SIKE'd Up: Fast Hardware Architectures for Supersingular Isogeny Key Encapsulation

Brian Koziel, A-Bon Ackie, Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani

Abstract

In this work, we present a fast parallel architecture to perform supersingular isogeny key encapsulation (SIKE). We propose and implement a fast isogeny accelerator architecture that uses fast and parallelized isogeny formulas. On top of our isogeny accelerator, we build a novel architecture for the SIKE primitive, which provides both quantum and IND-CCA security. We synthesized this architecture on the Xilinx Artix-7, Virtex-7, and Kintex UltraScale+ FPGA families. Over Virtex-7 FPGA's, our constant-time implementations are roughly 14% faster than the state-of-the-art with a better area-time product. At the NIST security level 5 on a Kintex UltraScale+ FPGA, we can execute the entire SIKE protocol in 15.3 ms. This work continues to improve the speed of isogeny-based computations and also features all parameter sets of the SIKE round 2 specification, with results applicable to NIST's post-quantum standardization process.

Index Terms

Field-programmable gate array, isogeny-based cryptography, post-quantum cryptography, SIKE.

I. Introduction

Although it is unclear when large-scale quantum computers will be available, it is very clear that such an event will have huge ramifications on today's public-key cryptosystems. Notably, Shor's algorithm [1] could be used in conjunction with a quantum computer to break factorization, discrete logarithm, and elliptic curve discrete logarithm problems which are the security foundation for RSA, Diffie-Hellman, and elliptic curve cryptography, respectively. These quantum computer fears have existed for decades and inspired a new domain of cryptography: post-quantum cryptography (PQC). PQC focuses on cryptographic algorithms that are resistant to attackers armed with both classical and quantum computers.

However, the use of elliptic curves in public-key cryptography is not dead. Isogeny-based cryptography relies on the difficulty to compute isogenies between elliptic curves. Rather than finding a secret point with a secret point multiplication, the objective of isogeny-based cryptography is to find a secret elliptic curve isomorphism class by performing a secret walk on an isogeny graph. For large finite fields, it is *difficult* to construct an isogeny between two distant isomorphism classes of isogenous curves. This supersingular isogeny problem is related to the claw problem, which is hard even in the quantum sense.

The use of isogenies for a cryptosystem was first proposed in independent works by Couveignes [2] and Rostovtsev and Stolbunov [3] that were first published in 2006. This isogeny-based key-exchange was protected by the difficulty to compute isogenies between ordinary elliptic curves. In 2009, Charles *et al.* further explored this problem and designed an isogeny-based hash function [4]. In 2010, Childs, Jao, and Soukharev [5] proposed a quantum algorithm to compute isogenies between ordinary elliptic curves in subexponential time. Then, in 2011, Jao and De Feo [6] proposed a different isogeny-based cryptosystem that was instead protected by the difficulty to compute isogenies between *supersingular* elliptic curves. This was called the supersingular isogeny Diffie-Hellman (SIDH) key exchange and is based on the supersingular isogeny problem, for which there is no known quantum attack in subexponential time. In addition to the standard SIDH

B. Koziel, A. Ackie, R. El Khatib, and R. Azarderakhsh are with CEECS Department, Florida Atlantic University, Boca Raton, FL. Email addresses: (bkoziel2017, razarderakhsh, aackie, relkhatib2015)@fau.edu. M. Mozaffari Kermani is with the CSE Department at University of South Florida, Tampa FL. Email address: mehran20@usf.edu.

primitive, the supersingular isogeny problem has also been used to create digital signatures [7], [8] and undeniable signatures [9].

At PQCrypto 2016, NIST announced a standardization process for post-quantum algorithms [10]. Among the primary quantum-resilient candidates, there is no clear winner. There are various tradeoffs in underlying quantum security, key sizes, and efficiency. Isogeny-based cryptography sticks out as a strong candidate because it features the smallest key sizes of known PQC algorithms. Small key sizes reduce transmission cost and storage requirements. At NIST security level 5, the public keys in supersingular isogeny Diffie-Hellman (SIDH) key exchange are approximately 576 bytes and key compression [11], [12] further reduces this to 336 bytes. However, the primary downsides to SIDH are that static keys cannot be reused (as malicious public keys can reveal bits of a user's private keys [13]) and that it is slow.

