of multi-precision integer arithmetic *might*, depending on the use-case, be either advantageous or even necessary. Versus a hardware-based alternative, for example, any disadvantage related to efficiency might be offset by advantages such as flexibility; the latter can be rationalized by considering the value of algorithmic agility, which has been brought into sharp focus by, e.g., recent cryptanalytic results [CD23, MMP+23, Rob23] relating to the post-quantum construction SIKE [JAC+22].

We aim to explore implementations of this type based on use of RISC-V, an ISA whose strongly RISC-like design principles present various challenges. For example, the base ISA, e.g., RV64I, is sparse enough that *any* form of multiplication instruction is captured in the standard M (multiplication) [RV19, Chapter 7] extension, and, e.g., to facilitate efficient pipelined micro-architectures, no special-purpose status register that reflects carries is included. However, the increased deployment of RISC-V cores and by-design modularity of the ISA acts as motivation for and a vehicle to address such challenges. For example, Stoffelen [Sto19] explores implementation of multi-precision integer arithmetic on RISC-V (specifically, an RV32I-compliant E31 core), and, in doing so, 1) evaluates the efficiency of different data-structures (e.g., reduced- and full-radix representations) and algorithms [Sto19, Section 6] and 2) asymptotically studies the impact of a hypothetical ISE, namely an add-with-carry instruction [Sto19, Section 7.4]. At a high level, this work aims to offer broader, more concrete insight into similar questions. We claim it makes two lower-level contributions:

1. We study the efficient ISA-only implementation (i.e., using only base ISA instructions) of multi-precision integer arithmetic, including both full- and reduced-radix, on RV64GC base ISA. We take (the prime-field arithmetic of) X25519 [Ber06] and CSIDH [CLM+18] key-exchange as the case study, with developing and benchmarking their highly-optimized assembly implementations. Although no carry flag exists on RISC-V architecture, our timing results show that the full-radix representation is a more efficient option for CSIDH-512 on the 64-bit Rocket host core. For X25519, full-radix and reduced-radix are equally efficient.

2. We propose two instruction set extensions for multi-precision integer arithmetic; one for full-radix representation while the other one for reduced-radix representation. Each ISE includes a pair of novel fused multiply-add instructions and one instruction to accelerate carry propagation. After using our custom instructions, the reduced-radix representation achieves a better performance; ISE-supported X25519 and CSIDH-512 reach respectively 1.60× and 1.71× faster compared to their most efficient ISA-only implementation.

Source code. Note that all source code (software and hardware) and documentation associated with this work are publicly available at `https://github.com/scarv/mpise`.

Organization. The paper is organized as follows. In Section 8.2 we present background information on related RV64GC instructions, X25519 and CSIDH, which forms the basis for our work. In Section 8.3 we survey ISA-only implementation techniques applicable to RISC-V, before turning our attention to the design and implementation of associated ISEs in Section 8.4. We present a set of evaluation results in Section 8.5, then, finally, a concluding summary in Section 8.6.

8.2 Background

8.2.1 Related instructions

The frequently-used instructions in this work for the integer arithmetic computation are integer addition `add`, integer subtraction `sub`, and bitwise shift `slli`, `srli`, `srai` from RV64I; integer multiplication `mul` and `mulhu` from RV64M. Notably, there is no carry flag existing on RISC-V, hence the corresponding carry propagation on RISC-V takes two instructions, i.e., one `sltu` (from RV64I) for carry-out (i.e., overflow) check, and one `add` for propagating the carry bit.

8.2.2 X25519

X25519 [Ber06] is an efficient Elliptic Curve Diffie-Hellman (ECDH) key exchange scheme that uses a Montgomery curve (called Curve25519) defined over a 255-bit prime field. Montgomery curves allow for a remarkably simple method to compute a variable-base scalar multiplication $Q = kP$, the *Montgomery ladder*, which gets the (affine) x-coordinate of a point P as input and returns the x coordinate of Q. The core operation is a *ladder step* consisting of five multiplications, four squarings, a multiplication by a small constant, and nine additions/subtractions in the underlying prime field $\mathbb{F}_p$. Performing a scalar multiplication according to the Montgomery ladder is not only extremely fast, but also provides some intrinsic protection against timing attacks since it executes always the same sequence of field operations, independent of the actual value of the scalar k. The prime p used by Curve25519 is a so-called *pseudo-Mersenne prime*, i.e., a prime of the form $p = 2^k - c$ where c is small compared to 2^k (ideally, c fits into a single register of the target platform). More concretely, Curve25519 is defined over the pseudo-Mersenne prime $p = 2^{255} - 19$, which means the elements of $\mathbb{F}_p$ are integers of a length of up to 255 bits. Pseudo-Mersenne prime allow for efficient modular reduction by taking advantage of the congruence $2^k \equiv c \bmod p$. In this way, a $2k$-bit product can be reduced modulo p by multiplying the

higher half (i.e., the upper k bits) of the product by c and adding the result to the lower half. Repeating this step a second time and performing a few conditional subtractions of p will eventually yield a fully reduced result in the range of $[0, p-1]$.

8.2.3 CSIDH-512

CSIDH [CLM+18], the short for Commutative Supersingular Isogeny Diffie-Hellman, is an isogeny-based key exchange protocol and can serve as a "drop-in" post-quantum replacement for the standard elliptic curve Diffie-Hellman protocol. The core component of CSIDH is the action of an ideal class group on a set of supersingular elliptic curves, which is built on Montgomery curve arithmetic and with $\mathbb{F}_p$ operations at the low level. In this work, we focus on CSIDH-512 (NIST PQ level 1) and particularly its $\mathbb{F}_p$ arithmetic. The prime p is 511 bits long and in a form of $p = 4 \cdot \ell_1 \cdots \ell_{74} - 1$, where $\ell_1, \ldots, \ell_{73}$ are the first odd primes starting from $\ell_1 = 3$, and $\ell_{74} = 587$. Unlike SIKE, which uses Montgomery-friendly primes [CLN16] resulting in faster Montgomery reduction, the primes of CSIDH do not offer any opportunities for optimizing generic reduction techniques such as Montgomery or Barrett reduction. Therefore, the software implementation for CSIDH $\mathbb{F}_p$ arithmetic are in some sense more ordinary, i.e., it can be extended to other similar-size primes with only a minor modification.

8.2.4 Notation

We use GPR$[i]$ to denote the i-th entry of the General-Purpose Register file, where $0 \leq i < 32$. GPR$[0]$ is hard-wired 0, which means writes to it are ignored and reads from it always get 0. We use $x \ll y$ (resp. $x \gg y$) to denote logical left-shift (resp. logical right-shift) of x by y bits. Additionally, taking the same way to [RVK22], we use prefix EXTS (Sign-EXTended) for arithmetic shift; EXTS$(x \ll y)$ (resp. EXTS$(x \gg y)$) denote arithmetic left-shift (resp. arithmetic right-shift) of x by y bits. We use $x \parallel y$ to denote concatenation of x and y, and $x_{\{h \ldots l\}}$ denote extraction of bits h (the high, or more-significant index) through l (the low, or less-significant index) inclusive from some x.

8.3 Implementation: ISA-only

Both case studies we are considering (i.e., X25519 and CSIDH-512) have to perform *basic* arithmetic operations in $\mathbb{F}_p$, namely addition, subtraction, multiplication, and squaring. In this Section, we first focus on the ISA-only implementation of the most performance-critical operation, namely $\mathbb{F}_p$ multiplication. We then discuss two different (constant-time) implementation options of the fast modulo-p reduction, which is used in CSIDH and in fact dominates the execution time of its $\mathbb{F}_p$ addition and subtraction (per Table 8.6). As described before, we consider both full-radix and reduced-radix representations in this work. Concretely, the full-radix in this work means 64-bit-per-word (i.e., 2^{64}) representation, while the reduced-radix means a classic 51-bit-per-limb (i.e., 2^{51}) representation for operands on Curve25519 and 57-bit-per-limb (i.e., 2^{57}) representation[1] for CSIDH-512. The implementation of other basic $\mathbb{F}_p$ operations are relatively straightforward, we refer readers to our source code for more details.

8.3.1 $\mathbb{F}_p$ multiplication

High-level techniques. The $\mathbb{F}_p$ multiplication (i.e., modular multiplication) is the most costly and also the most complicated among the basic $\mathbb{F}_p$ operations. It is composed of two main steps, namely integer multiplication and modular reduction. There exist different techniques for optimizing integer multiplication or modular reduction itself, e.g., operand-scanning (aka. schoolbook), product-scanning [Com90], Karatsuba [KO63] multiplications. According to our experiments, the basic multiplication

[1] A prime-field element in CSIDH-512 in full-radix representation needs 8 words, so an ideal reduced-radix representation would need just 9 limbs but with the most headroom, hence we compute $\lceil 511/9 \rceil = 57$.

Table 8.1: Information about our $\mathbb{F}_p$ multiplication implementations.

$\mathbb{F}_p$	Radix	Multiplication	Reduction	Fashion
X25519	full	product-scanning	pseudo-Mersenne prime	separated
X25519	reduced	operand-scanning	pseudo-Mersenne prime	integrated
CSIDH-512	full	product-scanning	Montgomery reduction	separated
CSIDH-512	reduced	product-scanning	Montgomery reduction	separated

with quadratic complexity (i.e., operand-scanning and product-scanning) is more efficient than Karatsuba for the two case studies on our base core. In addition, various fashions about how to integrate integer multiplication and modular reduction have also been studied, e.g., separated, coarsely-integrated, finely-integrated (see more details in [KAK96]). These integration fashions mainly differ regarding the number of memory accesses when the register-space is limited. However, RV64GC register-space is large enough to store all operands and intermediates for our case, and we fully unroll the loops. Therefore, all integration fashions are very similar in performance on our base ISA. The detailed information about our implementations of $\mathbb{F}_p$ multiplication in X25519 and CSIDH-512 is shown in Table 8.1.

Algorithm 8.1: ISA-only full-radix MAC operation.

Input: 192-bit accumulator $e \parallel h \parallel l$; 64-bit multiplicands a and b.
Output: 192-bit accumulator $e \parallel h \parallel l$.

```
1 mulhu   z, a, b
2 mul     y, a, b
3 add     l, l, y
4 sltu    y, l, y
5 add     z, z, y
6 add     h, h, z
7 sltu    z, h, z
8 add     e, e, z
9 return  e || h || l
```

Algorithm 8.2: ISA-only reduced-radix MAC operation.

Input: 128-bit accumulator $h \parallel l$; 64-bit multiplicands a and b.
Output: 128-bit accumulator $h \parallel l$.

```
1 mulhu   z, a, b
2 mul     y, a, b
3 add     l, l, y
4 sltu    y, l, y
5 add     z, z, y
6 add     h, h, z

7 return  h || l
```

Low-level building block. Although there are various high-level techniques for implementing modular multiplication, at a very low level all the different implementations (of integer multiplication and Montgomery reduction) essentially rely on the same building block, namely $w_{i+j} \leftarrow u_i \cdot v_j + w_{i+j}$. In detail, it first computes a partial product of two long-integer operands u and v, and then accumulates the generated partial product $u_i \cdot v_j$ to a corresponding word/limb accumulator w_{i+j} of the result w. This Multiply-then-ACcumulate (MAC) operation is used the most frequently in integer multiplication and also Montgomery reduction, which is actually the inner-loop operation if integer multiplication and/or Montgomery reduction are implemented in a nested-loop way. Due to the purpose of this work which is to design the efficient instructions, we take more care of this MAC operation. We list the RV64GC assembly implementation of the mentioned MAC operation in Algorithm 8.1 for full-radix, i.e., computation of $(e \parallel h \parallel l) \leftarrow (e \parallel h \parallel l) + a \cdot b$, and in Algorithm 8.2 for reduced-radix, i.e., computation of $(h \parallel l) \leftarrow (h \parallel l) + a \cdot b$.

For the MAC operation only, reduced-radix requires fewer instructions compared to full-radix (i.e., 6 vs. 8). However, for the overall modular multiplication, there are some implicit overheads in the reduced-radix implementation. First of all, an integer represented in reduced-radix (in most cases) needs more limbs than in full-radix. Obviously, the more limbs, the more MAC operations are required.

In addition, reduced-radix needs extra instructions to align the accumulator and to propagate the carry bits. In contrast, the alignment of accumulator in full-radix is automatic (i.e., no extra cost), and the carry propagation is already included in Algorithm 8.1. Based on the above analysis, it is not obvious whether the full-radix or the reduced-radix is more efficient on RISC-V. Instead, it is influenced by several different factors, e.g., the design of cryptographic algorithm itself, multiplication technique, RISC-V core design, and etc.

8.3.2 Fast reduction modulo p

In the implementation of CSIDH-512, when the range of an operand u is known to be $[0, 2p-1]$, a *fast* modulo-p reduction is performed instead of the relatively-costly Montgomery reduction to reduce it to $[0, p-1]$. There are two steps in this fast reduction; 1) it subtracts the modulus p from the operand and gets $t = u - p$; 2) if $t < 0$ which means u is already in $[0, p-1]$, it outputs the result $r = u$, otherwise it outputs the result $r = t$. Regarding the second step, there are two constant-time options to implement it, i.e., the *addition-based* and the *(conditional-)swap-based*. We show both options respectively in Algorithm 8.3 and 8.4. Note that this fast modulo-p reduction is used in the $\mathbb{F}_p$ addition and the final step of Montgomery reduction, and a variant of Algorithm 8.3 (Line 1 changes to be $t = u - v$, where v is another operand instead of modulus p) can be used as the $\mathbb{F}_p$ subtraction.

Algorithm 8.3: Addition-based fast modulo-p reduction.

Input: an operand $u \in [0, 2p)$ and the modulus p.
Output: the result $r \in [0, p)$.

1 $t \leftarrow u - p$
2 **if** $t < 0$ **then** /* constant-time */
3 $\quad m \leftarrow$ -1 /* all 1 in binary */
4 **else**
5 $\quad m \leftarrow 0$ /* all 0 in binary */
6 $m \leftarrow m \wedge p$
7 $r \leftarrow t + m$
8 **return** r

Algorithm 8.4: Swap-based fast modulo-p reduction.

Input: an operand $u \in [0, 2p)$ and the modulus p.
Output: the result $r \in [0, p)$.

1 $t \leftarrow u - p$
2 **if** $t < 0$ **then** /* constant-time */
3 $\quad m \leftarrow$ -1 /* all 1 in binary */
4 **else**
5 $\quad m \leftarrow 0$ /* all 0 in binary */
6 $m \leftarrow m \wedge (u \oplus t)$
7 $r \leftarrow t \oplus m$
8 **return** r

Full-radix. On most ISAs, the addition-based one is a more efficient option for the full-radix since it saves one operation compared to a swap-based version (line 6 in Algorithm 8.3 and 8.4). However, this is not the case for RISC-V, due to the lack of carry flag. On RISC-V, the last operation in Algorithm 8.3, line 7, is relatively expensive, which in turn makes the swap-based version become a faster option for our full-radix assembly implementation.

Reduced-radix. It is obvious that the swap-based version is a better choice for the final step of Montgomery reduction, since it can avoid the carry propagation caused by addition. But for the $\mathbb{F}_p$ addition, the swap-based version is less efficient because it needs extra instructions to make the limbs of the sum, i.e., the operand u, to be short enough.

8.4 Implementation: ISE-supported

This section discusses our ISE-supported implementation. After introducing some requirements of ISE design in Section 8.4.1, we respectively explain our integer fused multiply-add instructions in Sec-

Table 8.2: The overview of our custom instructions.

Functionality	Custom instructions	
	full-radix	reduced-radix
integer fused multiply-add	`maddlu` `maddhu`	`madd[51|57]lu` `madd[51|57]hu`
carry propagation	`cadd`	`sraiadd`

tion 8.4.2 and our carry-propagation instructions in Section 8.4.3. Further, we in Section 8.4.4 analyze the impact of our ISE on software side. At last in Section 8.4.5 we give the details of hardware implementation of our ISE on Rocket core.

Proposed ISEs. Before going into details, we present an overview of our custom instructions in Table 8.2. Each custom instruction is designed to be *single-cycle*. Instructions are grouped into two ISEs, and each ISE contains three instructions: 1) one ISE is designed for the full-radix representation including `maddlu`, `maddhu`, and `cadd`; 2) the other ISE for the reduced-radix presentation including `madd51lu` (resp. `madd57lu`), `madd51hu` (resp. `madd57hu`), and `sraiadd`. Note that in the case of using reduced-radix ISE, in order to have a straightforward comparison (between full-radix and reduced-radix), we suppose the core is extended with either the pair of `madd51lu` and `madd51hu` or the pair of `madd57lu` and `madd57hu`, depending on the target algorithm (e.g., X25519 or CSIDH-512) or operand length (e.g., 256-bit integers or 512-bit integers).

Developed implementations. With the ISEs we proposed, we are able to develop in total four different types of the highly-optimized assembly implementation. Concretely, they are 1) ISA-only full-radix; 2) ISA-only reduced-radix; 3) ISE-supported full-radix; 4) ISE-supported reduced-radix.

8.4.1 Constraints

In their research of RISC-V ISE for AES, Marshall et al. codify a set of ISE requirements [MNP+21, Section 3] to guide their design process. We aim to adopt the same requirements, and list here ((numbered to match, noting we omit their AES-specific requirement 1)):

Requirement 2. The ISE must align with the wider RISC-V design principles. This means it should favour simple building-block operations, and use instruction encodings with at most 2 source register addresses and 1 destination register address.

Requirement 3. The ISE must use the RISC-V general-purpose scalar register file to store operands.

Requirement 4. The ISE must not introduce special-purpose architectural state, nor rely on special-purpose micro-architectural state (e.g., caches or scratch-pad memory).

These requirements (resp. constraints) maximize potential utility of ISE designs and simplify their implementation. For example, within the context of RISC-V they 1) support multiple implementation options, including a more traditional integrated approach or via the in-development Custom Function Unit (CFU)[2] specification, and 2) offer an easier route to standardization and deployment as a result of limiting impact on other aspects of the ISA. Beyond this, the constraints also permit extrapolation to other ISAs, e.g., via the ARMv8-M custom instruction mechanism [CP20]; doing so would be more difficult otherwise. However, we also recognize that adopting these constraints means potential ISE designs might be ignored; this fact potentially renders results sub-optimal (e.g., with respect to the

[2] https://cfu.readthedocs.io

Table 8.3: Examples of existing integer fused multiply-add instructions.

Instruction	ISA/ISE	Operation	Radix
mla	ARM	rd ← lo(rs1*rs2)+rs3	F + R
umlal	ARM	(rd2||rd1) ← (rs1*rs2)+(rd2||rd1)	F + R
umaal	ARM	(rd2||rd1) ← (rs1*rs2)+rd2+rd1	F + R
vpmadd52luq	AVX-512	rd ← lo52(rs1*rs2)+rs3	R
vpmadd52huq	AVX-512	rd ← hi52(rs1*rs2)+rs3	R

achievable execution-time reduction), at least versus a more permissive alternative where one or more of the constraints are *not* adhered to.

Notably and relevantly, we need to stress the only exception (violating the listed requirements) that already exists on our base ISA is floating-point fused multiply-add `F[N]MADD/F[N]MSUB` instructions [RV19, Section 11.6]. In detail, this sort of instructions violates Requirement 2 and exclusively takes advantage of a new standard instruction format R4-type, which specifies 3 *source register addresses* and 1 destination register address. This makes sense for multiply-add instructions, because with using 2 source registers, one can only design an instruction that accumulates the (partial) product into a multiplicand, e.g.,

$$\texttt{rd} \leftarrow \texttt{rs1 * rs2 + rs1.}$$

This kind of conceived instruction is of little use for floating-point kernels used in digital-signal processing and other domains since these kernels require accumulation of products into a third register. Without violating Requirement 2, it would not be possible to design an instruction that is capable to accelerate fused multiply-add operations. However, the integration of R4-type instructions comes at a cost since a third read port has to be added to the register file and an additional bus is necessary to be able to transport all 3 source operands to the floating-point unit in one cycle. Nonetheless, since fused multiply-add operations are so fundamentally important and performance-critical for many kernels, the designers of the F extension came to the conclusion that the benefits of R4-type instructions outweighs issues related to their implementation and integration.

ISE design for long-integer arithmetic for public-key cryptography faces similar challenges as ISE design for floating-point kernels in digital-signal processing since 1) the most performance-critical operation are fused multiply-add (executed in, e.g., the inner loop of the product-scanning method) and 2) these operations can not be accelerated through ISE without violating Requirement 2. Similarly as argued above, it makes sense to consider R4-type instructions (and violate Requirement 2) when designing ISE for multi-precision integer arithmetic. By relaxing the ISE requirements, the operand-scanning and product-scanning methods can benefit from multiply-add instructions with three source register addresses (similar to the `F[N]MADD/F[N]MSUB` instructions). This naturally raises the question of whether it would make sense to permit R4-type instructions for operations other than fused multiply-add. We argue that instructions with 3 source and 1 destination register addresses should be used sparingly due to the limited instruction space. Namely, since an R4-type instruction contains 4 register addresses, consuming 20 bits altogether, only 12 bits are left for the encoding of the actual instruction. Therefore, we decided to consider R4-type instructions only for the most performance-critical operations, namely the operations in the inner loop of integer multiplication, squaring, and Montgomery reduction, i.e., the MAC operation described in Section 8.3.1.

