Spring 5-11-2018

Optimization of Supersingular Isogeny Cryptography for Deeply Embedded Systems

Jeffrey Denton Calhoun
University of New Mexico - Main Campus

Jeffrey Denton Calhoun

Candidate

Electrical and Computer Engineering

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

James Plusquellic , Chairperson

Marios Pattichis

Manel Martinez-Ramon

Optimization of Supersingular Isogeny Cryptography for Deeply Embedded Systems

by

Jeffrey D. Calhoun

B.S., Computer Engineering, University of New Mexico, 2016

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Engineering

The University of New Mexico

Albuquerque, New Mexico

July, 2018

Dedication

Dedicated to those I love and everyone else, even though it might not be appreciated.

Acknowledgments

I would like to thank my friends and colleagues, Donald Owen and Andrew Targhetta, for their mentorship and support in this research. I would also like to thank my adviser, Jim Plusquellic, for his helpful suggestions and timely feedback. Finally, thanks to friends, family, and everyone else who helped out along the journey.

Optimization of Supersingular Isogeny Cryptography for Deeply Embedded Systems

by

Jeffrey D. Calhoun

B.S., Computer Engineering, University of New Mexico, 2016

M.S., Computer Engineering, University of New Mexico, 2018

Abstract

Public-key cryptography in use today can be broken by a quantum computer with sufficient resources. Microsoft Research has published an open-source library of quantum-secure supersingular isogeny (SI) algorithms including Diffie-Hellman key agreement and key encapsulation in portable C and optimized x86 and x64 implementations. For our research, we modified this library to target a deeply-embedded processor with instruction set extensions and a finite-field coprocessor originally designed to accelerate traditional elliptic curve cryptography (ECC). We observed a 6.3-7.5x improvement over a portable C implementation using instruction set extensions and a further 6.0-6.1x improvement with the addition of the coprocessor. Modification of the coprocessor to a wider datapath further increased performance 2.6-2.9x. Our results show that current traditional ECC implementations can be easily refactored to use supersingular elliptic curve arithmetic and achieve post-quantum security.

Contents

List of Figures **ix**

List of Tables **x**

Glossary **xi**

1 Introduction **1**

2 Related Work **4**

3 Evaluated Microarchitectures **8**

3.1 Embedded Core with Instruction Set Extensions 9

3.2 Embedded Core with Coprocessor 10

4 Underlying Mathematics **15**

4.1 Overview of SI Operations 15

4.2 SI Diffie-Hellman (SIDH) 17

4.3 SI Key Encapsulation (SIKE) 18

5 Implementation and Optimizations **20**

5.1 Algorithm Optimizations 20

5.2 ISE Optimizations 22

5.3 Coprocessor Optimizations 27

6 Methodology **29**

7 Experimentation and Results **31**

7.1 $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ Arithmetic Evaluation 32

7.2 Elliptic Curve Arithmetic Evaluation 33

7.3 SIDH Evaluation 34

7.4 SIKE Evaluation 36

7.5 Code Size 37

7.6 FPGA Implementation 38

8 Conclusion **39**

8.1 Future Work 40

A Supporting Data **42**

B Supporting Algorithms **46**

References **48**

List of Figures

3.1 Pete with Monte 11

3.2 The Finite-Field Arithmetic Unit of Monte 12

4.1 Supersingular Elliptic Curve Isogeny Algorithm Computational Hierarchy 16

7.1 Millions of Clock Cycles vs. Field Size of SIDH Algorithms 34

7.2 Millions of Clock Cycles vs. Field Size in Log Scale Across Architectures 36

7.3 Millions of Clock Cycles vs. Field Size of SIKE Algorithms 37

List of Tables

3.1 Core ISEs 10

3.2 Monte Instructions 12

6.1 GNU Make Options for x86/x64 30

7.1 Field Math Cycle Count 33

7.2 EC Operation Cycle Count 33

7.3 CPU Time of Finite-Field Arithmetic for Alice's SIDH Key Generation on $\text{Pete}_{\text{Monte64}}$ 35

7.4 SI Library Code Size (kB) 37

7.5 Hardware Resource Usage 38

A.1 Field Math Cycle Count 43

A.2 EC Operation Cycle Count 44

A.3 SIDH Cycle Count - cc * 10^3 44

A.4 SIKE Cycle Count - cc * 10^3 45

Glossary

abelian group	a group that satisfies commutativity
cryptography	the study of techniques for secure communication in the presence of adversaries
elliptic curve	a smooth, projective, algebraic curve with points that form an abelian group
field	a set of elements on which addition, subtraction, multiplication, and division are defined
finite field	a field that contains a finite number of elements
group	a set of elements and an operation that satisfies closure, associativity, identity, and invertiblity
hardware acceleration	the use of computer hardware to perform a function more efficiently than is possible in software
isogeny	a group homomorphism and rational map

Chapter 1

Introduction

Over the last decade, the convergence of multiple technologies including communications, real-time analytics, and machine learning has evolved the vision of the Internet of things (IoT). Experts predict that the IoT will consist of nearly 30 billion devices and reach a global market value of $7.1 trillion by 2020 [9]. These devices will have a range of applications such as environmental monitoring, home and building automation, transportation, and medical and healthcare. Because IoT device data is currently following cryptographic standards and using encryption in end-to-end scenarios [26], post-quantum cryptography for embedded devices will be necessary in the years to come.

The last few years have seen a tremendous surge in the study of post-quantum cryptography. This is mainly due to the ongoing development of quantum computers and their ability to compromise currently used cryptographic protocols. The problem with current public-key algorithms is that their security relies on one of three hard mathematical problems: the integer factorization problem [23], the discrete logarithm problem [5], or the elliptic curve discrete logarithm problem [19]. Shor's algorithm can solve the integer factorization problem in polynomial time on

a quantum computer [25], which is substantially faster than the sub-exponential runtime of the most efficient known algorithm running on a classical computer [1]. Furthermore, quantum polynomial time algorithms for solving the hidden subgroup problem over finite abelian groups have been shown [6] to weaken the discrete logarithm and elliptic curve discrete logarithm problems.

In 2006, Rostovtsev et al. created a key agreement algorithm that relies on the difficulty of computing isogenies between ordinary elliptic curves with the aim of making it quantum-resistant [24]. While the best known classical algorithm for solving this problem requires exponential time [7], its quantum variant has been shown to recover keys in sub-exponential time [2]. In 2011, De Feo et al. improved upon the work of Rostovtsev and created what is believed to be a quantum-resistant key agreement algorithm whose security is based on the hardness of finding isogenies between supersingular elliptic curves [13]. This algorithm has the advantages that it can be built on top of many of the same primitives already in use for elliptic curve Diffie-Hellman and that it breaks the abelian group structure, thus making it secure against the quantum attack on the hidden subgroup problem. Currently, there are no known polynomial time algorithms that can solve this problem and the best known quantum and classical algorithms are both exponential time [4]. In 2016, Costello et al. of Microsoft Research released an open-source library[1] of SI Diffie-Hellman (SIDH) and key exchange (SIKE) algorithms.

For our research, we modified this open-source library to target a deeply embedded system with instruction set extensions and a reconfigurable finite-field coprocessor originally designed to improve the performance of traditional elliptic curve cryptography (ECC) [28]. It is our intent to showcase how SI algorithms can be improved using these ECC optimizations. Furthermore, we modify the

[1] https://github.com/Microsoft/PQCrypto-SIDH

pre-existing coprocessor to use a wider datapath and show the performance improvement achieved.

The contributions of this paper are:

- Evaluation and optimization of SI algorithms on a deeply embedded system across the p503, p751, and p964 primes
- Evaluation of instruction set extensions and a reconfigurable finite-field coprocessor developed for traditional ECC applied to SI algorithms, showing 6.3-7.5x and 38.7-45.3x speedups over portable C code, respectively, with the highest speedups at larger prime sizes
- Modification of finite-field coprocessor from 32- to 64-bit datapath, showing additional 2.65-2.95x speedup
- Resource usage of optimized embedded system on 2 different FPGA platforms
- Comparison of optimized embedded system against 32- and 64-bit x86 system, showing comparable performance on a cycle-by-cycle basis

Chapter 2

Related Work

In 2016, Costello et al. created an open-source library containing SIDH algorithms and defined two different primes: p503 and p751. The library is implemented in portable C as well as hand-tuned x86-64 (x64) assembly. The implementation is notable because it uses constant-time algorithms for all operations, making it resistant to timing and cache-timing side channel attacks. Results showed a 3x speedup compared to other non-constant-time implementations at the time. Furthermore, the finite-field arithmetic routines are algorithmically optimized to take advantage of common processor architectures - data is accessed sequentially to improve cache performance, bit-wise operations are used for constant-time operations, and loops are unrolled for improved branch prediction at the cost of code size. Running on an Intel Haswell processor, the SIDHp751 algorithm was capable of generating ephemeral public keys in 46 million cycles for Alice and 52 million cycles for Bob, and shared secret computation took 44 million and 50 million cycles, respectively [3]. While Costello et al. evaluate SIDH on a desktop/server grade processor architecture with aggressive out-of-order execution, our work evaluates SIDH on a resource limited embedded architecture with different levels of hardware acceleration.

