A High-Performance and Scalable Hardware Architecture for Isogeny-Based Cryptography

Brian Koziel, Reza Azarderakhsh, *Member, IEEE*, and Mehran Mozaffari Kermani, *Senior Member, IEEE*

Abstract—In this work, we present a high-performance and scalable architecture for isogeny-based cryptosystems. In particular, we use the architecture in a fast, constant-time FPGA implementation of the quantum-resistant supersingular isogeny Diffie-Hellman (SIDH) key exchange protocol. On a Virtex-7 FPGA, we show that our architecture is scalable by implementing at 83, 124, 168, and 252-bit quantum security levels. This is the first SIDH implementation at close to the 256-bit quantum security level to appear in literature. Further, our implementation completes the SIDH protocol 2 times faster than performance-optimized software implementations and 1.34 times faster than the previous best FPGA implementation, both running a similar set of formulas. Our implementation employs inversion-free projective isogeny formulas. By replicating multipliers and utilizing an efficient scheduling methodology, we can heavily parallelize quadratic extension field arithmetic and the isogeny evaluation stage of the large-degree isogeny computation. For a constant-time implementation of 124-bit quantum security SIDH on a Virtex-7 FPGA, we generate ephemeral public keys in 8.0 and 8.6 ms and generate the shared secret key in 7.1 and 7.9 ms for Alice and Bob, respectively. Finally, we show that this architecture could also be used to efficiently generate undeniable and digital signatures based on supersingular isogenies.

Index Terms—Elliptic curve cryptography, field-programmable gate array, isogeny-based cryptography, post-quantum cryptography

1 Introduction

It is widely accepted that much of today's public-key cryptosystems could be broken with the emergence of a large-scale quantum computer. Notably, RSA and elliptic curve cryptography (ECC), which are protected by the difficulty to factor extremely large integers and to perform elliptic curve discrete logarithms, respectively, will be effectively broken by a quantum computer utilizing Shor's algorithm [1]. Although it is unclear when such a quantum computer will be operational, NIST has taken the initiative to begin standardizing candidates for a post-quantum future [2]. Among the post-quantum cryptography (PQC) candidates, isogeny-based cryptography, based on the difficulty of computing isogenies between elliptic curve isomorphism classes, has been gaining momentum.

An isogeny between elliptic curves is a morphism between elliptic curves that preserves the point at infinity. The idea to use isogenies between elliptic curves as a cryptosystem was first published by Rostovtsev and Stolbunov in [3]. Originally defined as isogenies between ordinary elliptic curves, this cryptosystem was later broken by Childs, Jao, and Stolbunov [4] with a subexponential quantum attack. Jao and De Feo subsequently published an isogeny-based key exchange instead over *supersingular* elliptic curves, for which there is no known subexponential attack. This was introduced as the supersingular isogeny Diffie-Hellman (SIDH) key-exchange [5]. From there, various other important applications of isogeny-based cryptosystems have appeared in the literature, namely fast isogeny arithmetic [6], [7], undeniable signatures [8], strong designated verifier signatures [9], key compression [10], [11], static-static key agreement [12], digital signature schemes [13], [14], and efficient implementations [6], [7], [15], [16], [17], [18].

The case for isogeny-based cryptography as a quantum-resistant alternative for public-key cryptography has been so compelling because it features the smallest public and private keys among known quantum-resistant algorithms. As Table 1 shows, the SIDH key exchange over 128-bit quantum security level features a public key of 576 Bytes. With public key compression, the public key size is reduced further by almost a factor of two to only 336 Bytes. Small public key sizes reduce both the transmission cost over a wire as well as the storage requirement, which are extremely valuable as we begin the transition to a post-quantum secure world. SIDH features public and private keys that are only a fraction of other PQC public-encryption schemes. Furthermore, the SIDH algorithm features perfect forward secrecy, which means that compromising long-term keys does not compromise past session keys. Unfortunately, the primary

- *B. Koziel is with Texas Instruments, Dallas, TX 75243. E-mail: kozielbrian@gmail.com.*
- *R. Azarderakhsh is with the Computer and Electrical Engineering and Computer Science Department and I-SENSE, Florida Atlantic University, Boca Raton, FL 33431. E-mail: razarderakhsh@fau.edu.*
- *M. Mozaffari Kermani is with the Computer Science and Engineering Department, University of South Florida, Tampa, FL 33620. E-mail: mehran2@usf.edu.*

Manuscript received 28 May 2017; revised 23 Jan. 2018; accepted 26 Jan. 2018. Date of publication 12 Mar. 2018; date of current version 16 Oct. 2018. (Corresponding author: Brian Koziel.)
Recommended for acceptance by Ç. K. Koç, Z. Liu, and P. Longa.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2815605

TABLE 1
Comparison of Different Post-Quantum Key Exchange and Encryption Algorithms at 128-bit Quantum Security Level

Algorithm	NTRU [19]	New Hope [20]	McBits [21]	SIDH [7]	SIDH (with Compression) [11]
Type	Lattice	Ring-LWE	Code	Isogeny	Isogeny
Public Key	6,130	2,048	1,046,739	576	336
Private Key	6,743	2,048	10,992	48	48
Perfect Forward Secrecy	×	✓	×	✓	✓
Performance	Slow	Very Fast	Slow	Very Slow	Very Slow

Key sizes are in Bytes.

downside of isogeny-based cryptography is that it is currently a few orders of magnitude slower than other PQC candidates in both hardware and software.

Just as elliptic curve cryptography has improved by leaps and bounds through decades of research, we believe that isogeny-based cryptography will continue to improve in performance and security. To alleviate these performance concerns, this paper provides a fast and scalable architecture for SIDH and other isogeny-based cryptosystems at multiple security levels. On a Virtex-7 FPGA, our hardware architecture can simulate a full SIDH key-exchange (both party's perspectives) in a constant-time 31.6 ms at the 124-bit quantum security level. This work is a major extension of the work, "Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA", which was presented at Indocrypt 2016 [17]. This extension features a complete look at an efficient architecture for isogeny-based cryptography, with improved results, more security levels, and more analysis. Our contributions can be summarized as follows:

- We provide a constant-time SIDH field-programmable gate array (FPGA) implementation that is 2 times faster than an optimized software implementation on Haswell architectures and 1.34 times faster than the previous best FPGA implementation, both running similar projective isogeny formulas.
- We show that our SIDH core is scalable by implementing it at the 83, 124, 168, and 252-bit security levels.
- We illustrate how to achieve high parallelization in quadratic extension field arithmetic and isogeny evaluations for isogeny-based cryptosystems.
- We analyze the necessary requirements for deployment of our isogeny architecture.

2 Preliminaries

Here, we briefly discuss isogeny-based cryptography. As we review elliptic curve and isogeny theory, we point the reader to [22] for an in-depth explanation of elliptic curve theory. For this implementation, we most closely follow the algorithms from [7].

2.1 Isogeny Theory

An elliptic curve defined over a finite field $\mathbb{F}_q$ can be written in its Short Weierstrass form as

$$E_{(a,b)}/\mathbb{F}_q : y^2 = x^3 + ax + b,$$

where $a, b \in \mathbb{F}_q$. An elliptic curve is composed of all points (x, y) that satisfy the above equation as well as the point at infinity. This forms an abelian group over point addition, the underlying basis of the scalar point multiplication in elliptic curve Diffie-Hellman, $Q = kP$, where $P, Q \in E$ and k is a scalar. By using abstract geometry to define point addition and doubling formulas, one can efficiently perform a scalar point multiplication by performing a series of point doublings and additions. However, instead of performing affine point addition and affine point doubling for a scalar point multiplication, we define projective formulas over projective coordinates $(X : Y : Z)$ such that $x = X/Z$ and $y = Y/Z$. With this representation, only a single inversion is performed at the end of the scalar point multiplication.

We define an isogeny over a finite field $\mathbb{F}_q$, $\phi : E \rightarrow E'$ as a non-constant rational map defined over $\mathbb{F}_q$ such that ϕ satisfies group homomorphism from $E(\mathbb{F}_q)$ to $E'(\mathbb{F}_q)$ [22]. An isogeny can be thought of as a mapping from one elliptic curve class to another that preserves the point at infinity. Two curves are isogenous if an isogeny exists between them. Specifically, for two elliptic curves to be isogenous over a finite field, they must have the same number of points [23]. The degree of an isogeny is its degree as a rational map. For every prime, $\ell \neq p$, there exist $\ell + 1$ isogenies of degree ℓ from a specific isomorphism class. Unique isogenies can be computed over a kernel, κ, such that $\phi : E \rightarrow E/\langle\kappa\rangle$ by using Vélu's formulas [24].

The j-invariant is an identifier for an elliptic curve isomorphism class. The curves within this isomorphism class share various complex features. An isogeny is essentially a morphism from one isomorphism class to another. Thus, we can create a graph of all isogenies by using nodes that represent each isomorphism class and edges that represent an ℓ degree isogeny. When considering a specific ℓ, each node has $\ell + 1$ neighbors. When considering two distant nodes on a large graph, it is very difficult to determine a path from one node to the other. The difficulty to compute an isogeny between supersingular isomorphism classes forms the security basis for SIDH and several other isogeny-based cryptosystems. Supersingular elliptic curve isogeny classes are interesting in that their endomorphism ring is not commutative. The commutative endomorphism ring of ordinary elliptic curves was key to the subexponential quantum algorithm attacking ordinary isogeny cryptosystems in [4]. The best known attack to compute an isogeny between two isomorphism classes is based on the claw problem with complexity $O(\sqrt{s})$ for classical computers and $O(root3s)$ for quantum computers [5], where s is the degree of the isogeny. There is no known subexponential solution to this problem, even when quantum computers are available. However, it should also be noted that SIDH does reveal the image of a few torsion points between rounds,

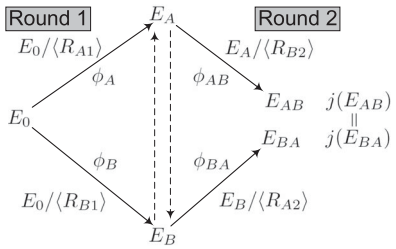

Fig. 1. Visualization of SIDH key exchange. Alice and Bob separately perform their secret isogeny walks on their respective isogeny graphs.

which may make the problem easier [25]. To this date, no faster attack on the Jao and De Feo SIDH scheme has been proposed.

2.2 Finite Field Arithmetic

For supersingular isogeny-based cryptosystems, we can represent all arithmetic over a quadratic extension field $\mathbb{F}_{p^2}$, where p is a prime number. A finite field $\mathbb{F}_p$ consists of p elements that are closed under addition and multiplication, which can be thought of as modular addition and multiplication. We can extend the base finite field $\mathbb{F}_p$ to a quadratic extension field by defining the extension field over an irreducible polynomial. Notably, we choose the irreducible polynomial x^2+1, which is irreducible since $i=\sqrt{-1}$ does not exist in our base field. Thus, we can represent an element $A \in \mathbb{F}_{p^2}$ as being composed of elements $a_0, a_1 \in \mathbb{F}_p$ in the form $A = a_0 + ia_1$. a_1 is considered to be the most significant $\mathbb{F}_p$ element in the quadratic extension field representation.

2.3 SIDH Key Exchange

In the supersingular isogeny Diffie-Hellman key exchange protocol, Alice and Bob seek to agree on a secret elliptic curve class by separately taking seemingly random walks on their respective isogeny graphs. SIDH is composed of a double-point multiplication to generate a secret kernel point followed by a large-degree isogeny over that kernel point. The difficulty to determine a connection between two distant supersingular isomorphism classes provides security for this protocol.