8.4.2 Integer multiply-add instructions

Existing designs. Integer fused multiply-add instructions exist on almost all mainstream ISAs, e.g., x86 and ARM. From a viewpoint of cryptographic software implementation, such instructions are undoubtedly beneficial for the public key cryptosystems like RSA and ECC, whose underlying arithmetic

31 30 29 28 27	26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
rs3	00	rs2	rs1	111	rd	1111011	maddlu
rs3	01	rs2	rs1	111	rd	1111011	maddhu

- **maddlu rd, rs1, rs2, rs3**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
z       ← GPR[rs3]
m       ← (1 << 64) - 1
r       ← (x * y + z) & m
GPR[rd] ← r
```

- **maddhu rd, rs1, rs2, rs3**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
z       ← GPR[rs3]
m       ← (1 << 64) - 1
r       ← ((x * y + z) >> 64) & m
GPR[rd] ← r
```

Figure 8.1: Our integer multiply-add instructions for full-radix implementation.

is based on large integers. For example, Intel in [Int22b] explicitly states their newest AVX-512 vector integer fused multiply-add instructions are "two new instructions for big number multiplication for acceleration of RSA vectorized SW and other Crypto algorithms (Public key) performance". We list in Table 8.3 some integer multiply-add instructions currently existing on ARM and Intel AVX-512. These instructions are chosen to showcase different design forms/styles, therefore we omit the different instruction variants that have a similar operation (e.g., unsigned multiplication vs. signed multiplication) to the listed instructions. In Table 8.3, `lo` (resp. `hi`) means the lower half (resp. the higher half) of the product; `lo52` (resp. `hi52`) precisely takes the lower 52-bit (resp. the higher 52-bit) of a 104-bit product; F (resp. R) is the short for full-radix (resp. reduced-radix). We first analyze these existing instructions and then introduce our own design. We discuss these instructions from three aspects; 1) **operation**: all the instructions except for `umaal` can be formalized into a Multiply-Shift-And-Add (MSA2) paradigm, i.e.,

$$\texttt{rd} \leftarrow \texttt{(((rs1 * rs2) >> j) \& m) + rs3.}$$

We default the length of multiplier here is in line with the register length (or the element width in the case of vector instructions). The offset j and the mask m jointly control whether the whole product or the partial product will be accumulated, and if the latter, concretely which part. Moreover, in order to accumulate the whole product, the instructions, like `umlal` and `umaal`, need to accordingly generate a *double-length* output. Further, based on this fact, the `umaal` instruction can even perform two additions. However, considering we use standard R4-type, accumulating the whole product in our custom instruction is impossible; 2) **instruction encoding**: all these instructions use *at least* 3 *source register addresses*, and some of these instructions overwrite 1 or 2 source registers (i.e., some source registers at the same time serve as the destination registers); 3) **supported radix**: normally the multiply-add instructions could support both full- and reduced-radix representations. The exceptions are `vpmadd52luq` and `vpmadd52huq`, which work properly with the reduced-radix only.

Our design for the full-radix. As a result, when designing the integer fused multiply-add instructions for full-radix implementation, `mla` can serve as a good starting point. We first design an instruction `maddlu` having the same functionality, used for accumulating the lower half of the product. It makes sense to accordingly design a counterpart `maddhu` to accumulate the higher half of the product (of the unsigned multiplication). More detailed information of our `maddlu` and `maddhu` instructions is shown in Figure 8.1. It should be noted that our `maddhu` is designed to be in a paradigm of Multiply-Add-Shift-And instead of the typical MSA2. Because in this way, it can handle the carry propagation *inside* the `maddhu` thus to save a carry-out check (i.e., one `sltu` instruction) on the software side.

31 30 29 28 27	26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
rs3	00	rs2	rs1	111	rd	1111011	madd51lu
rs3	01	rs2	rs1	111	rd	1111011	madd51hu
rs3	10	rs2	rs1	111	rd	1111011	madd57lu
rs3	11	rs2	rs1	111	rd	1111011	madd57hu

- **madd51lu rd, rs1, rs2, rs3**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
z       ← GPR[rs3]
m       ← (1 << 51) - 1
r       ← ((x * y) & m) + z
GPR[rd] ← r
```

- **madd51hu rd, rs1, rs2, rs3**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
z       ← GPR[rs3]
m       ← (1 << 64) - 1
r       ← (((x * y) >> 51) & m) + z
GPR[rd] ← r
```

- **madd57lu rd, rs1, rs2, rs3**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
z       ← GPR[rs3]
m       ← (1 << 57) - 1
r       ← ((x * y) & m) + z
GPR[rd] ← r
```

- **madd57hu rd, rs1, rs2, rs3**

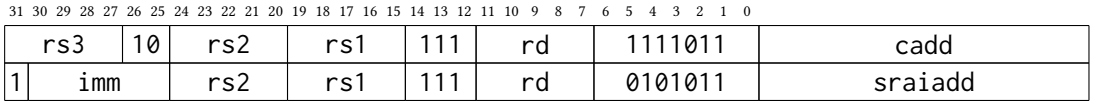
```
x       ← GPR[rs1]
y       ← GPR[rs2]
z       ← GPR[rs3]
m       ← (1 << 64) - 1
r       ← (((x * y) >> 57) & m) + z
GPR[rd] ← r
```

Figure 8.2: Our integer multiply-add instructions for reduced-radix implementation.

Our design for the reduced-radix. As for the reduced-radix implementation, `vpmadd52luq` and `vpmadd52huq` (AVX-512IFMA) offer the helpful experience for our custom instruction design. However, as pointed out in [DeV18] and [CFGR22, Section 3.1], when using a representation whose limb length is 52-bit (or slightly shorter than 52-bit), there exists a so-called *multiplier saturating problem* of using these instructions. In specific, because of the delayed carry propagation, during the intermediate computation of a cryptographic implementation, a limb may increase few bits therefore exceeding the multiplier length. In such case, if using AVX-512IFMA instructions on these limbs, the extra instructions are required to instantly propagate carry bits and increase the burden on execution time and dependency chain. From a software perspective, the implementer should use a more conservative data representation, i.e., a shorter limb length, to avoid multiplier saturation. However, in this work, we can solve this saturating problem at the instruction design level (for our custom instructions). Concretely, instead of using a reduced-length multiplier with respect to element width (e.g., 52-bit versus 64-bit in AVX-512IFMA), we take advantage of a full 64-bit multiplier and use the offset j and the mask m to control the accumulated partial product. The offset j equals to the limb bit-length of a selected data representation; m equals to $(1 \ll j) - 1$ in the instruction accumulating lower part of the product and equals to $(1 \ll 64) - 1$ in the instruction accumulating higher part. The instruction accumulating higher part has to return more bits (i.e., a larger m), because due to the delayed carry propagation the product is usually larger than $2j$ bits. Details of our custom integer multiply-add instructions for radix-2^{51} and radix-2^{57} are illustrated in Figure 8.2, and our custom instructions are designed in MSA2 style.

8.4.3 Carry-propagation instructions

Our design for the full-radix. In the full-radix MAC operation (Algorithm 8.1), there are two carry propagations (i.e., two `sltu` and two corresponding `add` instructions). Although our `maddhu` saves one carry propagation, there is the other one still remaining. We therefore design an instruction to further decrease the cost of this carry propagation, and our design is shown in Figure 8.3, and `cadd` stands for get-Carry-then-ADD.



31 30 29 28 27	26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
rs3	10	rs2	rs1	111	rd	1111011	cadd
1 imm		rs2	rs1	111	rd	0101011	sraiadd

- **cadd rd, rs1, rs2, rs3**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
z       ← GPR[rs3]
r       ← ((x + y) >> 64) + z
GPR[rd] ← r
```

- **sraiadd rd, rs1, rs2, imm**

```
x       ← GPR[rs1]
y       ← GPR[rs2]
r       ← x + EXTS(y >> imm)
GPR[rd] ← r
```

Figure 8.3: Our custom carry-propagation instructions. `cadd` is designed for full-radix implementation. `sraiadd` is designed for reduced-radix implementation.

Our design for the reduced-radix. Unlike full-radix implementation propagates the carry bit instantly, the reduced-radix implementation delays some carry propagations in a certain intermediate computation, and performs one-time propagation at the end. Even though in this way the delayed carry propagation has already saved a number of instructions, the final one-time propagation still has high latency and, in particular, strong dependency, compared to arithmetic computations. We therefore design a custom instruction aiming to accelerate this final propagation in the reduced-radix implementation. Our design (see Figure 8.3) in essence fuses the `sari` and `add` instructions into one.

Algorithm 8.5: ISE-supported full-radix MAC operation.

Input: 192-bit accumulator $e \parallel h \parallel l$;
64-bit multiplicands a and b.
Output: 192-bit accumulator $e \parallel h \parallel l$.

```
1 maddhu   z, a, b, l
2 maddlu   l, a, b, l
3 cadd     e, h, z, e
4 add      h, h, z
5 return   e ∥ h ∥ l
```

Algorithm 8.6: ISE-supported reduced-radix MAC operation.

Input: 128-bit accumulator $h \parallel l$;
64-bit multiplicands a and b.
Output: 128-bit accumulator $h \parallel l$.

```
1 madd57hu h, a, b, h
2 madd57lu l, a, b, l


3 return   h, l
```

8.4.4 Impact

On the full-radix implementation. Apparently, after applying `madd[l|h]u` and `cadd` to the full-radix implementation, the instruction number for MAC operation is decreased. We show the ISE-supported MAC implementation in Algorithm 8.5 for full-radix, i.e., computation of $(e \parallel h \parallel l) \leftarrow (e \parallel h \parallel l) + a \cdot b$. Compared with the ISA-only implementation in Algorithm 8.1, Algorithm 8.5 saves the half of the instructions (i.e., the number of required instructions decreases from 8 to 4).

On the reduced-radix implementation. We show the ISE-supported MAC operation in Algorithm 8.6 for reduced-radix, taking radix-2^{57} as the example, i.e., computation of $l \leftarrow l + (a \cdot b)_{\{56...0\}}$ and $h \leftarrow h + (a \cdot b)_{\{120...57\}}$. Algorithm 8.6 (compared with Algorithm 8.2) not only reduces instructions (i.e., the number of required instructions decreases from 6 to 2) but also makes the accumulators automatically aligned so that even saving further overheads. Hence, one can expect the reduced-radix

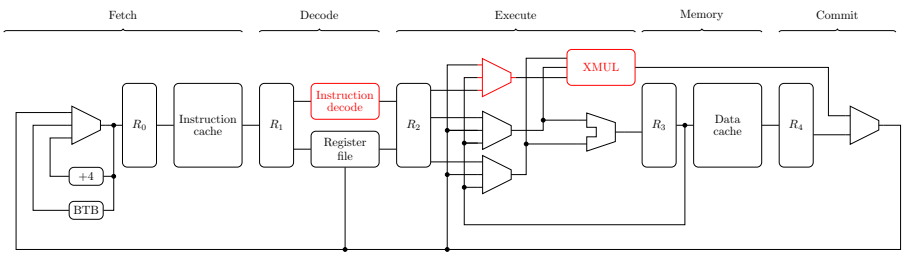


Figure 8.4: A block diagram highlighting features in our hardware implementation (e.g., integration of the extended multiplier (XMUL), and modified instruction decoder) in red. Note that R_i denotes the i-th pipeline register, a Branch Target Buffer (BTB) is shown toward the left-hand end of the pipeline.

is supposed to get more performance improvement from an ISA-only to an ISE-supported implementation.

In addition, the instruction sequence for propagating the carry bits from the previous limb `rx` to the next limb `ry` in an ISA-only radix-2^w implementation is shown as follows (where `rm` stands for a mask and holds the value of $2^w - 1$).

```
srai    rz, rx, w
add     ry, ry, rz
and     rx, rx, rm
```

After utilizing the custom `sraiadd` instruction, the instruction sequence is simplified as follows.

```
sraiadd ry, ry, rx, w
and     rx, rx, rm
```

This `sraiadd` reduces the number of instructions for the final propagation and meanwhile weakens the dependency chain.

8.4.5 Hardware implementation

Host core. To realize our ISE design, the highly configurable, RISC-V compliant Rocket-Chip [AAB+16] host core is used. At a high level, the core executes instructions using a 5-stage, in-order pipeline; support is included within the core for a branch prediction mechanism, and in the wider system for a 16 kB instruction cache and a 16 kB data cache. To support the execution of the proposed instructions in our ISE design, two modifications are made to the host core. First, an extended multiplier (XMUL) is added to execute our proposed custom instructions. The XMUL unit *extends* the original pipelined multiplier of the Rocket core, which means it supports not only our integer multiply-add and carry-propagation instructions but also the original multiply instructions of base ISA. Second, new ISE modifications are made to the instruction decoder, which, e.g., allows it to correctly provide input operands to XMUL, control the XMUL so it performs the required computation, and accept output operands from XMUL. See details in Figure 8.4.

XMUL. XMUL is extended from the original multiplier so that it has the additional third input operand to support the integer fused multiply-add instructions and the carry-propagation instruction, defined in Section 8.4. Similar to the normal operands, the additional operand can be fetched from the forwarding path to resolve the read-after-write hazard associated with the previous instructions. Like the original multiplier, XMUL is implemented with a 2-stage pipelined architecture (including one register stage at input operands and another at the output result) to avoid timing-critical paths, and it does not extend the existing critical path so has no impact on the clock frequency. The implementation of XMUL computation stems directly from the definition of the associated instructions with no effort invested

Table 8.4: Results of hardware-oriented evaluation.

Core	LUTs	Regs	DSPs	CMOS
RV64GC base core	4935	2156	16	596660
RV64GC extended core (full-radix)	5188	2390	16	644172
RV64GC extended core (reduced-radix)	5275	2352	16	650964

Table 8.5: Results of software-oriented evaluation: the execution time (in clock cycles) of our implementations of X25519. The most efficient ISA-only implementation is used as the baseline for computing the speed-up factor.

Operation	Full-radix		Reduced-radix	
	ISA-only	ISE-sup.	ISA-only	ISE-sup.
$\mathbb{F}_p$ mod add	52	51	33	32
$\mathbb{F}_p$ mod sub	57	56	37	36
$\mathbb{F}_p$ mod mul	200	138	238	134
$\mathbb{F}_p$ mod sqr	178	138	163	89
Montgomery ladder step	2205	1698	2204	1344
variable-base scalar multiplication	634903	494192	633893	395578
(speed-up)	(1.00×)	(1.28×)	(1.00×)	(1.60×)

in optimizing the arithmetic circuitry. Certainly, the extension of XMUL causes an increased hardware usage overhead compared to the original multiplier of the Rocket core.

8.5 Evaluation

Experimental platform. To produce an experimental platform which permits evaluation of, e.g., area and cycle-accurate execution latency, we make use of the Arty-100T board[3], which hosts a Xilinx Artix-7 (model `XC7A100TCSG324`) FPGA device. We synthesize the stand-alone design for our implementations using Xilinx Vivado 2019.1; default synthesis settings are used, with no effort invested in synthesis or post-implementation optimization. The FPGA uses a 100 MHz external clock input, which is adjusted into a 50 MHz internal clock signal for use by the host core itself.

Hardware. Table 8.4 presents a summary of synthesis results for each implementation. We measure (cumulative) hardware cost in terms of the number of LUTs, Regs, DSPs, and CMOS. The measurement is compared to the RV64GC core alone, and so excludes the wider system: doing so seems more representative, in that, e.g., the caches, would dominate otherwise. In line with our definition of base ISA, what we term the **base core**, i.e., a baseline for our work, is a 64-bit Rocket core supporting RV64GC only. We then further extend this RV64GC core with support for our proposed custom instructions, yielding what we term an **extended core**. The implementation of full-radix (resp. reduced-radix) XMUL produces an increase of only 5% (resp. 7%) in the number of LUTs, and an increase of 11% (resp. 9%) in the number of Regs, on the extended core versus the RV64GC base core.

Software of X25519. As mentioned in Section 8.4, we developed four different constant-time X25519 implementations to evaluate our ISE designs, whose execution times are shown in Table 8.5. First of all, in terms of ISA-only implementations, the full-radix and the reduced-radix are equally efficient, because compared with reduced-radix, the full-radix implementation has faster multiplication and squaring due to a smaller word/limb number but slower addition and subtraction due to the costly carry-out check

[3]See https://digilent.com/reference/programmable-logic/arty-a7/start.

Table 8.6: Results of software-oriented evaluation: the execution time (in clock cycles) of our CSIDH-512 implementations. The timing for CSIDH class group action *includes* a key validation. The most efficient ISA-only implementation is used as the baseline for computing the speed-up factor.

Operation	Full-radix		Reduced-radix	
	ISA-only	ISE-sup.	ISA-only	ISE-sup.
integer multiplication product-scan.	608	371	625	303
integer multiplication Karatsuba	650	500	708	342
integer squaring	440	371	398	216
Montgomery reduction	730	469	818	389
fast modulo-p reduction	107	107	112	104
field addition	163	163	148	132
field subtraction	143	143	139	123
field multiplication	1446	954	1561	799
field squaring	1279	951	1334	712
point addition	9344	6709	9877	5484
point doubling	9687	7053	10174	5759
Montgomery ladder step	18544	13279	19588	10835
CSIDH class group action	701.0 M	502.9 M	736.2 M	411.1 M
(speed-up)	(1.00×)	(1.39×)	(0.95×)	(1.71×)

and propagation (while the reduced-radix can simply use a delayed carry propagation). After using our ISEs, the $\mathbb{F}_p$ multiplication and squaring in reduced-radix implementation become even faster than the full-radix ones. This enhancement (from multiplication and squaring) propagates up to the variable-base scalar multiplication, and thus makes the reduced-radix 1.62× faster compared to the ISA-only implementations, which verifies the statement in Section 8.4.4 that reduced-radix is expected to get a larger improvement after using our ISE.

Software of CSIDH-512. Table 8.6 shows the performance of different CSIDH-512 implementations. Since this work focuses solely on the ISE design (i.e., instruction-level), so the Table 8.6 (i.e., the comparison) is done to mainly evaluate the impact of ISE. Per Table 8.6, the speed-up factors scale very well from point operation to group action (e.g., in the case of reduced-radix ISE-supported implementation, the speed-up factors are both 1.71× at these two levels). This means using different flavours of CSIDH (which differ in higher levels rather than point and $\mathbb{F}_p$ arithmetic) does not really affect the speed-up factor of ISE, so we simply use the original CSIDH software[4] for the higher-level arithmetic and computation. All of our prime-field implementations are ensured to be constant-time. In order to have the optimal field multiplication implementation, we develop two versions of integer multiplication with respectively product-scanning and Karatsuba technique. The result shows the product-scanning outperforms Karatsuba across all cases. The full-radix pure-software implementation of CSIDH-512 group action takes 701.0 M clock cycles on our base ISA. However, after using our custom instructions, the group action can just take 411.1 M cycles, which means our ISE can result in a 1.71× speed-up.

8.6 Conclusion

In this work we provided new insights on the efficient implementation of multi-precision integer arithmetic, in particular multiplication and modular reduction, on the RISC-V platform, both with and without ISE. In the context of "pure" software implementation using only the base RV64GC instructions, we studied the impact of full-radix versus reduced-radix representation of the operands. Intuitively,

[4] https://csidh.isogeny.org/software.html

one would expect a reduced-radix implementation to outperform its full-radix counterpart since the RISC-V architecture does not include an add-with-carry instruction. However, our results demonstrate that, for relatively short operands of a length of 255 bits (using X25519 as case study), the two radix representations achieve almost equal execution times, while for 511-bit operands (i.e., CSIDH-512), the full-radix representation is the better option. Overall, this leads to the conclusion that implementers of public-key cryptosystems should represent operands with the full radix of 64 bits. We then designed a small set of custom instructions to speed up the execution of long-integer arithmetic, whereby we focused primarily on the inner-loop operation of the product-scanning multiplication. Using our ISE, we again analyzed the performance of reduced-radix versus full-radix representation and found that the reduced-radix option is more suitable for ISE-supported long-integer arithmetic on RISC-V. This is again somewhat counter-intuitive since virtually all previous ISE for multi-precision integer arithmetic were designed for full-radix representation.

Part V

Side-Channel Leakage Analysis and Elimination

CHAPTER

9

A LEAKAGE-FOCUSED RISC-V ISE FOR MASKED IMPLEMENTATION

This Chapter is based on our paper [CP23].

9.1 Introduction

Use of masking to mitigate information leakage. Modern embedded computing devices are increasingly used in applications that can be deemed security-critical in some sense. This role is challenging due to the inherent constraints on storage, computation, and communication, and also because such devices may be deployed in an adversarial environment. Set within this context, implementation attacks, which focus on the concrete implementation rather than abstract specification of some functionality, represent a particularly potent threat. A side-channel attack is a category of implementation attack: the idea is that an attacker passively observes a target device while it executes some target functionality, using the observed behavior to make inferences about 1) the computation performed and/or 2) the data said computation is performed on. Doing so affords the attacker an advantage with respect to some goal, such as recovery of any security-critical information (e.g., key material) involved; we say such information is leaked via (or is leakage with respect to) the mechanism used for observation (i.e., the side-channel in question).

Although alternatives exist, we focus on Differential Power Analysis (DPA) [KJJ99] and variants thereof. The importance of robust countermeasures against DPA has motivated a significant amount of research activity, with techniques often classified as being based on hiding [MOP07, Chapter 7] and/or masking [MOP07, Chapter 10]. We focus on the latter, and, more specifically, the concept of a d-th order Boolean masking scheme. Such a scheme represents a variable x as $\hat{x} = \langle \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_d \rangle$, i.e., as $d + 1$ statistically independent shares, where

$$x = \bigoplus_{i=0}^{i \leq d} \hat{x}_i.$$

Application of the scheme to some functionality $r = f(x)$ can be described as three high-level steps: 1) x is masked to yield $\hat{x}$, 2) an alternative but compatible functionality $\hat{r} = \hat{f}(\hat{x})$ is executed, then 3) $\hat{r}$ is

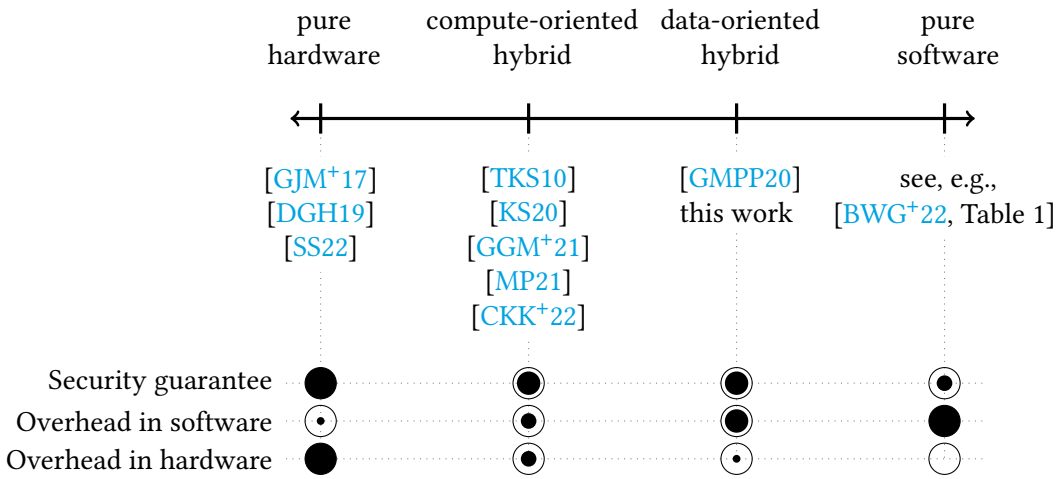


Figure 9.1: A selective overview of the design space for masked software implementation; indicative assessment of overhead and security guarantee is reflected by zero (◯), low (⊙), and high (●), plus various intermediate points.

unmasked to yield r. An attacker is now tasked with recovering $\hat{x}_i$ for all $0 \leq i \leq d$ using leakage which stems from $\hat{f}$, because x can no longer be recovered directly (as it might have been using leakage which stems from f). Put another way, such a scheme is designed to prevent a t-th order attack, in which the attacker is able to combine leakage from $t < d + 1$ points of interest. For example, a 1-st order scheme prevents a 1-st order attack but may be vulnerable to a 2-nd order attack.