The supersingular isogeny key-encapsulation (SIKE) scheme [14] was submitted to NIST's standardization process as an isogeny-based key exchange alternative to SIDH that *can* safely support static keys. This scheme resembles SIDH in computations, but also adds additional hashing operations to provide indistinguishability under chosen ciphertext attack (IND-CCA). Over a public channel at NIST security level 5, a 564 byte public key and 596 byte ciphertext are exchanged and a 24 byte shared secret is computed. SIKE is currently in the second round of the NIST PQC standardization process, featuring the smallest public key sizes of known PQC key encapsulation algorithms.

On the efficiency side, much research has gone into bringing SIDH and isogeny computation times down. Notably, faster isogeny arithmetic [15], [16], double-point multiplication schemes [17], and large-degree isogeny parallelization [18] have improved the performance of isogeny computations. On the software side at NIST security level 2, the SIDH protocol on a high-performance processor has dropped from the order of 1.3 s [6] to 10 ms [14]. Other software implementations have improved the isogeny computation time on smaller ARM processors [19], [20]. On the hardware side at NIST security level 2, the SIDH protocol on a high-performance FPGA has dropped from 34 ms [21] to 14 ms [22].

Contribution. In this paper, we improve upon the high-performance isogeny accelerator presented in [22] and present a fast FPGA SIKE implementation that continues to push isogeny-based computations faster. In this SIKE architecture, we have proposed new optimizations to the field adder, field multiplier, field arithmetic unit architecture, scheduler, and full interface between an isogeny accelerator and Keccak accelerator to accomplish SIKE computations. We have also utilized new, faster isogeny formulas over an optimized scheduling methodology. Our constant-time implementation performs isogeny-based primitives 14% faster than the previous fastest known FPGA implementation with a better area-time product. On Xilinx Artix-7, Virtex-7, and UltraScale+ FPGA boards, we implemented all SIKE round 2 parameter sets, `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751`, which conservatively provide NIST security levels 1, 2, 3, and 5 over large finite fields of 434, 503, 610, and 751 bits, respectively.

Organization. Our paper is organized as follows. In Section 2, we review isogeny-based cryptography preliminaries. In Section 3, we present design choices in our finite field accelerator and scheduling that achieve faster isogeny acceleration than the previous state-of-the-art. In Section 4, we describe our architecture to encapsulate all SIKE functionalities on top of our isogeny accelerator. In Section 5, we present our FPGA results and compare to the previous state-of-the-art. Lastly, in Section 6, we conclude this paper.

II. Preliminaries

Here, we review some preliminaries of isogeny-based cryptography that are necessary for SIKE. We point the reader to [23] for a more in-depth review of isogeny fundamentals.

A. Isogeny-Based Cryptography

Elliptic curve cryptography deals with the study of elliptic curves over finite fields with many useful applications to public-key cryptography. An elliptic curve over a finite field $\mathbb{F}_q$ is the collection of all points (x, y) as well as the point at infinity that satisfy the short Weierstrass form:

$$E/\mathbb{F}_q : y^2 = x^3 + ax + b$$

where a,b, $x, y \in \mathbb{F}_q$. The set points form an abelian group over addition [24]. Consider a point $P = (x, y)$. We can perform consecutive point addition and doublings to compute an elliptic curve point multiplication, $Q = kP$ where $k \in \mathbb{Z}$ and $P, Q \in E$. Scalar point multiplication forms the basis for the elliptic curve discrete logarithm problem, for as the abelian group becomes very large (such as 2^{256} points), it becomes infeasible to find k given Q and P. However, this is only in a classical security model. Shor's algorithm [1] will break the elliptic curve discrete logarithm problem by computing discrete logarithms in polynomial time on a quantum computer.