To initiate SIDH, Alice and Bob decide on a smooth isogeny prime p of the form $\ell_A^a \ell_B^b \cdot f \pm 1$ where ℓ_A and ℓ_B are small primes, a and b are positive integers, and f is a small cofactor to make the number prime. They determine a supersingular elliptic curve over a quadratic extension field, $E_0(\mathbb{F}_{p^2})$, with cardinality $(p \mp 1)^2$. Over this starting supersingular curve E_0, Alice and Bob pick the bases $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ which generate the torsion groups $E_0[\ell_A^{e_A}]$ and $E_0[\ell_B^{e_B}]$, respectively, such that $\langle P_A, Q_A\rangle = E_0[\ell_A^{e_A}]$ and $\langle P_B, Q_B\rangle = E_0[\ell_B^{e_B}]$.

The SIDH protocol proceeds as follows. Alice and Bob each perform a double-point multiplication with two selected private keys that span $\mathbb{Z}/\ell^a\mathbb{Z}$ and $\mathbb{Z}/\ell^b\mathbb{Z}$, respectively. This generates a secret kernel point that identifies a large-degree isogeny. Alice's secret kernel is $R_{A1} = m_A P_A + n_A P_A$ and Bob's secret kernel is $R_{B1} = m_B P_B + n_B P_B$, where $\{m_A, n_A\}$ are Alice's secret keys and $\{m_B, n_B\}$ are Bob's secret keys. Next, Alice and Bob use that secret kernel to perform their own secret isogeny walk by computing a large-degree isogeny. Alice computes $\phi_A : E_0 \rightarrow E_A = E_0/\langle R_{A1}\rangle$ and Bob computes $\phi_B : E_0 \rightarrow E_B = E_0/\langle R_{B1}\rangle$. For the first round, the opposite party's basis points are also pushed through the isogeny mapping. Alice computes $\phi_A(P_B), \phi_A(Q_B)$ with her hidden isogeny and Bob computes $\phi_B(P_A), \phi_B(Q_A)$ with his hidden isogeny. At the end of the first round, Alice transmits $\{E_A, \phi_A(P_B), \phi_A(Q_B)\}$ and Bob transmits $\{E_B, \phi_B(P_A), \phi_B(Q_A)\}$.

With the exchanged information, Alice and Bob again perform their hidden isogeny walk, but this time with the transmitted public keys as the starting point. Similar to the first round, Alice and Bob calculate a secret kernel with their secret key. Alice's secret kernel for the second round is $R_{A2} = m_A\phi_B(P_A) + n_A\phi_B(P_A)$ and Bob's secret kernel is $R_{B2} = m_B\phi_A(P_B) + n_B\phi_A(P_B)$. With this secret kernel, Alice and Bob compute a large-degree isogeny which is the same isogeny walk as the first round. Alice computes $\phi_{BA} : E_B \rightarrow E_{BA} = E_B/\langle R_{A2}\rangle$ and Bob computes $\phi_{AB} : E_A \rightarrow E_{AB} = E_A/\langle R_{B2}\rangle$. Essentially, curves E_{BA} and E_{AB} were obtained by performing the same two isogeny walks, but in a different order. In effect, Alice and Bob now share isomorphic curves with a common j-invariant that can be used as a shared secret. We illustrate the SIDH key exchange in terms of the isogeny computation in Fig. 1.

2.4 Projective Isogeny Formulas

The supersingular isogeny Diffie-Hellman protocol was first proposed by David Jao and Luca De Feo in [5] in 2011. Since then it has been interesting to see how further papers have improved the protocol. The two main papers that have improved the protocol are [6] by De Feo, Jao, and Plût and [7] by Costello, Longa, and Naehrig. Here, we highlight the main protocol optimizations that we adapt. As introduced in [6], we utilize points on Montgomery curves [26] and optimize arithmetic around them. We define a Montgomery curve, E, as the set of all points (x, y) that satisfy the Montgomery form

$$E_{(a,b)}: \; by^2 = x^3 + ax^2 + x,$$

and a point at infinity. When the value $a_{24} = (a+2)/4$ is known, these curves feature extremely fast point arithmetic along their Kummer line, $(x, y) \rightarrow (X : Z)$, where $x = X/Z$. Isogenies still work for this representation because P and $-P$ generate the same set subgroup of points. Interestingly, we are performing isogenies along the Kummer varieties of Montgomery curves. This reduces the total number of computations as the y-coordinate does not need to be updated for point arithmetic or when the point is pushed to a new curve by evaluating an isogeny.

Projective isogeny formulas over Montgomery curves were introduced in [7]. These formulas projectivize the curve equation with a numerator and denominator, similar to projective point arithmetic. We define a projective Montgomery curve, $\hat{E}$, as the set of all points (x, y) that satisfy the projectivized Montgomery form

$$\hat{E}_{(A,B,C)}: \; By^2 = Cx^3 + Ax^2 + Cx,$$

and a point at infinity. In this representation, the corresponding affine Montgomery curve would have coefficients $a = A/C$ and $b = B/C$. The authors of [7] also show that the b curve coefficient is not needed in any of the computations for SIDH and can omit recalculating it after each isogeny computation.

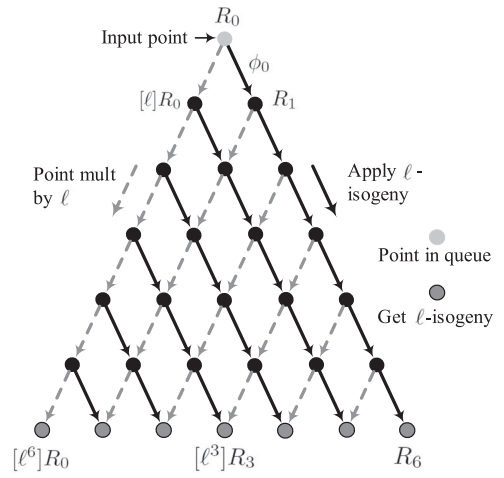

Fig. 2. Acyclic graph structure for performing isogeny computation of ℓ^6.

To perform a double point multiplication, we specify that one of Alice and Bob's secret keys is 1, as introduced in [6], so that the double point multiplication produces a secret kernel, $R = \langle P + mQ \rangle$. Costello et al. [7] also greatly simplified the starting parameters for SIDH by proposing to use the starting Montgomery curve

$$E_0/\mathbb{F}_{p^2} : \ y^2 = x^3 + x.$$

By specifying points in the base field and trace-zero torsion subgroup, the first round of the SIDH protocol can be performed as a Montgomery [26] ladder followed by a point addition, with all operations in $\mathbb{F}_p$. The second round of the protocol involves a double-point multiplication with elements in $\mathbb{F}_{p^2}$. For this calculation, we utilize the 3-point Montgomery ladder proposed in [6] that computes $P + mQ$ in $\log_2(m)$ steps. Each step requires 2 point additions and 1 point doubling. We closely follow the projective isogeny formulas presented in [7] for isogenies of degree $\ell_A = 4$ and $\ell_B = 3$. For the first round, we push the Kummer coordinates of the other party's basis P, Q, and $Q - P$ through the large-degree isogeny rather than the projective version of P and Q to remove a point subtraction before the 3-point ladder.

As proposed by [27], large-degree isogenies can be decomposed into a chain of smaller degree isogeny computations and computed iteratively. From a base curve E_0 and kernel point $R_0 = R$ of order ℓ^e, we compute a chain of ℓ-degree isogenies

$$E_{i+1} = E_i/\langle \ell^{e-i-1} R_i \rangle, \ \phi_i : E_i \rightarrow E_{i+1}, \ R = \phi_i(R_i).$$

This problem can be visualized as an acylic graph, which is shown in Fig. 2. We can traverse this graph by storing multiple pivot points to efficiently compute the large-degree isogeny. De Feo et al. [6] show that we can calculate an optimal strategy, or path of least cost, by solving a combinatorics problem.

Lastly, Costello et al. [7] show that we can perform a simple compression of the SIDH keys by exchanging the image of the opposite party's basis $[x_P, x_Q, x_{Q-P}]$. This gives sufficient information to recalculate the Montgomery curve coefficient a and only costs 3 $\mathbb{F}_{p^2}$ elements to transmit over the wire. For instance, Alice's public key would be $\{x_{\phi_A(P_B)}, x_{\phi_A(Q_B)}, x_{\phi_A(Q_B - P_B)}\}$.

2.5 SIDH Public Parameters

Table 2 contains the chosen public parameters for our SIDH implementations. These were selected to be a representative for parameters close to the 85, 128, 170, and 256-bit quantum security levels. Their actual quantum security is just smaller at 83, 124, 168, and 252-bit quantum security levels, respectively. Each of these primes have an even power for $\ell_A = 2$ so that the fast isogeny formulas of degree 4 could be fully used. Prime p_{503} was selected since it was utilized in [16], [17] and prime p_{751} was selected since it was utilized in [7]. Primes p_{1019} and p_{1533} were selected to be slightly smaller than 1,024 bits and 1,536 bits and also have fairly balanced isogeny graphs. Unfortunately, the candidates for the 1,024-bit quantum security level were fairly limited, and thus, we chose a prime with an f term that is not 1.

We utilize the method proposed by Costello et al. [7] to find generator points for the torsion subgroups $\ell_A^{e_A}$ and $\ell_B^{e_B}$. In this method, we start with the Montgomery curve, $E_0/\mathbb{F}_{p^2} : \ y^2 = x^3 + x$. For the $\ell_A^{e_A}$-torsion points P_A and Q_A, we find a point $P_A \in E_0(\mathbb{F}_p)[\ell_A^{e_A}]$ as $[f\ell_B^{e_B}](z, \sqrt{z^3 + z})$, where z is the smallest positive integer such that $\sqrt{z^3 + z} \in \mathbb{F}_p$ and P_A has order $\ell_A^{e_A}$. We apply a distortion map over E_0 to P_A to find Q_A such that it is the endomorphism $\tau : \ E_0(\mathbb{F}_{p^2}) \rightarrow E_0(\mathbb{F}_{p^2}), (x + 0i, y + 0i) \rightarrow (-x + 0i, 0 + iy)$. Therefore, $Q_A = \tau(P_A)$. We repeat this process to find the $\ell_B^{e_B}$-torsion points. We find $P_B \in E_0(\mathbb{F}_p)[\ell_B^{e_B}]$ as $[f\ell_A^{e_A}](z, \sqrt{z^3 + z})$, where z is the smallest positive integer such that $\sqrt{z^3 + z} \in \mathbb{F}_p$ and P_B has order $\ell_B^{e_B}$. Lastly, $Q_B = \tau(P_B)$. We also list the generator points P_A and P_B in Table 2.