Challenges stemming from production of a masked implementation. Consider a software implementation of some f, intended for execution by a micro-processor that supports a given Instruction Set Architecture (ISA), and the task of producing an associated masked implementation, i.e., an implementation of $\hat{f}$. At least two significant challenges stem from this task. The first challenge relates to efficiency, i.e., ensuring the masked implementation is efficient enough to be viable. Doing so is challenging because masking implies a notoriously high overhead due to factors such as computation on shares (i.e., overhead related to each "gadget" which represent the masked version of some non-masked functionality), storage of shares (e.g., register pressure due to the larger working set), and the requirement for generation of randomness; all the above are amplified when scalability to larger d is considered. The second challenge relates to security, i.e., translating theoretical security guarantees related to the masking scheme into practical guarantees related to the masked implementation. There is significant evidence that doing is challenging (cf. Beckers et al. [BWG+22]), e.g., due to the invalidity of theoretical assumptions on a given device. One common example is the occurrence of micro-architectural leakage (see, e.g., [PV17, MPW22]), which can invalidate 1) the only computation leaks assumption ("*computation, and only computation, leaks information*" [MR04, Section 2, Axiom 1]), and 2) independent leakage assumption ("*information leakage is local*" [MR04, Section 2, Axiom 4]).

A design space for masked implementation. Given the task outlined above, Figure 9.1 attempts to illustrate the design space of viable implementation strategies; a given strategy within said design space essentially selects whether software and/or hardware is responsible for (resp. aware of) or not responsible for (resp. unaware of) masking-specific properties of instructions and their execution. Toward the right-hand side are pure software or ISA-based implementation strategies, which place responsibility in software alone. These imply zero overhead in hardware, e.g., in relation to metrics such as area, but high overhead in software, e.g., in relation to metrics such as execution latency and memory footprint. Since hardware is unaware of masking, it cannot eliminate micro-architectural leakage; soft-

ware must address micro-architectural leakage via purely architectural means, e.g., using the ISA-based rewrite rules presented by Shelton et al. [SSB+21, Section V.C]. Toward the left-hand side are pure hardware implementation strategies, which place responsibility in hardware alone (typically via an entirely masked micro-architecture). These imply high overhead in hardware, but close to zero overhead in software. Since hardware is aware of masking, it can eliminate micro-architectural leakage; hardware can address micro-architectural leakage via micro-architectural means, e.g., through careful management of instruction execution. A variety of hybrid, implementation strategies exist between the two extremes. Generalizing a little, such strategies will typically share responsibility by 1) adding some limited, hardware-supported functionality for masking, and 2) exposing this functionality to software via an Instruction Set Extension (ISE); an ISE-based implementation strategy of this type naturally implies a compromise, namely some overhead in hardware and some overhead in software. Addressing micro-architectural leakage could be a shared responsibility, although, since hardware is aware of masking, a security guarantee more in line with a pure hardware implementation strategy is at least plausible.

It seems reasonable to claim there is no definitively best implementation strategy. Rather, each strategy will simply offer a different trade-off in terms of the metrics above plus other important examples such as usability (i.e., the burden on a software developer) and invasiveness (i.e., whether alteration of hardware is possible, and the scope and form of said alterations).

Contributions and organization. Within Figure 9.1, we claim there are (at least) two classes of hybrid, ISE-based implementation strategy:

1. a class of *compute*-oriented ISEs (which are closer to a pure hardware implementation strategy), where software indicates that the micro-architecture should execute masking-specific computation (e.g., a gadget) on masking-specific data (i.e., the shares used to represent a variable), and

2. a class of *data*-oriented ISEs (which are closer to a pure software implementation strategy), where software indicates that the micro-architecture should execute generic computation on masking-specific data.

We note that the data-oriented ISE class is at best less[1] explored than the compute-oriented ISE class, and at worst a gap in existing literature. In this Chapter we explore a specific instance of it. Conceptually, the ISE allows a leakage-focused behavioral hint[2] to be communicated from software to the micro-architecture; doing so informs how existing, generic computation is realized when applied to masking-specific data. After presenting relevant background information in Section 9.2, we organize the Chapter content as follows:

- In Section 9.3 we provide some technical analysis that fixes the scope of (i.e., provides a problem statement for) subsequent content. In short, we aim support an ISE-based implementation strategy which eliminates leakage stemming from architectural and micro-architectural overwriting.

- In Section 9.4 we present a concrete ISE design. We stress that although the design is based on RISC-V, or, more specifically, the RV32I [RV19, Section 2] base ISA, the concepts involved are more generally applicable.

[1] We note that the RISC-V Zkt [RVK22, Chapter 5] (meta-)extension is conceptually analogous: we do not include it in Figure 9.1, however, because it focuses on execution latency and so not masking nor micro-architectural leakage per se.

[2] As an aside, note that the same concept has been harnessed for various non-security use-cases across a range of existing ISAs. For example the ARMv6-M [ARM18, Section A6.6] and ARMv7-M [ARM21, Section A7.6] ISAs include a generic mechanism that can "*provide advance information to memory systems about future memory accesses, without actually loading or storing any data*"; the RISC-V RV32I [RV19, Section 2.9] and RV64I [RV19, Section 5.4] ISAs include a generic mechanism that can be "*used to communicate performance hints to the microarchitecture*"; the x86 ISA includes various specific mechanisms with applications that span branch prediction (e.g., branch taken and not taken prefixes [Int22a, Page 2-2]), pre-fetching (e.g., as in `prefetch` [Int22a, Page 4-414]), and non-temporal memory access (e.g., as in `movntdq` [Int22a, Page 4-99]).

- In Section 9.5 we explore prototype, latency- and area-optimized implementations of our ISE design, each based on the open source Ibex[3] base core. We stress that *any* implementation of the ISE will depend inherently on the base core (resp. micro-architecture); our implementations are intended to act as exemplars, therefore, rather than a limit on how the ISE could or should be implemented in general.
- In Section 9.6 we evaluate our prototype ISE implementations with respect to their impact on area, execution latency, and security guarantee; for the latter, we utilize the Coco [GHP+21, HB21] formal verification framework.

  Among existing[4] work with a similar remit, we view the Rosita tool of Shelton et al. [SSB+21] and FENL design of Gao et al. [GMPP20] as the most closely related; Section 9.6 offers a comparative evaluation of the ISE relative to such work.

Source code. Note that all material associated with this Chapter, e.g., documentation and source code relating to all hardware and software implementations, is openly available at `https://github.com/scarv/eliminate` under an open source license.

9.2 Background

9.2.1 RISC-V

We focus, without loss of generality, on a non-standard extension for RV32I, i.e., the 32-bit integer RISC-V base ISA; although such base ISAs use XLEN to denote the word size abstractly, our focus means we assume XLEN = 32 concretely throughout. We assume that some mechanism is available which supports the generation of randomness and hence fresh masks, so deem this out of scope; such a mechanism might, for example, but without loss of generality, be constructed using the RISC-V Zkr [RVK22, Chapter 4] extension.

9.2.2 Notation

Let $x_{(b)}$ denote x expressed in radix- or base-b; if the base is omitted, it is safe to assume use of decimal (i.e., that $b = 10$). Let $x \leftarrow y$ denote assignment of y to x, and $x \xleftarrow{\$} y$ denote selection of x uniformly at random from (e.g., a set) y. Let $\neg$, $\wedge$, $\vee$, and $\oplus$, denote the Boolean NOT, AND, (inclusive) OR, and (exclusive OR, or) XOR operators respectively, and $x \ll y$ and $x \lll y$ (resp. $x \gg y$ and $x \ggg y$) denote left-shift and left-rotate (resp. right-shift and right-rotate) of x by y bits respectively. Let $x \parallel y$ denote concatenation of x and y. Let $\text{ext}_0^w(x)$ and $\text{ext}_\pm^w(x)$ respectively denote zero- or sign-extension of x to w-bits. Let $\text{MEM}[i]^b$ denote a b-byte access to some byte-addressable memory, using the address i; where $b = 1$, the access granularity may be omitted. Let $\text{GPR}[i]$, for $0 \leq i < r$, denote the i-th, w-bit entry in the r-entry general-purpose register file. Note that our focus on RV32I means $\text{GPR}[0]$ is fixed to 0, in the sense reads from it always yield 0 and writes to it are ignored, $w = \text{XLEN} = 32$, and $r = 32$. We allow reference to Control and Status Registers (CSRs) using either a numeric- or mnemonic-based notation; per [RV21, Chapter 2], for example, $\text{CSR}[C00_{(16)}] \equiv$ `cycle` both refer to the cycle counter CSR.

[3] `https://github.com/lowRISC/ibex`.

[4] We note that the RISC-V Zkt [RVK22, Chapter 5] (meta-)extension is conceptually analogous: we omit it from Figure 9.1, however, because it focuses on execution latency and so not masking nor micro-architectural leakage per se. Likewise, we omit other fence instructions, e.g., [WSG+20, LHP20], due to the same lack of specificity.

The micro-architectural implementation of instructions may involve one or more steps. For example, the RISC-V load word instruction

$$\texttt{lw rd, imm(rs1)} \mapsto \text{GPR}[\texttt{rd}] \leftarrow \text{MEM}[\text{GPR}[\texttt{rs1}] + \texttt{imm}]^4$$

might be executed by 1) latching $s = \text{GPR}[\texttt{rs1}] + \texttt{imm}$ in a Memory Address Register (MAR), 2) carrying out a memory access to yield $v = \text{MEM}[s]^4$ then latching v in a Memory Buffer Register (MBR), 3) writing-back MBR into GPR[rd]. When describing the semantics of such an instruction, it can be important to show the cycle a given step is performed in. For example, we could describe the above as

$$\texttt{lw rd, imm(rs1)} \mapsto \begin{cases} 1 : \text{MAR} \leftarrow \text{GPR}[\texttt{rs1}] + \texttt{imm} \\ 2 : \text{MBR} \leftarrow \text{MEM}[\text{MAR}]^4 \\ 3 : \text{GPR}[\texttt{rd}] \leftarrow \text{MBR} \end{cases}$$

to show that the three steps are performed in cycles 1, 2 and 3, within what is therefore a 3-cycle execution stage. Said annotation may include ranges, e.g., $1 \ldots 3$ denotes cycles 1 to 3 inclusive: a step annotated as such is itself multi-cycle therefore. Annotation of multiple steps with the same cycle means they are performed in parallel; if no annotation appears, this means all steps are performed, in parallel, in cycle 1.

9.2.3 Terminology

Modulo details such as access granularity, memory and the register file can both be viewed as addressable forms of storage. As such, transfers between them can be modelled using the operation $\mathcal{T}[t] \leftarrow \mathcal{S}[s]$ noting that if $\mathcal{S} = \text{GPR}$ and $\mathcal{T} = \text{MEM}$ this models a store instruction type, whereas if $\mathcal{S} = \text{MEM}$ and $\mathcal{T} = \text{GPR}$ this models a load instruction type; in both cases, s and t are the (effective) source and target addresses respectively.

Terminology 1. We focus on data, and so, e.g., the MBR throughout: noting that neither use of nor terminology for the MBR is consistent, if $\mathcal{S} = \text{GPR}$ and $\mathcal{T} = \text{MEM}$ we term it the **store buffer**, if $\mathcal{S} = \text{MEM}$ and $\mathcal{T} = \text{GPR}$ we term it the **load buffer**, and if the MBR is bi-directional (i.e., one MBR is used to support both operations) we term it the **load/store buffer**.

Terminology 2. We distinguish between resources which are physically internal or external to the micro-architecture: we term such resources **intra-core** or **extra-core** resources respectively.

Terminology 3. We distinguish between resources which permit **direct control** (e.g., via specific control signals) or require **indirect control** (i.e., via an abstraction layer or interface).

For example, a load/store buffer might be intra-core or extra-core (e.g., exist within an SRAM module, or bus connecting such a module to the core): the former would permit direct control by the micro-architecture but require indirect control by software, whereas the latter would require indirect control by both the micro-architecture and software.

Various work has identified architectural and micro-architectural leakage effects which relate to unintentional share recombination shown to occur during transfer of shares between forms of storage. For example, using an ST-based ARM Cortex-M0 [ARM09] target device, Shelton et al. [SSB+21, Section IV.E] carry out experiments which identify leakage stemming from overwriting one value with another 1) within $\mathcal{T} = \text{GPR}$ (see [SSB+21, Section IV.E.1]) or $\mathcal{T} = \text{MEM}$ (see [SSB+21, Section IV.E.2]), and 2) within the interface, i.e., a load or store buffer between $\mathcal{S}$ and $\mathcal{T}$ (see [SSB+21, Section IV.E.4]).

Terminology 4. We refer to the cases above as **architectural overwriting** and **micro-architectural overwriting**, because they stem from architectural and micro-architectural resources respectively.

9.3 Analysis

Some leakage-focused requirements for share transfer. Gaspoz and Dhooghe [GD23] introduce what they term horizontal [GD23, Definition 5] and vertical [GD23, Definition 6] non-completeness requirements on the representation of variables: their goal is to prevent unintentional share recombination that might stem from inter- and intra-register interaction respectively. One could imagine attempting to introduce analogous requirements to guide the transfer of shares between memory and the register file. For example:

Requirement 5 (Architectural overwriting)**.** Imagine instructions of the form $\mathcal{T}[t] \leftarrow v_0$ and $\mathcal{T}[t] \leftarrow v_1$ are executed in cycles i and $j > i$ respectively, and that no intermediate instructions that update $\mathcal{T}[t]$ are executed, i.e., no instruction of the form $\mathcal{T}[t] \leftarrow v_2$ is executed in cycle k where $i < k < j$. If v_0 equals $\hat{x}_p$ for some $0 \leq p \leq d$, one must ensure that $v_1 \neq \hat{x}_q$ for all $0 \leq q \leq d$.

Requirement 6 (Micro-architectural overwriting)**.** Imagine instructions of the form $\mathcal{T}[t_0] \leftarrow \mathcal{S}[s_0]$ and $\mathcal{T}[t_1] \leftarrow \mathcal{S}[s_1]$ are executed in cycles i and $j > i$ respectively, and that no intermediate instructions of the same type are executed, i.e., no instruction of the form $\mathcal{T}[t_2] \leftarrow \mathcal{S}[s_2]$ is executed in cycle k where $i < k < j$. If $\mathcal{S}[s_0]$ equals $\hat{x}_p$ for some $0 \leq p \leq d$, one must ensure that $\mathcal{S}[s_1] \neq \hat{x}_q$ for all $0 \leq q \leq d$.

The aim of these requirements is to eliminate leakage stemming from $\hat{x}_p$ being overwritten with some $\hat{x}_q$: put simply, the former requirement does so by preventing architectural overwriting while the latter requirement does so by preventing micro-architectural overwriting. Note that the former requirement is more general than required by the context, in the sense it captures *any* instruction which updates $\mathcal{T}[t]$ (rather than load or store instructions specifically).

ISA-based requirement satisfaction. As part of a pure software implementation strategy, both architectural and micro-architectural overwriting must be prevented by using the ISA alone: for architectural resources this fact implies use of direct control, whereas for micro-architectural resources it implies use of indirect control. The Rosita tool of Shelton et al. [SSB+21, Section V.C] offers an excellent example of how to do so concretely. [SSB+21, Section V.A] outlines the main strategy: Rosita reserves a (random) mask register `r7`, and uses this to flush architectural and micro-architectural state, i.e., shares, by rewriting pertinent instructions. For example:

1. Imagine GPR[4] $= \hat{x}_p$. Per [SSB+21, Section V.A], Rosita might rewrite

   ```
   movs r3, r4  ↦  movs r3, r7 ; movs r3, r4
   ```

   to prevent architectural overwriting: doing so randomizes GPR[3] before it is overwritten.

2. Imagine MEM[GPR[3]] $= \hat{x}_p$. Per [SSB+21, Section V.E], Rosita might rewrite

   ```
   ldr r2, [ r3 ]  ↦  push r7 ; pop r2 ; ldr r2, [ r3 ]
   ```

   to prevent architectural and micro-architectural overwriting: doing so randomizes GPR[2] and the load buffer before they are overwritten.

3. Imagine MEM[GPR[2]] $= \hat{x}_p$. Per [SSB+21, Section V.E], Rosita might rewrite

   ```
   str r2, [ r3 ]  ↦  str r7, [ r3 ] ; str r2, [ r3 ]
   ```

   to prevent architectural and micro-architectural overwriting: doing so randomizes MEM[GPR[3]] and the store buffer before they are overwritten.

Even given a set of requirements, whose specification is a challenge in and of itself, we make two claims about an ISA-based strategy for their satisfaction along the lines above. First, an ISA-based strategy may be sub-optimal with respect to efficiency. Consider the example above, where Rosita prevents architectural and micro-architectural overwriting related to an `ldr` instruction:the rewrite translates 1 load instruction (resp. memory access) into 3. Although the overhead differs on a case-by-case basis (both per-instruction and per-ISA), it clearly may be significant. Second, an ISA-based strategy may be sub-optimal with respect to security. In particular, there are clear limitations on how effective indirect control of a micro-architectural resource can be. Consider the same example above: the security guarantee offered will depend on validity of assumptions about the micro-architecture, e.g., that the `push` and `pop` instructions use the same data-path and hence store buffer as the `ldr` instruction. In fact, some instructions can prevent either direct or indirect control over pertinent micro-architectural resources. Consider the store instruction variants in ARMv6-M: in contrast to the single-access variant `str` [ARM18, Section A6.7.60], the multi-access variant `stm` [ARM18, Section A6.7.58] "*store[s] multiple registers to consecutive memory locations using an address from a base register*". So if GPR[1] = $\hat{x}_p$ and GPR[2] = $\hat{x}_q$, then while executing `stm r0, { r1, r2 }` one cannot prevent $\hat{x}_q$ from overwriting $\hat{x}_p$ within a store buffer: the instruction semantics mean one cannot control the order registers are accessed in, nor take a Rosita-like approach by randomizing the store buffer between accesses.

An argument for ISE-based requirement satisfaction. We claim the points above stem from the role of an ISA as an abstraction of the micro-architecture, and thus relevant resources. Somewhat aligned with the argument of Ge, Yarom, and Heiser [GYH18] for a "*new security-oriented hardware/-software contract*", we propose to address this fact using an ISE-based strategy. Specifically, we aim to design a data-oriented ISE class which is leakage-focused: the ISE should eliminate leakage stemming from architectural and micro-architectural overwriting. This goal can be described as necessary but not sufficient, in the sense that additional forms of micro-architectural leakage may also need to be considered.

It is important to stress that doing so has inherent limitations, reflecting the idea in Section 9.1 that it simply offers a different trade-off. For example, relative to a compute-oriented ISE, a data-oriented ISE cannot be competitive in terms of execution latency because it does not add support for masking-specific computation; we focus on comparison with an ISA-based strategy therefore. Likewise, under the conservative assumption that extra-core resources require indirect control by the micro-architecture, neither a compute-oriented nor data-oriented ISEs can deliver an "ideal" security guarantee: again, we focus on comparison with an ISA-based strategy therefore.