Isogeny-based cryptography on the other hand, deals with the relationships between elliptic curves. We define an elliptic curve isogeny over $\mathbb{F}_q$, $\phi : \ E \rightarrow E'$ as a non-constant rational map from $E(\mathbb{F}_q)$ to $E'(\mathbb{F}_q)$ that preserves the point at infinity. This can be thought of as a mapping of points from one elliptic curve to another. The j-invariant of a curve acts as its unique identifier for an elliptic curve isomorphism class. We can compute a unique isogeny by using Vélu's formulas [25] over a kernel, $\phi : \ E \rightarrow E/\langle ker \rangle$. The degree of an isogeny is its degree as a rational map. We can compute large-degree isogenies of the form ℓ^e by chaining e isogenies of degree ℓ.

In this work, we are particularly interested in *supersingular* elliptic curves rather than *ordinary* elliptic curves as Childs *et al.* [5] have proposed a quantum subexponential attack on ordinary curves. The non-commutative nature of a supersingular curve's endomorphism ring renders the quantum attack in [5] useless. Thus, for supersingular elliptic curves, $q = p^2$ and there are approximately $p/12$ isomorphism classes.

Isogeny-based cryptography relies on the difficulty to compute isogenies between elliptic curves. For $\phi : \ E \rightarrow E'$ where ϕ is given as a product of small-degree isogenies, it is simple to perform an isogeny from E to E', but it is difficult to find an isogeny from E to E'. This problem can be visualized as a walk on an isogeny graph where each node represents an isomorphism class and the edges are isogenies of degree ℓ. Considering a specific ℓ, this is a complete graph where each node has $\ell + 1$ unique isogenies up to isomorphism of degree ℓ. We point the reader to [26] for an analysis of best-known classical computer attacks and [27] for an analysis of the best-known quantum computer attacks.

B. Supersingular Isogeny Diffie-Hellman

The supersingular isogeny Diffie-Hellman (SIDH) key-exchange [6] protocol utilizes the supersingular isogeny problem for two parties to securely agree on a shared secret. The idea behind the protocol is that Alice and Bob have secret isogeny walks on their respective isogeny graphs. They each perform their secret walk over public parameters, exchange them, then perform their secret walk on the public keys. Alice and Bob each perform two secret walks, but in a different order. The end result is that both parties end up at a secret isomorphism class where the j-invariant can be used as a shared secret.

To perform this protocol, Alice and Bob first agree on a prime p of the form $\ell_A^{e_A} \ell_B^{e_B} \pm 1$, where ℓ_A and ℓ_B are small primes and e_A and e_B are positive integers. They then agree on a supersingular elliptic curve $E_0(\mathbb{F}_{p^2})$ and find torsion bases $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ that generate $E_0[\ell_A^{e_A}]$ and $E_0[\ell_B^{e_B}]$, respectively. Lastly, Alice chooses a private scalar key $n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$ and Bob likewise chooses a private scalar key $n_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$.

To perform a secret isogeny walk, Alice and Bob separately generate a secret kernel by performing a double-point multiplication, $R = P + nQ$ and computing a unique isogeny over that kernel $\phi : \ E \rightarrow E/\langle R \rangle$. SIDH is composed of two rounds of isogenies. In the first round, Alice computes the isogeny $\phi_A : \ E_0 \rightarrow E_A = E_0/\langle P_A + [n_A]Q_A \rangle$ and also projects Bob's basis points under the new curve, $\{\phi_A(P_B), \phi_A(Q_B)\} \subset E_A$. Bob likewise computes the isogeny $\phi_B : E_0 \rightarrow E_B = E_0/\langle P_B + [n_B]Q_B \rangle$ and projects Alice's basis points under the new curve, $\{\phi_B(P_A), \phi_B(Q_A)\} \subset E_B$. Alice's public key is the tuple $\{E_A, \phi_A(P_B), \phi_A(Q_B)\}$ and Bob's public key is the tuple $\{E_B, \phi_B(P_A), \phi_B(Q_A)\}$. For the second round, Alice and Bob perform their secret isogeny walk over the public keys from the other party. Alice computes her isogeny $\phi'_A : E_B \rightarrow E_{AB} = E_B/\langle \phi_B(P_A) + [n_A]\phi_B(Q_A) \rangle$ and Bob computes his isogeny $\phi'_B : E_A \rightarrow E_{BA} = E_A/\langle \phi_A(P_B) + [n_B]\phi_A(Q_B) \rangle$. The curves E_{AB} and E_{BA} reside in the same isomorphism class, so the j-invariant can be used as the shared secret.