3 Proposed Finite Field Arithmetic Unit for Isogeny Computations

In this section, we investigate an efficient field arithmetic unit for isogeny computations. SIDH features a large

TABLE 2
SIDH Public Parameters

Curve: $E_0/\mathbb{F}_{p^2} : \ y^2 = x^3 + x$				
Prime	Classical/Quantum Security (bits)	Public Key Size (Bytes)	P_A	P_B
$p_{503} = 2^{250}3^{159} - 1$	125/83	378	$[3^{159}](14, \sqrt{14^3 + 14})$	$[2^{250}](6, \sqrt{6^3 + 6})$
$p_{751} = 2^{372}3^{239} - 1$	186/124	564	$[3^{239}](11, \sqrt{11^3 + 11})$	$[2^{372}](6, \sqrt{6^3 + 6})$
$p_{1019} = 2^{508}3^{319}35 - 1$	253/168	765	$[3^{319}35](13, \sqrt{13^3 + 13})$	$[2^{508}35](7, \sqrt{7^3 + 7})$
$p_{1533} = 2^{776}3^{477} - 1$	378/252	1,150	$[3^{477}](5, \sqrt{5^3 + 5})$	$[2^{776}](6, \sqrt{6^3 + 6})$

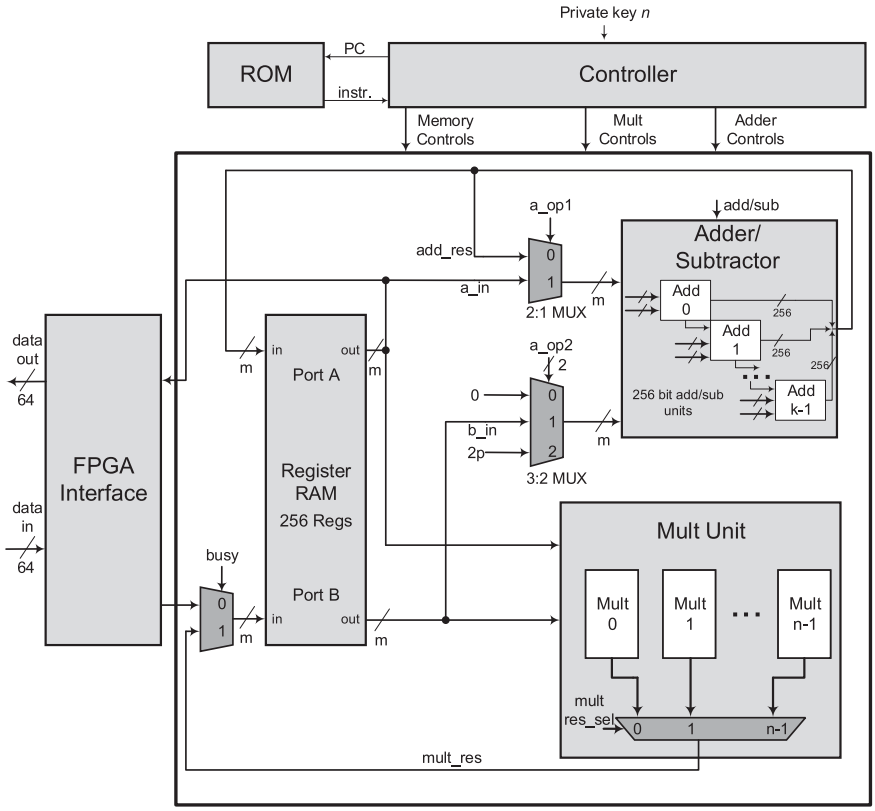

Fig. 3. High-level architecture of our SIDH core. This architecture is optimized for performance, configurable with the number of replicated multipliers, and scalable for isogeny-based computations at varying security levels.

amount of finite field arithmetic, which is primarily based on modular addition and multiplication.

3.1 Field Arithmetic Unit Methodology

Our field arithmetic unit centers around a dual-port RAM block for 256 values in $\mathbb{F}_p$, which is shown in Fig. 3. In addition to RAM, we have highly optimized adder/subtractor and multiplication accelerators. We include so many registers so that we can parallelize as much of the SIDH formulas as possible, which is illustrated in Section 4. In using the dual-port RAM, we can read two registers simultaneously to initiate a modular addition or modular multiplication immediately. Further, we opted to directly store the output of the adder/subtractor to Port A and the output of the multiplication unit to Port B.

As is described later, we utilized Montgomery multiplication [28] for our multiplier. This implies that all results are in the Montgomery domain, and that the result of a Montgomery multiplication could be c or $c + p$, since the result is modulo $2p$. Thus, we assume that all values are modulo $2p$ inside the register RAM. In other designs, such as [16], [17], the result of the multiplier is sent into the adder/subtractor so that it can be reduced. Thus, when compared to these other implementations, this design decision allows us to reduce the multiplication latency by the add delay, allows us to store up to two values to the register file simultaneously, and was found to also slightly increase the max frequency of the device.

The RAM file contains 256 values in $\mathbb{F}_p$, which allows for up to 128 values in $\mathbb{F}_{p^2}$. The beginning of the register file holds 0, 1, 2, and 6 in the Montgomery domain as well as the Montgomery constant R^2. This is followed by the SIDH parameters. The rest of the registers are for intermediate values throughout the protocol. We place $\mathbb{F}_{p^2}$ values starting at an even register position by placing the most significant $\mathbb{F}_p$ element at the even position and the least significant $\mathbb{F}_p$ element at the odd position.

3.2 Finite Field Adder

Finite field addition computes the sum $C = A + B$, where $A, B, C \in \mathbb{F}_p$. If the sum C is greater than p, then there is a reduction by performing the subtraction $C = C - p$ so that $C \in \mathbb{F}_p$. A similar situation occurs for finite field subtraction, $C = A - B$, where $A, B, C \in \mathbb{F}_p$. However, as noted, we represent all numbers modulo $2p$. Therefore, we will subtract by $2p$ in the case of addition and add by $2p$ in the case of subtraction to ensure that the result is modulo $2p$.

Since SIDH uses very large inputs, we split the adder/subtractor into approximately 256-bit add/sub stages. We chose 256 since it is a nice division of the prime sizes for approximate security levels and also features a critical path delay approximately equal to that of our multiplier (described in Section 3.3). Each cycle we can compute a different 256-bit chunk of the result by adding or subtracting the aligned bits with the carry/borrow of the previous operation. We fully pipeline this cascaded addition chain so that the adder/subtractor can compute on new inputs every cycle. Operand 1 can be the feedback of the previous addition result or Port A out. Operand 2 can be 0, Port B out, or $2p$. The controller selects the inputs. The possible inputs to the adder/subtractor are shown in Fig. 3.

Every modular addition and subtraction operation performs both $A \pm B$ and $A \pm B \mp 2p$ and takes the appropriate

TABLE 3
Time Complexity Comparison of Montgomery Multipliers

Work	Critical Path Delay	Latency (cc)	
		Mult	Interleave
512-bit Montgomery Multiplication			
[29] ($d = 4$)	$5T_{FA} + T_{AND}$	130	130
[30] ($k = 16$)	$2T_{16\times} + T_{32+}$	192	192
[31]($s = 16$)	$2T_{32\times} + T_{64+}$	66	66
[16] ($k = 16$)	$T_{16\times} + T_{16+} + T_{32+}$	100	69
1,024-bit Montgomery Multiplication			
[29] ($d = 4$)	$5T_{FA} + T_{AND}$	258	258
[30] ($k = 16$)	$2T_{16\times} + T_{32+}$	384	384
[31]($s = 32$)	$2T_{64\times} + T_{128+}$	66	66
[16] ($k = 16$)	$T_{16\times} + T_{16+} + T_{32+}$	196	133

Note that $T_{32\times}$ indicates the critical path of a 32-bit multiplication and T_{16+} indicates the critical path of a 16-bit addition.

result modulo $2p$ so that it is a constant set of operations. To efficiently check that the modulo $2p$ condition is enforced, we always check to see if the results $A + B - 2p$ and $A - B$ are negative for addition and subtraction, respectively.

3.3 Field Multiplier

Finite field multiplication is essentially a modular multiplication. Multiplying two elements in a finite field $\mathbb{F}_p$ will produces an output double the size of the inputs, so finite field multiplication is a combination of a standard multiplication with a reduction. It is well known that when optimizing for high-performance hardware architectures, the conventional wisdom is to perform multiple partial multiplications in parallel. For instance, although [7] and [32] present methods to reduce the number of partial multiplications when computing SIDH arithmetic, this method is not altogether useful for hardware architectures that would perform a number of these partial multiplications in parallel anyways.

Specifically, we examined the Montgomery [28] multiplication literature to find a strong multiplier for our architecture. Montgomery multiplication performs a modular multiplication by transforming integers to m-residues, or the Montgomery domain, and performing multiplications with this representation. Thus, for SIDH, we will initially transform our input parameters to the Montgomery domain and use Montgomery multiplication throughout the protocol. At the end of computations, the result can be converted out of the Montgomery domain with a single Montgomery multiplication.

In Table 3, we provide a time complexity comparison of several Montgomery multiplication schemes over the 512 and 1,024-bit modulus levels. Some of these are high-performance multipliers that also have been designed for fast RSA computations. Our target is a high-performance Montgomery multiplier, which is a balance between critical path for the frequency and multiplication/interleave latency. We chose to not include the provided hardware results of these works because there are too many differing implementation specifics that make comparisons unfair.

Ultimately, we chose the interleaved systolic Montgomery multiplier proposed in [16] as it is high-performance, high-throughput, and scalable, which is a strong fit for a high-performance implementation of isogeny-based cryptography. This multiplier utilizes the high-radix Montgomery multiplication procedure. This Montgomery multiplication, $S_{m+3} = A \times B \times R^{-1} \bmod M$ centers on computing $m + 2$ rounds of the computations

$$q_i = (S_i) \bmod 2^k \tag{1}$$

$$S_{i+1} = (S_i + q_i\overline{M})/2^k + a_iB, \tag{2}$$

where k is the high radix, $R = 2^{km}$ is the auxiliary modulus, M is the target modulus, m is the total number of k-bit chunks in the modulus, A and B are inputs with a_i being the ith k-bit chunk of A, $\overline{M} = ((-M^{-1} \bmod R) \bmod 2^k)M$, and $S_0 = 0$ [33]. Essentially, we can perform the large multiplications $q_i\overline{M}$ and a_iB by computing them in k-bit parallel chunks.

Notably, we can use a systolic architecture to perform the iterative computations in Equations (1) and (2). This is built on a 1-dimensional systolic array of $m + 2$ processing elements that each compute $S_{i+1,j} = (S_i + q_i\overline{m_j})/2^k + a_ib_j$, where j is the number of the processing element in the array. In this sense, we push the k-bit chunks of A from processing element 0 to processing element $m + 2$ and also perform a feedback for $S_i + q_i\overline{M}$. The multiplication sequence begins by pushing a 0 through the systolic array by triggering a synchronous reset within each processing element. Then, the first k-bit chunk of A, a_0, is pushed into the first processing element where it performs the simpler computation, $S_{1,0} = (S_0 + q_0\overline{M})/2^k + a_0b_0 = a_0b_0$. This corresponds to the first k-bits of the S_1. a_0 continues through the entire systolic array to produce all of S_1. This only occupies a single processing element at a time, so we also push other chunks of A through the array, such that the processing elements are working in parallel. After $m + 3$ cycles, the least significant k-bit word of the result is ready. The last word is ready after $3m + 7$ cycles. When a single multiplication is issued, only half of the processing elements are used on a specific cycle. Thus, we can issue two multiplications simultaneously, at the cost of multiplexers on the input and output to select between the "even" and "odd" multiplication.

For our Montgomery multiplier, we chose the radix 2^{16} as it approximately matched the addition critical path and a 16×16 bit multiplication could be computed with a single DSP48 block. The critical path of the Montgomery multiplier was the basic processing equation, $S_{i+1,j} = (S_i + q_i\overline{m_j})/2^k + a_ib_j$. With the radix 2^{16}, we compute two 16-bit multiplications $q_i\overline{m_j}$ and a_ib_j in parallel, which is then accumulated in a 4-way adder that added two 32-bit values and two 16-bit values. The adder summed $q_i\overline{m_j}$, a_ib_j, S_i, and a carry. Regardless of the prime size, this will be the critical path of the multiplier, hence the scalability.