Also in 2016, Koziel et al. investigated the efficiency of implementing SIDH on an A8 and A15 processor with NEON SIMD extensions. Their studies produced three new primes: p512, p768, and p1024. They discovered that affine coordinate operations were faster in ARM devices than the Projective coordinates used by Costello et al. Optimized assembly routines resulted in a 2x speedup over a portable C implementation and were nearly 3x faster than other ARMv7 implementations using the NEON extensions. Although the A8 and A15 are 32-bit processors, they still fall into a higher performance category than the processor we evaluate in this work. For example, the A8 is a dual-issue superscalar architecture, and the A15 is a dual-issue out-of-order architecture. Furthermore, the NEON extensions fall into the Single Instruction Multiple Data (SIMD) class of data-level parallelism, whereas we utilize a dedicated accelerator for finite-field arithmetic. Koziel also presented a constant-time hardware implementation of SIDH synthesized to a Virtex-7 FPGA. Hardware datapaths can be designed to take advantage of a heavily parallelized workload in $\mathbb{F}_{p^2}$, which forms the foundation of SI operations. By further replicating these arithmetic logic units, the computation of large-degree isogenies was improved by 2x over the fastest SIDHp751 implementation at the time. The hardware implementation of SIDHp751 was capable of generating ephemeral public keys in 10.6 and 11.6 milliseconds and compute the shared secret key in 9.5 and 10.8 milliseconds for Alice and Bob, respectively [18, 17]. This implementation is faster than ours in terms of raw performance at the expense of large area (26-56k flops, 192-470 DSP blocks) and being tied to a specific field size after synthesis. Our design is lighter weight (4.6-6k flops, 1-17 DSP blocks) and has the benefit of being software-configurable for any of the proposed SI fields.

In 2017, Jalali et al. performed a rigorous study of different implementations of SIDH on a 64-bit A57 processor for 125- and 160-bit quantum security levels. Their optimized assembly routines resulted in a 5x speedup over other portable

ARMv8 implementations. Though experiments showed using affine coordinates is more efficient in the final round of SIDH, it was concluded that Projective coordinates showed better overall performance over the entire protocol. Specifically, over larger finite-fields, Projective formulas show much better performance over affine formulas [11]. The A57 is a superscalar, out-of-order core targeting the mobile device market, and although it has SIMD extensions, Jalali et al. found that they could achieve higher performance multiplication using the normal 64-bit datapath without utilizing the SIMD extensions.

With the help of Jao in 2017, Costello et al. expanded the library to include a SIKE algorithm and p964 was defined. At the time of this writing, optimized implementations for p964 are not included in the library. The SIKE protocol uses many of the same functions as SIDH and three hash functions instantiated with the SHA-3 derived function *cSHAKE256* [15], and was submitted to NIST as part of the post-quantum cryptography standardization effort. Running on an Intel Skylake processor with hand-tuned x64 assembly, the SIKEp751 algorithm was able to generate an ephemeral public key in 31 million cycles and key encapsulation and decapsulation took 50 million and 54 million cycles, respectively. In addition to the software, a SIKE accelerator was written in VHDL and synthesized to a Virtex-7 FPGA. Results showed a total key encapsulation mechanism (KEM) time of 33.35 milliseconds. The same SIKE accelerator was synthesized using TSMC 65-nm CMOS standard technology and CORE65LPSVT standard cell library. This implementation resulted in a KEM time of 18.87 milliseconds [12]. While the SIKE accelerator drastically improved the performance of the algorithm, it is designed specifically for a single prime. Our implementation strikes a balance between performance and area by using a small coprocessor to target the routines that contribute most to computation time.

Later in 2017, Yoo et al. presented the first supersingular elliptic curve isogeny

digital signature algorithm. They have an open-source library[1] containing portable C code as well as hand-optimized x64 assembly. Interestingly, the library employs the functions written by Costello et al. for finite-field arithmetic. Running on an Intel Xeon processor with optimized assembly routines, the digital signature algorithm using p751 took 123 million cycles to generate an ephemeral public key, 57 billion cycles to sign, and 37 billion cycles to verify. A significant fraction of the cost incurred by the signing algorithm can be computed offline, in which case the signing algorithm only needs to compute a hash function [30].

[1] https://github.com/yhyoo93/isogenysignature

Chapter 3

Evaluated Microarchitectures

For our work, we chose to evaluate SIDH and SIKE on an ultra-low energy embedded platform, representative of a microarchitecture one might find in an IoT device. The baseline processor has a 32-bit datapath similar to a PIC32 but has additional Instruction Set Extensions (ISEs) for accelerating traditional ECC. The fully-accelerated microarchitecture marries the baseline processor with a coprocessor specifically designed for finite-field arithmetic. The RTL for the embedded platform was obtained from Targhetta et al. [28] who refer to the processor core as "Pete" and the finite-field arithmetic coprocessor as "Monte." The following sections will describe in detail the evaluated microarchitectures, along with the modifications we made in order to support the larger field sizes found in the SI algorithms.

3.1 Embedded Core with Instruction Set Extensions

The microarchitecture of Pete is a traditional 5-stage, in-order, RISC pipeline with a 32-bit datapath. The integer multiplication unit lies outside of the pipeline and uses special result registers in addition to the 32 general purpose registers. Dedicated instructions move data between the result registers and the general purpose registers, allowing multiplication to happen in parallel with other instructions flowing through the pipeline. This facilitates an energy-efficient, multi-cycle 32x32 multiplication unit that employs a single 16x16 parallel multiplier instead of instantiating a single-cycle, fully parallel multiplier. Even with a tightly nested multi-precision multiplication loop, the extra latency of multiplication can be hidden with loop maintenance and data movement instructions.

On top of the base processor design, Targhetta et al. extended the multiplication unit to support integer multiply-accumulate, overflow accumulation, addition direct to the accumulator, and shift accumulator operations. Accumulating instructions take one more cycle than a non-accumulating instruction. These ISEs, adapted from the work of Groβschädl et al., use the special result registers as an accumulator and were shown to increase the performance of non-constant time traditional elliptic curve algorithms by a factor of 1.30 to 1.43 and energy efficiency by a factor of 1.28 to 1.41 [8, 28]. In short, the ISEs allow intermediate computation to remain within the accumulator, and consequently, improve the efficiency of multi-precision addition and product-scanning multiplication.

Several instructions tailored to support binary fields — $GF(2^m)$ — are also part of the architecture but were not used for this work. In keeping with the theme of targeting a pre-existing design intended for use with traditional ECC, none of the ISEs were modified, and the binary field-specific instructions were

Table 3.1: Core ISEs

Format	Operation	Latency (cycles)
maddu rs, rt	$\{Ov, Hi, Lo\} \leftarrow \{Ov, Hi, Lo\} + rs * rt$	5
m2addu rs, rt	$\{Ov, Hi, Lo\} \leftarrow \{Ov, Hi, Lo\} + 2 * rs * rt$	5
addau rs, rt	$\{Ov, Hi, Lo\} \leftarrow \{Ov, Hi, Lo\} + rs + rt$	3
sha	$\{Ov, Hi, Lo\} \leftarrow \{0, Ov, Hi\}$	1

not removed from the RTL. In addition to the multi-cycle multiplication unit, a small 2-bit counter branch predictor is included and functions primarily as a loop predictor for the tightly nested arithmetic loops present in many cryptographic algorithms.

The ISEs and their operations are detailed in Table 3.1 with respective latencies. This instruction set extended architecture formed the baseline for our implementation of the SI algorithms. Table 3.1 does not include the normal suite of signed and unsigned integer multiplication/division instructions and $GF(2^m)$ extensions that are also present in the core. These ISEs will be used in Algorithms 3, 4, 5, and 9 to accelerate multi-precision and finite-field arithmetic detailed in Chapter 5.

3.2 Embedded Core with Coprocessor

In addition to the core ISEs, Targhetta et al. also designed a coprocessor tailored for prime finite-field arithmetic. The coprocessor, colloquially referred to as "Monte," is depicted in Figure 3.1. The microarchitecture is implemented as a multi-processing system such that the coprocessor shares memory with the microprocessor, reducing potential performance bottlenecks common amongst bus-style accelerators. Coprocessor instructions are fetched and partially decoded by Pete and then forwarded to Monte via the coprocessor interface. Within Monte, copro-

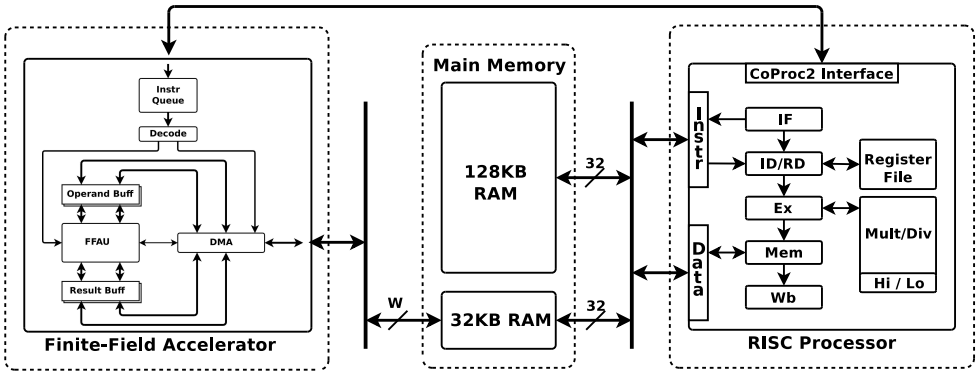


Figure 3.1: Pete with Monte
The processor core with the prime finite-field coprocessor and shared memory. W represents the machine width of Monte. Our work evaluates $W = 32$ and $W = 64$.

cessor instructions are queued, enabling Pete to continue with the higher-level computation while Monte works on the lower-level field math. At the center of Monte is a Finite-Field Arithmetic Unit (FFAU) that computes three multi-precision operations over prime finite-fields: addition, subtraction, and Montgomery multiplication. Operand and result buffers store intermediate field elements and a Direct Memory Access (DMA) engine allows the transfer of field elements between the internal buffers and the shared memory. To allow the overlap of data movement with computation, Monte uses a double buffering scheme for both operands and results.

The coprocessor instructions and their operations are detailed in Table 3.2. Monte follows the simple load-store model in which computation takes place on field elements within the buffers, and separate instructions for loading and storing field elements to and from memory are provided. The first three instructions in Table 3.2 handle the loading of input operands and the prime into the operand buffers. The next three instructions are the computation instructions, and the Store is for storing the result back into memory. Algorithm 1 illustrates the assembly routine that coordinates finite-field addition using the coprocessor.