9.4 Design

In this Section, we present the ISE design. To explain it at a high level, consider, without loss of generality, the RISC-V load word instruction

$$\texttt{lw rd, imm(rs1)} \mapsto \texttt{GPR[rd]} \leftarrow \texttt{MEM[GPR[rs1]} + \texttt{imm]}^4$$

and some (abstractly defined) mechanism denoted

$$\texttt{GPR[rd]} \overset{\triangle}{\longleftarrow} \texttt{MEM[GPR[rs1]} + \texttt{imm]}^4$$

which represents a variant of the existing semantics. The existing and variant semantics are functionally identical but may be behaviorally different: the existing semantics are expressed in [RV19, Section 2.6] as "*loads a* 32*-bit value from memory into* `rd`", whereas the variant semantics might be expressed as "*loads a* 32*-bit value from memory into* `rd`*, preventing any architectural and micro-architectural overwriting while doing so*" by appending a written hint which controls how they are implemented: the

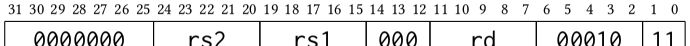
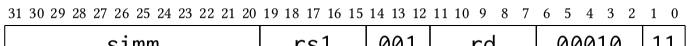
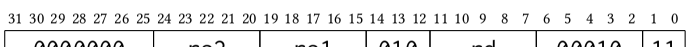
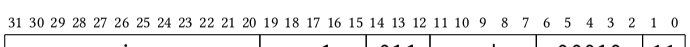
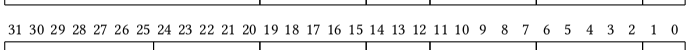
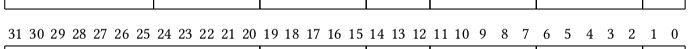
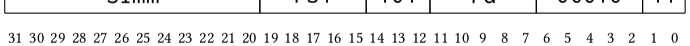
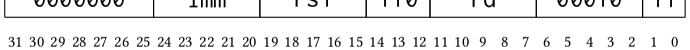
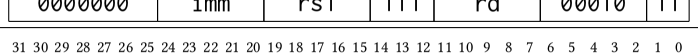
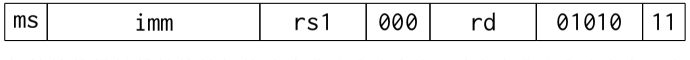
Class-1	`sec.and rd, rs1, rs2`	:		$\mapsto \quad \mathsf{GPR}[\mathtt{rd}] \xleftarrow{\triangle} \mathsf{GPR}[\mathtt{rs1}] \wedge \mathsf{GPR}[\mathtt{rs2}]$
	`sec.andi rd, rs1, imm`	:		$\mapsto \quad \mathsf{GPR}[\mathtt{rd}] \xleftarrow{\triangle} \mathsf{GPR}[\mathtt{rs1}] \wedge \mathrm{ext}^{w}_{\pm}(\mathtt{simm})$
	`sec.or rd, rs1, rs2`	:		$\mapsto \quad \mathsf{GPR}[\mathtt{rd}] \xleftarrow{\triangle} \mathsf{GPR}[\mathtt{rs1}] \vee \mathsf{GPR}[\mathtt{rs2}]$
	`sec.ori rd, rs1, imm`	:		$\mapsto \quad \mathsf{GPR}[\mathtt{rd}] \xleftarrow{\triangle} \mathsf{GPR}[\mathtt{rs1}] \vee \mathrm{ext}^{w}_{\pm}(\mathtt{simm})$
	`sec.xor rd, rs1, rs2`	:		$\mapsto \quad \mathsf{GPR}[\mathtt{rd}] \xleftarrow{\triangle} \mathsf{GPR}[\mathtt{rs1}] \oplus \mathsf{GPR}[\mathtt{rs2}]$
	`sec.xori rd, rs1, imm`	:		$\mapsto \quad \mathsf{GPR}[\mathtt{rd}] \xleftarrow{\triangle} \mathsf{GPR}[\mathtt{rs1}] \oplus \mathrm{ext}^{w}_{\pm}(\mathtt{simm})$
	`sec.slli rd, rs1, imm`	:		$\mapsto \quad \mathsf{GPR}[\mathtt{rd}] \xleftarrow{\triangle} \mathsf{GPR}[\mathtt{rs1}] \ll \mathtt{imm}$
	`sec.srli rd, rs1, imm`	:		$\mapsto \quad \mathsf{GPR}[\mathtt{rd}] \xleftarrow{\triangle} \mathsf{GPR}[\mathtt{rs1}] \gg \mathtt{imm}$
Class-2	`sec.lw rd, rs1, imm, ms`	:		$\mapsto \quad \mathsf{GPR}[\mathtt{rd}] \xleftarrow{\triangle} \mathsf{MEM}[\mathsf{GPR}[\mathtt{rs1}] + \mathtt{imm}]^4$
	`sec.sw rs2, rs1, imm, ms`	:		$\mapsto \quad \mathsf{MEM}[\mathsf{GPR}[\mathtt{rs1}] + \mathtt{imm}]^4 \xleftarrow{\triangle} \mathsf{GPR}[\mathtt{rd}]$

Table 9.1: A summary of additional instructions that constitute the ISE, described in terms of assembly language syntax (left), encoding (middle), and semantics (right).

hint essentially captures a guarantee that leakage stemming from architectural and micro-architectural overwriting will be eliminated by said implementation. Note that expression of the written hint requires some care, because under-specification means any value it affords is degraded (because the semantics offer too weak a security guarantee), whereas over-specification means the semantics may be unimplementable. We attempt to balance these facts by capturing the goal in Section 9.3 while also permitting some degree of micro-architectural flexibility, i.e., multiple viable micro-architectural implementations.

Framed as such, the ISE can be viewed as two somewhat orthogonal components. First, a general mechanism by which such a hint can be encoded programmatically: Section 9.4.1 explores and selects from a range of candidate mechanisms, each of which offers advantages and disadvantages given the goal at hand. Second, a specific set of instructions: as summarized by Table 9.1, we divide this set into instruction classes outlined in Section 9.4.2 and Section 9.4.3.

9.4.1 Encoding

Candidate #1. One could define variant instructions, providing the necessary hint via their use. For example, one could define and then use

$$\texttt{sec.lw rd, imm(rs1)} \;\mapsto\; \texttt{GPR[rd]} \xleftarrow{\triangle} \texttt{MEM[GPR[rs1]} + \texttt{imm]}^4$$

for security-critical cases.

Candidate #2. One could redefine existing instructions, providing the necessary hint via management of a processor mode. For example, given SEC, a Control and Status Register (CSR)for said mode, one could redefine

$$\texttt{lw rd, imm(rs1)} \;\mapsto\; \begin{cases} \texttt{GPR[rd]} \xleftarrow{\triangle} \texttt{MEM[GPR[rs1]} + \texttt{imm]}^4 & \text{if SEC} = 1 \\ \texttt{GPR[rd]} \leftarrow \texttt{MEM[GPR[rs1]} + \texttt{imm]}^4 & \text{otherwise} \end{cases}$$

and then use SEC = 1 for security-critical cases. This approach is conceptually similar to those now employed by ARM via Data Independent Timing (DIT) and by Intel via Data Operand Independent Timing (DOIT). For a capability-enabled ISA (e.g., one that supports CHERI [WMSN19]), it may be possible to control the mode via a capability associated with the program counter: this would allow the variant semantics to be applied while executing the masked implementation, and the existing semantics otherwise.

Candidate #3. One could redefine existing instructions, providing the necessary hint via their operands. For example, given SEC, a set of distinguished registers, one could redefine

$$\texttt{lw rd, imm(rs1)} \;\mapsto\; \begin{cases} \texttt{GPR[rd]} \xleftarrow{\triangle} \texttt{MEM[GPR[rs1]} + \texttt{imm]}^4 & \text{if rd} \in \text{SEC} \\ \texttt{GPR[rd]} \leftarrow \texttt{MEM[GPR[rs1]} + \texttt{imm]}^4 & \text{otherwise} \end{cases}$$

and then use an rd $\in$ SEC for security-critical cases. This approach is conceptually similar to that outlined by Escouteloup et al. [EFLL20, Recommendation 1], who apply some security-focused requirements and semantics to a set of general-purpose registers, e.g., SEC = $\{8, 9, \ldots, 15\}$, deemed confidential. As above, and at least for load and store instructions (where GPR[rs1] can be viewed as a pointer) within a capability-enabled ISA, an alternative way to designate a register as distinguished might be via a capability.

Candidate #4. One could redefine existing instructions, providing the necessary hint via micro-architectural compliance with a suitable specification. For example, a non-compliant micro-architecture

could retain

$$\texttt{lw rd, imm(rs1)} \;\mapsto\; \mathsf{GPR}[\texttt{rd}] \leftarrow \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4,$$

whereas a compliant micro-architecture could redefine

$$\texttt{lw rd, imm(rs1)} \;\mapsto\; \mathsf{GPR}[\texttt{rd}] \overset{\triangle}{\leftarrow} \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4.$$

This approach is conceptually similar to the RISC-V Zkt [RVK22, Chapter 5] (meta-)extension, which, rather than defining functionality per se, simply "*attests that the machine has data-independent execution time for a safe subset of instructions*".

Summary. The candidates presented above can be summarized as follows

	Definition	Invocation
Candidate #1	Variant	Unconditionally dynamic
Candidate #2	Existing	Conditionally dynamic
Candidate #3	Existing	Conditionally dynamic
Candidate #4	Existing	Static

using two properties. First, a given mechanism can either define variant instructions or redefine (or overload) existing instructions. Second, invocation of a given mechanism can either be 1) static, i.e., the variant semantics are "always on" or "always off", 2) conditionally dynamic, i.e., the variant semantics are "opt in" but there is some overhead or constraint, or 3) unconditionally dynamic, i.e., the variant semantics are "opt in" and there is no overhead nor constraint.

We anticipate that a masked implementation will use a limited subset of the ISA and form a limited component of the overall workload. Although all candidates are viable, these factors suggest candidate #1 would be an effective choice: although it consumes encoding space, it permits a targeted, self-contained extension (limiting impact on the ISA as a whole) with no overhead related to invocation (for masking-specific instruction sequences) or non-invocation (for generic instruction sequences).

9.4.2 Class-1 instructions: computation-related

Concept. Consider the (optimized) SecAnd and SecOr gadgets for $d = 2$ presented by Biryukov et al. [BDLU17, Table 1], for example, which represent the masked versions of AND and OR respectively:

function SecAND$((\hat{x}_0, \hat{x}_1), (\hat{y}_0, \hat{y}_1))$
- $\hat{r}_1 \leftarrow (\hat{x}_1 \wedge \hat{y}_1) \oplus (\hat{x}_1 \vee \neg\hat{y}_0)$
- $\hat{r}_0 \leftarrow (\hat{x}_0 \wedge \hat{y}_1) \oplus (\hat{x}_0 \vee \neg\hat{y}_0)$
- **return** $(\hat{r}_0, \hat{r}_1)$

function SecOR$((\hat{x}_0, \hat{x}_1), (\hat{y}_0, \hat{y}_1))$
- $\hat{r}_1 \leftarrow (\hat{x}_1 \wedge \hat{y}_1) \oplus (\hat{x}_1 \vee \hat{y}_0)$
- $\hat{r}_0 \leftarrow (\hat{x}_0 \vee \hat{y}_1) \oplus (\hat{x}_0 \wedge \hat{y}_0)$
- **return** $(\hat{r}_0, \hat{r}_1)$

Although generalization to other functionality and larger d is clearly important, we claim these gadgets act as exemplars: they are implemented using a (short) sequence of bit-wise logical and shift instructions. As such, the goal of this instruction class is to provide a minimal set of such instructions to support the implementation of a maximal set of gadgets.

Instructions. Per Table 9.1, this instruction class includes `sec.andi` (resp. `sec.and`), `sec.ori` (resp. `sec.or`), `sec.xori` (resp. `sec.xor`), `sec.slli`, and `sec.srli`: these instructions support register-immediate (resp. register-register) variants of AND, OR, XOR, left-shift, and right-shift respectively. In line with the approach taken by the base ISA, note that NOT can be synthesized by using XOR: doing so relies on the fact that $\neg x \equiv x \oplus \mathrm{ext}^w_{\pm}(-1)$.

Number	Privilege	Name	Description
$800_{(16)}$	read/write	`ms0`	Mask seed #0
$801_{(16)}$	read/write	`ms1`	Mask seed #1
$802_{(16)}$	read/write	`ms2`	Mask seed #2
$803_{(16)}$	read/write	`ms3`	Mask seed #3

Table 9.2: Additional mask seed CSRs which support class-2 instructions.

9.4.3 Class-2 instructions: storage-related

Concept. The goal of this instruction class is to support transfer of shares between memory and the register file using load and store instructions. During execution of the masked implementation, we claim use of such instructions is dominated by spilling, i.e., temporary use of (a larger) memory to deal with pressure on the register file (stemming from the smaller size); this implies some structure, in the sense that loads (to pop, or restore some shares) and stores (to push, or preserve some shares) will be "grouped" into phases rather than used in a more isolated, ad hoc manner.

As well as a destination (resp. source) register address, load (resp. store) instruction provided by this class must specify 1) an effective address (via a base register address, plus an immediate offset), and 2) a mask seed; the former mirrors existing RISC-V load (resp. store) word instructions, whereas the latter is an addition to and so deviation from them. Furthermore, two constraints apply to their use. First, from a functional perspective, we assume load and store instructions operate in pairs. For example, consider two instructions: the first stores v_0 at address a_0 using mask seed m_0, whereas the second loads v_1 from address a_1 using mask seed m_1. These instructions form a load/store pair iff. $a_0 = a_1$ and $m_0 = m_1$; otherwise, there is no guarantee that $v_0 = v_1$. Second, from a behavioral perspective, a security guarantee is offered iff. each load/store pair uses a unique combination of address and mask seed. For example, consider two instructions: the first stores v_0 at address a_0 using mask seed m_0, whereas the second stores v_1 at address a_1 using mask seed m_1. If $a_0 \neq a_1$ or $m_0 \neq m_1$, the guarantee offered is that no leakage will stem from v_1 overwriting v_0.

As will becomes more obvious later in Section 9.5, the design represents an interface that allows several micro-architectural implementations, e.g., enable an approach which randomizes (or remasks) shares while storing them into memory then derandomizes shares while loading them from memory. Variants of this approach are applied, e.g., De Mulder, Gummalla, and Hutter [DGH19, Section 4], and Stangherlin and Sachdev [SS22, Section G]; one could also view it as an realization of Escouteloup et al. [EFLL20, Recommendation 2], i.e., to "*encrypt the confidential data in memory, as soon as it leaves the pipeline*".

State. Per Table 9.2, instructions in this class are supported by four additional CSRs: $\mathsf{CSR}[800_{(16)} + i]$ for $0 \leq i < 4$ denotes the i-th mask seed. The CSRs must be initialized with fresh randomness *before* execution of the masked implementation (or at least before their first use); we assume the overhead of doing so is amortized by the execution latency of said implementation as a whole. The CSRs may need to be refreshed *during* execution of the masked implementation, e.g., to satisfy the two constraints outlined above.

Instructions. Per Table 9.1, this instruction class includes `sec.lw` and `sec.sw`: these instructions support load word and store word memory access respectively. The encodings for `sec.lw` and `sec.sw` reserve two MSBs of `imm` for some meta-data `ms`, which is used to specify the mask seed, namely $\mathsf{CSR}[800_{(16)} + \mathtt{ms}]$. Note that doing so reduces `imm` from 12 to 10 bits, and thus the range from $2^{12} = 4096$ to $2^{10} = 1024$.

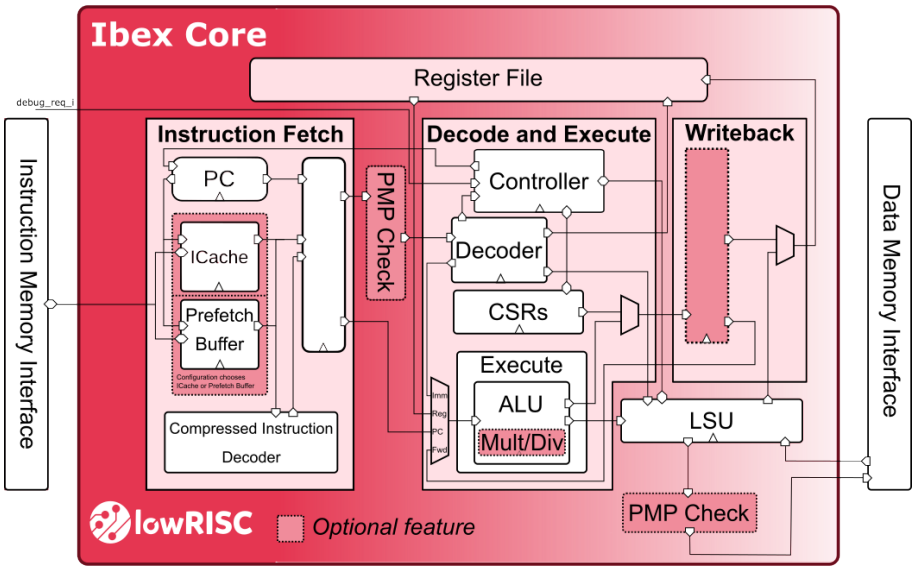


Figure 9.2: A block diagram describing the Ibex micro-architecture (image source: `blockdiagram.svg`, obtained from `https://github.com/lowRISC/ibex`).

9.5 Implementation

In this Section, we present prototype implementations of our ISE design: Section 9.5.1 introduces the base core, after which Section 9.5.2 and Section 9.5.3 then describe latency- and area-optimized implementations of the ISE within it.

We stress (again) that *any* implementation of the ISE will depend on the base core (resp. micro-architecture), meaning certain aspects of them are tailored to suit Ibex specifically. For example, we assume use of a generic SRAM module: we can neither select nor modify the specific SRAM module combined with the core. This suggests a conservative approach, where *potential* leakage (stemming, e.g., from a potential load/store buffer within the SRAM module) is eliminated using indirect control; doing so means greater overhead, but also a more robust security guarantee. However, we note that it is clearly possible and, depending on the context, attractive to do the opposite: in their CocoIbex core, for example, Gigerl et al. [GHP+21, Section 4] use a special-purpose SRAM module that eliminates certain forms of leakage.

9.5.1 Base core: Ibex

General overview. Ibex is a 32-bit, RISC-V compliant micro-processor core, which is designed for embedded use-cases; originally developed as part of the PULP[5] platform, the core (and a suite of associated resources) is now maintained by lowRISC. Ibex supports both FPGA- and ASIC-based synthesis targets: the block diagram in Figure 9.2 describes the micro-architectural design, which is highly configurable. For example, the core can support either the integer (i.e., RV32I) or embedded (i.e., RV32E) RISC-V base ISA; said base ISA can be supplemented by the multiplication [RV19, Chapter 7], compressed [RV19, Chapter 16], or bit manipulation [RV19, Chapter 17] extensions; the micro-architecture can use either a 2- or 3-stage pipeline (by excluding or including a dedicated write-back stage), and supports options relating to the multiplier, branch prediction, and Physical Memory Protection (PMP). Beyond this, implementation of specific units can be specialized to suit the underlying technology; the register file can be implemented using flip-flops, latches, or RAM elements, for example, in order to suit the synthesis target.

[5] `https://pulp-platform.org`.

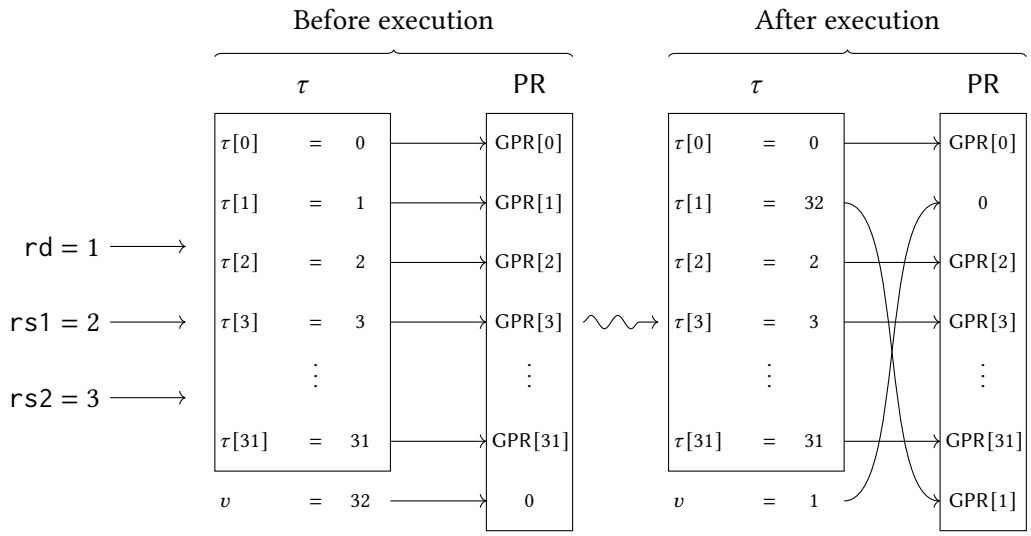


Figure 9.3: A diagrammatic description of how PR, GPR, τ, and υ are managed during execution of `sec.xor x1, x2, x3` by the latency-optimized implementation.

Specific configuration. We develop the prototype ISE implementation and perform our experiments on Ibex Demo System[6], which is also developed by lowRISC and comprises the Ibex core. We select to use a flip-flop-based register file. For other settings, we adopt the default configuration of Ibex Demo System; the ISA is RV32IMC, i.e., bit manipulation extension is not enabled; a fast multi-cycle multiplier and an iterative divider are used; 2-stage pipeline is used, which includes an Instruction Fetch (IF) stage and an Instruction Decode and Execute (ID+EX) stage, while Write-Back (WB) is not enabled as a dedicated stage; PMP and instruction cache are disabled.

9.5.2 ISE implementation #1: latency-optimized

Class-1 instructions. For the implementation of class-1 instructions, we design a new mechanism for indexing the registers. In order to make it clearer, we introduce a term Physical Register, denoted by PR. In the base core, GPR$[i]$ and PR$[i]$ refer to exactly the same registers; given a register index (or say address) k there is only one map $k \Rightarrow$ PR$[k]$ (equally $k \Rightarrow$ GPR$[k]$) used by reading (resp. writing) the data from (resp. to) the target physical register. In our implementation, first of all, GPR and PR are different: the use of GPR is viewable to software in the sense that developers can see what GPRs are being used and can choose what GPRs to use in their code (instructions); however, the use of PR is not viewable to software and is controlled by only micro-architecture. We add a new index look-up table τ between the two objects of the original map, i.e., the register index and the target physical register, and change to use two maps to link them such that $k \Rightarrow \tau[k]$ then $\tau[k] \Rightarrow$ PR$[\tau[k]]$. But for GPR, it is still a direct map from k to GPR$[k]$. Furthermore, we add one more physical register PR$[32]$ to the register file. In our setting, in each clock cycle there are 32 general-purpose registers and 1 *idle register* that holds a value 0 all the time. In more detail, there are 32 entries in τ (see a diagrammatic description in Figure 9.3), each of which stores the index of a general-purpose register, and there is another variable υ used to hold the index of the idle register. At the beginning (i.e., after a reset signal), each entry of τ is initialized as $\tau[i] \leftarrow i$, and υ points to PR$[32]$. In each instruction executed, we always use the current idle register, i.e., PR$[\upsilon]$, as the destination (physical) register, and set PR$[\tau[$`rd`$]]$ to be the new idle register. In this way, the data is always written to a cleared register, which prevents the architectural overwriting in the register file. Of course, the values of entries in τ and of υ update dynamically according to the different instructions executed. Note if `rd` is 0, we will not update τ and υ.

[6] https://github.com/lowRISC/ibex-demo-system.