The design in [16] features an interleaved version of [34]. As k-bit chunks of A are pushed through the systolic architecture, the earlier processing elements are no longer processing inputs. Thus, we can interleave multiplications every $2m + 3$ cycles by pushing in a new set of A and B. It is also known that for SIDH primes of the form $2^{e_a}\ell_b^{e_b}f - 1$, $\bar{M} = M$ since $M' = 1$. This is applicable to our test primes. This simplification removes the need for one processing element and reduces the multiplication latency by 3 cycles and

TABLE 4
Scheduling Parameters

Prime	Latency (cc)				
	Read	Write	Add	Multiplication	
				Mult.	Interleaved
p_{503}	2	1	2	100	69
p_{751}			3	148	101
p_{1019}			4	196	133
p_{1533}			6	292	197
p_m			$\lceil \frac{m}{256} \rceil$	$3\lceil \frac{m+2}{16} \rceil + 4$	$2\lceil \frac{m+2}{16} \rceil + 5$

interleave latency by 2 cycles. Unfortunately, the only caveat to using the modulo $2p$ methodology is that there will be overflow for p_{751} since the input operands can be 752 bits. Thus, p_{751} actually performs a 768-bit Montgomery multiplication for this field arithmetic unit, which raises the multiplication latency by 3 cycles and the interleave delay by 2 cycles.

Similar to the design in [16], we replicate the Montgomery multiplier depending on how many resources we require. Since the multiplication latency is much higher than that of the addition latency and the multiplier is not fully pipelined, we found this to be necessary to greatly increase the speed of isogeny-based arithmetic. We issue and read the multiplication results by using a circular FIFO buffer. It is the job of the program ROM within the control unit to keep track of which multiplications are ready and read them when appropriate, which is explained in Section 4.1. Each time a multiplication is started, the issue index is incremented and each time a multiplication result is read, i.e., through a write to Port B, the read index is incremented.

4 Parallelizing SIDH with New Architectures

This section details our techniques to maximize the throughput of our architecture throughout the SIDH protocol.

4.1 Scheduling

In this work, we examined various methods to efficiently schedule isogeny-based algorithms with our high throughput architecture. Ultimately, we generate a constant program ROM using a greedy algorithm to schedule the multipliers, adder, and register file for critical computations first and effectively interleave other, less important computations when resources are available.

Since there is a variety of operations that are used throughout SIDH, we opted to divide each function, such as a Montgomery ladder step, compute an isogeny of degree 3, etc., into an instruction block or subroutine. To generate these blocks of code, we created a custom assembly language which takes in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ addition, subtraction, multiplication, and squaring operations, and schedules a fast implementation of the assembly code using the allocated number of resources and appropriate delays. The latency for a read, write, add, multiply, and multiplication interleave are shown in Table 4 for a given prime.

For our isogeny computations, we most closely use the projective isogeny and Montgomery arithmetic formulas presented in [7]. For all subroutines, we rearranged the order of the operations and included register renaming to minimize data and read dependencies. To schedule instructions, we dynamically create a data dependency and read dependency graph to issue instructions as soon as possible. We schedule the instructions from our assembly code in order, such that the first instructions have priority over the multiplier unit, adder, and register file. These first instructions are recorded into the current program ROM snapshot and used to create further dependency graphs. Data dependencies ensure that read after write dependencies are not violated, i.e., trying to read and use a result before it is ready. Read dependencies ensure that write after read dependencies are not violated, i.e., writing a result to a register that has not been read by all of its consumers. Algorithm 1 illustrates our procedure. We note that the order of our assembly code is essential in that the first instructions have the highest priority to be "placed", where we lock the hardware resources for these instructions to ensure that the instruction's output is ready for whichever future instructions might need it as a data dependency. Further instructions can also be interleaved with these critical computations as long as the necessary hardware resources are available.

Algorithm 1. Proposed Scheduling Methodology

Input: SIDH Instruction File
Output: FPGA ROM File
for each instruction block **do**
  for each $\mathbb{F}_p$ or $\mathbb{F}_{p^2}$ operation **do**
    current_cycle = $\max(\text{data\_dependency(opa,opb)}, \text{read\_dependency(opout)})$
    while instruction can't be issued **do**
      current_cycle += 1
    end while
    Insert instruction sequence
    Update data and read dependencies
  end for
  commit_cycle = 0
  while commit_cycle < last_cycle **do**
    if multiplier_mismatch **then**
      reschedule_mismatched_multiplier()
    end if
    commit_cycle += 1
  end while
end for
return FPGA ROM file

In addition to the dependencies, the resources to carry out an operation have to also be available. For an $\mathbb{F}_p$ addition, we engage a memory load, addition, addition reduction, and memory store. Thus, for a given cycle t, we ensure that the memory unit is available for a read at cycle t, addition unit is available at cycles $t + \text{read\_delay}$ and $t + \text{read\_delay} + \text{add\_delay}$, and finally that the memory unit can store to Port A at cycle $t + \text{read\_delay} + 2 \times \text{add\_delay}$. From the compiler's point of view, the result would be ready at cycle $t + \text{read\_delay} + 2 \times \text{add\_delay} + \text{write\_delay}$. Similarly, an $\mathbb{F}_p$ multiplication requires that the memory at cycle t can be used as a load, the multiplier at cycle $t + \text{read\_delay}$ can issue a multiplication, and the memory at cycle $t + \text{read\_delay} + \text{mult\_delay}$ can store to Port B.

TABLE 5
Quadratic Extension Field Arithmetic Unrolling When $i = \sqrt{-1}$

$\mathbb{F}_{p^2}$ operation	Addition	Subtraction	Squaring	Multiplication	Inversion
Step 1	$c_0 = a_0 + b_0$	$c_0 = a_0 - b_0$	$t_0 = a_0 \times a_1$	$t_0 = a_0 + a_1$	$t_0 = a_0 \times a_0$
Step 2	$c_1 = a_1 + b_1$	$c_1 = a_1 - b_1$	$t_1 = a_0 + a_1$	$t_1 = b_0 - b_1$	$t_1 = a_1 \times a_1$
Step 3	-	-	$t_2 = a_0 - a_1$	$t_2 = a_1 \times b_0$	$t_0 = t_0 + t_1$
Step 4	-	-	$c_0 = t_1 \times t_2$	$t_3 = a_0 \times b_1$	$t_0 = t_0^{-1}$
Step 5	-	-	$c_1 = t_0 + t_0$	$t_0 = t_0 \times t_1$	$t_1 = 0 - a_1$
Step 6	-	-	-	$c_1 = t_2 + t_3$	$c_0 = a_0 \times t_0$
Step 7	-	-	-	$t_1 = t_0 - t_2$	$c_1 = t_1 \times t_0$
Step 8	-	-	-	$c_0 = t_1 - t_3$	-
Cost in $\mathbb{F}_p$	$2A$	$2A$	$2M + 3A$	$3M + 5A$	$1I + 2M + 2S + 2A$

Inputs are $A = a_0 + ia_1$ and $B = b_0 + ib_1$ and the output is $C = c_0 + ic_1$. Temporary registers are t_0 through t_3.

We perform all scheduling outside of our hardware with the aid of a Python script. This script outputs the controls for *every* cycle for our register file, adder, and multiplier. Our controller is composed of a program ROM and program counter to issue the correct operations to the field arithmetic unit. Each instruction is 26 bits long. Bits 0-7 and 8-15 are the addresses sent to Port A and B, respectively, for read or write operations. Bits 16 and 17 indicate a write to Port A and Port B, respectively. Bit 18 indicates a read of Port A and Port B. Bits 19-21 indicate which add operation is taken. The possible addition sequences are memory add, memory sub, reduction add, reduction sub, or do nothing. Bits 22-23 indicate the multiplier operation, which includes starting a multiplication, resetting the circular FIFO buffer, or doing nothing. Bit 24 is to indicate that the address for Port A is special, which is used to pop the last point in the point queue in the isogeny evaluation sequence. Lastly, if bits 0-24 are all 0 for more than 1 cycle, then bit 25 indicates that there is a stall, with the number of stalled cycles represented with bits 0-24. Stalls are typically encountered when the processor is waiting on a result from a multiplier or adder and no new controls need to pushed.

4.2 Rescheduling Multiplier Mismatches

Since we used an even-odd multiplier, we scheduled our instructions with a greedy algorithm that incurs stalls if a multiplication is not on the right even-odd cycle. We force each instruction block to start on an even cycle by performing an even-odd bit reset on the last cycle of a block (which is a memory store). In our scheduling methodology, we schedule all instructions and then iteratively commit each cycle to verify that our multipliers are working on the correct cycle. We must issue the first multiplication of an instruction block on an even cycle, followed by an odd cycle, followed by an even cycle, and so on. We commit each cycle in order to verify this condition. If there is a multiplier mismatch, i.e., trying to issue an odd multiplier on an even cycle, then we must resolve this violation.

The first two rules for rescheduling mismatched multiply cycles is that any stores and multiplications before the erring multiplication are unaffected so that dependencies, arithmetic pipelines, and previous multiplications are not violating a scheduling rule. Next, we push the erring multiplication cycle one cycle forward so that it is now on the correct even-odd cycle. If there are any store sequences that coincide with the new load cycle for this stalled multiplication, then we push these stores forward as well. From there, we redo the scheduling process starting from the erring cycle. By iterating cycle by cycle through the initial instructions, we again keep track of the available resources and dynamically create a read and write dependency graph to fit the instructions into the new schedule with a corrected even-odd multiplication.

Overall, we found that completely rescheduling instructions after a mismatched multiplication greatly reduced the stalls encountered as a result of using the even-odd multiplier. For a few replicated multipliers, the majority of new schedules did not increase the total number of cycles in an instruction block.

4.3 Extension Field Arithmetic

As was previously stated, SIDH operates in the extension field $\mathbb{F}_{p^2}$. For this extension field, we use the irreducible polynomial $x^2 + 1$, applicable to SIDH primes of the form $2^{e_A}\ell_B^{e_B}f - 1$. By combining fast $\mathbb{F}_p$ addition and multiplication primitives, we can efficiently schedule these $\mathbb{F}_{p^2}$ computations. The $\mathbb{F}_{p^2}$ equations were chosen to minimize the total number of $\mathbb{F}_p$ multiplications. Let $i = \sqrt{-1}$ be the most significant $\mathbb{F}_p$ element in $\mathbb{F}_{p^2}$. Let $A, B, C \in \mathbb{F}_{p^2}$ and $a_0, b_0, a_1, b_1, c_0, c_1 \in \mathbb{F}_p$, where $A = a_0 + ia_1$, $B = b_0 + ib_1$ and the result of some $\star$ operation is $C = A \star B$ where $C = c_0 + ic_1$. We define the extension field arithmetic $\mathbb{F}_{p^2}$ in terms of $\mathbb{F}_p$ with the series of operations shown in Table 5. Based on these representations, parallel calculations could easily be performed for a single operation in $\mathbb{F}_{p^2}$. For instance, three separate multiplications in $\mathbb{F}_p$ could be carried out simultaneously for the calculation of a multiplication in $\mathbb{F}_{p^2}$. With other non-dependent instructions in the scheduling, many multipliers can be used in parallel.

Unfortunately, an inversion in $\mathbb{F}_p$ was difficult to parallelize, and suffered as a result. We performed an exponentiation with Fermat's little theorem by utilizing a k-ary method with $k = 4$. We were able to parallelize the generation of the windows $1, 2, 3, \ldots, 2^k - 1$, but after that, the inversion was done serially. k squarings were done in serial followed by a multiplication. The inversion added many lines to the program ROM, and was difficult to parallelize, showing that there may still be some merit to having a dedicated inversion unit. One could also perform the window method for exponentiation proposed in [35], but again serial window methods are not parallel in nature.

4.4 Scheduling Isogeny Computations and Evaluations

As is briefly discussed in Section 2.4, a large-degree isogeny is efficiently computed by iteratively computing the base degree

isogeny with specific points derived from the secret kernel point. As is shown in Fig. 2, this can be visualized as an acycle graph in the shape of a triangle. By beginning at the secret kernel point at the top, we compute the large-degree isogeny by computing the base degree isogeny at the leaves of the graph. We traverse the graph by moving left with a point multiplication by ℓ or moving right with an ℓ-degree isogeny evaluation. Since we only need to compute an isogeny at each of the leaves, we can determine an optimal strategy or least cost traversal of the graph based on the relative weight of a point multiplication by ℓ and an ℓ-degree isogeny evaluation. This combinatorial problem was studied in [6]. The primary takeaway is that an optimal strategy is composed of two optimal sub-strategies. Therefore, we can recursively determine the least cost traversal from the leaves up to the head of the graph. We calculated fast strategies for traversing this graph with the Magma code provided by [7], which closely follows the algorithms proposed by [6].