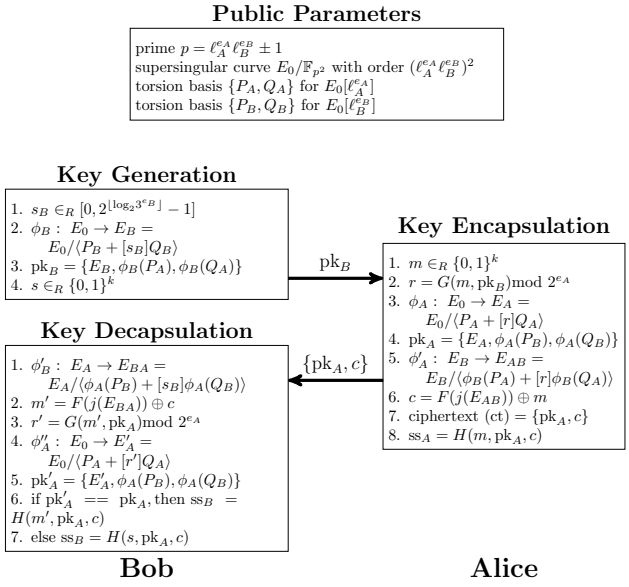

Figure 1. Supersingular isogeny key encapsulation protocol [14].

C. Supersingular Isogeny Key Encapsulation

The SIKE protocol is an IND-CCA variant of SIDH. Since this is a key encapsulation mechanism, SIKE produces a random shared secret that is encrypted and broadcast over an open channel. This was created by applying the Hofheinz, Hövelmanns, and Kiltz transform [28] to the supersingular isogeny public-key encryption scheme first proposed by Jao and De Feo [6]. This protocol consists of three phases: generate keys, encapsulate key, and decapsulate key. Figure 1 shows the full SIKE protocol. Let F, G, H be hashing functions.

Similar to SIDH as described in the previous section, Alice and Bob first agree on a prime p of the form $\ell_A^{e_A}\ell_B^{e_B} \pm 1$, where ℓ_A and ℓ_B are small primes and e_A and e_B are positive integers. They then agree on a supersingular elliptic curve $E_0(\mathbb{F}_{p^2})$ and find torsion bases $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ that generate $E_0[\ell_A^{e_A}]$ and $E_0[\ell_B^{e_B}]$, respectively. However, in contrast to SIDH, it is only Bob that chooses a private scalar key $s_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$. The security properties of SIKE allow Bob to safely reuse any of his public keys.

To illustrate the SIKE protocol, let us assume that Alice and Bob want to agree on a shared secret. Bob starts by choosing public parameters (similar to SIDH) and broadcasts a public key with the key generation phase. For this step, Bob computes the secret isogeny, $\phi_B : E_0 \to E_B = E_0/\langle P_B + [s_B]Q_B \rangle$. Bob publishes his public key, $\text{pk}_B = \{E_B, \phi_B(P_A), \phi_B(Q_A)\}$, and also generates a hidden key of length k bits, $s =_R \{0,1\}^k$.