For the software side, there is no difference in the use of GPR. Taking `sec.xor` as an example, a formal definition is as follows:

$$\texttt{sec.xor rd, rs1, rs2} \mapsto \begin{cases} \mathrm{PR}[\upsilon] \leftarrow \mathrm{PR}[\tau[\texttt{rs1}]] \oplus \mathrm{PR}[\tau[\texttt{rs2}]] \\ \mathrm{PR}[\tau[\texttt{rd}]] \leftarrow 0 \\ \tau[\texttt{rd}] \leftarrow \upsilon \\ \upsilon \leftarrow \tau[\texttt{rd}] \end{cases}$$

For easy understanding, in Figure 9.3 we consider an example that `sec.xor x1, x2, x3` is executed. It reads operands $\mathrm{PR}[\tau[2]]$ and $\mathrm{PR}[\tau[3]]$, i.e., in essence $\mathrm{PR}[2]$ and $\mathrm{PR}[3]$ per current τ, computes the result, writes the result to the idle register $\mathrm{PR}[\upsilon]$, i.e., $\mathrm{PR}[32]$, and clears the register $\mathrm{PR}[\tau[1]]$, i.e., $\mathrm{PR}[1]$. In the meantime, υ changes to be $\tau[1]$, i.e., 1, which indicates the idle register is now $\mathrm{PR}[1]$, and $\tau[1]$ should accordingly update to be 32 as well, i.e., $\mathrm{GPR}[1]$ is now essentially the register $\mathrm{PR}[32]$.

Class-2 instructions. We implement the class-2 instructions based on the re-masking method of De Mulder et al. [DGH19, Section 4]. When storing (resp. loading) a share to (resp. from) the memory, the share is always masked with a Load/Store Mask (LSM), which eliminates both the architectural overwriting in the memory and the micro-architectural overwriting in the MBR. Plus our new mechanism for indexing the registers, both `sec.sw` and `sec.lw` prevent the architectural and micro-architectural overwriting. As described in [DGH19], it suggests using 2 or 3 rounds of Keccak-f100 permutation [BDPA13] to generate an LSM, where the state is formed with the memory address and a mask seed from a specific CSR. In our implementation, we use 2 rounds of Keccak-f100 and define the generation of an LSM as (given `rs1`, `imm`, and `ms` from a class-2 instruction):

$$\mathrm{LSM} := \textsc{keccak-f100-2rounds}(0 \cdots 0 \;\|\; (\mathrm{PR}[\tau[\texttt{rs1}]] + \texttt{imm}) \;\|\; \mathrm{CSR}[800_{(16)} + \texttt{ms}])$$

We implement the 2 rounds of Keccak-f100 in an unrolled way to ensure a single-cycle execution. As indicated in Table 9.2, we add four mask seed CSRs with the addresses of $800_{(16)}$ to $803_{(16)}$, which, defined in [RV21, Table 2.1], are preserved for the use of custom read/write. `ms` selects which CSR to be used. In formal, `sec.sw` and `sec.lw` are defined as:

$$\texttt{sec.sw rs2, rs1, imm, ms} \mapsto \mathrm{MEM}[\mathrm{PR}[\tau[\texttt{rs1}]] + \texttt{imm}]^4 \leftarrow \mathrm{PR}[\tau[\texttt{rs2}]] \oplus \mathrm{LSM}$$

$$\texttt{sec.lw rd, rs1, imm, ms} \mapsto \begin{cases} \mathrm{PR}[\upsilon] \leftarrow \mathrm{MEM}[\mathrm{PR}[\tau[\texttt{rs1}]] + \texttt{imm}]^4 \oplus \mathrm{LSM} \\ \mathrm{PR}[\tau[\texttt{rd}]] \leftarrow 0 \\ \tau[\texttt{rd}] \leftarrow \upsilon \\ \upsilon \leftarrow \tau[\texttt{rd}] \end{cases}$$

9.5.3 ISE implementation #2: area-optimized

Class-1 instructions. We again take the `sec.xor` as an example to elaborate the implementation details of class-1 instructions. Note that we do not import and use the concept of PR in area-optimized implementation. At the high-level operation viewpoint, the `sec.xor` instruction is composed of two steps, namely:

$$\texttt{sec.xor rd, rs1, rs2} \mapsto \begin{cases} 1 : \mathrm{GPR}[\texttt{rd}] \leftarrow 0 \\ 2 : \mathrm{GPR}[\texttt{rd}] \leftarrow \mathrm{GPR}[\texttt{rs1}] \oplus \mathrm{GPR}[\texttt{rs2}] \end{cases}$$

For the low-level hardware implementation, in the decoder, a dedicated variable named `sec_bwlogic` (secure bitwise logical instruction) will be set to 1 when the class-1 instruction is decoded, and to 0 otherwise. When `sec_bwlogic` is 1, the ID stage stalls in the first clock cycle, and at the same time the

Core	Registers	LUTs
Base core	2363 (1.00×)	3602 (1.00×)
Base core + latency-optimized class-1	2585 (1.09×)	4829 (1.34×)
Base core + latency-optimized class-1+2	2713 (1.15×)	5000 (1.39×)
Base core + area-optimized class-1	2365 (1.00×)	3565 (0.99×)
Base core + area-optimized class-1+2	2365 (1.00×)	3847 (1.07×)

Table 9.3: Comparison of area, stemming from synthesis of the base core plus implementation #1 (latency-optimized, per Section 9.5.2) and implementation #2 (area-optimized, per Section 9.5.3) of the ISE; note that cumulative support for instruction classes is presented to highlight their individual contribution.

signal of `sec_bwlogic` is transmitted to register file and drives to clear the destination register. In the next clock cycle, it just works the same as the case of a normal `xor`, i.e., computing the result and writing it to the destination register. The computation of class-1 instructions is realized by simply (re-)using the hardware implementation of normal bitwise logical instructions (in ALU), hence its hardware cost is negligible.

Class-2 instructions. Essentially, the memory access instructions are implemented based on a pure-software implementation strategy used in Rosita [SSB+21]. Therefore, the LSM as well as the mask seed are not needed in this implementation; we also do not add four mask seed CSRs to further save the area overhead, and `ms` has no impact on the action of instruction. In detail, `sec.sw` consists of two steps whereas `sec.lw` needs three steps, and they are shown as follows:

$$\texttt{sec.sw rs2, rs1, imm, ms} \mapsto \begin{cases} 1\text{–}2 : \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 \leftarrow 0 \\ 3\text{–}4 : \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 \leftarrow \mathsf{GPR}[\texttt{rs2}] \end{cases}$$

$$\texttt{sec.lw rd, rs1, imm, ms} \mapsto \begin{cases} 1\text{–}2 : \mathsf{MEM}[\mathsf{GPR}[\texttt{sp}] + (\texttt{-4})]^4 \leftarrow 0 \\ 3\text{–}4 : \mathsf{GPR}[\texttt{rd}] \leftarrow \mathsf{MEM}[\mathsf{GPR}[\texttt{sp}] + (\texttt{-4})]^4 \\ 5\text{–}6 : \mathsf{GPR}[\texttt{rd}] \leftarrow \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 \end{cases}$$

In the low-level hardware implementation, in order to make class-2 instructions work correctly in each of their different steps, it is required to introduce some new states to the finite state machine (FSM) of the ID stage and of the load-store unit respectively, which constitute the most of additional hardware cost of class-2. Similar to class-1 instructions, we add two dedicated variables `sec_store` (secure store) and `sec_load` (secure load), whose values get updated in the decoder, and they are used by the ID-stage FSM and the load-store unit FSM. Furthermore, some other modifications in the decoder are also needed; e.g., in the first two steps of `sec.lw`, it reads the data of stack pointer register `sp` instead of the source register `rs1`.

9.6 Evaluation

To produce an experimental platform which permits evaluation of area and cycle-accurate execution latency, we use an Arty-100T board[7], which hosts a Xilinx Artix-7 (model `XC7A100TCSG324`) FPGA device. We synthesise the stand-alone design for our implementations using Xilinx Vivado 2019.1; default synthesis settings are used, with no effort invested in synthesis or post-implementation optimization.

[7] https://digilent.com/reference/programmable-logic/arty-a7/start

	Class-1								Class-2	
	[sec.]and	[sec.]andi	[sec.]or	[sec.]ori	[sec.]xor	[sec.]xori	[sec.]slli	[sec.]srli	[sec.]lw	[sec.]sw
Base core	1	1	1	1	1	1	1	1	2	2
Base core + latency-optimized	1	1	1	1	1	1	1	1	2	2
Base core + area-optimized	2	2	2	2	2	2	2	2	6	4

Table 9.4: Comparison of execution latency (measured in clock cycles), stemming from use of the base core plus implementation #1 (latency-optimized, per Section 9.5.2) and implementation #2 (area-optimized, per Section 9.5.3) of the ISE; note that functionally comparable instructions are included in the ISE-based (e.g., `sec.and`) and ISA-(e.g., `and`) cases respectively.

9.6.1 Area

Table 9.3 summarizes the ISE overhead in terms of area: it lists the resource utilization for base core and base core plus ISE (for both latency and area-optimized implementation variants), using an incremental approach to demonstrate the overhead of each instruction class. For the case of latency-optimized implementation, the extra area overhead of class-1 instructions mostly comes from our new mechanism of register indexing, e.g., the cost brought by index look-up table τ and an extra register PR[32]; the overhead of class-2 is obviously due to the generation of LSM, e.g., additional hardware resources for four mask seed CSRs and for the associated 2 rounds of keccak-f100 permutation. The total overhead of class-1 and class-2 amounts to 15% more Regs and 39% more LUTs compared to the base core. For the area-optimised variant, compared to the base core, the class-1 plus class-2 instructions take nearly no extra Regs and only 7% more LUTs.

9.6.2 Latency

Table 9.4 summarizes the ISE overhead in terms of execution latency: it lists the number of cycles required to execute each instruction on base core and base core plus ISE (for both latency and area-optimized implementation variants). The latency of the class-1 and class-2 instructions in latency-optimized implementation is as same as the latency of their counterparts on the base core, which is as designed and as expected. The latency of instructions in area-optimized implementation is as same as the latency of ISA-based strategy, e.g., the latency of a `sec.xor x1, x2, x3` equals the latency of a `mv x1, x0` plus an `xor x1, x2, x3`. This translates to, when using area-optimized variant of our ISE for a masked implementation, it might take a similar execution time as an ISA-based strategy. This is less attractive for a use case where the execution time is the priority. However, apart from the timing, the efficiency of a software also includes other metrics, e.g., instruction footprint. When using the area-optimized version of our ISE, the instruction footprint of a masked implementation can get significantly reduced compared to an ISA-based strategy, which is important for the deployment of masked implementation on memory-constrained devices, e.g., C0 and C1 devices defined in RFC 7228 [BEK14, Table 1].

9.6.3 Security

We use Coco [GHP+21, HB21] to evaluate the security (i.e., no leakage stemming from overwriting) of our ISE. In [GHP+21, Section 3], it states when using base Ibex core there are two constraints that a masked implementation should fulfill:

Constraint 1. Shares of the same secret must not be accessed within two successive instructions.

Constraint 2. A register or memory location which contains one share must not be overwritten with its counterparts.

Architectural overwriting. Recall that our ISE aims at eliminating the architectural and micro-architectural overwriting, which means, when using our secure instructions the constraint 2 should be no longer required. We then use Coco to perform the evaluation, where we label GPR[5] and GPR[12] to hold two *shares of the same secret* while we label GPR[6] and GPR[7] to hold *the static random values*. We use the following micro-benchmarks[8] to evaluate the class-1 instructions (using `[sec.]xor` as an example):

```
1 /* base core          */
2 xor       x5,  x5, x7
3 and       x6,  x6, x6
4 xor       x12, x5, x7
5 /* leakage captured   */
```

```
1 /* latency-optimized  */
2 sec.xor   x5,  x5, x7
3 and       x6,  x6, x6
4 sec.xor   x12, x5, x7
5 /* no leakage         */
```

```
1 /* area-optimized     */
2 sec.xor   x5,  x5, x7
3 and       x6,  x6, x6
4 sec.xor   x12, x5, x7
5 /* no leakage         */
```

Line 4 checks if there is a leakage when writing a share to a register that already contains another share of the same secret. This leakage is captured in the base core whereas it does not exist in the core extended with our ISE, which proves our class-1 instructions are secure in the sense of eliminating the architectural overwriting. We use the following micro-benchmarks to evaluate the store:

```
1 /* base core          */
2 li        x13, 0x20
3 sw        x5,  0(x13)
4 and       x6,  x6, x6
5 sw        x12, 0(x13)
6 /* leakage captured   */
```

```
1 /* latency-optimized  */
2 li        x13, 0x20
3 sec.sw    x5,  x13, 0, 0
4 and       x6,  x6, x6
5 sec.sw    x12, x13, 0, 1
6 /* no leakage         */
```

```
1 /* area-optimized     */
2 li        x13, 0x20
3 sec.sw    x5,  x13, 0, 0
4 and       x6,  x6, x6
5 sec.sw    x12, x13, 0, 0
6 /* no leakage         */
```

A leakage caused by overwriting in memory is expected in the base core since we write two shares to the same memory address, and it is successfully captured by Coco. When using our secure store instructions, this leakage does not exist, i.e., `sec.sw` eliminates this leakage as expected. Note that when evaluating the latency-optimized implementation, the initialization of mask seed CSRs is already done before the micro-benchmark. The micro-benchmarks used for evaluating the load are shown as follows:

```
1 /* base core          */
2 li        x13, 0x20
3 sw        x5,  0(x13)
4 and       x6,  x6, x6
5 lw        x12, 0(x13)
6 /* leakage captured   */
```

```
1 /* latency-optimized  */
2 li        x13, 0x20
3 sec.sw    x5,  x13, 0, 0
4 and       x6,  x6, x6
5 sec.lw    x12, x13, 0, 0
6 /* no leakage         */
```

```
1 /* area-optimized     */
2 li        x13, 0x20
3 sw        x5,  0(x13)
4 and       x6,  x6, x6
5 sec.lw    x12, x13, 0, 0
6 /* no leakage         */
```

The same results are also obtained from the evaluation of load instructions, i.e., no leakage is captured in the micro-benchmarks of our secure load.

Micro-architectural overwriting. One thing must be noted is that there is no MBR in the Ibex core, which means the above security evaluation can only *straightforwardly* prove our ISE is capable to eliminate the architectural overwriting (i.e., in the register file and in the memory). As we mentioned before, the location of MBR can be intra-core or extra-core, and our ISE is designed to be capable to work in both cases. It is not trivial to directly perform the security evaluation in this situation, especially when

[8] For the case of latency-optimized ISE implementation, we make sure that the register indices do not get updated before execution of the micro-benchmarks.

extra-core MBR is used. However, it is possible and easy to conclude that (if it is correctly implemented) our `sec.sw` and `sec.lw` can eliminate the micro-architectural overwriting (i.e., in MBR) based on the above security evaluation for architectural overwriting; 1) in the case of latency-optimized implementation, what `sec.sw` and `sec.lw` act in MBR is completely the same as what `sec.sw` acts in the memory, i.e., overwriting with a re-masked share; 2) as for area-optimized implementation, `sec.sw` also acts the same in both MBR and the memory. What the last two steps (i.e., steps that interact with the load buffer) of `sec.lw` act in MBR is the same as what `sec.sw` acts in the memory. In other words, for both latency-optmized and area-optmized implementations, if there is no architectural overwriting leakage captured in the micro-benchmarks of `sec.sw`, then we can claim `sec.sw` and `sec.lw` can eliminate the micro-architectural overwriting.

9.6.4 Usability

The latency-optimized implementation demands a software developer correctly manages use of the mask seed CSRs to eliminate architectural and micro-architectural overwriting; in contrast, the area-optimized implementation does so transparently. This implies a clear difference in terms of their usability.

9.6.5 Comparison with related work

The two closest alternatives, and hence most natural comparison points, are 1) the ISA-based strategy provided by Rosita [SSB+21], and 2) the ISE-based strategy provided by FENL [GMPP20]; we focus on the most similar, zeroization-based variant of FENL. Use of all three can be framed as rewriting instructions within an existing software implementation, with the goal of eliminating leakage. Rosita and FENL introduce additional instructions; eLIMInate replaces existing instructions (from ISA- to ISE-based, e.g., `xor` to `sec.xor`). FENL and eLIMInate use ISE-based instructions, so require support from hardware; Rosita uses ISA-based instructions, so requires no support from hardware. Note that Rosita, FENL, and eLIMInate are all largely agnostic to properties of the masking scheme or attacks on them. For example, Rosita++ [SCS+21] addresses the challenge of higher-order leakage elimination using the same set of rewrite rules as Rosita [SSB+21].

A direct comparison is difficult, because the ISAs, cores, and indeed stated remits differ. However, Table 9.5 attempts to offer a somewhat quantitative, somewhat qualitative summary that is derived from the analysis below:

- **Security.** The scope of Rosita addresses both architectural and micro-architectural leakage. As an ISA-based strategy, it uses indirect control of extra- and intra-core resources; per Section 9.3, doing so offers a weaker guarantee than, e.g., direct control. The scope of FENL addresses only micro-architectural leakage with any extra-core resources deemed out of scope. As an ISE-based strategy, it uses direct control of intra-core resources.

- **Usability.** A careful security analysis is required to identify where Rosita rewrite rules are applied. They can be described as local, in the sense they can be applied by using "peephole-like" translation which has no global impact (and so does not require any global functional analysis). The difficulty of doing so is significantly reduced by the associated, automated tooling. A similar argument to that above for can be applied to FENL, in the sense one needs to analyze 1) how to configure and 2) where to place fence instructions. However, although it is plausible to use Rosita-like automation, a tool to do so for FENL currently does not exist. Application of eLIMInate is local for the area-optimized implementation, but is local (for class-1 instructions) and global (for class-2 instructions) for the latency-optimized implementation.

		Security	Usability	Footprint	Latency	Area	Invasiveness
Base core + latency-optimized versus	Rosita	+	−	+	+	−	−
	FENL	+	−	+	+	−	+
Base core + area-optimized versus	Rosita	+	−	+	≈	−	−
	FENL	+	≈	+	≈	≈	+

Table 9.5: A somewhat quantitative, somewhat qualitative comparison versus Rosita [SSB+21] and FENL [GMPP20] (the two closest alternatives). Note that +, −, and ≈ suggest the comparison respectively positive, negative, and approximately equal versus Rosita or FENL.

- **Footprint.** Both Rosita and FENL imply marginal overhead in memory footprint, since their application demands at least one additional instruction; all else being equal, the additional memory access required to fetch said instructions could plausibly contribute to greater energy consumption.
- **Latency.** For both Rosita and FENL, the global impact on execution latency depends where the mechanism is or is not applied, so we focus only on local instances where it is. Translating the ARM-based Rosita rewrite rules to RISC-V yields a similar outcome: as suggested by Section 9.3, this means a 2-cycle latency for class-1 instructions, a 6-cycle latency for the class-2 instruction `lw`, and a 4-cycle latency for the class-2 instruction `sw` when executed on the base core. For FENL, comparison is more difficult. For the case most similar to the base core, [GMPP20, Section 3.3.3] lists two options in which `fenl.fence` has 1) a 1 cycle (non-bubbling) or 2) a 1 or 4 cycle execution latency (bubbling: depending whether or not a pipeline stall is required to deliver the security guarantee). Using the former, this *suggests* a 2-cycle latency for class-1 instructions, a 3-cycle latency for the class-2 instruction `lw`, and a 3-cycle latency for the class-2 instruction `sw`. However, note that FENL deems extra-core resources such as SRAM out of scope; the comparison is only reasonable for class-1 instructions, therefore.
- **Area.** Rosita implies no overhead in hardware area. FENL implies modest overhead in hardware area: for the core most similar to Ibex, [GMPP20, Table 2] cites 0.7% additional flip-flops plus 1.0% additional LUTs.
- **Invasiveness.** Rosita is an ISA-base strategy, so is not invasive. FENL is an ISE-base strategy, so is somewhat invasive: assuming existing instructions to manage CSRs, it adds 1 instruction and 1 CSR. Implementation of that instruction could be viewed as invasive, however, because it 1) has a global impact, potentially throughout the micro-architecture, and 2) intentionally exposes micro-architectural detail to software.

9.7 Conclusion

Summary. In this Chapter, we presented a functionally light-weight, leakage-focused ISE with the aim of supporting masked software implementation. By developed two concrete, prototype implementations of an underlying design concept, we demonstrate that use of the ISE can close the gap between assumptions about and actual behavior of a device and thereby deliver an improved security guarantee.

In our view, it is important to stress that use of our ISE enables a subtle shift in how masked implementations can be developed. Currently, the starting point is a masked implementation consisting

of instructions from the ISA this is functionally correct, insecure but efficient, implying a need to improve security (e.g., by identifying and eliminating micro-architectural leakage). Anecdotal evidence suggests that doing so is both conceptually difficult (and thus error-prone), and labour-intensive; the impact of failure can be catastrophic, in the sense it can renders the implementation insecure. Now, the starting point is a masked implementation consisting of instructions from the ISE: this is functionally correct, secure but inefficient, implying a need to improve efficiency (e.g., by selectively replacing ISE instructions, with ISA alternatives). We claim that doing so is conceptually easier, and the impact of failure is lessened; it aligns with a more general secure-by-default ethos.

Future work. Given the scope of this Chapter, and work presented within it, the following points seem to represent either useful or interesting future work:

1. Section 9.3 highlights an inherent limitation of the ISE, namely that extra-core resources require indirect control; improvement beyond this requires a change to the resource interface. For example, consider an SRAM module whose interface supports direct control via a "flush state" control signal: by removing the need for assumptions around indirect control, securing access to the SRAM can be more efficient and yield a more robust security guarantee. Realizing such a systemic change is of course non-trivial, not least because of trade-offs between security and other metrics, but seems an important long-term goal.

2. Section 9.3 is clear about insufficiency of the ISE, in the sense that additional forms of micro-architectural leakage may also need to be considered. Doing so by extending the scope is somewhat open ended, but, for example, Section 9.4.2 includes `sec.slli` and `sec.srli` for left- and right-shift; it would be plausible to extend the variant semantics for these instructions to, e.g., address the observation by Gao et al. [GMPO20] that bit-interaction within a barrel shifter can produce leakage.

3. For the latency-optimized implementation, Section 9.6 highlights a challenge with respect to usability: a software developer must correctly manage use of the mask seed CSRs. Alongside generation of ISE-based instructions rather than their ISA-based instructions analogue, this aspect seems ripe for automation within an appropriate compilation tool-chain.