Table 3.2: Monte Instructions

Format	Description	Operation
cop2lda rt	Load A into operand buffer	OpBuff[A] $\leftarrow$ Mem[GPR[rt]]
cop2ldb rt	Load B into operand buffer	OpBuff[B] $\leftarrow$ Mem[GPR[rt]]
cop2ldn rt	Load N into operand buffer	OpBuff[N] $\leftarrow$ Mem[GPR[rt]]
cop2mul	Modular multiplication	ResultBuff $\leftarrow A * B \bmod N$
cop2add	Modular addition	ResultBuff $\leftarrow A + B \bmod N$
cop2sub	Modular subtraction	ResultBuff $\leftarrow A - B \bmod N$
cop2st rt	Store result to memory	Mem[GPR[rt]] $\leftarrow$ ResultBuff
cop2sync	Coprocessor 2 Sync	Stall until Monte is idle
ctc2 rt,rd	Move to control register	cop2CR[rd] $\leftarrow$ GPR[rt]

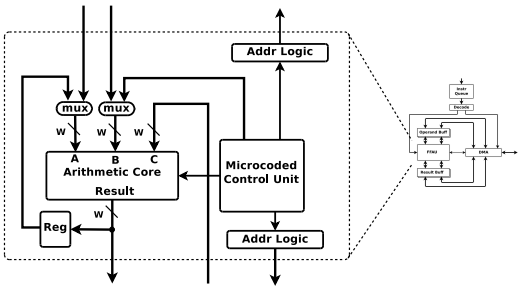


Figure 3.2: The Finite-Field Arithmetic Unit of Monte
Our work evaluates $W = 32$ and $W = 64$.

Subtraction and multiplication assembly routines are similar. There is no implicit synchronization between the microprocessor and the coprocessor; all data synchronization is performed through a *cop2sync* instruction that stalls the microprocessor until the coprocessor is idle. The coprocessor is configured at runtime with constants needed for Montgomery domain computation and with constants that set the correct field size for multi-precision computation. The last instruction in Table 3.2 is used for this configuration.

Figure 3.2 provides a zoomed in view of the FFAU, which is the primary datapath within Monte. Notable features include a 32-bit multiply-add unit, buffer address generation logic, and a microcoded control unit. The multiply-add unit is a 3-stage pipelined arithmetic unit that in multiplication mode performs the

following computation: $Result = OpA * OpB + OpC + Carry$, such that *carry* is the upper 32-bit result from the previous operation. Note that the throughput of the multiply-add unit is one operation per cycle, meaning a multiply-add can start on every clock cycle but takes 3 clock cycles to complete. This type of design lends itself nicely to multi-precision multiplication. In addition mode, the multiply-add unit performs the following computation: $Result = OpA + OpB + OpC + Carry$. The address generation logic blocks are simple, dedicated units for loop maintenance within the tightly nested multi-precision computations. The microcoded control unit orchestrates the arithmetic unit and address generation logic to carry out the Coarsely Integrated Operand Scanning (CIOS) Montgomery multiplication algorithm [16] in addition to modular addition and subtraction. It should be noted that Monte's datapath and control unit were specifically optimized for the CIOS algorithm because modular multiplication is the primary operation to optimize for any cryptographic scheme based on elliptic curves. This has to do with the number of invocations of multiplications of $O(n^2)$ complexity, as compared to $O(n)$ for addition and subtraction.

Algorithm 1 Field Addition using Monte, $c = a + b \bmod n$

Input: $a, b \in F_p, n = F_p$
cop2ldA a ▷ load A operand buffer from pointer a using DMA
cop2ldB b ▷ load B operand buffer from pointer b using DMA
cop2ldN n ▷ In practice, ldN is omitted here; n is pre-loaded at initialization
cop2add
cop2stC c ▷ store result from buffer to pointer c using DMA

Crucially, all Monte operations are constant-time by virtue of the way they are implemented in microcode. Algorithm 2 is a pseudocode representation of the internals of the coprocessor as it processes a *cop2add* instruction. This algorithm is constant-time because it unconditionally processes the prime value subtraction and only selects a buffer that contains the correct result. Previous work did not construct higher-level constant time algorithms that used Monte, but this work

does as detailed in Chapter 5.

Algorithm 2 Constant-Time Modular Addition Using Monte

Input: $a, b \in F_p, n = F_p$
$x \leftarrow a + b$ ▷ Add operands into the first buffer
$\{borrow, x'\} \leftarrow x - n$ ▷ Subtract the modulus into the second buffer
if *borrow* **then** ▷ $a + b < n$, select the first buffer
    $c \leftarrow x$
else ▷ $a + b \geq n$, select the second buffer
    $c \leftarrow x'$
end if

For our research, the architecture of Monte was only initially modified to increase the size of the operand buffers to a suitable size for the p964 extension field. The change needed to enable this functionality was sizing the operand buffers such that they were $\geq$ 384 bytes in size - enough to hold a, b, and n operands up to 1024 bits in size. Note that the previous work using Monte only sized operand buffers for up to a 521-bit field, the largest prime field standardized by NIST for traditional elliptic curve cryptography [22]. In subsequent work, we increased the datapath width to 64 bits wide to further accelerate the larger field sizes required for the SI algorithms. The modifications required for the 64-bit datapath were minimal. The FFAU core, the coprocessor top level buses, and the configuration registers were extended to 64 bits. We did not modify any microcode or control logic.

Chapter 4

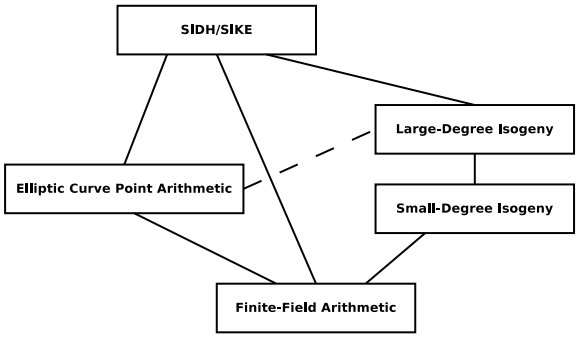
Underlying Mathematics

In this chapter we first give a brief description of the underlying operations used in SI algorithms. Next we will "zoom in" and discuss some of the optimizations that can be made to the algorithms. We will then provide a high-level description of the SIDH and SIKE protocols. Our contributions to the finite-field arithmetic implementation are discussed in Chapter 5, but here we provide the context for their use in the class of SI algorithms.

4.1 Overview of SI Operations

For traditional ECC, the primary computational burden is scalar point multiplication over a single elliptic curve, while for SIDH and SIKE the primary computation is a large-degree isogeny from one supersingular elliptic curve to another. Figure 4.1 illustrates the computational hierarchy of the protocols, showing that large-degree isogenies are computed using many small-degree isogeny evaluations along with scalar point multiplications. Finally, scalar point multiplications and small-degree isogenies are evaluated using finite-field arithmetic. The primary takeaway

Figure 4.1: Supersingular Elliptic Curve Isogeny Algorithm Computational Hierarchy



here is that, like traditional ECC, SIDH and SIKE are built on top of finite-field arithmetic. Our measurements in Table 7.3 show that, even for highly accelerated architectures, 89-94% of the key generation computation time is spent in finite-field arithmetic routines. Thus, we can apply many of the same hardware acceleration techniques used for traditional ECC to SIDH and SIKE. The question our research attempts to answer is: "how well do these techniques carry over?"

The implementation operates over finite-field $\mathbb{F}_{p^2}$ with a characteristic of the form $p = \ell_A^{e_A} \ell_B^{e_B} f \pm 1$, where ℓ_A and ℓ_B are small primes and f is a cofactor such that p is prime. A Montgomery curve $E_{a,b}/\mathbb{F}_{p^2}$ is a special form of an elliptic curve and is defined to be the set of points $P = (x, y)$ of solutions in $\mathbb{F}_{p^2}$ to the equation $by^2 = x^3 + ax^2 + x$, as well as the point at infinity ∞. An isogeny $\phi : E_1 \to E_2$ is a group homomorphism from E_1 to E_2, and isogenies of Montgomery curves can be efficiently computed using elliptic curve point arithmetic. The j-invariant of the Montgomery curve defined by the implementation is computed as $j(E_{a,b}) = \frac{256(a^2-3)^3}{a^2-4}$ [3, 12, 13].

The computational burden of the SIDH and SIKE protocols is in the Montgomery ladder and large-degree isogeny calculations. Let $\{P, Q\}$ be two points in $E_{a,b}$ and $\{m, n\}$ be two random integers both less than ℓ^e. The SI protocols require computation of a random point $mP + nQ$ in the kernel of the isogenies.

This is equivalent to $P + (m^{-1}n)Q$, which can be efficiently computed using the double-and-add technique. The downside to using this standard technique is that it is vulnerable to simple power analysis (SPA) and is non-constant time. To avoid SPA and timing side channels, the significantly slower Montgomery ladder is used to compute the point. De Feo et al. optimized the Montgomery ladder routine to take advantage of the fact that, on each iteration, the values xQ, $(x+1)Q$, and $P + xQ$ are stored for x equal to the leftmost bits of $m^{-1}n$. By using differential addition of points on a Montgomery curve, the ladder can be made comparable in efficiency to double-and-add on twisted Edwards curves. Costello et al. implement $LADDER3PT : (x(P), x(Q), x(Q-P), a, m) \rightarrow x(P + mQ)$ as specified by De Feo [13, 3].

The SI protocols also require calculation of $\phi : E_0 \rightarrow E_n$, where $\phi = \phi_{n-1} \circ \cdots \circ \phi_0$ is a large-degree isogeny composed of a chain of n isogenies. Let $\phi_i : E_i \rightarrow E_{i+1}$, then E_{i+1} and the isogeny ϕ_i can be computed using Vélu's formulas [21]. De Feo et al. provide a detailed description of the computational structure of the large-degree isogeny and define a well-formed strategy to reduce the number of scalar-point multiplications and isogeny evaluations needed. Costello et al. generalized this further with a MAGMA[1] script that computes the optimal strategy given the weights of computing a scalar point multiplication versus evaluating an isogeny [13, 3].