By utilizing multiple pivot points, we can avoid traversing every node. This is opposed to the multiplication or isogeny-based traversals originally proposed in [5]. The time complexity of an optimal strategy is approximately $O(e \log e)$, which is quite an improvement over $O(e^2)$ for hitting every point.

The general procedure to compute a large-degree isogeny is composed of performing serial point multiplications on a specified point to reach a leaf node and then performing an isogeny at that leaf node. While performing the serial point multiplications, multiple pivots are stored to a point queue. Following the isogeny computation, each of these saved pivots are pushed through the isogeny mapping to a new curve. In software implementations such as [6], [7], [15], these isogeny evaluations are computed serially. As a contrast, we emphasize that our hardware architecture can compute each of these isogeny evaluations in parallel, as there are no data dependencies between pivot points. Essentially, this can be thought of as loop unrolling the isogeny evaluation step of the large-degree isogeny computation. By performing multiple isogeny evaluations in a single instruction block, we can take advantage of instruction reordering and a massive amount of parallelism with the pipelined adder unit and replicated multipliers.

To demonstrate the speedup achieved by this approach, we consider an implementation with 4 replicated multipliers in our architecture under the public key parameters for p_{752}. For Bob's first round of the protocol, he could compute 1941 serial 3-isogeny evaluations, or he could compute 6 single, 15 double, 274 triple, ... which in array form is [6, 15, 274, 35, 56, 28, 36, 10, 10, 5, 1, 1] loop unrolled 3-isogeny evaluations. For our scheduling algorithm, the serial version of the isogeny evaluation stage requires 1.6 million cycles, whereas the parallel version requires 0.685 million cycles. This is a speedup of approximately 2.34. In addition, the isogeny evaluation step already requires a large portion of the total time in SIDH. By parallelizing the isogeny evaluations, we reduced the total time of Bob's first round from 2.9 million cycles to 1.9 million cycles for this example, a speed improvement of 1.53. The only downside to including each of these parallel routines is that the program ROM must include many more routines, costing more blocks of memory. However, we still only use a small proportion of the available block memories so this was not much of a problem.

To implement the large-degree isogeny computation, we held the selected strategy for our implementation in a block ROM. For our largest prime, this was 2,048 8-bit entries indicating which pivot points to take. Since we have a finite number of registers, we chose to hold a maximum of 12 points in the point queue with up to 6 evaluations being parallelized at a time. This occupied a total of 96 registers in our register file. With these constraints in mind, we generated fast strategies for Alice and Bob with a point multiplication cost to isogeny evaluation cost of 2:1, the only exception being p_{1533} which was utilized with a ratio 1.9:1 so that a maximum of 12 points were in the point queue. Our isogeny control unit kept track of the current isogeny, point multiplication index of each of the points in the point queue, and loop conditions for the isogenies. To add a point to the point queue, we utilized a special queue_size variable that was incremented on an isogeny split. The split was facilitated by using a special control bit to point the address of Port A to the address of the last point in the point queue so that it could be appended. Overall, the control logic used to implement the isogeny logic was minimal compared to the size of the field arithmetic unit.

4.5 Scheduling Example

In Fig. 4, we illustrate the data-dependency graph of the Get 3 Isogeny function, which is based on the same formula from [7]. This function takes a point of order 3, $P = (X_3, Z_3)$ and outputs an isogenous curve with projective Montgomery coefficients $(A, C) = (Z_3^4 + 18Z_3^2X_3^2 - 27X_3^4, 4X_3Z_3^3)$.

Since we use a dual-port RAM as our register file, we can only issue 1 load each cycle to initiate an $\mathbb{F}_p$ operation. Thus, as Fig. 4 illustrates, we break our instructions into "steps", or the order as it appears in our assembly code. We can issue these instructions once the data and read dependencies are resolved and the hardware resources are available. We can generally ignore read dependencies since we chose a large register file and have plenty of temporary registers to work with. We specifically ordered our assembly code so that we calculate the datapath critical computations first to allow more parallelism.

Although Fig. 4 shows a linear progression of instructions, the instructions can be initiated and completed out of order. For instance, consider a system with only two available multipliers. The squaring in Step 1 requires both multipliers. Thus, only the initial additions and subtractions in the squaring and multiplication in Step 2 and 3, respectively, can be issued until these multipliers are free. Upon the completion of Step 1's interleave delay, the two multiplications from Step 2 will be issued. As Step 2 is processing these multiplications, the addition unit is free and is utilized by Step 4 to perform the $\mathbb{F}_{p^2}$ addition. Thus, the timing cost of Step 4 is interleaved with the cost of Step 2.

We highlight the two blocks of multipliers in Fig. 4 to highlight the critical computations that limit the computation time of computing the 3-isogeny computation. Notably, there are 7 total multiplications in Mult Block 1 and 8 total multiplications in Mult Block 2. The number of multipliers is the primary configurability in our architecture to achieve a desired performance. If we have 8 total multiplier resources, then the minimum number of clock cycles is achieved for the 3-isogeny computation. This is illustrated later in our summary of subroutine costs in Table 6 of Section 4.6 which shows a consistent

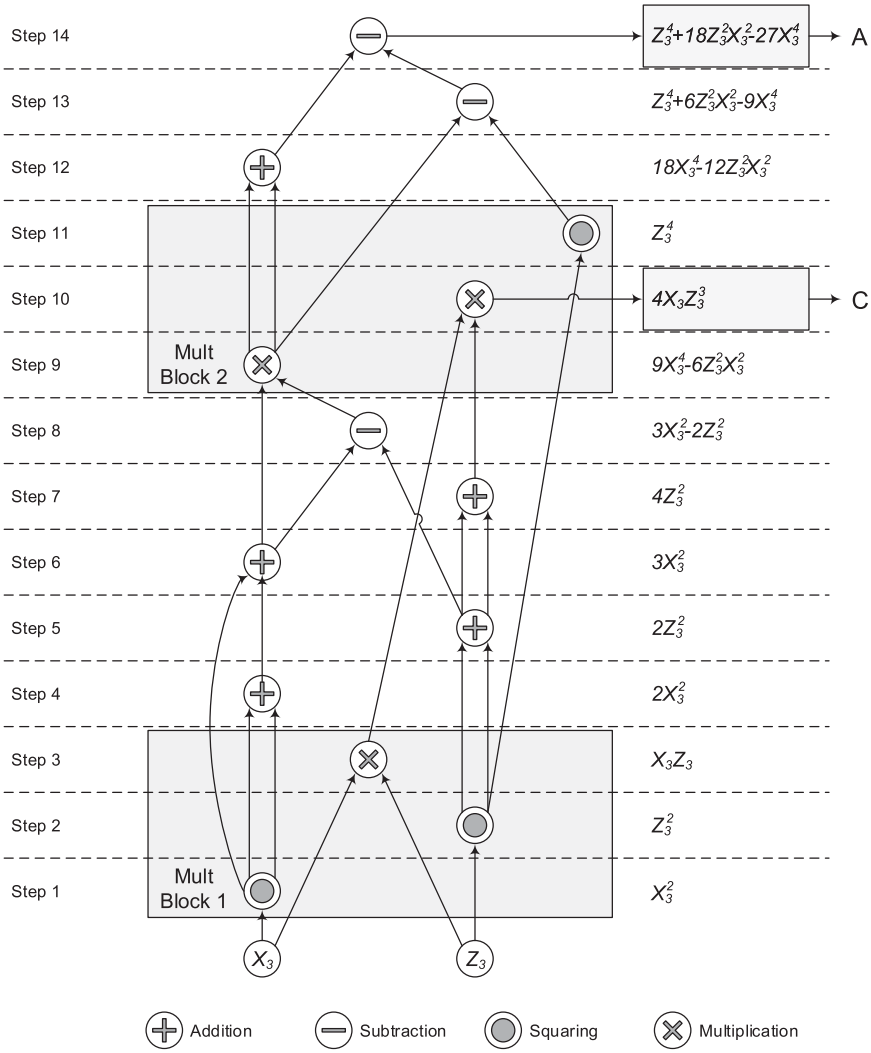

Fig. 4. Data-dependency graph for computing an isogeny of degree 3. This function takes a point of order 3, $P = (X_3, Z_3)$ and outputs an isogenous curve with projective Montgomery coefficients $(A, C) = (Z_3^4 + 18Z_3^2X_3^2 - 27X_3^4, 4X_3Z_3^3)$. Depending on data dependencies, read dependencies, and available resources, the greedy scheduler will attempt to schedule these instructions one-by-one at the first available opportunity, perhaps in an out-of-order but correct methodology.

424 cycles per "Get 3 Isog" subroutine for 8, 10, and 12 multipliers. For some subroutines, we do not have many data dependencies and can issue many multiplications in parallel, up to our total number of multipliers. Thus, adding more multipliers will continue to reduce the total number of cycles for subroutines with large amounts of parallelism.

4.6 Total Cost of Routines

Here, we break up the relative costs of routines within our implementation of the SIDH protocol. Table 6 contains the results of various routines for p_{751}, which closely follows the formulas provided in [7]. $\tilde{A}$, $\tilde{S}$, and $\tilde{M}$ refer to addition, squaring, and multiplication, respectively, in $\mathbb{F}_{p^2}$. Routines with a note of $(\mathbb{F}_p)$ count operations in $\mathbb{F}_p$. The purpose of each routine can be summarized as follows:

- *Mont. Ladder Step* $(\mathbb{F}_p)$*:* Perform a single step of the Montgomery ladder [26] in $\mathbb{F}_p$, which requires 1 point addition and 1 point doubling.
- *3-point Ladder Step*: Perform a single step of the 3-point Montgomery ladder [6], which requires 2 point additions and 1 point doubling.
- *Mont. Quadruple/Triple*: Perform a scalar point multiplication by 4 in the case of quadrupling and scalar point multiplication by 3 in the case of tripling.
- *Get ℓ Isog*: Compute an isogeny of degree ℓ. Alice operates over isogenies of degree 4 and Bob operates over isogenies of degree 3.
- *Eval. ℓ Isog (x times)*: Push points through the isogenous mapping from their old curve to their new curve. This code is unrolled x times from 1 point to 11 points.
- $\mathbb{F}_{p^2}$ *inversion* $(\mathbb{F}_p)$: Compute the inverse of an element using Fermat's little theorem.

We also include the total number of operations in $\mathbb{F}_p$ for the SIDH protocol in Table 7. As was noted previously, operations in $\mathbb{F}_{p^2}$ can be broken down into operations in $\mathbb{F}_p$, so this table shows the total cost of each SIDH round.

5 FPGA Implementations

In this section, we present the results of our supersingular isogeny accelerator on a Virtex-7 FPGA. For a proof of concept, we implemented the SIDH key exchange protocol.