Part VI

Concluding Remarks

CHAPTER

10 CONCLUSION

10.1 Summary

This thesis described research on engineering aspects related to the deployment of next-generation cryptography in the real world, and presented some useful approaches and techniques. We showed that lattice-based cryptography is suitable to be employed on embedded devices. With fixed parameters, many trade-offs (between time and memory) of lattice-based crypto implementations are possible, which provide flexibility for application scenarios with different requirements. Isogeny-based cryptography is very new; there is thus great room for improving its performance both from theoretical (focusing on the underlying mathematics) and implementation perspectives. We concentrated on vector instructions, and proved that their strong computing power, combined with smart vectorization techniques, can result in a significant gain for both the latency and the throughput of isogeny-based crypto software. Instruction set extensions are a very helpful tool to further accelerate and assist software implementations in both their efficiency and/or security. They are like a bridge linking software and hardware, and on RISC-V the developers have the right to decide how to build such bridges. For efficiency bottlenecks, ISE offers multiple different solutions for both symmetric and asymmetric cryptography, and there exist various designs and associated trade-offs. For the threats arising from power side-channel analysis, especially the recently observed effects of micro-architectural leakage, we designed a leakage-focused ISE and demonstrated that the use of our ISE can close the gap between assumptions and the actual behavior of a device and thereby deliver an improved security guarantee.

10.2 Impact

The implementation techniques and evaluation results presented in this thesis have (potential) impact on both the NIST PQC and NIST LWC cryptography standardization processes as well as, more generally, on cryptographic engineering.

Impact on the NIST PQC standardization process.

- There are currently only a few papers in the literature discussing and evaluating the software performance of NIST PQC candidates on AVR microcontrollers. Hence, Chapter 3 and Chapter 4

make contributions to filling this gap to some extent. The most important observation from these two Chapters is that with their carefully-optimized assembly implementations lattice-based NIST PQC candidates are feasible to be deployed on *extremely* constrained devices like AVR microcontrollers.

- On the other hand, regarding Chapter 6, SIKE is the only isogeny-based candidate in the NIST PQC standardization. SIKE has the highest execution time among the NIST PQC third round candidates but is already the fastest among various isogeny-based schemes in the literature. Hence, how to further improve its performance is very important for not only SIKE but also other isogeny-based algorithms. To the best of our knowledge, previous works about vectorization of SIKE only focus partially on one or two arithmetic layers, and Chapter 6 is the first one which *thoroughly* analyzes all the layers of SIKE from a vectorization perspective. Chapter 6 plus also Chapter 5 show that SIKE and other isogeny-based algorithms are vectorization-friendly, and vector units and instructions can significantly improve their software performance, especially given that vector units widely exist on high-performance CPUs and are constantly being enhanced.

Impact on the NIST LWC standardization process.

- To the best of our knowledge, Chapter 7 is the only work in the literature that benchmarks NIST LWC candidates from an ISE perspective. It provides useful evaluation results for NIST LWC standardization, and the associated paper [CGM+23] is taken into consideration in the NIST LWC official report [TMC+23]. In addition, not limited to NIST LWC, we note that currently NIST does not consider ISE in its cryptography competitions, e.g., in the way of defining a reference platform that could support work on ISE, like what NIST usually did with (pure) software implementations. However, ISE can clearly impact the evaluation results, and after standardization commonly appear in real ISAs. Chapter 7 serves as an evidence to support the view that NIST should take ISE into account in its cryptography standardizations.

Impact on cryptographic engineering.

- First of all, some techniques proposed in this thesis have already created impact, and inspired and/or assisted follow-up work of other researchers, e.g., [PSRH23] of Phalakarn et al. builds on Chapter 6.

- In particular, regarding the design and development of high-performance vectorized cryptographic software, Part III demonstrates the analyses from three different perspectives, namely operation, optimization, and vectorization, and provides a methodology rather than just a method: though the effectiveness of our vectorization methodology is demonstrated using isogeny-based cryptography as case study, it is widely applicable and potentially beneficial for a wide variety of cryptographic algorithms such as RSA, ECC, and pairing-based cryptography.

- In addition to observations regarding ISE design, Chapter 7 also discussed some observations regarding ISA design, which are helpful for RISC-V community, and some observations regarding symmetric algorithm design, which aim to assist the algorithm design phase.

- Given that Chapter 9 aims to harden masked software implementations at an instruction level, it natively supports a large scope of cryptographic algorithms. Meanwhile, the presented concept can be extended to other different instructions.

10.3 Future work

For each of the four research topics described in this thesis, we propose some high-level ideas that can potentially be either interesting or worthy for further exploration and research.

Lightweight implementation of lattice-based cryptography. CRYSTALS-Kyber [SAB+21] has been selected by NIST as a standard KEM in the NIST PQC standardization[1]. There is thus great value and significance in researching how it can be deployed in the real world in an efficient and secure way.

1. Although many papers have studied the efficient software implementation of CRYSTALS-Kyber on various constrained devices such as ARM Cortex-M and Cortex-A series, it is still unexplored how CRYSTALS-Kyber can be optimized on "extremely-constrained" microcontrollers, e.g., 8-bit AVR and 16-bit MSP430, and what performance a highly-optimized assembler implementation can achieve.

2. Apart from RV32I [RV19, Chapter 2], RISC-V also defines another 32-bit base integer instruction set RV32E [RV19, Chapter 4], which is a reduced version of RV32I and designed for *embedded systems*. It would make sense to consider using it to design a power-efficient PQC-oriented processor for the IoT but still retain high flexibility, e.g., one can look at first 1) a low-area, simple, and power-efficient RISC-V RV32EM core design (i.e., a multiplier is also included and its lightweight implementation should be considered); then 2) based on the core designed and developed, research an extremely power-efficient software implementation of CRYSTALS-Kyber; finally 3) propose various ISEs to assist and/or improve the implementations regarding, e.g., the efficiency and more importantly the security (namely masked implementation).

Vectorized implementation of isogeny-based cryptography. Isogeny-based cryptography is a booming research area, and in particular many new isogeny-based signatures were proposed in the recent years. A signature scheme has been submitted to NIST standardization for additional PQC digital signature schemes[2], i.e., SQISign [DKL+20, DLLW23]. Compared to other categories such as code-based, hash-based, and multivariate, the execution time of isogeny-based signatures is usually orders of magnitude slower, which means it makes sense to use them (only) on high-performance processors.

1. There were recently some new SIMD/vector instructions released, in particular the RISC-V vector extension [RVV21] and vector cryptography extension [RVK23]. They are more attractive than other vector instruction sets in the sense that the developer can also add customized ISE to enhance the vector core/unit. Therefore, one could 1) explore the efficient vectorization of the arithmetic used in isogeny-based signatures (e.g., isogeny arithmetic, curve arithmetic) and develop the associated efficient vectorized implementation of the full signature, and then 2) think about an efficient vector ISE design.

2. Compared to CPUs, GPUs have more computing power. We could also consider efficient GPU implementation of isogeny-based signatures. However, the transmission cost of data from memory/CPU to GPU might be a point to be wary of.

Efficient cryptographic instruction set extension design. Apart from the points already mentioned in the text above, there are also some other ideas and thoughts which seem to be interesting to take note of. We are still considering RISC-V ISA in the context of ISE design.

[1] See, https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.
[2] See, https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

1. Multi-precision integer arithmetic is always important in the field of cryptography; however, the speed-up gained from the scalar ISEs presented in Chapter 8 is *good* but cannot be claimed as *significant*. It makes sense to consider the stronger computing power unleashed by vector unit/instructions. To not overlap with the idea introduced before, i.e., designing efficient vector ISEs for isogeny-based signatures, it could be useful to think about Residue Number System (RNS) [Ana16] arithmetic. To be specific, RNS is believed to be efficient in SIMD fashion [PFPB19], so we could design dedicated vector ISEs for RNS arithmetic, and evaluate whether a *vector-ISE-assisted* RNS-based implementation is more efficient than a *vector-ISE-assisted* conventional implementation for, e.g., Montgomery multiplication, as the first step.

2. In current RISC-V scalar and vector cryptography extensions [RVK22, RVK23], there is no dedicated instruction for Keccak [BDPA13]. Since Keccak is not only used in hash functions but also in lattice-based and hash-based PQC schemes, it would make sense to think about the scalar and vector ISE design of Keccak.

Side-channel leakage analysis and elimination. Since we spent only six months working on this topic during the PhD studies, far from saying we are familiar with it, we therefore regard this topic as our primary research focus in the coming years, and particularly pay more attention to the questions and problems regarding the micro-architectural leakage.

1. Based on the ISE designed and developed in Chapter 9, our next step is to implement and show the complete first-order and higher-order masked implementations of AES using this data-oriented ISE, and we will evaluate and compare them with corresponding implementations using compute-oriented ISE, e.g., [GGM+21]. We will also seek whether there is an opportunity to combine both, i.e., the data-oriented ISE with the compute-oriented ISE, on either software perspective or ISE perspective to yield a win-win outcome.

2. It would be useful to research how to utilize the micro-architectural leakage (e.g., stemming from MBR) as the main leakage source to perform attacks on various devices, and then study the countermeasures.

ACRONYMS

ADK	:	Arbitrary Degree Karatsuba
AES	:	Advanced Encryption Standard
ALU	:	Arithmetic Logic Unit
ASIP	:	Application Specific Instruction Processor
AVX	:	Intel Advanced Vector Extensions
AVX2	:	Intel Advanced Vector Extensions 2
AVX-512	:	Intel Advanced Vector Extensions 512
AVX-512F	:	Intel AVX-512 Integer Fused Foundation
AVX-512IFMA	:	Intel AVX-512 Integer Fused Multiply-Add
BPS	:	Block Product Scanning
CIHS	:	Coarsely Integrated Hybrid Scanning
CMOS	:	Complementary Metal-Oxide Semiconductor
CPU	:	Central Processing Unit
CSIDH	:	Commutative Supersingular Isogeny Diffie-Hellman
CSR	:	Control/Status Register
DES	:	Data Encryption Standard
ECC	:	Elliptic Curve Cryptography
EM	:	ElectroMagnetic
FEC	:	Forward Error Correction
FIPS	:	Finely Integrated Product Scanning
FP	:	Floating Point
GAIP	:	Group Action Inverse Problem
HW	:	HardWare
IFMA	:	Integer Fused Multiply-Add
I-MLWE	:	Integer Module Learning with Errors
IoT	:	Internet of Things
ISA	:	Instruction Set Architecture
ISE	:	Instruction Set Extension
KCM	:	Karatsuba-Comba-Montgomery
KEM	:	Key Encapsulation Mechanism
LSU	:	Load-Store Unit

LUT	:	Look-Up Table
LWC	:	LightWeight Cryptography
LWE	:	Learning With Errors
MAC	:	Multiply-ACcumulate
MAR	:	Memory Address Register
MBR	:	Memory Buffer Register
MCU	:	Micro-Controller (Unit)
NIST	:	(US) National Institute of Standards and Technology
NSA	:	(US) National Security Agency
NTT	:	Number Theoretic Transform
PKE	:	Public-Key Encryption
PQC	:	Post-Quantum Cryptography
QKD	:	Quantum Key Distribution
QKE	:	Quantum Key Establishment
RAM	:	Random-Access Memory
RFC	:	Request For Comments
RISC	:	Reduced Instruction Set Computer
RNS	:	Residue Number System
ROM	:	Read-Only Memory
RPS	:	Reverse Product-Scanning
RSA	:	Rivest-Shamir-Adleman cryptosystem
SCA	:	Side-Channel Attack
SHA	:	Secure Hash Algorithm
SIDH	:	Supersingular Isogeny Diffie-Hellman
SIKE	:	Supersingular Isogeny Key Encapsulation
SIMBA	:	Splitting Isogeny computations into Multiple BAtches
SIMD	:	Single Instruction Multiple Data
SIPKE	:	Supersingular Isogeny Public Key Encryption
SIS	:	Shortest Integer Solution
SPS	:	Separated Product Scanning
SSL	:	Secure Socket Layer
SW	:	SoftWare
TLS	:	Transport Layer Security

BIBLIOGRAPHY

[AAA+20] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone. Status report on the second round of the nist post-quantum cryptography standardization process. National Institute of Standards and Technology (NIST), Interagency Report 8309, 2020. https://doi.org/10.6028/NIST.IR.8309.

[AAB+16] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D.A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[AAC+22] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and Y.-K. Liu. Status report on the third round of the nist post-quantum cryptography standardization process. National Institute of Standards and Technology (NIST), Interagency Report 8413, 2022. https://doi.org/10.6028/NIST.IR.8413-upd1.

[AAK21] M. Anastasova, R. Azarderakhsh, and M.M. Kermani. Fast strategies for the implementation of SIKE round 3 on ARM Cortex-M4. *IEEE Transactions on Circuits and Systems I (TCSI)*, 68(10):4129–4141, 2021. https://doi.org/10.1109/TCSI.2021.3096916.

[ABB+22] N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Guneysu, C.A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, G. Zemor, V. Vasseur, S. Ghosh, and J. Richter-Brokmann. BIKE: Bit flipping key encapsulation, 2022. https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf.

[ABC+22] M. Alsahli, A. Borgognoni, L. Cardoso dos Santos, H. Cheng, C. Franck, and J. Großschädl. Lightweight permutation-based cryptography for the ultra-low-power internet of things. In G. Bella, M. Doinea, and H. Janicke, editors, *Security for Information Technology and Communications (SECITC)*, LNCS 13809, pages 17–36. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-031-32636-3_2.

[ABJK18] R. Azarderakhsh, E. Bakos Lang, D. Jao, and B. Koziel. EdSIDH: supersingular isogeny Diffie-Hellman key exchange on Edwards curves. In A. Chattopadhyay, C. Rebeiro, and Y. Yarom, editors, *Security, Privacy, and Applied Cryptography Engineering (SPACE)*, LNCS 11348, pages 125–141. Springer-Verlag, 2018. https://doi.org/10.1007/978-3-030-05072-6_8.

[ACR22] G. Adj, J. Chi-Domínguez, and F. Rodríguez-Henríquez. Karatsuba-based square-root Vélu's formulas applied to two isogeny-based protocols. *Journal of Cryptographic Engineering (JCEN)*, 2022. https://doi.org/10.1007/s13389-022-00293-y.

[ADPS16] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - A new hope. In T. Holz and S. Savage, editors, *USENIX Security Symposium*, pages 327–343, 2016. https://doi.org/10.5555/3241094.3241120.

[AHJM11] M. Agren, M. Hell, T. Johansson, and W. Meier. Grain128a: a new version of Grain-128 with optional authentication. *International Journal of Wireless and Mobile Computing (IJWMC)*, 5(1):48–59, 2011. https://doi.org/10.1504/IJWMC.2011.044106.

[Ajt96] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In G.L. Miller, editor, *Symposium on Theory of Computing (STOC)*, pages 99–108. ACM, 1996. https://doi.org/10.1145/237814.237838.

[Ana16] P.V. Ananda Mohan. *Residue Number Systems*. Springer-Verlag, 2016. https://doi.org/10.1007/978-3-319-41385-3.

[ANP20] A. Adomnicai, Z. Najm, and T. Peyrin. Fixslicing: A new GIFT representation: Fast constant-time implementations of GIFT and GIFT-COFB on ARM Cortex-M. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(3):402–427, 2020. https://doi.org/10.13154/tches.v2020.i3.402-427.

[AO21] Ö. Altınay and B. Örs. Instruction extension of RV32I and GCC back end for Ascon lightweight cryptography algorithm. In *International Conference on Omni-Layer Intelligent Systems (COINS)*, pages 1–6, 2021. https://doi.org/10.1109/COINS51742.2021.9524190.

[AP20] A. Adomnicai and T. Peyrin. Fixslicing - application to some NIST LWC round 2 candidates. In 4-*th Lightweight Cryptography Workshop*, 2020. https://csrc.nist.gov/Events/2020/lightweight-cryptography-workshop-2020.

[AP21] A. Adomnicai and T. Peyrin. Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(1):402–425, 2021. https://doi.org/10.46586/tches.v2021.i1.402-425.

[ARM09] Cortex-M0 Technical Reference Manual. Technical Report DDI-0432C, ARM Ltd., 2009. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/index.html.

[ARM18] ARMv6-M Architecture Reference Manual. Technical Report DDI-0419E, ARM Ltd., 2018. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0419e/index.html.

[ARM20] Arm architecture reference manual: Armv8, for Armv8-A architecture profile. Technical report, 2020. https://static.docs.arm.com/ddi0487/fa/DDI0487F_a_armv8_arm.pdf.

[ARM21] ARMv7-M Architecture Reference Manual. Technical Report DDI-0403E.e, ARM Ltd., 2021. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0403e.e/index.html.

[AVR21] AVR instruction set manual. Technical report, Microchip Technology, Inc., 2021. https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf.

[BB03] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Security Symposium*, 2003. https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical.

[BB14] C.H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science (TCS)*, 560:7–11, 2014. https://doi.org/10.1016/j.tcs.2014.05.025.

[BBC+20a] C. Beierle, A. Biryukov, L. Cardoso dos Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, and Q. Wang. Alzette: a 64-bit ARX-box (feat. CRAX and TRAX). In *Advances in Cryptology (CRYPTO)*, LNCS 12172, pages 419–448. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-56877-1_15.

[BBC+20b] C. Beierle, A. Biryukov, L. Cardoso dos Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, and Q. Wang. Lightweight AEAD and hashing using the Sparkle permutation family. *IACR Transactions on Symmetric Cryptology (TOSC)*, 2020(S1):208–261, 2020. https://doi.org/10.13154/tosc.v2020.iS1.208-261.

[BBC+20c] D.J. Bernstein, B.B. Brumley, M.-S. Chen, C. Chuengsatiansup, T. Lange, A. Marotzke, B.-Y. Peng, N. Tuveri, C. van Vredendaal, and B.-Y. Yang. NTRU Prime, 2020. https://ntruprime.cr.yp.to/nist/ntruprime-20201007.pdf.

[BBC+21a] G. Banegas, D.J. Bernstein, F. Campos, T. Chou, T. Lange, M. Meyer, B. Smith, and J. Sotáková. CTIDH: faster constant-time CSIDH. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):351–387, 2021. https://doi.org/10.46586/tches.v2021.i4.351-387.

[BBC+21b] C. Beierle, A. Biryukov, L. Cardoso dos Santos, J. Großschädl, A. Moradi, L. Perrin, A.R. Shahmirzadi, A. Udovenko, V. Velichkov, and Q. Wang. Schwaemm and esch: Lightweight authenticated encryption and hashing using the sparkle permutation family. Submission to NIST (version 1.2), 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/sparkle-spec-final.pdf.

[BCC+22] D.J. Bernstein, T. Chou, C. Cid, J. Gilcher, T. Lange, V. Maram, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, N. Sendrier, J. Szefer, C.J. Tjhai, M. Tomlinson, and W. Wang. Classic mceliece: conservative code-based cryptography, 2022. https://classic.mceliece.org/mceliece-spec-20221023.pdf.

[BCD+21] Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K. Yasuda. PHOTON-beetle. Submission to NIST, 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/photon-beetle-spec-final.pdf.

[BCDM21] T. Beyne, Y.L. Chen, C. Dobraunig, and B. Mennink. Elephant. Submission to NIST (version 2.0), 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf.

[BCE+01] D.V. Bailey, D. Coffin, A.J. Elbirt, J.H. Silverman, and A.D. Woodbury. NTRU in constrained devices. In C.K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2162, pages 262–272. Springer-Verlag, 2001. https://doi.org/10.1007/3-540-44709-1_22.

[BCI+21] S. Banik, A. Chakraborti, T. Iwata, K. Minematsu, M. Nandi, T. Peyrin, Y. Sasaki, S.M. Sim, and Y. Todo. GIFT-COFB. Submission to NIST (version 1.1), 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/gift-cofb-spec-final.pdf.

[BCLV17] D.J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. NTRU Prime: Reducing attack surface at low cost. In C. Adams and J. Camenisch, editors, *Selected Areas in Cryptography (SAC)*, LNCS 10719, pages 235–260. Springer-Verlag, 2017. https://dl.acm.org/doi/10.5555/3241094.3241120.

[BCLV19] D.J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. NTRU Prime: Round 2 specification, 2019. https://ntruprime.cr.yp.to/nist/ntruprime-20190330.pdf.

[BDLS20] D.J. Bernstein, L. De Feo, A. Leroux, and B. Smith. Faster computation of isogenies of large prime degree. Cryptology ePrint Archive, Paper 2020/341, 2020. https://eprint.iacr.org/2020/341.

[BDLU17] A. Biryukov, D. Dinu, Y. Le Corre, and A. Udovenko. Optimal first-order Boolean masking for embedded IoT devices. In *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 10728, pages 22–41. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-75208-2_2.

[BDPA13] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche. Keccak. In *Advances in Cryptology (EUROCRYPT)*, LNCS 7881, pages 313–314. Springer-Verlag, 2013. https://doi.org/10.1007/978-3-642-38348-9_19.

[BEK14] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. Internet Engineering Task Force (IETF) Request for Comments (RFC) 7228, 2014. http://tools.ietf.org/html/rfc7228.

[Ber06] D.J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography (PKC)*, LNCS 3958, pages 207–228. Springer-Verlag, 2006. https://doi.org/10.1007/11745853_14.

[Ber20] D.J. Bernstein. Cryptographic competitions. Cryptology ePrint Archive, Report 2020/1608, 2020. https://eprint.iacr.org/2020/1608.

[BF20] J.W. Bos and S. Friedberger. Faster modular arithmetic for isogeny-based crypto on embedded devices. *Journal of Cryptographic Engineering (JCEN)*, 10(2):97–109, 2020. https://doi.org/10.1007/s13389-019-00214-6.

[BFG+17] J. Balasch, S. Faust, B. Gierlichs, C. Paglialonga, and F.-X. Standaert. Consolidating inner product masking. In T. Takagi and T. Peyrin, editors, *Advances in Cryptology (ASIACRYPT)*, LNCS 10624, pages 724–754. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-70694-8_25.

[BGH22] B. Buhrow, B.K. Gilbert, and C.R. Haider. Parallel modular multiplication using 512-bit advanced vector instructions. *Journal of Cryptographic Engineering (JCEN)*, 12(1):95–105, 2022. https://doi.org/10.1007/s13389-021-00256-9.