4.2 SI Diffie-Hellman (SIDH)

The protocol fixes Montgomery curve $E_0/\mathbb{F}_{p^2} : y^2 = x^3 + x$ and bases $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ as public parameters. To compute her public key, Alice chooses two secret integers $m_A, n_A < \ell_A^{e_A}$ and computes R_A using the $LADDER3PT$ function.

[1] http://magma.maths.usyd.edu.au/magma/

Her secret key is computed as $\phi_A : E_0 \to E_A$ with kernel R_A, and her public key is E_A together with the image points $\{\phi_A(P_B), \phi_A(Q_B)\}$. Bob goes through the same process with the A and B subscripts swapped. Alice and Bob must send two sets of information to ensure non-commutative isogenies. This property makes SIDH different from other Diffie-Hellman-like key agreement protocols, but this property is what breaks the abelian group structure. To compute the shared secret, Alice uses her secret integers and Bob's public key to compute $\phi'_A : E_B \to E_{BA}$ whose kernel is the point $\phi_B(R_A)$. Bob computes E_{AB} in the same way. Because E_{AB} and E_{BA} are isomorphic, Alice and Bob can compute a shared secret as the common j-invariant $j(E_{AB}) = j(E_{BA})$ [3].

4.3 SI Key Encapsulation (SIKE)

SIKE allows Alice to encrypt a secret key before sending to Bob. The SIKE protocol uses the same public parameters used in SIDH for elliptic curve and isogeny calculations. The protocol also fixes $n \in \{192, 256, 320\}$ to represent the length of random bitstrings and hash outputs. Let G be a function that hashes a random bit string $m \in \mathcal{M} = \{0,1\}^n$ concatenated with a public key pk. The function F is used as a key derivation function (KDF) on the j-invariant. H is used to derive the k-bit shared key K from a random bit string m and ciphertext c. All three hash functions are instantiated with the NIST-specified function *cSHAKE256* [12, 15].

Here we'll focus on the public-key encryption (PKE) scheme used by the protocol, which uses the operations already established for SIDH. First, let's assume that Alice and Bob have each already generated a public-private key pair offline. Alice will compute the common j-invariant using Bob's public key and her own private key. Alice can now encrypt message m and produce ciphertext $c_1 = F(j) \oplus m$.

Alice sends the pair (c_0, c_1) to Bob, where c_0 represents Alice's public key. Bob is able to decrypt the ciphertext and retrieve the message $m = c_1 \oplus F(j)$ after computing the j-invariant [12].

The key encapsulation mechanism (KEM) can be built by applying a transformation to the PKE. Once again, we will assume that both Alice and Bob have each generated ephemeral keys. Alice will now choose a value for m and generate the (c_0, c_1) pair to send to Bob as she did using PKE. However, rather than deriving c_0 from her private key, Alice will derive it from the random value $G(m||pk)$. The shared secret can be computed using $H(m||(c_0, c_1))$. Upon receipt of (c_0, c_1), Bob can decrypt the message in the same manner as PKE to retrieve m'. Assuming there are no errors, Bob can compute the same shared secret using $H(m'||(c_0, c_1))$. With the shared secret, Bob can decrypt any messages Alice may have sent [12].

Chapter 5

Implementation and Optimizations

Because SIDH and SIKE are composed of elliptic curve and isogeny operations built on top of finite-field arithmetic, the use of ISEs and a finite-field coprocessor should naturally improve the performance of the protocols. In this chapter, we give an overview of the algorithmic optimizations incorporated in the SI library, then describe how we modified the library to use instruction set extensions and the coprocessor.

5.1 Algorithm Optimizations

Costello et al. wrote the library to use fast constant-time algorithms for elliptic curve operations. Constant time in this context means that the execution time of an algorithm is not dependent on the input. Therefore, no secret information is leaked under a timing or cache-timing attack.

One design decision that led to improved performance was in the selection of the finite-field characteristic. For efficient isogeny calculations, the parameters $\ell_A = 2, \ell_B = 3, f = 1$ were used by the implementation. The parameters e_A, e_B were carefully chosen such that the difference between Alice and Bob's order was minimized ($2^{e_A} \approx 3^{e_B}$) in order to balance the computational cost of the protocols. This proved to be convenient for the 4-isogeny calculations because $\frac{e_A}{2} \approx e_B$. Additionally, for p503, p751, and p964, the bit length is slightly less than a multiple of 64 which causes there to be a number of zero bits at the top of the most significant word. In order to improve the performance of the constant-time implementation, an additional bit of the most significant word is used to allow intermediate results in the range $[0, 2p)$. A constant-time modular correction routine is used to move a $\mathbb{F}_p$ term into the range $[0, p)$. The primes chosen have a form that facilitates fast modular reduction. The Montgomery reduction [20] residue for an input $a < pR$ can be computed using $c = \frac{(a+p(ap' \bmod R))}{R}$ for Montgomery constants R and $p' = -p^{-1} \bmod R$. The special primes result in $p' - 1$ containing approximately $\frac{n}{2}$ zero words in the lower half, where n is the length of the prime in words. These zero words cut the number of multiplications in the reduction algorithm in half. A Comba-based algorithm that separates the multiplication from the modular reduction was used, which allowed the multiplication routine to be implemented using Karatsuba. All these optimizations drastically improve the performance of the modular multiplication which, as seen in Table 7.3, is the largest bottleneck in elliptic curve arithmetic. In the case of p751, the modular reduction showed a speedup of 1.85x when applying these optimizations to the algorithm [3].

In traditional ECC, a widely-used technique involves avoiding inversions by working in Projective space $(X : Y : Z)$. The implementation uses these same optimizations, but also works projectively with curve coefficients. $E_{a,b}$ can also be written as $E_{A:B:C} : By^2 = Cx^3 + Ax^2 + Cx$ such that $a = \frac{A}{C}$ and $b = \frac{B}{C}$. The j-

invariant can then be computed as $j(E_{A:B:C}) = \frac{256(A^2-3C^2)^3}{C^4(A^2-4C^2)}$, which does not depend on the B coefficient. Using projective space in conjunction with Montgomery curves leads to efficient point arithmetic that allows computation dependent only on $(X : Z)$, and new isogeny arithmetic that depends only on $(A : C)$. Using these optimizations, only one inversion is required when generating ephemeral public keys or computing the shared secret. Field inversions can be easily implemented using the binary GCD algorithm, which is not constant-time. Alternatively, by capitalizing on Fermat's little theorem and exponentiation by squaring, a constant-time inversion can be performed. This method is approximately 9x slower than using binary GCD, but was reported to have less than a 1% impact on the overall latency of SIDH [3].

Other optimizations to the elliptic curve and isogeny algorithms include an efficient Montgomery point tripling function used in the construction of large-degree isogenies, a 4-isogeny function that was found to be faster than computing a pair of 2-isogenies, and the use of base-field and trace-zero subgroups reduce the size of the public parameters. The implementation strikes a balance between efficiency and simplicity of design [3]. Our research leverages the implementation of these high level routines when executing the protocol. Next, we discuss our contributions to the finite-field arithmetic.

5.2 ISE Optimizations

Multi-precision integers are represented as an array of unsigned integers. Each integer is the size of a processor word (32 bits on Pete), and multi-precision integer routines operate on these word-sized data. Using the extended instruction set, we were able to perform multi-precision integer addition and subtraction in fewer data movement operations with the *addau* instruction (see Table 3.1). Algorithm

3 shows the multi-precision addition function specific to Pete using a pseudocode representation.

Algorithm 3 Multi-precision Addition

```
procedure MpAdd(a,b,c,n) ▷ c = a + b, where n is the array length in words.
    ACC ← 0                                          ▷ Clear the accumulator.
    for i in 0 to n − 1 do
        ACC ← ACC + a_i + b_i                        ▷ Using the addau instruction.
        c_i ← ACC_LO                  ▷ Move data out of the accumulator into c.
        Shift ACC                                    ▷ Using the sha instruction.
    end for
    return ACC_LO                                    ▷ Carry out bit (0 or 1).
end procedure
```

Though ISEs do not make provisions for a subtract accumulator instruction, we can still accelerate subtraction and reduce the number of data movement instructions by using the *addau* instruction. By taking advantage of two's complement representation for negative integers, we can compute $a - b = a + (-b)$. Algorithm 4 shows the multi-precision subtraction function specific to Pete. This function is constant-time as the if-else statement can be implemented as a bit-wise XOR operation.

Algorithm 4 Multi-precision Subtraction

```
procedure MpSub(a,b,c,n) ▷ c = a − b, where n is the array length in words.
    ACC ← 1                                          ▷ Load the accumulator with 1.
    for i in 0 to n − 1 do
        ACC ← ACC + a_i + b_i^C          ▷ b_i^C denotes the one's complement of b_i.
        c_i ← ACC_LO                  ▷ Move data out of the accumulator into c.
        Shift ACC                                    ▷ Using the sha instruction.
    end for
    if ACC_LO = 1 then
        return 0
    else
        return 1
    end if          ▷ If-else statement can be implemented with bit-wise XOR.
end procedure
```

Multi-precision multiplication is not as straightforward, with a larger trade space of potentially optimal algorithms. The optimized x64 implementation of SIDH written by Costello et al. uses the Karatsuba method for fast multi-precision multiplication in software, but this proved to be inefficient in a constant-time implementation using our hardware. Karatsuba trades one multi-precision multiply of quadratic complexity for additional additions and subtractions of linear complexity. Using our hardware and the *maddu* instruction, the multi-precision multiplications required by Karatsuba could be performed quickly. However, the instructions needed to move the data into and out of the accumulator created a "hiccup" in the dataflow of the algorithm. Instead, we settled on using a product scanning algorithm [10] that allowed us to keep the result stored in the accumulator. This led to fewer data movement operations, allowed us to write tight loops efficiently, and showed improved performance over Karatsuba. Algorithm 5 shows what the multi-precision multiplication algorithm looks like targeted for Pete.