Alice wants to engage in secure communications with Bob. Upon receiving Bob's public key, Alice performs key encapsulation by first generating a random message of length k bits, $m =_R \{0,1\}^k$. She finds a secret scalar by hashing the random message with Bob's public key, $r = G(m, \text{pk}_B)$. With this secret scalar, Alice performs two secret isogenies, one over the public parameters, $\phi_A : E_0 \to E_A = E_0/\langle P_A + [r]Q_A \rangle$ and another over Bob's public key: $\phi'_A : E_B \to E_{AB} = E_B/\langle \phi_B(P_A) + [r]\phi_B(Q_A) \rangle$. Alice's public key is $\text{pk}_A = \{E_A, \phi_A(P_B), \phi_A(Q_B)\}$. Following the supersingular isogeny public-key encryption, Alice hashes her secret curve's j-invariant, $h = F(j(E_{AB}))$, and XORs this with her random message, $c = h \oplus m$. Alice then computes the shared secret by hashing her random message with her public key and encrypted j-invariant, $\text{ss}_A = H(m, \text{pk}_A, c)$. Lastly, Alice broadcasts her ciphertext which is her public key and

Table I
SUMMARY OF ROUND 2 SIKE PUBLIC PARAMETERS [14]. EACH OF THE NIST SECURITY LEVELS ARE BASED ON THE DIFFICULTY TO BREAK EXISTING CRYPTOSYSTEMS PROPOSED IN [26]. FOR INSTANCE, AES DIFFICULTY IS BASED ON EXHAUSTIVE KEY SEARCH AND SHA DIFFICULTY IS BASED ON COLLISION SEARCH. NOTE THAT SIKEp434$= 2^{216}3^{137} - 1$, SIKEp503$= 2^{250}3^{159} - 1$, SIKEp610$= 2^{305}3^{192} - 1$, SIKEp751$= 2^{372}3^{239} - 1$.

Curve: $E_0/\mathbb{F}_{p^2}: \ y^2 = x^3 + 6x^2 + x$ (All sizes in bytes).				
Parameter Set	NIST Security Level	Public Key	Cipher Text	Shared Secret
SIKEp434	1 (AES128)	330	346	16
SIKEp503	2 (SHA256)	378	402	24
SIKEp610	3 (AES192)	462	486	24
SIKEp751	5 (AES256)	564	596	32

encrypted j-invariant, ct $= \{\text{pk}_A, c\}$.

To decapsulate Alice's secret scalar, Bob decrypts the random message by computing his secret isogeny walk over Alice's public key, $\phi'_B: \ E_A \rightarrow E_{BA} = E_A/\langle \phi_A(P_B) + [s_B]\phi_A(Q_B)\rangle$. Using this secret curve's j-invariant, Bob can recover the random message, $m' = F(j(E_{BA})) \oplus c$. To ensure nothing went wrong with the key encapsulation, Bob performs the first step of encapsulation to check if the public keys match. He recalculates Alice's secret scalar, $r' = G(m', \text{pk}_B)$ and recomputes Alice's secret isogeny walk, $\phi''_A: \ E_0 \rightarrow E'_A = E_0/\langle P_A + [r']Q_A\rangle$. If the resulting public key, $\text{pk}'_A = \{E'_A, \phi''_A(P_B), \phi''_A(Q_B)\}$, matches Alice's public key then the key encapsulation was performed correctly (as well as honestly) and Bob can finalize the protocol by computing the shared secret, $\text{ss}_B = H(m', \text{pk}_A, c)$. If for any reason the public key validation fails, Bob instead uses his hidden key to compute an invalid shared secret, $\text{ss}_B = H(s, \text{pk}_A, c)$.

The instantiated version of SIKE uses a similar set of public parameters as SIDH. Notably, the SIKE round 2 specification lists four sets of public parameters: SIKEp434, SIKEp503, SIKEp610, and SIKEp751. These are intended to give a low, medium-low, medium, and high assurance of quantum security, respectively. The public primes are chosen with $\ell_A = 2$ and $\ell_B = 3$ and the hash functions F, G, H are each SHAKE256, the SHAKE function based on the Keccak sponge construction [29]. We summarize the security levels, key sizes, and ciphertext sizes in Table I. In particular, we implement SIKEp434, SIKEp503, SIKEp610, and SIKEp751, for which the SIKE proposal also provides optimized software implementations [14].