[BGM09] S. Bartolini, R. Giorgi, and E. Martinelli. Instruction set extensions for cryptographic applications. In Ç.K. Koç, editor, *Cryptographic Engineering*, chapter 9, pages 191–233. Springer, 2009. https://doi.org/10.1007/978-0-387-71817-0_9.

[BHKL13] D.J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In A.-R. Sadeghi, V.D. Gligor, and M. Yung, editors, *Computer and Communications Security (CCS)*, pages 967–980. ACM, 2013. https://doi.org/10.1145/2508859.2516734.

[BI21] C. Bouvier and L. Imbert. An alternative approach for SIDH arithmetic. In J.A. Garay, editor, *Public-Key Cryptography (PKC)*, LNCS 12710, pages 27–44. Springer-Verlag, 2021. https://doi.org/10.1007/978-3-030-75245-3_2.

[BJK+16] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S.M. Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *Advances in Cryptology (CRYPTO)*, LNCS 9815, pages 123–153. Springer-Verlag, 2016. https://doi.org/10.1007/978-3-662-53008-5_5.

[BKL+07] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 4727, pages 450–466. Springer-Verlag, 2007. https://doi.org/10.1007/978-3-540-74735-2_31.

[BKL+13] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. SPONGENT: The design space of lightweight cryptographic hashing. *IEEE Transactions on Computers*, 62(10):2041–2053, 2013. https://doi.org/10.1109/TC.2012.196.

[BKV19] W. Beullens, T. Kleinjung, and F. Vercauteren. CSI-FiSh: efficient isogeny based signatures through class group computations. In S.D. Galbraith and S. Moriai, editors, *Advances in Cryptology (ASIACRYPT)*, LNCS 11921, pages 227–247. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-34578-5_9.

[BLMP19] D.J. Bernstein, T. Lange, C. Martindale, and L. Panny. Quantum circuits for the CSIDH: Optimizing quantum evaluation of isogenies. In *Advances in Cryptology (EUROCRYPT)*, LNCS 11477, pages 409–441. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-17656-3_15.

[BPP+17] S. Banik, S.K. Pandey, T. Peyrin, Y. Sasaki, S.M. Sim, and Y. Todo. GIFT: A small present - towards reaching the limit of lightweight encryption. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 10529, pages 321–345. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-66787-4_16.

[BS20] X. Bonnetain and A. Schrottenloher. Quantum security analysis of CSIDH. In A. Canteaut and Y. Ishai, editors, *Advances in Cryptology (EUROCRYPT)*, LNCS 12106, pages 493–522. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-45724-2_17.

[BT11] B.B. Brumley and N. Tuveri. Remote timing attacks are still practical. In V. Atluri and C. Díaz, editors, *European Symposium on Research in Computer Security (ESORICS)*, LNCS 6879, pages 355–371. Springer-Verlag, 2011. https://doi.org/10.1007/978-3-642-23822-2_20.

[BWG+22] A. Beckers, L. Wouters, B. Gierlichs, B. Preneel, and I. Verbauwhede. Provable secure software masking in the real-world. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 13211, pages 215–235. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-030-99766-3_10.

[CCC+19] D. Cervantes-Vázquez, M. Chenu, J. Chi-Domínguez, L. De Feo, F. Rodríguez-Henríquez, and B. Smith. Stronger and faster side-channel protections for CSIDH. In *Progress in Cryptology (LATINCRYPT)*, LNCS 11774, pages 173–193. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-30530-7_9.

[CCJR20] J. Chávez-Saab, J.-J. Chi-Domínguez, S. Jaques, and F. Rodríguez-Henríquez. The SQALE of CSIDH: Sublinear vélu quantum-resistant isogeny action with low exponents. Cryptology ePrint Archive, Paper 2020/1520, 2020. https://eprint.iacr.org/2020/1520.

[CD23] W. Castryck and T. Decru. An efficient key recovery attack on SIDH. In C. Hazay and M. Stam, editors, *Advances in Cryptology (EUROCRYPT)*, LNCS 14008, pages 423–447. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-30589-4_15.

[CDG18] H. Cheng, D. Dinu, and J. Großschädl. Efficient implementation of the SHA-512 hash function for 8-bit AVR microcontrollers. In *Security for Information Technology and Communications (SECITC)*, LNCS 11359, pages 273–287. Springer-Verlag, 2018. https://doi.org/10.1007/978-3-030-12942-2_21.

[CDG+19] H. Cheng, D. Dinu, J. Großschädl, P.B. Rønne, and P.Y.A. Ryan. A lightweight implementation of NTRU Prime for the post-quantum internet of things. In *Workshop on Information Security Theory and Practices (WISTP)*, LNCS 12024, pages 103–119. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-41702-4_7.

[CDH+19] C. Chen, O. Danba, J. Hoffstein, A. Hulsing, J. Rijneveld, J.M. Schanck, P. Schwabe, W. Whyte, Z. Zhang, T. Saito, T. Yamakawa, and K. Xagawa. NTRU, 2019. https://ntru.org/f/ntru-20190330.pdf.

[CDPA16] C. Celio, P. Dabbelt, D.A. Patterson, and K. Asanović. The renewed case for the reduced instruction set computer: Avoiding ISA bloat with macro-op fusion for RISC-V. *CoRR*, abs/1607.02318, 2016. https://arxiv.org/abs/1607.02318.

[CFG+21] H. Cheng, G. Fotiadis, J. Großschädl, P.Y.A. Ryan, and P.B. Rønne. Batching CSIDH group actions using AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):618–649, 2021. https://doi.org/10.46586/tches.v2021.i4.618-649.

[CFG+23] H. Cheng, G. Fotiadis, J. Großschädl, D. Page, T. Pham, and P.Y.A. Ryan. RISC-V instruction set extensions for multi-precision integer arithmetic. 2023. To be submitted.

[CFGR22] H. Cheng, G. Fotiadis, J. Großschädl, and P.Y.A. Ryan. Highly vectorized SIKE for AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(2):41–68, 2022. https://doi.org/10.46586/tches.v2022.i2.41-68.

[CGGS22] L. Cardoso dos Santos, F. Gérard, J. Großschädl, and L. Spignoli. Rivain-prouff on steroids: Faster and stronger masking of the AES. In I. Buhan and T. Schneider, editors, *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 13820, pages 123–145. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-031-25319-5_7.

[CGM+23] H. Cheng, J. Großschädl, B. Marshall, D. Page, and T. Pham. RISC-V instruction set extensions for lightweight symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2023(1):193–237, 2023. https://doi.org/10.46586/tches.v2023.i1.193-237.

[CGRR20] H. Cheng, J. Großschädl, P.B. Rønne, and P.Y.A. Ryan. Lightweight post-quantum key encapsulation for 8-bit AVR microcontrollers. In *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 12609, pages 18–33. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-68487-7_2.

[CGRR21] H. Cheng, J. Großschädl, P.B. Rønne, and P.Y.A. Ryan. AVRNTRU: Lightweight NTRU-based post-quantum cryptography for 8-bit AVR microcontrollers. In *Design, Automation, and Test in Europe (DATE)*, pages 1272–1277, 2021. https://doi.org/10.23919/DATE51398.2021.9474033.

[CGT+20] H. Cheng, J. Großschädl, J. Tian, P.B. Rønne, and P.Y.A. Ryan. High-throughput elliptic curve cryptography using AVX2 vector instructions. In *Selected Areas in Cryptography (SAC)*, LNCS 12804, pages 698–719. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-81652-0_27.

[CGZ20] J.-S. Coron, A. Greuet, and R. Zeitoun. Side-channel masking with pseudo-random generator. In A. Canteaut and Y. Ishai, editors, *Advances in Cryptology (EUROCRYPT)*, LNCS 12107, pages 342–375. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-45727-3_12.

[Cho15] T. Chou. Sandy2x: New Curve25519 speed records. In O. Dunkelman and L. Keliher, editors, *Selected Areas in Cryptography (SAC)*, LNCS 9566, pages 145–160. Springer-Verlag, 2015. https://doi.org/10.1007/978-3-319-31301-6_8.

[CJL+20] F. Campos, L. Jellema, M. Lemmen, L. Müller, D. Sprenkels, and B. Viguier. Assembly or optimized C for lightweight cryptography on RISC-V? In *Cryptology and Network Security (CANS)*, LNCS 12579, pages 526–545. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-65411-5_26.

[CJS14] A.M. Childs, D. Jao, and V. Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *Journal of Mathematical Cryptology (JMC)*, 8(1):1–29, 2014. https://doi.org/10.1515/jmc-2012-0016.

[CKK+22] P. Choi, W. Kong, J.-H. Kim, M.-K. Lee, and D.K. Kim. Architectural supports for block ciphers in a RISC CPU core by instruction overloading. *IEEE Transactions on Computers (TOC)*, 71(11):2844–2857, 2022. https://doi.org/10.1109/TC.2021.3050515.

[CLM+18] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes. CSIDH: an efficient post-quantum commutative group action. In *Advances in Cryptology (ASIACRYPT)*, LNCS 11274, pages 395–427. Springer-Verlag, 2018. https://doi.org/10.1007/978-3-030-03332-3_15.

[CLN16] C. Costello, P. Longa, and M. Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In *Advances in Cryptology (CRYPTO)*, LNCS 9814, pages 572–601. Springer-Verlag, 2016. https://doi.org/10.1007/978-3-662-53018-4_21.

[Com90] P.G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990. https://doi.org/10.1147/sj.294.0526.

[Coo66] S.A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, 1966.

[Cor14] J.-S. Coron. Higher order masking of look-up tables. In P.Q. Nguyen and E. Oswald, editors, *Advances in Cryptology (EUROCRYPT)*, LNCS 8441, pages 441–458. Springer-Verlag, 2014. https://doi.org/10.1007/978-3-642-55220-5_25.

[COR22] D. Cervantes-Vázquez, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez. Parallel strategies for SIDH: toward computing SIDH twice as fast. *IEEE Transactions on Computers (TOC)*, 71(6):1249–1260, 2022. https://doi.org/10.1109/TC.2021.3080139.

[Cos19] C. Costello. Supersingular isogeny key exchange for beginners. Cryptology ePrint Archive, Report 2019/1321, 2019. https://eprint.iacr.org/2019/1321.

[CP20] L. Choquin and F. Piry. Arm custom instructions: Enabling innovation and greater flexibility on Arm. Technical report, Arm Ltd., 2020. https://www.arm.com/why-arm/technologies/custom-instructions.

[CP23] H. Cheng and D. Page. eLIMInate: a Leakage-focused ISE for Masked Implementation. Cryptology ePrint Archive, Paper 2023/966, 2023. https://eprint.iacr.org/2023/966.

[CR22] J. Chi-Domínguez and F. Rodríguez-Henríquez. Optimal strategies for CSIDH. *Advances in Mathematics of Communications (AMC)*, 16(2):383–411, 2022. https://doi.org/10.3934/amc.2020116.

[Cro06] Micaz wireless measurement system. Technical report, Crossbow Technology, Inc., 2006. Data sheet, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[CS09] N. Costigan and P. Schwabe. Fast elliptic-curve cryptography on the cell broadband engine. In B. Preneel, editor, *Progress in Cryptology (AFRICACRYPT)*, LNCS 5580, pages 368–385. Springer-Verlag, 2009. https://doi.org/10.1007/978-3-642-02384-2_23.

[DeF17] L. De Feo. Mathematics of isogeny based cryptography. *CoRR*, abs/1711.04062, 2017. http://arxiv.org/abs/1711.04062.

[DEM+21] C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel, B. Mennink, R. Primas, and T. Unterluggauer. ISAP. Submission to NIST (version 2.0), 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/isap-spec-final.pdf.

[DEMS21] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon. Submission to NIST (version 1.2), 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf.

[DeV18] H. De Valence. An AVX512-IFMA implementation of the vectorized point operation strategy, 2018. https://github.com/dalek-cryptography/curve25519-dalek/blob/main/docs/ifma-notes.md.

[DG18] N. Drucker and S. Gueron. Fast modular squaring with AVX512IFMA. Cryptology ePrint Archive, Paper 2018/335, 2018. https://eprint.iacr.org/2018/335.

[DG19] L. De Feo and S.D. Galbraith. SeaSign: compact isogeny signatures from class group actions. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology (EUROCRYPT)*, LNCS 11478, pages 759–789. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-17659-4_26.

[DGH19] E. De Mulder, S. Gummalla, and M. Hutter. Protecting RISC-V against side-channel attacks. In *Design Automation Conference (DAC)*, pages 45:1–45:4, 2019. https://doi.org/10.1145/3316781.3323485.

[DGK19] N. Drucker, S. Gueron, and V. Krasnov. Making AES great again: The forthcoming vectorized AES instruction. In *Information Technology New Generations (ITNG)*, AISC 800, pages 37–41. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-14070-0_6.

[DHAK18] J. Daemen, S. Hoffert, G. van Assche, and R. van Keer. The design of Xoodoo and Xoofff. *IACR Transactions on Symmetric Cryptology (TOSC)*, 2018(4):1–38, 2018. https://doi.org/10.13154/tosc.v2018.i4.1-38.

[DHH+15] M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A.H. Sánchez, and P. Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography (DCC)*, 77(2-3):493–514, 2015. https://doi.org/10.1007/s10623-015-0087-1.

[DHM+21] J. Daemen, S. Hoffert, S. Mella, M. Peeters, G. van Assche, and R. van Keer. Xoodyak, a lightweight cryptographic scheme. Submission to NIST (version 2.0), 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/xoodyak-spec-final.pdf.

[DJP14] L. De Feo, D. Jao, and J. Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *Journal of Mathematical Cryptology (JMC)*, 8(3):209–247, 2014. https://doi.org/10.1515/jmc-2012-0015.

[DKL+20] L. De Feo, D. Kohel, A. Leroux, C. Petit, and B. Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In S. Moriai and H. Wang, editors, *Advances in Cryptology (ASIACRYPT)*, LNCS 12491, pages 64–93. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-64837-4_3.

[DLLW23] L. De Feo, A. Leroux, P. Longa, and B. Wesolowski. New algorithms for the deuring correspondence - towards practical and secure sqisign signatures. In C. Hazay and M. Stam, editors, *Advances in Cryptology (EUROCRYPT)*, LNCS 14008, pages 659–690. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-30589-4_23.

[Dwo15] M. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. National Institute of Standards and Technology (NIST), Federal Information Processing Standards (FIPS), 2015. https://doi.org/10.6028/NIST.FIPS.202.

[EFLL20] M. Escouteloup, J.J.A. Fournier, J.-L. Lanet, and R. Lashermes. Recommendations for a radically secure ISA. In *Computer Architecture Research with RISC-V (CARRV)*, 2020. https://carrv.github.io/2020.

[Eri23] Ericsson mobility report. Technical report, Ericsson Inc., 2023. https://www.ericsson.com/49dd9d/assets/local/reports-papers/mobility-report/documents/2023/ericsson-mobility-report-june-2023.pdf.

[ET19] T. Edamatsu and D. Takahashi. Accelerating large integer multiplication using intel AVX-512IFMA. In S. Wen, A.Y. Zomaya, and L.T. Yang, editors, *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, LNCS 11944, pages 60–74. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-38991-8_5.

[FL15] A. Faz-Hernández and J.C. López-Hernández. Fast implementation of Curve25519 using AVX2. In K.E. Lauter and F. Rodríguez-Henríquez, editors, *Progress in Cryptology (LATINCRYPT)*, LNCS 9230, pages 329–345. Springer-Verlag, 2015. https://doi.org/10.1007/978-3-319-22174-8_18.

[FLD19] A. Faz-Hernández, J. López, and R. Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Transactions on Mathematical Software (TOMS)*, 45(3):1–35, 2019. https://doi.org/10.1145/3309759.

[FLOR18] A. Faz-Hernández, J.C. López-Hernández, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. *IEEE Transactions on Computers (TOC)*, 67(11):1622–1636, 2018. https://doi.org/10.1109/TC.2017.2771535.

[Gal99] S.D. Galbraith. Constructing isogenies between elliptic curves over finite fields. *LMS Journal of Computation and Mathematics*, 2:118–138, 1999. https://doi.org/10.1112/S1461157000000097.

[GAST05] J. Großschädl, R.M. Avanzi, E. Savaş, and S. Tillich. Energy-efficient software implementation of long integer modular arithmetic. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 3659, pages 75–90. Springer-Verlag, 2005. https://doi.org/10.1007/11545262_6.

[GCC17] GCC Team. AVR-GCC wiki, 2017. http://gcc.gnu.org/wiki/avr-gcc#Exceptions_to_the_Calling_Convention.

[GD23] J. Gaspoz and S. Dhooghe. Threshold implementations in software: Micro-architectural leakages in algorithms. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2023(2):155–179, 2023. https://doi.org/10.46586/tches.v2023.i2.155-179.

[GGM+21] S. Gao, J. Großschädl, B. Marshall, D. Page, T. Pham, and F. Regazzoni. An instruction set extension to support software-based masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):283–325, 2021. https://doi.org/10.46586/tches.v2021.i4.283-325.

[GHP+21] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem. Coco: Co-design and co-verification of masked software implementations on CPUs. In *USENIX Security Symposium*, pages 1469–1468, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl.

[GIK+21] C. Guo, T. Iwata, M. Khairallah, K. Minematsu, and T. Peyrin. Romulus. Submission to NIST (version 1.3), 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/romulus-spec-final.pdf.

[GJM+17] H. Gross, M. Jelinek, S. Mangard, T. Unterluggauer, and M. Werner. Concealing secrets in embedded processors designs. In *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 10146, pages 89–104. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-54669-8_6.

[GK16] S. Gueron and V. Krasnov. Accelerating big integer arithmetic using intel IFMA extensions. In P. Montuschi, M.J. Schulte, J. Hormigo, S.F. Oberman, and N. Revol, editors, *Computer Arithmetic (ARITH)*, pages 32–38. IEEE, 2016. https://doi.org/10.1109/ARITH.2016.22.

[GMPO20] S. Gao, B. Marshall, D. Page, and E. Oswald. Share slicing: friend or foe? *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(1):152–174, 2020. https://doi.org/10.13154/tches.v2020.i1.152-174.

[GMPP20] S. Gao, B. Marshall, D. Page, and T.H. Pham. FENL: an ISE to mitigate analogue micro-architectural leakage. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(2):73–98, 2020. https://doi.org/10.13154/tches.v2020.i2.73-98.

[GPP11] J. Guo, T. Peyrin, and A. Poschmann. The PHOTON family of lightweight hash functions. In *Advances in Cryptology (CRYPTO)*, LNCS 6841, pages 222–239. Springer-Verlag, 2011. https://doi.org/10.1007/978-3-642-22792-9_13.

[GPW+04] N. Gura, A. Patel, A. Wander, H. Eberle, and S.C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 3156, pages 119–132. Springer-Verlag, 2004. https://doi.org/10.1007/978-3-540-28632-5_9.

[Gu17] C. Gu. Integer version of Ring-LWE and its applications. Cryptology ePrint Archive, Paper 2017/641, 2017. https://eprint.iacr.org/2017/641.

[Gue09] S. Gueron. Intel's new AES instructions for enhanced performance and security. In *Fast Software Encryption (FSE)*, LNCS 5665, pages 51–66. Springer-Verlag, 2009. https://doi.org/10.1007/978-3-642-03317-9_4.

[GYH18] Q. Ge, Y. Yarom, and G. Heiser. No security without time protection: we need a new hardware-software contract. In *Asia-Pacific Workshop on Systems (APSys)*, pages 1:1–1:9, 2018. https://doi.org/10.1145/3265723.3265724.

[Ham15] M. Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Paper 2015/625, 2015. https://eprint.iacr.org/2015/625.

[Ham19] M. Hamburg. ThreeBears, 2019. https://www.shiftleft.org/papers/threebears/threebears-spec.pdf.

[HB21] V. Hadzic and R. Bloem. COCOALMA: A versatile masking verifier. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10. IEEE, 2021. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_9.

[HBD+22] A. Hulsing, D.J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, P. Kampanakis, S. Kolbl, T. Lange, M.M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, J.-P. Aumasson, B. Westerbaan, and W. Beullens. SPHINCS+, 2022. https://sphincs.org/data/sphincs+-r3.1-specification.pdf.

[HEY20] H. Hisil, B. Egrice, and M. Yassi. Fast 4 way vectorized ladder for the complete set of Montgomery curves. Cryptology ePrint Archive, Paper 2020/388, 2020. https://eprint.iacr.org/2020/388.

[HHK17] D. Hofheinz, K. Hövelmanns, and E. Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Y. Kalai and L. Reyzin, editors, *Theory of Cryptography (TCC)*, LNCS 10677, pages 341–371. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-70500-2_12.

[HJM07] M. Hell, T. Johansson, and W. Meier. Grain: a stream cipher for constrained environments. *International Journal of Wireless and Mobile Computing (IJWMC)*, 2(1):86–93, 2007. https://doi.org/10.1504/IJWMC.2007.013798.

[HJM+21] M. Hell, T. Johansson, A. Maximov, W. Meier, J. Sönnerup, and H. Yoshida. Grain-128AEADv2. Submission to NIST (version 2.0), 2021. https://csrc.nist.

gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/grain-128aead-spec-final.pdf.

[HKSS12] Y. Hori, T. Katashita, A. Sasaki, and A. Satoh. SASEBO-GIII: A hardware security evaluation board equipped with a 28-nm FPGA. In *IEEE Global Conference on Consumer Electronics*, pages 657–660, 2012. https://doi.org/10.1109/GCCE.2012.6379944.

[HLKA20] A. Hutchinson, J. LeGrow, B. Koziel, and R. Azarderakhsh. Further optimizations of CSIDH: A systematic approach to efficient strategies, permutations, and bound vectors. In *Applied Cryptography and Network Security (ACNS)*, LNCS 12146, pages 481–501. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-57808-4_24.

[HMV04] D.R. Hankerson, A.J. Menezes, and S.A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004. https://doi.org/10.1007/b97644.

[HPS98] J. Hoffstein, J. Pipher, and J.H. Silverman. NTRU: a ring-based public key cryptosystem. In J. Buhler, editor, *Algorithmic Number Theory (ANTS-III)*, LNCS 1423, pages 267–288. Springer-Verlag, 1998. https://doi.org/10.1007/BFb0054868.