Algorithm 5 Multi-precision Multiplication

1: **procedure** MpMul(a,b,c,n) ▷ $c = ab$, where n is the array length in words.
2: $ACC \leftarrow 0$ ▷ Clear the accumulator.
3: **for** i in 0 to $n-1$ **do** ▷ Lower words of result.
4: **for** j in 0 to i **do**
5: $ACC \leftarrow ACC + a_j b_{i-j}$ ▷ Using the *maddu* instruction.
6: $c_i \leftarrow ACC_{LO}$ ▷ Move data out of accumulator into c.
7: Shift ACC ▷ Using the *sha* instruction.
8: **end for**
9: **end for**
10: **for** i in n to $2n-2$ **do** ▷ Upper words of result.
11: **for** j in $i-n+1$ to $n-1$ **do**
12: $ACC \leftarrow ACC + a_j b_{i-j}$
13: $c_i \leftarrow ACC_{LO}$
14: Shift ACC
15: **end for**
16: **end for**
17: $c_{2n-1} \leftarrow ACC_{LO}$ ▷ Store most significant word of result.
18: **end procedure**

For multi-precision squaring, the original library code simply called the *MpMul* routine with the input duplicated. Unfortunately, this method misses out on a slight optimization. Looking at Algorithm 5, if $b = a$ then it follows that $a_i b_j + a_j b_i = a_i a_j + a_j a_i = 2a_i a_j$. By using the *m2addu* instruction, we can eliminate half of the multiplication instructions. Algorithm 9 shows the squaring algorithm using the *m2addu* instruction on Pete.

Up to now we have not discussed the latencies associated with the extended instructions. For example, the *maddu* instruction requires five clock cycles to complete. However, because the accumulator and the arithmetic units that act on it lie parallel to the processor's regular datapath, instructions that do not use the accumulator can be overlapped. Therefore, the latency of multi-cycle accumulator instructions can be hidden behind the pointer increments and memory reads that prepare the next accumulator operation. This leads to efficient algorithms that reduce the number of data movement instructions. Algorithm 6 "zooms in" on the first loop of the *MpMul* routine (lines 3-9) and shows the interleaving of instructions. The body of the innermost loop contains one accumulator instruction (line 10) that is allowed to execute in parallel with the other instructions. The processor will not stall during execution of this loop. However, once the termination condition of this loop is met, the processor will continue to the *mflo* instruction (line 14) and must stall until the result is ready. In this case, the processor must stall for only one clock cycle.

Finite field operations can now be built using the multi-precision functions. Constant-time field addition involves three passes through the operands. The first pass performs the multi-precision addition, the second pass subtracts the modulus value, and the final pass performs a constant-time conditional add of either the modulus value or zero to ensure that the result is in $[0, p)$. Algorithm 7 shows the constant-time field addition using Pete. This algorithm is still constant-time

Algorithm 6 Multi-precision Multiplication in Assembly

```
1:  loop1:                                   ▷ For i in 0 to n − 1 do
2:      addu $t0,$a1,$0                      ▷ t0 ← address of a_0
3:      addu $t1,$a2,$t4                     ▷ t1 ← address of b_i
4:      addu $t9,$a1,$t4
5:      addiu $t9,$t9,4                      ▷ t9 ← loop termination value
6:      loop2:                               ▷ For j in 0 to i do
7:          lw $t5,0($t0)                    ▷ t5 ← a_j
8:          lw $t6,0($t1)                    ▷ t6 ← b_{i−j}
9:          addiu $t0,$t0,4                  ▷ Increment a pointer.
10:         maddu $t5,$t6                    ▷ ACC ← ACC + a_j b_{i−j} (5 cycles remaining)
11:         addiu $t1,$t1,-4                 ▷ Decrement b pointer (4 cycles remaining)
12:         bne $t0,$t9,loop2                ▷ If j ≠ i go to loop2, otherwise continue (3 cycles
        remaining)
13:     addu $t2,$a0,$t4                     ▷ t2 ← address of c_i (2 cycles remaining)
14:     mflo $t1                             ▷ t1 ← ACC_LO (stall for 1 cycle)
15:     sha                                  ▷ Shift ACC
16:     addiu $t4,$t4,4                      ▷ Increment i
17:     sw $t1,0($t2)                        ▷ c_i ← ACC_LO
18:     bne $t4,$v1,loop1                    ▷ If i ≠ n − 1 go to loop1, otherwise continue
```

despite the if-else statement as the conditional add can be achieved using the bit-wise AND operator.

All the finite-field functions are implemented using the multi-precision functions. By improving the performance of the multi-precision functions with the

Algorithm 7 $\mathbb{F}_p$ Addition

```
procedure FpAdd(a,b,c)        ▷ c = a + b mod p, where p is intrinsic to the protocol.
    MpAdd(a, b, c, N_p)       ▷ N_p is the length of p in words.
    mask = 0 − MpSub(c, p, c, N_p)        ▷ Subtract the prime.
    if mask = 0 then
        MpAdd(c, 0, c, N_p)
    else
        MpAdd(c, p, c, N_p)
    end if                    ▷ Conditional addition of p.
end procedure
```

extended instruction set, the improvements will propagate up into the higher-level routines.

5.3 Coprocessor Optimizations

Using Monte, the field additions, subtractions, and multiplications are all constant-time by design of the coprocessor's microcode and datapath. The double buffering scheme allows execution of modular arithmetic while simultaneously queuing operands for the next operation. The prime that defines the finite-field remains static throughout the protocol, so the Montgomery constant used by Monte and the prime only need to be loaded at initialization time. This leads to a slight improvement compared to reloading the prime in each of the software routines. Because Monte performs modular multiplication using the CIOS algorithm, the reduction is interleaved with the multiplication. This eliminates the need for a software modular reduction routine. Additionally, Monte always outputs values in the range $[0, p)$, so there is no longer a need to call the correction routine. This drastically improves the performance of the protocol as every aspect of the finite-field arithmetic is improved. Algorithm 1 shows an example of what the field math looks like using Monte.

Monte is not capable of natively computing modular division or inversion. Instead, inversion is calculated via Fermat's little theorem using Monte to accelerate modular exponentiation. Division can be calculated via inversion and multiplication: $\frac{x}{y} = xy^{-1}$. However, the library eliminates as much modular division in the algorithm as possible, and a fast *FpDiv2* is used by the protocols. Using this algorithm, division can be computed without the need for inversion. Algorithm 8 shows how the division-by-2 algorithm operates.

The algorithm performs finite-field division by 4 by chaining two calls to the

Algorithm 8 Divide by $2 \in \mathbb{F}_p$

procedure FpDiv2(a,c) ▷ $c = \frac{a}{2} \bmod p$
  if a_0 is odd **then**
    $MpAdd(a, p, c, N_p)$ ▷ N_p is the length of p in words.
  else
    $MpAdd(a, 0, c, N_p)$
  end if ▷ Conditional addition of p.
  $MpShiftR1(c, N_p)$ ▷ Multi-precision shift-right-by-1 (divide-by-2).
end procedure

FpDiv2 function back-to-back. Though the use of Monte renders instruction set extensions in the multi-precision functions moot, the *MpAdd* function used in *FpDiv2* can still be improved. The *MpShiftR1* function is trivial to implement and it does not use ISEs. Using Monte, the performance improvement of using ISEs becomes a small percentage of the overall improvement. However, because the multi-precision functions are still required by the underlying operations, the ISEs can still improve performance.

Chapter 6

Methodology

In this chapter we will describe the build process used to collect data for two different implementations provided by the open-source library. One implementation is portable C compiled into x86 assembly. The other implementation is mixed C/assembly that is compiled into x64 assembly.

The library comes with a build script that makes compiling for the different architectures simple. Table 6.1 details the command used to build for the two provided architectures. We used gcc version 4.4.7 to compile the open-source library targets. The x86 target compiles portable C code into 32-bit instructions. The x64 target compiles mixed C/assembly into 64-bit assembly with optimized routines for the finite-field math. At the time of writing, we did not have a processor capable of executing the Intel *adx* (multi-precision add-carry) and *mulx* (unsigned multiply without affecting condition codes) instructions. Because these instructions were new at the time of writing the library, the code allows their use to be disabled. As a result, our reported numbers for the assembly-optimized x64 will be slower than previous publications. The timing data for x86 and x64 was gathered on an Intel Xeon 2.9GHz processor with 64GB RAM, 32KB L1D/L1I

Table 6.1: GNU Make Options for x86/x64

Architecture	Build Command
x86	make CC=gcc ARCH=x86 OPT_LEVEL=GENERIC OPT=-O3 SET=EXTENDED
x64	make CC=gcc ARCH=x64 OPT_LEVEL=FAST OPT=-O3 SET=EXTENDED USE_ADX=FALSE USE_MULX=FALSE

cache, 256KB L2 cache, and 2MB L3 cache.

Code for Pete is built using a custom toolchain compiled using crosstools-ng version 1.22. The Pete compiler is gcc version 5.2.0 and customized to recognize the extended instruction set. The Pete timing data was gathered on the Pete microprocessor synthesized to a Zedboard FPGA and represents the portable C code compiled with our custom toolchain; this served as the baseline to compare our future improvements against. Pete_{ISE} denotes the build for which the finite-field and multi-precision arithmetic is optimized with the ISEs described in Table 3.1. $\text{Pete}_{\text{Monte32}}$ denotes the build that uses both ISEs and Monte with an unaltered, 32-bit datapath. $\text{Pete}_{\text{Monte64}}$ denotes the build that uses both ISEs and Monte with datapath modifications from 32- to 64-bit.