TABLE 6
Cost of Major Routines for p_{751}

Routine	Ops. in $\mathbb{F}_{p^2}$			#ops. in	Latency for n mults. (cc)					
	$(\tilde{A})$	$(\tilde{S})$	$(\tilde{M})$	protocol	2	4	6	8	10	12
Mont. Ladder Step ($\mathbb{F}_p$)	9	4	5	751	621	485	485	485	485	485
3-point Ladder Step	14	6	9	751	2,137	1,167	901	715	706	685
Mont. Quadruple	11	4	8	1,094	1,816	1,284	1,132	1,132	1,132	1,132
Mont. Triple	15	5	8	1,512	1,812	1,192	1,034	1,031	978	978
Get 4 Isog.	7	5	0	370	590	384	363	363	363	363
Eval. 4 Isog.	6	1	9	10	1,981	1,299	1,151	1,151	1,151	1,151
Eval. 4 Isog. (3 times)	18	3	27	145	4,557	2,395	1,775	1,557	1,455	1,315
Eval. 4 Isog. (5 times)	30	5	45	66	7,515	3,845	2,625	2,105	1,850	1,658
Eval. 4 Isog. (7 times)	42	7	63	54	10,473	5,357	3,796	3,234	2,944	2,692
Eval. 4 Isog. (9 times)	54	9	81	18	13,431	6,803	4,655	3,765	3,299	3,157
Eval. 4 Isog. (11 times)	66	11	99	2	16,389	8,242	5,597	4,371	3,800	3,461
Get 3 Isog.	8	3	3	478	883	488	455	424	424	424
Eval. 3 Isog.	2	2	6	12	1,425	1,013	830	824	750	750
Eval. 3 Isog. (3 times)	6	6	18	274	3,437	1,839	1,354	1,145	1,086	965
Eval. 3 Isog. (5 times)	10	10	30	112	5,681	2,933	2,017	1,596	1,371	1,192
Eval. 3 Isog. (7 times)	14	14	42	72	7,925	4,094	2,880	2,409	2,167	1,996
Eval. 3 Isog. (9 times)	18	18	54	20	10,169	5,178	3,517	2,852	2,456	2,327
Eval. 3 Isog. (11 times)	22	22	66	2	12,413	6,301	4,265	3,322	2,801	2,500
$\mathbb{F}_{p^2}$ Inversion ($\mathbb{F}_p$)	2	757	196	4	148,958	148,556	148,404	148,310	148,310	148,310

TABLE 7
Operation Counts for SIDH Rounds for Various Security Levels

SIDH Round	Ops. in $\mathbb{F}_p$			
	p_{503}	p_{751}	p_{1019}	p_{1533}
Alice Round 1	$23,915A + 42,545M$	$37,417A + 66,958M$	$52,929A + 94,773M$	$84,605A + 149,771M$
Bob Round 1	$24,894A + 48,456M$	$39,648A + 76,413M$	$54,663A + 106,047M$	$88,820A + 163,972M$
Alice Round 2	$23,152A + 39,170M$	$36,288A + 61,936M$	$51,392A + 87,915M$	$82,264A + 139,295M$
Bob Round 2	$26,758A + 45,513M$	$42,442A + 71,970M$	$58,402A + 100,134M$	$93,628A + 155,161M$

A and M correspond to modular addition and multiplication, respectively, in $\mathbb{F}_p$.

TABLE 8
Implementation Results of SIDH Architectures on a Xilinx Virtex-7 FPGA

Primec	Quantum Security (bits)	# Mults.	Area					Time			SIDH/s
			# FFs	# LUTs	# Slices	# DSPs	# BRAMs	Freq. (MHz)	Latency ($cc \times 10^6$)	Total time (ms)	
p_{503}	83	6	24,908	18,820	7,491	192	43.5	202.1	3.34	16.5	60.5
		8	31,809	23,483	9,608	256	42	202.5	3.09	15.2	65.5
		10	38,701	28,687	11,517	320	40.5	202.9	3.00	14.7	67.6
		12	**45,615**	**33,969**	**13,203**	**384**	**40**	**207.0**	**2.94**	**14.2**	**70.4**
p_{751}	124	6	38,489	27,713	11,277	288	60.5	204.9	7.46	36.4	27.4
		8	48,688	34,742	14,447	384	58.5	203.7	6.86	33.7	29.7
		10	58,846	42,390	16,983	480	56	197.7	6.56	33.2	30.2
		12	**69,054**	**50,084**	**19,892**	**576**	**54.5**	**201.5**	**6.37**	**31.6**	**31.6**
p_{1019}	168	6	54,572	37,188	13,443	384	74	188.9	13.47	71.3	14.0
		8	68,364	46,640	16,618	512	72.5	204.0	12.45	61.0	16.4
		10	82,102	57,086	23,187	640	70.5	192.6	11.75	61.0	16.4
		12	**95,895**	**67,432**	**26,976**	**768**	**68**	**195.3**	**11.41**	**58.4**	**17.1**
p_{1533}	252	6	82,176	59,246	20,559	576	90	194.6	30.06	154.5	6.47
		8	102,832	73,438	25,692	768	89.5	193.3	27.73	143.4	6.97
		10	123,429	88,945	34,700	960	87	184.0	26.00	141.3	7.08
		12	**144,127**	**104,684**	**40,279**	**1152**	**85**	**193.1**	**25.16**	**130.3**	**7.67**

5.1 Implementation Results

The SIDH core was synthesized with Xilinx Vivado 2015.4 to a Xilinx Virtex-7 xc7vx690tffg1157-3 device. All results were obtained after place-and-route. The area and timing results of our SIDH core are shown in Table 8. We present results for 3-6 replicated multipliers in our architecture. Fewer multipliers could not benefit from the isogeny evaluation parallelism and are not included. The implementation was optimized with

TABLE 9
Hardware Comparison of SIDH Architectures on a Virtex-7 with 3 Replicated Multipliers

Work	Prime	Area					Time		
	(bits)	# FFs	# LUTs	# Slices	# DSPs	# BRAMs	Freq. (MHz)	Latency ($cc \times 10^6$)	Total time (ms)
Koziel et al. [16]	511	30,031	24,499	10,298	192	27	177	5.967	33.7
Koziel et al. [17]	503	26,659	19,882	8,918	192	40	181.4	3.80	20.9
This Work	503	24,908	18,820	7,491	192	43.5	202.1	3.34	16.5
Improvement over [17]	-	−7.0%	−5.6%	−19%	-	+8.1%	+10.3%	−13.8%	−27%

the PerfOptimized High and Performance Explore options in Xilinx Vivado. These are constant-time results. Our SIDH parameters are discussed in Section 2.

The final column of Table 8 indicates the number of SIDH protocol executions per second. An SIDH protocol execution refers to the completion of round 1 and round 2 for both Alice and Bob, which is composed of 4 double-point multiplications and 4 large-degree isogeny computations. For a single party, the approximate SIDH protocol execution time is expected to be half of the total SIDH protocol since the party will only be performing 1 set of round 1 and round 2 computations.

The target of this work is to push an FPGA implementation of SIDH to be as fast as possible. As expected, the heavy parallelism that is achieved by evaluating multiple isogenies in parallel and performing quadratic extension field arithmetic pushes the cycle counts lower when more multipliers are added. When comparing 6 replicated dual multipliers to 3 replicated dual multipliers, the timing results are improved by approximately 20 percent, but at the cost of almost double the resources.

By applying the architecture to SIDH primes of approximately size 512 bits, 768 bits, 1,024 bits, and 1,536 bits, we show that the architecture is scalable as the clock frequency remains fairly stable. The critical path delay of the design remains at approximately 5 ns, but routing and additional hardware resources account for the difference between implementations. Interestingly, the synthesis and implementation tools appeared to have trouble for an odd number of replicated multipliers as is illustrated in the slower clock frequency in 6 and 10 multipliers of p_{1019} and p_{1533}. For our largest implementation, p_{1533} with 6 replicated dual-multipliers, the highest utilization of resources was for DSP's at 32 percent. IO pins, LUTs, FF's, and BRAM's were 25.67, 24.17, 16.64, and 5.78 percent utilized, respectively. For the same prime with 3 replicated multipliers, 16 percent of DSP's and 13.68 percent of LUT's were utilized. With these details, a 3,072-bit SIDH implementation (∼512 bits quantum security) should also fit on the same Virtex-7 device.

Since our fastest results were achieved for 6 replicated dual-multipliers in each security level, we highlighted each of these results in Table 8. These results show that performance scaled quadratically with the size of the prime.

5.2 Comparison to Other Works

5.2.1 Other SIDH Implementations

The only other hardware implementations of SIDH are [16] and [17], both of which were geared towards speed on a Virtex-7 and use a similar systolic multiplier. In Table 9, we provide a rough comparison for 6 multipliers at the approximate 512-bit security level. [16] uses affine isogeny formulas, resulting in a non-constant key-exchange. [17] uses the same projective isogeny formulas as this work, but has a slower frequency and requires more clock cycles, a consequence of the architecture and scheduling methodology. Thus, the implementation from this work is 27 percent faster and uses fewer resources than the faster of the two. For p_{751}, our 6 replicated multiplier implementation pushes the speedup to 37 percent faster than the fastest SIDH core in [17], but at the cost of additional hardware.

As these results show, the architecture and methodology presented in this paper provide a major boost to speed. We especially emphasize that our implementation results are constant-time, which helps defend against simple power analysis and timing attacks. In Table 10, we compare the time our SIDH round times to both software and hardware implementations at the approximate 85, 128, and 170-bit quantum security levels. In assessing the comparison of our hardware implementation to others, it is important to note that not all implementations utilized the same isogeny formulas. It is difficult to make a fair comparison. We include other prototype results to show how SIDH has progressed over the years. Notably, we utilized the projective isogeny formulas over isogeny graphs of 4 and 3 that was proposed by [7]. The only other known implementations that utilize these formulas are [7] and [17]. Both of these previous implementations were geared towards speed and used high-performance computing devices. In summary, our implementation is 2 times faster than the software implementation presented for a Haswell architecture in [7] and 1.36 times faster than the Virtex-7 implementation proposed in [17], which each use a similar set of projective isogeny formulas.

5.2.2 Other PQC Implementations

Comparison to other PQC public-key encryption and key-exchange schemes is even more difficult. Notably, the device, methodology, and motivations differ with the current FPGA literature on PQC, which is sparse at the moment. These different factors mean that this comparison is not fair. Nevertheless, we include this comparison as a baseline for the state of the art PQC schemes on reconfigurable computing.

In the FPGA PQC literature, we found works related to code-based crypto with McBits [36] and Ring-LWE with NewHope [37]. As we compare these works in Table 11, we emphasize that isogeny-based cryptography features much smaller public-keys. We chose our p_{503} and p_{751} implementations with 3 replicated multipliers to compare with because they have the best time-area products for our results.