[HS03] J. Hoffstein and J.H. Silverman. Random small hamming weight products with applications to cryptography. *Discrete Applied Mathematics*, 130(1):37–49, 2003. https://doi.org/10.1016/S0166-218X(02)00588-7.

[HS15] M. Hutter and P. Schwabe. Multiprecision multiplication on AVR revisited. *Journal of Cryptographic Engineering (JCEN)*, 5(3):201–214, 2015. https://doi.org/10.1007/s13389-015-0093-2.

[HV11] A. Hakkala and S. Virtanen. Accelerating cryptographic protocols: A review of theory and technologies. In *Communication Theory, Reliability, and Quality of Service (CTRQ)*, pages 103–109, 2011.

[Int18a] Intel 64 and IA-32 architectures – software developer's manual (volume 1: Basic architecture). Technical Report 325383-067US, Intel Corp., 2018. http://software.intel.com/en-us/articles/intel-sdm.

[Int18b] Intel 64 and ia-32 architectures optimization reference manual. Technical report, Intel Corp., 2018. https://software.intel.com/content/dam/develop/public/us/en/documents/64-ia-32-architectures-optimization-manual.pdf.

[Int20] 10th generation intel core processor based on ice lake microarchitecture instruction throughput and latency. Technical report, Intel Corp., 2020. https://software.intel.com/content/www/us/en/develop/download/10th-generation-intel-core-processor-instruction-throughput-and-latency-docs.html.

[Int22a] Intel 64 and IA-32 architectures – software developer's manual (volume 2: Instruction set reference a-z). Technical Report 325383-078US, Intel Corp., 2022. http://software.intel.com/en-us/articles/intel-sdm.

[Int22b] Intel 64 and IA-32 architectures optimization reference manual. Technical report, Intel Corp., 2022. http://software.intel.com/en-us/articles/intel-sdm.

[JAC+22] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Hutchinson, A. Jalali, K. Karabina, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, G. Pereira, J. Renes, V. Soukharev, and D. Urbanik. Supersingular Isogeny Key Encapsulation, 2022. https://sike.org/files/SIDH-spec.pdf.

[JAKJ19] A. Jalali, R. Azarderakhsh, M. Kermani, and D. Jao. Towards optimized and constant-time CSIDH on embedded devices. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 11421, pages 215–231. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-16350-1_12.

[JD11] D. Jao and L. De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In B.-Y. Yang, editor, *Post-Quantum Cryptography (PQCrypto)*, LNCS 7071, pages 19–34. Springer-Verlag, 2011. https://doi.org/10.1007/978-3-642-25405-5_2.

[Jel19] L. Jellema. Optimizing Ascon on RISC-V. BSc thesis, Radboud University, 2019. https://www.cs.ru.nl/bachelors-theses/2019/Lars_Jellema___4388747___Optimizing_Ascon_on_RISC-V.pdf.

[JHH+11] K. Jang, S. Han, S. Han, S.B. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In D.G. Andersen and S. Ratnasamy, editors, *Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2011.

[KAK96] Ç.K. Koç, T. Acar, and B.S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996. https://doi.org/10.1109/40.502403.

[KBSV18] A. Karmakar, J.M. Bermudo Mera, S. Sinha Roy, and I. Verbauwhede. Saber on ARM: CCA-secure module lattice-based key encapsulation on ARM. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018(3):243–266, 2018. https://doi.org/10.13154/tches.v2018.i3.243-266.

[KCP16] J. Kelsey, S.-J. Chang, and R. Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash. National Institute of Standards and Technology (NIST), Special Publication 800-185, 2016. https://doi.org/10.6028/NIST.SP.800-185.

[KG19] D. Kostic and S. Gueron. Using the new VPMADD instructions for the new post quantum key encapsulation mechanism SIKE. In N. Takagi, S. Boldo, and M. Langhammer, editors, *Computer Arithmetic (ARITH)*, pages 215–218. IEEE, 2019. https://doi.org/10.1109/ARITH.2019.00050.

[KJJ99] P.C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology (CRYPTO)*, LNCS 1666, pages 388–397. Springer-Verlag, 1999. https://doi.org/10.1007/3-540-48405-1_25.

[KLM07] P.R. Kaye, R. Laflamme, and M. Mosca. *An Introduction to Quantum Computing*. Oxford University Press, 2007.

[KO63] A.A. Karatsuba and Y.P. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics - Doklady*, 7(7):595–596, 1963.

[Koç09] Ç.K. Koç. About cryptographic engineering. In Ç.K. Koç, editor, *Cryptographic Engineering*, chapter 1, pages 1–4. Springer, 2009. https://doi.org/10.1007/978-0-387-71817-0_9.

[Koc96] P.C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology (CRYPTO)*, LNCS 1109, pages 104–113. Springer-Verlag, 1996. https://doi.org/10.1007/3-540-68697-5_9.

[KRS19] M.J. Kannwischer, J. Rijneveld, and P. Schwabe. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In R.H. Deng, V. Gauthier-Umaña, M. Ochoa, and M. Yung, editors, *Applied Cryptography and Network Security (ACNS)*, LNCS 11464, pages 281–301. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-21568-2_14.

[KS20] P. Kiaei and P. Schaumont. Domain-oriented masked instruction set architecture for RISC-V. Cryptology ePrint Archive, Report 2020/465, 2020. https://eprint.iacr.org/2020/465.

[Kup05] G. Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM Journal on Computing*, 35(1):170–188, 2005. https://doi.org/10.1137/S0097539703436345.

[Kup13] G. Kuperberg. Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In S. Severini and F.G.S.L. Brandão, editors, *Theory of Quantum Computation (TQC)*, volume 22 of *LIPIcs*, pages 20–34. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. https://doi.org/10.4230/LIPIcs.TQC.2013.20.

[LDK+21] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehle, and S. Bai. CRYSTALS-DILITHIUM, 2021. https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf.

[Lem20] M. Lemmen. Optimizing Elephant for RISC-V. BSc thesis, Radboud University, 2020. https://www.cs.ru.nl/bachelors-theses/2020/Mauk_Lemmen___4798937___Optimizing_Elephant_for_RISC-V.pdf.

[LG14] Z. Liu and J. Großschädl. New speed records for Montgomery modular multiplication on 8-bit AVR microcontrollers. In D. Pointcheval and D. Vergnaud, editors, *Progress in Cryptology (AFRICACRYPT)*, LNCS 8469, pages 215–234. Springer-Verlag, 2014. https://doi.org/10.1007/978-3-319-06734-6_14.

[LHP20] T. Li, B. Hopkins, and S. Parameswaran. SIMF: Single-instruction multiple-flush mechanism for processor temporal isolation. *CoRR*, abs/2011.10249, 2020. https://arxiv.org/abs/2011.10249.

[LPR10] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In H. Gilbert, editor, *Advances in Cryptology (EUROCRYPT)*, LNCS 6110, pages 1–23. Springer-Verlag, 2010. https://doi.org/10.1007/978-3-642-13190-5_1.

[LSGK14] Z. Liu, H. Seo, J. Großschädl, and H. Kim. Reverse product-scanning multiplication and squaring on 8-bit AVR processors. In L.C.K. Hui, S.H. Qing, E. Shi, and S.-M. Yiu, editors, *Information and Communications Security (ICICS)*, LNCS 8958, pages 158–175. Springer-Verlag, 2014. https://doi.org/10.1007/978-3-319-21966-0_12.

[MAB+23] C.A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, J. Bos, A. Dion, J. Lacan, J.-M. Robert, and P. Veron. Hamming Quasi-Cyclic (hqc), 2023. http://pqc-hqc.org/doc/hqc-specification_2023-04-30.pdf.

[MBTM17] K. McKay, L. Bassham, M.S. Turan, and N. Mouha. Report on lightweight cryptography. Technical report, 2017. https://doi.org/10.6028/NIST.IR.8114.

[MCR19] M. Meyer, F. Campos, and S. Reith. On lions and elligators: An efficient constant-time implementation of CSIDH. In *Post-Quantum Cryptography (PQCrypto)*, LNCS 11505, pages 307–325. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-25510-7_17.

[MMP+23] L. Maino, C. Martindale, L. Panny, G. Pope, and B. Wesolowski. A direct key recovery attack on SIDH. In C. Hazay and M. Stam, editors, *Advances in Cryptology (EUROCRYPT)*, LNCS 14008, pages 448–471. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-30589-4_16.

[MNP+21] B. Marshall, G.R. Newell, D. Page, M.-J.O. Saarinen, and C. Wolf. The design of scalar AES instruction set extensions for RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(1):109–136, 2021. https://doi.org/10.46586/tches.v2021.i1.109-136.

[Mon85] P.L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation (MCOM)*, 44(170):519–521, 1985. https://doi.org/10.1090/S0025-5718-1985-0777282-X.

[Mon87] P.L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation (MCOM)*, 48(177):243–264, 1987. https://doi.org/10.1090/S0025-5718-1987-0866113-7.

[MOP07] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007. https://doi.org/10.1007/978-0-387-38162-6.

[MOV96] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[MP21] B. Marshall and D. Page. SME: Scalable Masking Extensions. Cryptology ePrint Archive, Paper 2021/1416, 2021. https://eprint.iacr.org/2021/1416.

[MPC00] L. May, L. Penna, and A. Clark. An implementation of bitsliced DES on the Pentium MMX™ processor. In *Australasian Conference on Information Security and Privacy (ACISP)*, LNCS 1841, pages 112–122. Springer-Verlag, 2000. https://doi.org/10.1007/10718964_10.

[MPW22] B. Marshall, D. Page, and J. Webb. MIRACLE: MIcRo-ArChitectural Leakage Evaluation. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(1):175–220, 2022. https://doi.org/10.46586/tches.v2022.i1.175-220.

[MR04] S. Micali and L. Reyzin. Physically observable cryptography. In *Theory of Cryptography (TCC)*, LNCS 2951, pages 278–296. Springer-Verlag, 2004. https://doi.org/10.1007/978-3-540-24638-1_16.

[MR18] M. Meyer and S. Reith. A faster way to the CSIDH. In D. Chakraborty and T. Iwata, editors, *Progress in Cryptology (INDOCRYPT)*, LNCS 11356, pages 137–152. Springer-Verlag, 2018. https://doi.org/10.1007/978-3-030-05378-9_8.

[MS16] D. Moody and D. Shumow. Analogues of Vélu's formulas for isogenies on alternate models of elliptic curves. *Mathematics of Computation (MCOM)*, 85(300):1929–1951, 2016. https://doi.org/10.1090/mcom/3036.

[Nat19] National Academies of Sciences, Engineering, and Medicine. *Quantum Computing: Progress and Prospects*. The National Academies Press, 2019. https://doi.org/10.17226/25196.

[NIK04] K. Nadehara, M. Ikekawa, and I. Kuroda. Extended instructions for the AES cryptography and their efficient implementation. In *Signal Processing Systems (SIPS)*, pages 152–157, 2004. https://doi.org/10.1109/SIPS.2004.1363041.

[NIS07] Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. National Institute of Standards and Technology (NIST), Special Publication 800-38D, 2007. http://csrc.nist.gov.

[NIS18] Submission requirements and evaluation criteria for the lightweight cryptography standardization process. National Institute of Standards and Technology (NIST), 2018. https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf.

[NOOS95] E. Nahum, S. O'Malley, H. Orman, and R. Schroeppel. Towards high performance cryptographic software. In *High Performance Communication Subsystems (HPCS)*, pages 69–72, 1995. https://doi.org/10.1109/HPCS.1995.662009.

[NS21] K. Nath and P. Sarkar. Efficient 4-way vectorizations of the Montgomery ladder. *IEEE Transactions on Computers (TOC)*, 71(3):712–723, 2021. https://doi.org/10.1109/TC.2021.3060505.

[OAL18] G. Orisaka, D. Aranha, and J. López. Finite field arithmetic using AVX-512 for isogeny-based cryptography. In *XVIII Simpósio Brasileiro de Segurança da Informação e Sistemas Computacionais (SBSeg)*, pages 49–56, 2018. https://sol.sbc.org.br/index.php/sbseg/article/view/4269/4200.

[OAYT19] H. Onuki, Y. Aikawa, T. Yamazaki, and T. Takagi. A faster constant-time algorithm of CSIDH keeping two points. In *International Workshop on Security (IWSEC)*, LNCS 11689, pages 23–33. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-26834-3_2.

[Pei20] C. Peikert. He gives C-sieves on the CSIDH. In A. Canteaut and Y. Ishai, editors, *Advances in Cryptology (EUROCRYPT)*, LNCS 12106, pages 463–492. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-45724-2_16.

[PFH+20] T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru, 2020. https://falcon-sign.info/falcon.pdf.

[PFPB19] L. Papachristodoulou, A.P. Fournaris, K. Papagiannopoulos, and L. Batina. Practical evaluation of protected residue number system scalar multiplication. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2019(1):259–282, 2019. https://doi.org/10.13154/tches.v2019.i1.259-282.

[Por18] T. Pornin. Why constant-time crypto? BearSSL, 2018. https://bearssl.org/constanttime.html.

[POW18] Power ISA. Technical Report 2.07 B, IBM, 2018. https://ibm.ent.box.com/s/jd5w15gz301s5b5dt375mshpq9c3lh4u.

[PQM4] M.J. Kannwischer, R. Petri, J. Rijneveld, P. Schwabe, and K. Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[Pre23] Precedence Research. Microcontroller (MCU) market - global market size, trends analysis, segment forecasts, regional outlook 2022 - 2030, 2023. https://www.precedenceresearch.com/microcontroller-mcu-market.

[PSRH23] K. Phalakarn, V. Suppakitpaisarn, F. Rodríguez-Henríquez, and M.A. Hasan. Vectorized and parallel computation of large smooth-degree isogenies using precedence-constrained scheduling. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2023(3):246–269, 2023. https://doi.org/10.46586/tches.v2023.i3.246-269.

[PV17] K. Papagiannopoulos and N. Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 10348, pages 282–297. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-64647-3_17.

[Reg04] O. Regev. A subexponential time algorithm for the dihedral hidden subgroup problem with polynomial space. *arXiv preprint*, 2004. https://arxiv.org/abs/quant-ph/0406151.

[Reg05] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In H.N. Gabow and R. Fagin, editors, *Symposium on Theory of Computing (STOC)*, pages 84–93. ACM, 2005. https://doi.org/10.1145/1060590.1060603.

[RI16] F. Regazzoni and P. Ienne. Instruction set extensions for secure applications. In *Design, Automation, and Test in Europe (DATE)*, pages 1529–1534, 2016.

[RNSL17] M. Roetteler, M. Naehrig, K.M. Svore, and K.E. Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. In T. Takagi and T. Peyrin, editors, *Advances in Cryptology (ASIACRYPT)*, LNCS 10625, pages 241–270. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-70697-9_9.

[Rob23] D. Robert. Breaking SIDH in polynomial time. In C. Hazay and M. Stam, editors, *Advances in Cryptology (EUROCRYPT)*, LNCS 14008, pages 472–503. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-30589-4_17.

[RP10] M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 6225, pages 413–427. Springer-Verlag, 2010. https://doi.org/10.1007/978-3-642-15031-9_28.

[RPM20] S. Renner, E. Pozzobon, and J. Mottok. A hardware in the loop benchmark suite to evaluate NIST LWC ciphers on microcontrollers. In *International Conference on Information and Communications Security (ICICS)*, LNCS 12282, pages 495–509. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-61078-4_28.

[RSA78] R.L. Rivest, A. Shamir, and L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. https://doi.org/10.1145/359340.359342.

[RV19] The RISC-V instruction set manual, Volume I: User-level ISA (version 20191213-base-ratified). Technical report, 2019. http://riscv.org/specifications.

[RV21] The RISC-V instruction set manual, Volume II: Privileged architecture (version 20211203-priv-msu-ratified). Technical report, 2021. http://riscv.org/specifications.

[RVB21] RISC-V bit-manipulation ISA-extensions (version 1.0.0). Technical report, 2021. https://github.com/riscv/riscv-bitmanip.

[RVK22] RISC-V cryptographic extension proposals, Volume I: Scalar & entropy source instructions (version 1.0.1). Technical report, 2022. https://github.com/riscv/riscv-crypto/releases/download/v1.0.1-scalar/riscv-crypto-spec-scalar-v1.0.1.pdf.

[RVK23] RISC-V cryptographic extension proposals, Volume II: Vector instructions (version 0.9.5). Technical report, 2023. https://github.com/riscv/riscv-crypto/releases/download/v20230509/riscv-crypto-spec-vector.pdf.

[RVV21] RISC-V "V" vector extension. Technical report, 2021. https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf.

[Saa20] M.-J.O. Saarinen. A lightweight ISA extension for AES and SM4. 2020. https://ascslab.org/conferences/secriscv/program.html.

[SAB+21] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J.M. Schanck, G. Seiler, and D. Stehle. CRYSTALS-Kyber, 2021. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[Sco18] M. Scott. Missing a trick: Karatsuba variations. *Cryptography and Communications*, 10(1):5–15, 2018. https://doi.org/10.1007/s12095-017-0217-x.

[SCS+21] M.A. Shelton, L. Chmielewski, N. Samwel, M. Wagner, L. Batina, and Y. Yarom. Rosita++: Automatic higher-order leakage elimination from cryptographic code. In *Computer and Communications Security (CCS)*, pages 685–699, 2021. https://doi.org/10.1145/3460120.3485380.

[Sho94] P.W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science (FOCS)*, pages 124–134. IEEE Computer Society, 1994. https://doi.org/10.1109/SFCS.1994.365700.

[Sin19] S. Sinha Roy. SaberX4: high-throughput software implementation of saber key encapsulation mechanism. In *International Conference on Computer Design (ICCD)*, pages 321–324. IEEE, 2019. https://doi.org/10.1109/ICCD46524.2019.00050.

[SL21] N. Smart and T. Lange. Post-quantum cryptography: Current state and quantum mitigation. Technical report, European Union Agency for Cybersecurity (ENISA), 2021. https://www.enisa.europa.eu/publications/post-quantum-cryptography-current-state-and-quantum-mitigation.

[SLLH18] H. Seo, Z. Liu, P. Longa, and Z. Hu. SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018(3):1–20, 2018. https://doi.org/10.13154/tches.v2018.i3.1-20.

[SP21] S. Steinegger and R. Primas. A fast and compact RISC-V accelerator for Ascon and friends. In *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 12609, pages 53–67. Springer-Verlag, 2021. https://doi.org/10.1007/978-3-030-68487-7_4.

[SPA16] Oracle SPARC architecture 2011. Technical Report D1.0.0, Oracle Corp., 2016. https://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf.

[SS22] K. Stangherlin and M. Sachdev. Design and implementation of a secure RISC-V microprocessor. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(11):1705–1715, 2022. https://doi.org/10.1109/TVLSI.2022.3203307.

[SSB+21] M.A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *Network and Distributed System Security Symposium (NDSS)*, 2021. https://doi.org/10.14722/ndss.2021.23137.

[Sto19] K. Stoffelen. Efficient cryptography on the RISC-V architecture. In *Progress in Cryptology (LATINCRYPT)*, LNCS 11774, pages 323–340. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-30530-7_16.

[Tak20] D. Takahashi. Fast multiple Montgomery multiplications using Intel AVX-512IFMA instructions. In *International Conference on Computational Science and Its Applications (ICCSA)*, LNCS 12253, pages 655–663. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-58814-4_52.

[Tan09] S. Tani. Claw finding algorithms using quantum walk. *Theoretical Computer Science (TCS)*, 410(50):5285–5297, 2009. https://doi.org/10.1016/j.tcs.2009.08.030.

[Tat66] J. Tate. Endomorphisms of abelian varieties over finite fields. *Inventiones Mathematicae*, 2(2):134–144, 1966. https://doi.org/10.1007/BF01404549.

[TGSD20] E. Tehrani, T. Graba, A. Si-Merabet, and J.-L. Danger. RISC-V extension for lightweight cryptography. In *Euromicro Conference on Digital System Design (DSD)*, pages 222–228, 2020. https://doi.org/10.1109/DSD51259.2020.00045.

[TKS10] S. Tillich, M. Kirschbaum, and A. Szekely. SCA-resistant embedded processors: The next generation. In *Annual Computer Security Applications Conference (ACSAC)*, pages 211–220, 2010. https://doi.org/10.1145/1920261.1920293.

[TLP05] B.L. Titzer, D.K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Information Processing in Sensor Networks (IPSN)*, pages 477–482. IEEE, 2005. https://doi.org/10.1109/IPSN.2005.1440978.

[TMC+21] M.S. Turan, K. McKay, D. Chang, C. Calik, L. Bassham, J. Kang, and J. Kelsey. Status report on the second round of the NIST lightweight cryptography standardization process. Technical report, 2021. https://doi.org/10.6028/NIST.IR.8369.

[TMC+23] M.S. Turan, K. McKay, D. Chang, L. Bassham, J. Kang, N. Waller, J. Kelsey, and D. Hong. Status report on the final round of the NIST lightweight cryptography standardization process. Technical report, 2023. https://doi.org/10.6028/NIST.IR.8454.

[Too63] A. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics-Doklady*, 7:714–716, 1963.

[TWL+22] J. Tian, P. Wang, Z. Liu, J. Lin, Z. Wang, and J. Großschädl. Efficient software implementation of the SIKE protocol using a new data representation. *IEEE Transactions on Computers (TOC)*, 71(3):670–683, 2022. https://doi.org/10.1109/TC.2021.3057331.

[Vél71] J. Vélu. Isogénies entre courbes elliptiques. *Comptes Rendus de l'Académie des Sciences de Paris*, 273:238–241, 1971.

[Wat16] A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California at Berkeley, 2016. https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf.

[WH21] H. Wu and T. Huang. TinyJAMBU. Submission to NIST (version 2.0), 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/tinyjambu-spec-final.pdf.

[WMSN19] R.N.M. Watson, S.W. Moore, P. Sewell, and P.G. Neumann. An introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, 2019. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf.

[WSG+20] N. Wistoff, M. Schneider, F.K. Gürkaynak, L. Benini, and G. Heiser. Prevention of microarchitectural covert channels on an open-source 64-bit RISC-V core. In *Computer Architecture Research with RISC-V (CARRV)*, 2020. https://carrv.github.io/2020.