The cycle counts reported in Chapter 7 were collected by running the tests contained in the library. The tests are provided to benchmark the various operations used in the algorithms and check the correctness of the implementations. For the finite-field and elliptic curve and isogeny benchmarks, the x86 and x64 builds perform 100,000 bench loops and 100 test loops. For each of the Pete builds we used 100 bench loops and test loops. Because Pete does not have a data cache, we saw only small variations between the execution times when looping and decided that 100 loops was enough to filter out the noise. With 100 test loops, we are able to show the same level of fidelity as the provided implementations.

Chapter 7

Experimentation and Results

In this chapter, we benchmark the implementations over the finite-field arithmetic functions, elliptic curve and isogeny functions, and finally the SIDH and SIKE protocols. By showing the benchmark times for p503, p751, and p964, we hope to showcase the following:

- hardware built to accelerate traditional ECC at the finite-field level can be easily adapted to accelerate SIDH and SIKE
- well-designed ISEs can accelerate software-only SI implementations with high payoff for minimal logic
- with a reconfigurable coprocessor, an embedded system can perform competitively with high-end processors
- higher levels of acceleration have a lower scaling factor for larger field sizes compared to unaccelerated architectures; increasing the security factor on a system with acceleration has a lower performance penalty than on a non-accelerated system

7.1 $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ Arithmetic Evaluation

Table 7.1 shows the timing measurements for the p751 finite-field algorithms. Primes p503 and p964 are omitted for brevity but are included in Table A.1 in the Appendix. The Pete column represents a portable C implementation compiled for the Pete processor. The Pete_{ISE} column represents a mixed C/assembly implementation that uses the ISEs for multi-precision and finite-field operations. Finally, the $\text{Pete}_{\text{Monte32}}$ and $\text{Pete}_{\text{Monte64}}$ columns represent mixed C/assembly implementations that coordinate finite-field operations with the coprocessor with a 32- and 64-bit datapath, respectively.

As with ECC, multiplication and squaring are the most significant operations in the SI algorithms due to their higher computational complexities, e.g. $O(n^2)$ as opposed to $O(n)$ for addition and subtraction. As seen in Table 7.1, the multiplication time for the baseline Pete implementation is nearly 3x slower than that of the x86 build, but the Pete_{ISE} multiplication is over 2x and 7x faster than the x86 and baseline Pete builds, respectively. For Monte, the 32-bit variant is a little over 2x slower than the optimized x86 64-bit build (x64), but the 64-bit variant is 1.64x faster than the x64 build. For squaring, only the Pete_{ISE} architecture takes advantage of a separate squaring algorithm facilitated by the accumulator in conjunction with the $m2addu$ instruction shown in Table 3.1. In such case, we see up to a 12% reduction over multiplication for p751.

When looking at the extension field level, we see that x64 closes the performance gap, i.e. $\text{Pete}_{\text{Monte64}}$ $GF(p751^2)$ multiplication is only 1.2x faster than that of x64. The reason for this is that $\text{Pete}_{\text{Monte64}}$ does not take advantage of some mathematical optimizations, such as lazy reduction, at the extension field level. Instead, the design of the accelerator slightly favors simplicity over efficiency and reduces the results $\bmod p$ for every field math operation. This fact also contributes

Table 7.1: Field Math Cycle Count

Op	x86	x64	Pete	Pete_{ISE}	PM32	PM64
p751						
$\mathbb{F}_p$ Add	591	63	1,413	952	79	43
$\mathbb{F}_p$ Sub	378	52	977	653	79	43
$\mathbb{F}_p$ Mul	23,102	667	75,650	10,164	1,378	406
$\mathbb{F}_p$ Sqr	†	†	†	9,075	†	†
$\mathbb{F}_p$ Red	9,533	268	30,320	5,295	5,317	5,319
$\mathbb{F}_p$ Inv	20,788,481	591,978	67,989,937	8,328,377	1,262,795	378,107
$\mathbb{F}_{p^2}$ Add	1,179	117	2,816	1,904	158	86
$\mathbb{F}_{p^2}$ Sub	781	96	1,944	1,308	158	86
$\mathbb{F}_{p^2}$ Mul	61,496	1,831	198,316	25,610	4,649	1,517
$\mathbb{F}_{p^2}$ Sqr	47,065	1,409	153,107	21,594	3,080	1,004
$\mathbb{F}_{p^2}$ Inv	20,870,580	594,527	68,294,317	8,368,154	1,268,626	379,918

† *Optimized implementation unavailable*

to the smaller buffer sizes within Monte.

7.2 Elliptic Curve Arithmetic Evaluation

Table 7.2 shows the timing measurements for elliptic curve operations over the $GF(p751^2)$ field across the differing architectures. Primes p503 and p964 are omitted for brevity but are included in Table A.2 in the Appendix.

Table 7.2: EC Operation Cycle Count

Op	x86	x64	Pete	Pete_{ISE}	PM32	PM64
p751						
$2P$	344,280	10,615	1,108,823	152,076	25,361	8,417
$3P$	675,372	20,669	2,175,303	302,112	49,431	16,479
$get3isog(P)$	281,556	9,497	895,004	142,788	20,899	7,375
$eval3isog(P)$	344,389	10,609	1,108,810	152,060	25,376	8,420
$get4isog(P)$	193,859	6,203	625,422	95,316	13,096	4,456
$eval4isog(P)$	470,043	14,615	1,510,109	206,492	34,962	11,610

7.3 SIDH Evaluation

Figure 7.1 illustrates the clock cycle measurements of the entire SIDH protocol over each of the primes for the various architectures we evaluated. Further details of the exact clock cycle timings for each protocol component are included in Table A.3 in the appendix. It should be noted that we did not include results for p964 on the x64 target because assembly optimized routines for p964 do not yet exist. Pete (without ISEs) took 4x the clock cycles of an x86 processor, and nearly 100x that of an x64 over each of the primes. By redesigning the finite-field operations to use the ISEs, we saw a speedup of approximately 6x over the portable implementation. Pete_{ISE} showed better performance than that of the x86 build representative of a 32-bit desktop processor.

With the addition of Monte we saw another 7x improvement over that of Pete_{ISE}. This is approximately a 42x speedup over the portable implementation. Though Monte renders modifications to the field math routines moot, the ISEs still improved the multi-precision integer routines. After modifying Monte to have a 64-bit datapath we saw another 2.97-3.17x speedup over the 32-bit datapath.

Close examination of Figure 7.1 shows that the Pete and Monte architectures

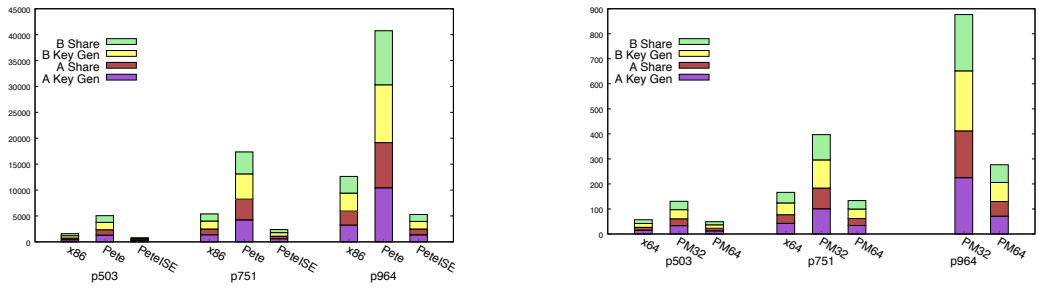


Figure 7.1: Millions of Clock Cycles vs. Field Size of SIDH Algorithms
x86, Pete running portable C, and Pete with ISEs running assembly optimized code (left) and x64, Pete with Monte32, and Pete with Monte64 (right). Note x64 assembly optimized code for p964 does not exist and was omitted.

Table 7.3: CPU Time of Finite-Field Arithmetic for Alice's SIDH Key Generation on $\text{Pete}_{\text{Monte64}}$

$GF(p)$ Operation	p503	p751	p964
Addition	12.7%	10.13%	8.35%
Subtraction	8.82%	7.04%	5.8%
Multiplication	68.06%	75.65%	80.49%
Total	89.58%	92.82%	94.64%

scale out to larger key sizes better than the baseline Pete and even slightly better than Pete with ISEs. In fact, Monte64 scaling is on par with the optimized x64 implementation going from p503 to p751. We expect similar results when we evaluate p964 on the optimized x64 platform. In order to view the clock cycle times for p503, p751, and p964 on all platforms, Figure 7.2 plots the results using a logarithm scale. This demonstrates the significant levels of acceleration provided by Monte32 and Monte64.

Table 7.3 shows the percentage of time spent in the field math for the implementation accelerated with Monte64. As shown, even the accelerated field math is taking up a significant portion of the computation time, which indicates, according to Amdahl's Law, that this architecture could possibly benefit from even greater hardware acceleration. We also see an expected trend of a higher percentage of field math computation with larger field sizes.

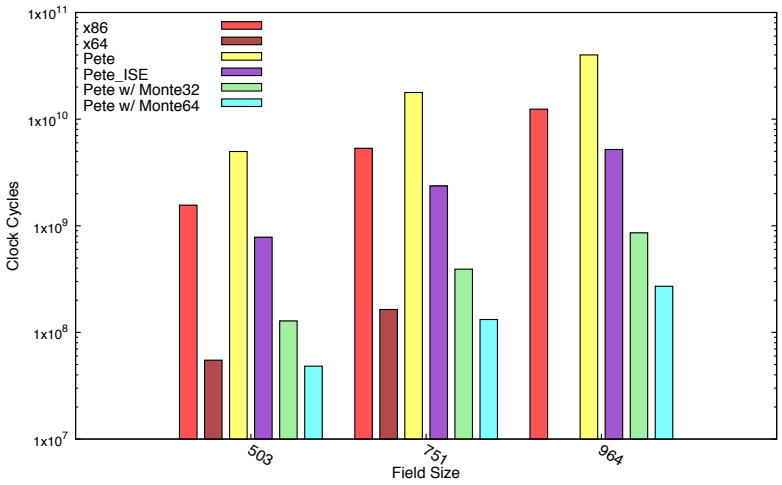


Figure 7.2: Millions of Clock Cycles vs. Field Size in Log Scale Across Architectures

Note x64 assembly optimized code for p964 does not exist and was omitted.