When comparing to the McBits implementation scheme by Wang et al. [36], we note that this implementation was

TABLE 10
Comparison to Other SIDH Implementations

Work	Quantum Security (bits)	Platform	Smooth Isogeny Prime	Time (ms)				
				Alice Rnd. 1	Bob Rnd. 1	Alice Rnd. 2	Bob Rnd. 2	Total Time
~85-bit Quantum Security Level								
Jao and De Feo [5]	84	2.4 GHz Opt.	$2^{253}3^{161}7-1$	365	318	363	314	1360
Jao et al. [6]	85	2.4 GHz Opt.	$2^{258}3^{161}186-1$	28.1	28.0	23.3	22.7	102.1
Azarderakhsh et al. [10]	85	4.0 GHz i7	$2^{258}3^{161}186-1$	-	-	-	-	54.0
Koziel et al. [16]	84	Virtex-7	$2^{253}3^{161}7-1$	9.35	8.41	8.53	7.41	33.70
Koziel et al. [17]	83	Virtex-7	$2^{250}3^{159}-1$	4.83	5.25	4.41	4.93	19.42
This Work (12 Mults.)	83	Virtex-7	$2^{250}3^{159}-1$	3.59	3.87	3.22	3.53	14.22
~128-bit Quantum Security Level								
Jao et al. [6]	128	2.4 GHz Opt.	$2^{387}3^{242}-1$	65.7	54.3	65.6	53.7	239.3
Azarderakhsh et al. [10]	128	4.0 GHz i7	$2^{387}3^{242}-1$	-	-	-	-	133.7
Costello et al. [7]	124	3.4 GHz i7	$2^{372}3^{239}-1$	15.0	17.3	13.8	16.8	62.9
Koziel et al. [17]	124	Virtex-7	$2^{372}3^{239}-1$	10.6	11.6	9.5	10.8	42.5
This Work (12 Mults.)	124	Virtex-7	$2^{372}3^{239}-1$	7.99	8.63	7.14	7.86	31.61
~170-bit Quantum Security Level								
Jao et al. [6]	170	2.4 GHz Opt.	$2^{514}3^{323}353-1$	122	101	125	102	450
Azarderakhsh et al. [10]	170	4.0 GHz i7	$2^{514}3^{323}353-1$	-	-	-	-	266.9
This Work (12 Mults.)	168	Virtex-7	$2^{508}3^{319}35-1$	14.97	15.72	13.43	14.28	58.4

TABLE 11
Hardware Comparison of PQC Public Encryption and Key-Exchange Schemes on FPGA

Work	Scheme	Platform	Quan. Sec. (bits)	Public Key Size (Bytes)	Area				Time	
					# FFs	# LUTs	# DSPs	# BRAMs	Freq. (MHz)	Total (μs)
Wang et al. [36]	McBits[1]	Ultrascale+	128	1,046,739	-	112,845	-	375	225	3,980
Oder and Güneysu [37]	NewHope-Simple[2]	Artix-7	128	2,176	4,452	5,142	2	4	125	1,369
					4,635	4,498	2	4	117	1,532
This Work (6 Mults)	SIDH	Virtex-7	83	**378**	24,908	18,820	192	43.5	202	16,533
This Work (6 Mults)	SIDH	Virtex-7	124	**564**	38,489	27,713	288	60.5	205	36,440

1. Performs key generation only, which has a 29% chance of success.
2. Row 1 is server-side results and row 2 is client-side results.

also performance focused. However, the focus of this McBits work was to only generate public keys, which is a very large invertible matrix. Not all generated matrices are invertible, so there is approximately a 29 percent chance of success. This large matrix means a much larger public key, which is approximately 1800 times larger than that of SIDH. In terms of hardware, the evaluation FPGA's are different, but our SIDH implementation appears to use a fraction of LUTs and BRAMs. Lastly, the key generation of this McBits works is faster by a factor of 4 and 9 for our 83 and 124-bit quantum security implementations, respectively.

When comparing to the NewHope-Simple work from Oder and Güneysu [37], we note that this NewHope implementation was geared towards a compact design for server and client, completely the opposite of our design rationale. They provide separate implementations for the server and client side. Since our implementation performs the entire protocol, we added separate entries for the server and client results. Directly combining both results results in hardware that uses approximately 3 times fewer LUTs and 4 times fewer FFs when compared to our 124-bit quantum security implementation. Nevertheless, this NewHope-Simple work performs the entire protocol 5 and 12 times faster than our p_{503} and p_{751} implementations, respectively, with very few resources. The primary upside of our implementation is that the public keys are about 4 times smaller and has a simpler selection of public parameters for a particular security level.

6 Real-World Implementation Consideration

Here, we discuss aspects related to the deployment of our supersingular isogeny accelerator. Notably, we discuss the security of our SIDH implementation and other isogeny-related applications. We present this implementation as an FPGA rapid prototype. The same architecture could be applied to an application-specific integrated circuit (ASIC) security coprocessor. The resulting deployment to smart card or less constrained environments determines certain security measures and side-channel countermeasures to ensure that the cryptosystem is not broken by the physical phenomena emitted by a real-world implementation.

6.1 Security Considerations

In this implementation, the only building block missing is a true-random number generator. Based on the oracle attack

proposed in [38] and fault attacks proposed in [39], [40], there are many uncertainties to using static keys. Thus, a random number generator, whether it is accessible to the hardware or a host CPU, is necessary to generate safe keys for use with isogeny-based cryptosystems.

6.1.1 Static Key Considerations

If a static key is to be used, then the Kirkwood et al. [41] validation model must be used to ensure that maliciously crafted keys will not divulge bits of the key, as is illustrated in [38]. This validation model requires the ephemeral key user to use a seed to generate a key from a key derivation function, compute the shared secret key with the static user's public keys, and encrypt the seed to the key derivation function with the resulting shared secret. The static user will then perform the SIDH protocol as normal and use the shared secret to decrypt the seed for the key derivation function. Upon retrieving the ephemeral user's private key, the static key user will then recompute the first round of the ephemeral user to ensure that the ephemeral user transmitted honestly generated public keys. If the two sets of ephemeral public keys do not match, then the static user rejects the key exchange session. This validation model is expensive as it requires the static key user to perform two rounds and only protects against the Galbraith et al. [38] oracle attack.

6.1.2 Side-Channel Attacks

Side-channel attacks attempt to break a cryptosystem by taking advantage of physical phenomena emitted in a physical implementation of a cryptosystem, such as power consumption, timing information, and electromagnetic radiation. Since isogeny-based cryptography resembles elliptic curve cryptography, much of the ECC side-channel literature can be applied to its isogeny-based cryptography counterpart. Notably, since this is a constant-time implementation with a constant set of operations, this architecture is naturally resistant to simple power analysis and timing attacks. Differential power analysis is only a consideration if a static key is used. In this case, one could perform the random (with values from the random number generator) projectivization of the input torsion basis to change the power signature of multiple runs [42]. This would add one multiplication per step for the standard Montgomery ladder and two multiplications per step for the three-point Montgomery ladder, which would be effectively parallelized since they do not increase data dependencies of the ladder step.

As a quick experiment for this random projectivization, we calculated the additional cost of randomizing the input torsion points. For our test, we first used p_{751} with 8 multipliers. Our Montgomery ladder over the base field increased from 485 cycles to 488 cycles and our 3-point Montgomery ladder increased from 715 cycles to 804 cycles. This increased the total runtime of our SIDH protocol by 69,092 cycles, which translated to an extra 1.008 percent of the protocol. When we tested p_{751} with 12 multipliers, the Montgomery ladder again increased from 485 cycles to 488 cycles, but the 3-point Montgomery ladder only increased from 685 to 693 cycles. Thus, the extra DPA countermeasure only cost an extra 0.13 percent in this scenario as the extra multiplications were effectively interleaved. This additional cost, however, does not include the cost of using a true random number generator to generate the random projectivization.

In [43], the authors propose three different zero value attacks to reveal a user's secret key. We note that the random projectivization will help defend against the partial-zero attack on the 3-point Montgomery ladder. Defending against the zero-point attack on the 3-point Montgomery ladder requires dynamic keys or key validation, which we discuss in the following section. Lastly, to defend against the refined power analysis attack on the large-degree isogeny computation, dynamic keys or a random curve isomorphism could be used.

6.1.3 Fault and Active Attacks

Lastly, fault and active attacks are a tricky topic in that it again depends on the application of the device. To protect against invalid public keys, such as points of invalid order or invalid curves, Costello et al. [7] propose to validate the public keys. They propose validating that the transmitted torsion basis points have the correct order, a Weil pairing with the maximum possible order, and have a valid supersingular elliptic curve. Of these operations, the most expensive operations are the Weil pairing, two scalar multiplications by ℓ^e, and a point multiplication by $\ell_A^{e_A}\ell_B^{e_B}f$. In general, these are fairly serial operations, but we could parallelize these computations by performing each of the scalar multiplications in parallel. If needed, this isogeny accelerator could efficiently implement key validation to help protect against invalid public keys. For the loop-abort fault attack proposed in [39], extra hardware is required to ensure that the isogeny controller is not faulted to finish early. The secret kernel point fault attack proposed in [40] could be used in the case that this hardware has a static key and recomputes the static public key, which would not be a typical use-case as the static public key would most likely simply be inserted into the RAM file.

6.2 Isogeny Cryptography Applications

In this implementation, we centered our results on the timing of SIDH round times. There are various other isogeny-based cryptosystems that this architecture could apply, notably SIDH compression, undeniable signatures, and digital signatures.

6.2.1 SIDH Key Compression

To continue the discussion of SIDH, public key compression and decompression [10], [11] appear to have useful applications for the transmission or long-term storage of SIDH keys. Compression is done by methodically generating a torsion basis from the j-invariant of the original public key's elliptic curve and performing two double-point multiplication discrete logarithms on the image of the other party's torsion basis to generate scalars. The j-invariant, double-point multiplication scalars, and a bit indicating if there is a twist are sent over the public channel instead of the normal public keys. Then, the other party would use the same method to generate a torsion basis from the transmitted j-invariant and perform two double-point multiplications to recover the supersingular elliptic curve and image of their torsion basis. This procedure compresses the size of the public key by a factor of $5/12$ in [11].

When assessing the viability of decompression and compression in this implementation, we note that the three

primary computations in compression and decompression include systematically constructing an n-torsion basis, performing various pairing computations, and solving a discrete logarithm with Pohlig-Hellman. The two main problems when implementing all of this functionality is that it requires a much larger amount of program ROM and many serial functionalities. Since our architecture explicitly instructs the addition, multiplication, and memory units each cycle, our program ROM grows very quickly, but allows a large amount of parallelism. One could change the controller to issue instruction by instruction at the cost of a massive slowdown if the number of memory units is a problem. Otherwise, the designer could bite the bullet and include all functionalities in the program ROM. For simple SIDH, only 6 percent of block RAM's were utilized for p_{1533}. Otherwise, there are many serial operations in solving each of the primary computations, including exponentiations, square roots, and Pohlig-Hellman. As was shown for the simple exponentiation for inversion in SIDH, it is difficult to achieve high speeds in serial computations since data dependencies restrict the usage of all available resources.

Although it is shown in [11] that compression and decompression increase the total protocol time by a factor of 2.7, it is expected that for our hardware implementation, this protocol time would be higher as a result of the loss in parallelism. More than likely, decompression can be added to the protocol and increase the total protocol time by a small margin, but compression does not seem practical as it requires a large amount of program ROM, temporary values, and serial calculations.

6.2.2 Undeniable and Digital Signatures Based on Isogenies

Undeniable signatures based on isogenies [8] are interesting in that the signer has the freedom to select who can verify the signature. This scheme can be thought of as a 3-dimensional equivalent of SIDH, whereby multiple isogenies are computed over 3 different isogeny graphs. The prime p is of the form $p = \ell_A^{e_A}\ell_M^{e_M}\ell_C^{e_C} f \pm 1$. In this scheme, the primary computations are double-point multiplications and large-degree isogenies, the same as SIDH. Since the modular arithmetic works for a generic prime, the isogeny-based architecture presented here will also efficiently perform computations in a prime of this form. After adding more optimized isogeny formulas, such as $\ell = 5$ or $\ell = 7$, this architecture can efficiently support undeniable signatures.

Digital signature schemes have recently been proposed in [14] and [13]. The first scheme in [14] and only scheme in [13] are both based on utilizing Unruh's construction [44] to transform the interactive zero-knowledge proof of isogeny-based cryptography to a non-interactive zero-knowledge proof to generate proofs for a signature. In this signature scheme, the public parameters are similar to SIDH, but requires only one torsion basis for the isogeny graph of $\ell_B^{e_B}$. Thus, this isogeny accelerator could be used to efficiently compute digital signatures.

7 Conclusion

In this work, we presented an efficient and scalable supersingular isogeny accelerator for isogeny-based cryptography. We illustrated a high-performance field arithmetic unit, efficient scheduling methodology, and achievable parallelization schemes in isogeny evaluations and $\mathbb{F}_{p^2}$ arithmetic. When applied to the supersingular isogeny Diffie-Hellman key exchange protocol, our architecture on FPGA is 2 times faster than a Haswell software implementation and 1.36 times faster than the fastest other FPGA implementation. We demonstrated the scalability of our hardware architecture by implementing over four different finite fields, ranging from a low level of quantum security to a high level of quantum security. Lastly, we showed that the same isogeny accelerator could also be used to efficiently accelerate undeniable signatures and digital signatures. Overall, isogeny-based cryptography appears to be a strong candidate for standardization since it utilizes small keys and this work demonstrates that hardware accelerators are indeed viable and can achieve a high degree of parallelization.