7.4 SIKE Evaluation

Figure 7.3 illustrates the clock cycle measurements of the SIKE algorithms over each of the primes for the different architectures. Refer to Table A.4 in the appendix for a full list of our experimental results for SIKE. The improvement factors for each of the Pete iterations are approximately the same as those observed in the SIDH protocol. However, of notable importance is the fact that x64 is only slightly slower than Monte64. In the case of SIKE, the SHA-3 computations are not accelerated or optimized for Pete, allowing the higher performance x64 to further close the performance gap.

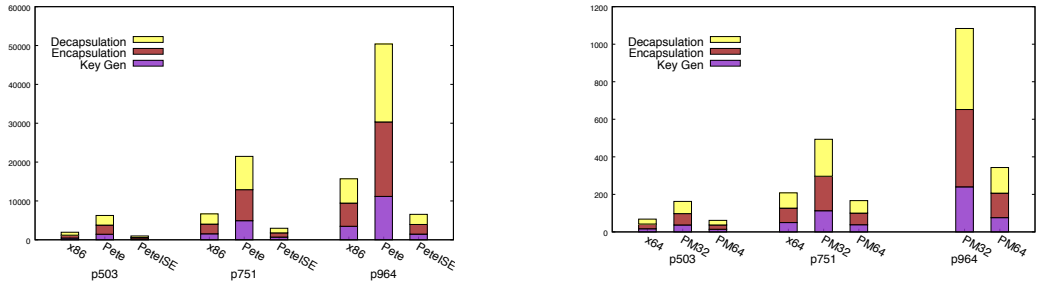


Figure 7.3: Millions of Clock Cycles vs. Field Size of SIKE Algorithms x86, Pete running portable C, and Pete with ISEs running assembly optimized code (left) and x64, Pete with Monte32, and Pete with Monte64 (right). Note x64 assembly optimized code for p964 does not exist and was omitted.

7.5 Code Size

Table 7.4 shows the compiled code size of the SIDH and SIKE libraries. The sizes reported are those of Pete_{ISE}. In each case, $\text{Pete}_{\text{Monte32}}$ and $\text{Pete}_{\text{Monte64}}$ reduce the sizes by approximately 1kB. The code size of p964 is smaller than that of p751 because of the unoptimized inversion routine used in the case of p964. Rather than performing aggressive loop unrolling, a small loop is used to perform the exponentiation by squaring. The main difference between the SIDH and SIKE libraries is in the cSHAKE256 hash function, which is approximately 11kB in size.

Table 7.4: SI Library Code Size (kB)

Library	p503	p751	p964
SIDH	20	22	20
SIKE	32	34	33

7.6 FPGA Implementation

Pete and Monte are designed to be synthesizable to a number of targets. Table 7.5 shows the resources consumed by the various studied architectures on two different Xilinx parts. Of particular note are the DSP primitive numbers; the Pete core only uses 1 16x16 multiplier to implement all of its ISA extensions. As the level of hardware acceleration increases, the DSP numbers scale accordingly. The FFAU core within the Monte accelerator synthesizes into 4 and 16 primitives in pipelined 32x32 and 64x64 configurations.

Table 7.5: Hardware Resource Usage

Primitive	Pete_{ISE}	$\text{Pete}_{\text{Monte32}}$	$\text{Pete}_{\text{Monte64}}$	$\text{Pete}_{\text{Monte64}}$
	Zynq 7Z020			**Kintex 7K325T**
LUTs	4685	5403	6074	6603
FFs	2944	3426	3700	3802
BRAM36	33	43	47	46
BRAM18	1	3	1	0
DSP	1	5	17	17

Chapter 8

Conclusion

We analyzed the new class of SI algorithms on a deeply-embedded processor and compared it to 32- and 64-bit x86 implementations representative of a desktop processor. After first compiling a constant-time portable C version of the open-source SI library, our results showed that a single key generation took over a billion cycles on Pete over all the primes analyzed. By modifying the finite-field and multi-precision routines to use the ISEs, we saw an improvement of 6.3-7.5x compared to the portable C implementation. With the addition of Monte32, results showed a 38.7-45.3x speedup over the portable C implementation. After upgrading to Monte64, the performance improved another 2.65-2.95x over using Monte32. Each of these improvements maintained constant-time algorithms. Monte64 showed better performance than an optimized 64-bit x86 implementation on a cycle-by-cycle basis.

Our design is unique in that Monte is small and completely reconfigurable in software. That is, it can be tailored for each prime extension field at initialization time and does not require resynthesis. This contrasts the SIDH and SIKE accelerators that others developed [17, 12], which are capable of achieving speed records

at the cost of a large area and inflexibility of design.

The ISEs and Monte were originally designed to optimize traditional ECC. Though we made some modifications to the design of Monte, we showed the performance improvements the original ECC optimizations had on the SI algorithms. We believe current ECC implementations could be redesigned to use the quantum secure SI algorithms and still leverage existing optimizations.

8.1 Future Work

The extension of prior work using Pete and Monte from traditional ECC algorithms to SI algorithms was natural considering the similarity of the underlying mathematics. However, most constructions submitted to the NIST PQC project are not based on SI or ECC. As such, we should study the application of the existing architectures to other proposed algorithms. Additionally, we should study whether modifications to or new instructions and accelerators can help make PQC algorithms practical in the embedded domain.

Given the substantial performance increase observed in scaling the Monte datapath from 32 to 64 bits, it would be interesting to explore how a similar scaling of the Pete multiplication/accumulator logic would affect performance. As noted earlier, Monte does not take advantage of the form of the primes for the SI algorithms. Modification of microcode to take advantage of the reduced computation allowed by the primes might be possible and should be explored.

As attacks and security estimates are refined, it is likely that the proposed primes and fields defining specific security levels will change. Because the Pete and Monte architectures are designed with agnosticism to the prime values, we should keep up to date with the latest proposed values, such as faster, smaller

primes less than p503 or larger, more secure primes greater than p964.

In addition to SIDH and SIKE, other SI-based cryptosystems have been proposed, such as SI variants of the Digital Signature Algorithm [14, 27, 12]. Because these algorithms are based on the same mathematics, we believe that the ISEs and coprocessor optimizations can be applied to them in a similar manner. Exploring the performance of these algorithms in an embedded system context should prove interesting.

Finally, we'd like to gather energy and power measurements of SI algorithms. Many embedded applications are energy- and power-limited and we'd like to marry the fields of post-quantum security with ultra-low energy computing. The existing instructions and coprocessor architecture were shown to be efficient across the range of existing ECC security levels [29] (NIST p192 - p521 [22]), but we have not yet explored how energy scales with SI prime sizes or how the trades of datapath size vs power and energy apply to acceleration of SI algorithms.

Appendix A

Supporting Data

Table A.1: Field Math Cycle Count

Op	x86	x64	Pete	Pete_{ISE}	PM32	PM64
p503						
$\mathbb{F}_p$ Add	415	51	955	648	55	31
$\mathbb{F}_p$ Sub	258	47	663	445	55	31
$\mathbb{F}_p$ Mul	11,963	390	34,361	5,161	666	210
$\mathbb{F}_p$ Sqr	†	†	†	4,677	†	†
$\mathbb{F}_p$ Red	5,055	165	13,845	2,670	2,696	2,696
$\mathbb{F}_p$ Inv	6,431k	229,513	20,944k	2,908k	417,670	134,582
$\mathbb{F}_{p^2}$ Add	719	90	1,902	1,296	110	62
$\mathbb{F}_{p^2}$ Sub	445	77	1,318	893	110	62
$\mathbb{F}_{p^2}$ Mul	28,163	1,041	90,347	13,076	2,369	857
$\mathbb{F}_{p^2}$ Sqr	21,534	841	69,928	11,187	1,568	568
$\mathbb{F}_{p^2}$ Inv	6,457k	206,928	21,082k	2,928k	420,565	125,565
p964						
$\mathbb{F}_p$ Add	792	†	1,867	1,256	103	55
$\mathbb{F}_p$ Sub	524	†	1,287	861	103	55
$\mathbb{F}_p$ Mul	41,135	†	133,125	16,816	2,347	666
$\mathbb{F}_p$ Sqr	†	†	†	14,862	†	†
$\mathbb{F}_p$ Red	16,732	†	53,249	8,794	8,820	8,822
$\mathbb{F}_p$ Inv*	71,852k	†	232,926k	27,431k	4,410k	1,183k
$\mathbb{F}_{p^2}$ Add	1,582	†	3,726	2,513	206	110
$\mathbb{F}_{p^2}$ Sub	1,047	†	2,566	1,725	206	110
$\mathbb{F}_{p^2}$ Mul	108,719	†	348,400	43,071	7,697	2,369
$\mathbb{F}_{p^2}$ Sqr	83,087	†	268,643	35,298	5,104	1,568
$\mathbb{F}_{p^2}$ Inv*	72,033k	†	233,461k	27,496k	4,150k	1,186k

k Clock cycles in thousands

** Algorithm does not employ loop-unrolling*

† Optimized implementation unavailable

Table A.2: EC Operation Cycle Count

Op	x86	x64	Pete	Pete_{ISE}	PM32	PM64
p503						
$2P$	157,895	5,385	507,458	79,085	13,041	4,817
$3P$	309,550	10,613	996,409	157,612	25,479	9,463
$get3isog(P)$	130,564	5,020	416,680	77,963	11,115	4,403
$eval3isog(P)$	157,762	5,356	507,441	79,068	13,048	4,816
$get4isog(P)$	89,362	3,256	288,430	50,853	6,824	2,600
$eval4isog(P)$	215,282	7,290	691,240	107,408	17,986	6,642
p964						
$2P$	607,931	†	1,943,241	251,371	41,777	13,041
$3P$	1,188,971	†	3,810,556	497,593	81,319	25,479
$get3isog(P)$	489,670	†	1,554,552	227,366	33,755	11,115
$eval3isog(P)$	607,640	†	1,943,228	251,362	41,800	13,048
$get4isog(P)$	340,409	†	1,091,817	152,981	21,416	6,824
$eval4isog(P)$	826,787	†	2,646,206	341,737	57,570	17,986

† *Optimized implementation unavailable*

Table A.3: SIDH Cycle Count - cc * 10^3

Op	x86	x64	Pete	Pete_{ISE}	PM32	PM64
p503						
A Public Key	404,077	15,141	1,290,674	201,914	33,433	12,534
B Public Key	454,014	15,394	1,422,263	225,263	36,753	13,848
A Shared Key	328,001	11,335	1,052,815	164,620	27,114	10,181
B Shared Key	373,884	13,001	1,201,573	190,536	30,952	11,686
p964						
A Public Key	3,233,546	†	10,425,433	1,350,260	224,892	70,923
B Public Key	3,463,766	†	11,168,205	1,457,806	239,759	75,866
A Shared Key	2,707,163	†	8,730,791	1,128,804	187,088	59,004
B Shared Key	2,990,773	†	9,647,917	1,258,500	206,177	65,271

† *Optimized implementation unavailable*

Table A.4: SIKE Cycle Count - cc * 10^3

Op	x86	x64	Pete	Pete_{ISE}	PM32	PM64
p503						
Public Key	442,746	16,533	1,422,268	225,279	36,758	13,853
Encapsulate	730,149	24,969	2,343,805	366,866	60,861	23,028
Decapsulate	775,677	26,616	2,492,566	392,787	64,703	24,536
p964						
Public Key	3,472,063	†	11,168,214	1,457,807	239,767	75,875
Encapsulate	5,961,490	†	19,156,714	2,479,541	412,467	130,413
Decapsulate	6,269,328	†	20,073,841	2,609,239	431,559	136,687

† *Optimized implementation unavailable*

Appendix B

Supporting Algorithms

Algorithm 9 Multi-precision Squaring

procedure MpSqr(a,c,n) ▷ $c = a^2$, where n is the array length in words.
  $ACC \leftarrow a_0^2$ ▷ Initialize the accumulator.
  $c_0 \leftarrow ACC_{LO}$ ▷ Move data out of accumulator into c.
  Shift ACC ▷ Using the *sha* instruction.
  for k in 1 to $n-1$ **do** ▷ Lower words of result.
    $i \leftarrow 0$
    $j \leftarrow k$
    while $i < j$ **do**
      $ACC \leftarrow ACC + 2a_j a_{i-j}$ ▷ Using the *m2addu* instruction.
      $i \leftarrow i + 1$
      $j \leftarrow j - 1$
    end while
    if $i = j$ **then**
      $ACC \leftarrow ACC + a_i^2$
    end if
    $c_k \leftarrow ACC_{LO}$
    Shift ACC
  end for
  for k in n to $2n-2$ **do** ▷ Upper words of result.
    $i \leftarrow k - (n-1)$
    $j \leftarrow n - 1$
    while $i < j$ **do**
      $ACC \leftarrow ACC + 2a_j a_{i-j}$
      $i \leftarrow i + 1$
      $j \leftarrow j - 1$
    end while
    if $i = j$ **then**
      $ACC \leftarrow ACC + a_i^2$
    end if
    $c_k \leftarrow ACC_{LO}$
    Shift ACC
  end for
  $ACC \leftarrow ACC + a_i^2$
  $c_{2n-1} \leftarrow ACC_{LO}$ ▷ Most significant word.
end procedure

References

[1] Buhler, J. P., Lenstra, H. W., and Pomerance, C. Factoring integers with the number field sieve. In *The development of the number field sieve* (Berlin, Heidelberg, 1993), A. K. Lenstra and H. W. Lenstra, Eds., Springer Berlin Heidelberg, pp. 50–94.

[2] Childs, A., Jao, D., and Soukharev, V. Constructing elliptic curve isogenies in quantum subexponential time. *Journal of Mathematical Cryptology 8*, 1 (2014), 1–29.

[3] Costello, C., Longa, P., and Naehrig, M. Efficient algorithms for supersingular isogeny diffie-hellman. In *Advances in Cryptology – CRYPTO 2016* (Berlin, Heidelberg, 2016), M. Robshaw and J. Katz, Eds., Springer Berlin Heidelberg, pp. 572–601.

[4] Delfs, C., and Galbraith, S. D. Computing isogenies between supersingular elliptic curves over $\mathbb{F}_p$. *Des. Codes Cryptography 78*, 2 (Feb. 2016), 425–440.

[5] Diffie, W., and Hellman, M. New directions in cryptography. *IEEE Trans. Inf. Theor. 22*, 6 (Sept. 2006), 644–654.

[6] Ettinger, M., Høyer, P., and Knill, E. The quantum query complexity of the hidden subgroup problem is polynomial. *Information Processing Letters 91*, 1 (2004), 43 – 48.

[7] Galbraith, S., and Stolbunov, A. Improved algorithm for the isogeny problem for ordinary elliptic curves. *Applicable Algebra in Engineering, Communication and Computing 24*, 2 (Jun 2013), 107–131.

[8] Groβschädl, J., and Savaş, E. Instruction set extensions for fast arithmetic in finite fields $gf(p)$ and $gf(2^m)$. In *Cryptographic Hardware and Em-*

bedded Systems - CHES 2004 (Berlin, Heidelberg, 2004), M. Joye and J.-J. Quisquater, Eds., Springer Berlin Heidelberg, pp. 133–147.

[9] Hsu, C.-L., and Lin, J. C.-C. An empirical examination of consumer adoption of internet of things services: Network externalities and concern for information privacy perspectives. *Computers in Human Behavior 62* (2016), 516 – 527.

[10] Hutter, M., and Wenger, E. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In *Cryptographic Hardware and Embedded Systems – CHES 2011* (Berlin, Heidelberg, 2011), B. Preneel and T. Takagi, Eds., Springer Berlin Heidelberg, pp. 459–474.

[11] Jalali, A., Azarderakhsh, R., Kermani, M. M., and Jao, D. Supersingular isogeny diffie-hellman key exchange on 64-bit arm. *IEEE Transactions on Dependable and Secure Computing PP*, 99 (2017), 1–1.

[12] Jao, D., Azarderakhsh, R., Campagna, M., Costello, C., Jalali, A., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Renes, J., et al. Supersingular isogeny key encapsulation. *NIST Post-Quantum Cryptography Standardization Round 1 Submissions* (Nov. 2017).

[13] Jao, D., and De Feo, L. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Proceedings of the 4th International Conference on Post-Quantum Cryptography* (Berlin, Heidelberg, 2011), PQCrypto'11, Springer-Verlag, pp. 19–34.

[14] Jao, D., and Soukharev, V. Isogeny-based quantum-resistant undeniable signatures. In *Post-Quantum Cryptography* (Cham, 2014), M. Mosca, Ed., Springer International Publishing, pp. 160–179.

[15] Kelsey, J. Sha-3 derived functions: cshake, kmac, tuplehash, and parallelhash. *NIST special publication 800* (2016), 185.

[16] Koc, C. K., Acar, T., and Kaliski, B. S. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro 16*, 3 (Jun 1996), 26–33.

[17] Koziel, B., Azarderakhsh, R., and Mozaffari Kermani, M. Fast hardware architectures for supersingular isogeny diffie-hellman key exchange on fpga, 12 2016.

[18] Koziel, B., Jalali, A., Azarderakhsh, R., Jao, D., and Mozaffari Kermani, M. Neon-sidh: Efficient implementation of supersingular isogeny diffie-hellman key exchange protocol on arm, 11 2016.

[19] Miller, V. S. Use of elliptic curves in cryptography. In *Advances in Cryptology* (London, UK, 1986), CRYPTO '85, Springer-Verlag, pp. 417–426.

[20] Montgomery, P. L. Modular multiplication without trial division. *Mathematics of computation 44*, 170 (1985), 519–521.

[21] Moody, D., and Shumow, D. Analogues of vélu's formulas for isogenies on alternate models of elliptic curves. *Math. Comput. 85* (2011), 1929–1951.

[22] NIST. Fips pub 186-4. digital signature standard (dss). *National Institute of Standards and Technology (NIST)* (2013).

[23] Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb. 1978), 120–126.

[24] Rostovtsev, A., and Stolbunov, A. Public-key cryptosystem based on isogenies. *IACR Cryptology ePrint Archive 2006* (2006), 145.

[25] Shor, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput. 26*, 5 (Oct. 1997), 1484–1509.

[26] Sklavos, N., and Zaharakis, I. D. Cryptography and security in internet of things (iots): Models, schemes, and implementations. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)* (Nov 2016), pp. 1–2.

[27] Sun, X., Tian, H., and Wang, Y. Toward quantum-resistant strong designated verifier signature. *Int. J. Grid Util. Comput. 5*, 2 (Mar. 2014), 80–86.

[28] Targhetta, A. D., Owen, D. E., and Gratz, P. V. The design space of ultra-low energy asymmetric cryptography. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (March 2014), pp. 55–65.

[29] Targhetta, A. D., Owen, D. E., Israel, F. L., and Gratz, P. V. Energy-efficient implementations of gf (p) and gf(2m) elliptic curve cryptography. In *2015 33rd IEEE International Conference on Computer Design (ICCD)* (Oct 2015), pp. 704–711.

[30] Yoo, Y., Azarderakhsh, R., Jalali, A., Jao, D., and Soukharev, V. A post-quantum digital signature scheme based on supersingular isogenies. Cryptology ePrint Archive, Report 2017/186, 2017.