Acknowledgments

The authors would like to thank the reviewers for their comments. This work is supported in parts by the grants NIST-60NANB16D246, ARO W911NF-17-1-0311, and US National Science Foundation CNS-1661557.

References

[1] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. Symp. Found. Comput. Sci.*, 1994, pp. 124–134.
[2] L. Chen and S. Jordan, "Report on post-quantum cryptography," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. NISTIR 8105, 2016.
[3] A. Rostovtsev and A. Stolbunov, "Public-key cryptosystem based on isogenies," Cryptology ePrint Archive, Tech. Rep. 2006/145, 2006.
[4] A. Childs, D. Jao, and V. Soukharev, "Constructing elliptic curve isogenies in quantum subexponential time," 2010. [Online]. Available: https://arxiv.org/abs/1012.4019
[5] D. Jao and L. De Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *Proc. Post-Quantum Cryptography*, 2011, pp. 19–34.
[6] L. De Feo, D. Jao, and J. Plut, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," *J. Math. Cryptology*, vol. 8, no. 3, pp. 209–247, Sep. 2014.
[7] C. Costello, P. Longa, and M. Naehrig, "Efficient algorithms for supersingular isogeny Diffie-Hellman," in *Proc. Annu. Int. Cryptology Conf. Advances Cryptology*, 2016, pp. 572–601.
[8] D. Jao and V. Soukharev, "Isogeny-based quantum-resistant undeniable signatures," in *Proc. Post-Quantum Cryptography*, 2014, pp. 160–179.
[9] X. Sun, H. Tian, and Y. Wang, "Toward quantum-resistant strong designated verifier signature from isogenies," in *Proc. Int. Conf. Intell. Netw. Collaborative Syst.*, 2012, pp. 292–296.
[10] R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi, "Key compression for isogeny-based cryptosystems," in *Proc. ACM Int. Workshop ASIA Public-Key Cryptography*, 2016, pp. 1–10.
[11] C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes, and D. Urbanik, "Efficient compression of SIDH public keys," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2017, pp. 679–706.
[12] R. Azarderakhsh, D. Jao, and C. Leonardi, "Post-quantum static-static key agreement using multiple protocol instances," in *Proc. Int. Conf. Sel. Areas Cryptography*, 2017, pp. 45–63.
[13] Y. Yoo, R. Azarderakhsh, A. Jalali, D. Jao, and V. Soukharev, "A post-quantum digital signature scheme based on supersingular isogenies," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2017, pp. 163–181.
[14] S. D. Galbraith, C. Petit, and J. Silva, "Identification protocols and signature schemes based on supersingular isogeny problems," in *Proc. Int. Conf. Theory Appl. Cryptology Inf. Secur.*, 2017, pp. 3–33.

[15] B. Koziel, A. Jalali, R. Azarderakhsh, D. Jao, and M. Mozaffari Kermani, "NEON-SIDH: Efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM," in *Proc. Int. Conf. Cryptology Netw. Secur.*, 2016, pp. 88–103.
[16] B. Koziel, R. Azarderakhsh, M. Mozaffari Kermani, and D. Jao, "Post-quantum cryptography on FPGA based on isogenies on elliptic curves," *IEEE Trans. Circuits Syst. I: Reg. Papers*, vol. 64, no. 1, pp. 86–99, Jan. 2017.
[17] B. Koziel, R. Azarderakhsh, and M. Mozaffari Kermani, "Fast hardware architectures for supersingular isogeny Diffie-Hellman key exchange on FPGA," in *Proc. Int. Conf. Cryptology India*, 2016, pp. 191–206.
[18] A. Jalali, R. Azarderakhsh, and M. Mozaffari Kermani, "Efficient post-quantum undeniable signature on 64-bit ARM," in *Proc. Sel. Areas Cryptography*, 2017, pp. 281–298.
[19] P. S. Hirschhorn, J. Hoffstein, N. Howgrave-Graham, and W. Whyte, "Choosing NTRUEncrypt parameters in light of combined lattice reduction and MITM approaches," in *Proc. Int. Conf. Appl. Cryptography Netw. Secur.*, 2009, pp. 437–455.
[20] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange—A new hope," in *Proc. USENIX Secur. Symp.*, 2016, pp. 327–343.
[21] D. J. Bernstein, T. Chou, and P. Schwabe, "McBits: Fast constant-time code-based cryptography," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2013, pp. 250–272.
[22] J. H. Silverman, *The Arithmetic of Elliptic Curves*, New York, NY, USA: Springer, 1992.
[23] J. Tate, "Endomorphisms of abelian varieties over finite fields," *Inventiones Mathematicae*, vol. 2, pp. 134–144, 1966.
[24] J. Vélu, "Isogénies entre courbes elliptiques," *Comptes Rendus de l'Académie des Sciences Paris Séries A-B*, vol. 273, pp. A238–A241, 1971.
[25] C. Petit, "Faster algorithms for isogeny problems using torsion point images," in *Proc. Int. Conf. Theory Appl. Cryptology Inf. Secur.*, 2017, pp. 330–353.
[26] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *J. Math. Comput.*, vol. 48, pp. 243–264, 1987.
[27] J.-M. Couveignes, "Hard homogeneous spaces," Cryptology ePrint Archive, Tech. Rep. 2006/291, 2006.
[28] P. L. Montgomery, "Modular multiplication without trial division," *J. Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.
[29] G. Sutter, J.-P. Deschamps, and J. L. Imaña, "Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation," *IEEE Trans. Ind. Electron.*, vol. 58, no. 7, pp. 3101–3109, Jul. 2011.
[30] G. Perin, D. G. Mesquita, and J. B. dos Santos Martins, "Montgomery modular multiplication on reconfigurable hardware: Systolic versus multiplexed implementation," *Int. J. Reconfigurable Comput.*, vol. 2011, pp. 1–10, 2011.
[31] A. Mrabet, N. E. Mrabet, R. Lashermes, J.-B. Rigaud, B. Bouallegue, S. Mesnager, and M. Machhout, "A systolic hardware architectures of Montgomery modular multiplication for public key cryptosystems," *J. Hardware Syst. Secur.*, vol. 1, no. 3, pp. 219–236, Sep. 2017.
[32] J. W. Bos and S. Friedberger, "Fast arithmetic modulo $2^x p^y \pm 1$," Cryptology ePrint Archive, Tech. Rep. 2016/986, 2016.
[33] T. Blum and C. Paar, "High-radix montgomery modular exponentiation on reconfigurable hardware," *IEEE Trans. Comput.*, vol. 50, no. 7, pp. 759–764, Jul. 2001.
[34] C. McIvor, M. McLoone, and J. V. McCanny, "High-radix systolic modular multiplication on reconfigurable hardware," in *Proc. IEEE Int. Conf. Field-Programmable Technol.*, Dec. 2005, pp. 13–18.
[35] B. Koziel, R. Azarderakhsh, D. Jao, and M. Mozaffari Kermani, "On fast calculation of addition chains for isogeny-based cryptography," in *Proc. Int. Conf. Inf. Secur. Cryptology*, 2016, pp. 323–342.
[36] W. Wang, J. Szefer, and R. Niederhagen, "FPGA-based key generator for the Niederreiter cryptosystem using binary Goppa codes," in *Proc. Int. Conf. Cryptographic Hardware Embedded Syst.*, 2017, pp. 253–274.
[37] T. Oder and T. Güneysu, "Implementing the NewHope-Simple key exchange on low-cost FPGAs," in *Proc. Int. Conf. Cryptology Inf. Secur. Latin America*, 2017.
[38] S. D. Galbraith, C. Petit, B. Shani, and Y. B. Ti, "On the security of supersingular isogeny cryptosystems," in *Proc. Int. Conf. Theory Appl. Cryptology Inf. Secur.*, 2016, pp. 63–91.
[39] A. Gélin and B. Wesolowski, "Loop-abort faults on supersingular isogeny cryptosystems," in *Proc. Int. Workshop Post-Quantum Cryptography*, 2017, pp. 93–106.
[40] Y. B. Ti, "Fault attack on supersingular isogeny cryptosystems," in *Proc. Int. Workshop Post-Quantum Cryptography*, 2017, pp. 107–122.
[41] D. Kirkwood, B. C. Lackey, J. McVey, M. Motley, J. A. Solinas, and D. Tuller, "Failure is not an option: Standardization issues for post-quantum key agreement," in *Talk Workshop Cybersecurity Post-Quantum World*, Apr. 2015.
[42] J.-S. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 1999, pp. 292–302.
[43] B. Koziel, R. Azarderakhsh, and D. Jao, "Side-channel attacks on quantum-resistant supersingular isogeny Diffie-Hellman," in *Proc. Int. Conf. Sel. Areas Cryptography*, 2017, pp. 64–81.
[44] D. Unruh, "Non-interactive zero-knowledge proofs in the quantum random oracle model," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2015, pp. 755–784.

Brian Koziel received the BSc and MSc degrees in computer engineering from the Rochester Institute of Technology, in 2016. Currently, he is a cryptographic designer at Texas Instruments. His current research interests include efficient software and hardware implementations of elliptic curve cryptography and post-quantum cryptography. At RIT, he was a recipient of the prestigious Outstanding Undergraduate Scholar award.

Reza Azarderakhsh received the PhD degree in electrical and computer engineering from the Western University, in 2011. He was a recipient of the NSERC Post-Doctoral Research Fellowship working with the Center for Applied Cryptographic Research and the Department of Combinatorics and Optimization, University of Waterloo. Currently, he is an assistant professor with the Department of Electrical and Computer Engineering, Florida Atlantic University. His current research interests include finite field and its application, elliptic curve cryptography, pairing based cryptography, and post-quantum cryptography. He is serving as an associate editor of the *IEEE Transactions on Circuits and Systems* (TCAS-I). He is a member of the IEEE.

Mehran Mozaffari Kermani (S'00-M'11-SM'16) received the BSc degree in electrical and computer engineering from the University of Tehran, Tehran, Iran, in 2005, and the MESc and PhD. degrees from the Department of Electrical and Computer Engineering, University of Western Ontario, London, Canada, in 2007 and 2011, respectively. He joined the Advanced Micro Devices as a senior ASIC/layout designer, integrating sophisticated security/cryptographic capabilities into accelerated processing. In 2012, he joined the Electrical Engineering Department, Princeton University, New Jersey, as an NSERC post-doctoral research fellow. From 2013-2017, he was an assistant professor with Rochester Institute of Technology and starting 2017, he has joined the Computer Science and Engineering Department, University of South Florida. Currently, he is serving as an associate editor for the *IEEE Transactions on VLSI Systems*, the *ACM Transactions on Embedded Computing Systems*, the *IEEE Transactions on Circuits and Systems I*, and the guest editor for the *IEEE Transactions on Dependable and Secure Computing* for the special issue of Emerging Embedded and Cyber Physical System Security Challenges and Innovations (2016 and 2017). He was the lead guest editor for the *IEEE/ACM Transactions on Computational Biology and Bioinformatics* and the *IEEE Transactions on Emerging Topics in Computing* for special issues on security. He has been the TPC member for a number of conferences including HOST (Publications Chair), DAC, DATE, RFIDSec, LightSec, WAIFI, FDTC, and DFT. He was a recipient of the prestigious Natural Sciences and Engineering Research Council of Canada Post-Doctoral Research Fellowship in 2011 and the Texas Instruments Faculty Award (Douglas Harvey) in 2014. He is a senior member of the IEEE.