In the NIST PQC competition, each parameter set is assigned a NIST security level. The NIST security levels are based on the difficulty to break existing cryptosystems on a scale from 1-5. NIST security level 1 represents the difficulty to break AES128 with exhaustive key search and NIST level 2 represents the difficulty to break SHA256 by finding a collision. This difficulty goes in the order AES128, SHA256, AES192, SHA384, and AES256. Initially, the first round SIKE submission used the security levels 1 and 3 for schemes SIKEp503 and SIKEp751, respectively. More exploration of the problem revealed that this estimate may have been too conservative. A work by Adj *et al.* [30] suggested that a 434-bit prime gives 128-bit classical security (NIST level 1) and subsequent work by Jaques and Schanck [27] supported this proposition and gave further insights into quantum attacks on SIDH/SIKE. Lastly, a new cryptanalysis paper by Costello *et al.* [26] proposed that the parameter set SIKEp751 is sufficient for NIST security level 5 and provided metrics that SIKEp503 is approximately NIST security level 2. The results of these cryptanalytic works were addressed as part of the SIKE team's round 2 submission:

- SIKEp934 was removed.
- SIKEp503 and SIKEp751 were updated to NIST security levels 2 and 5. Their corresponding public key, ciphertext, and shared secret sizes were also updated.
- Two additional parameter sets, SIKEp434 and SIKEp610, were added at NIST security levels 1 and 3, respectively.
- Instantiated hash functions F, G, H were changed from cSHAKE256 to SHAKE256.
- The starting curve was changed from Montgomery curve coefficient $A = 0$ to $A = 6$. Torsion bases are now defined over $\mathbb{F}_{p^2}$, which slows down the first round kernel generation.

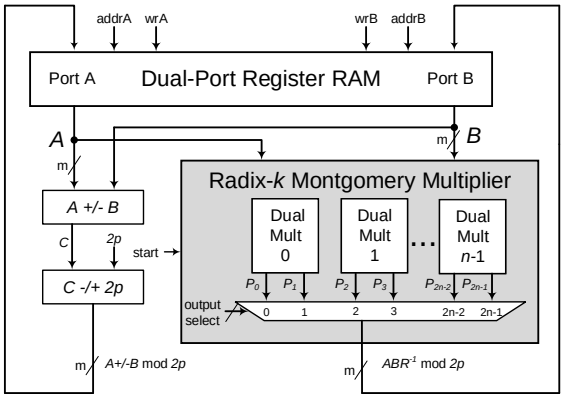

Figure 2. Proposed field arithmetic unit. This design centers on isolated field addition (left side) and multiplication (right side) pipelines. In this work, we implement over `SIKEp434`, `SIKEp503`, `SIKEp610` and `SIKEp751`.

- An additional implementation including key compression was added.

III. A Fast Isogeny Computation Accelerator

Here, we present our architecture to accelerate supersingular isogeny computations. Our methodology revolves around fast finite field arithmetic units and highly parallelized fast isogeny formula for a high performance implementation. We utilize constant-time finite-field arithmetic, constant-time isogeny and elliptic curve formulas, and constant-time ladders and large-degree isogeny algorithms, resulting in an implementation that will perform SIKE in the same amount of time, regardless of any secret inputs.

A. Fast Finite Field Arithmetic

At the lowest level of computations, elliptic curve and isogeny arithmetic is based on operations over finite fields. Specifically, since we are working on various supersingular elliptic curves, we define all arithmetic over the quadratic extension field $\mathbb{F}_{p^2}$. For isogeny-based computations, we are interested in finite field addition, subtraction, multiplication, squaring, and inversion. At an even lower level, we can build each of these operations from just addition and multiplication over $\mathbb{F}_p$. The field arithmetic unit is shown in Figure 2. The general idea is to have two separate pipelines, one for finite field addition and the other for finite field multiplication. At the top, we use a dual-port RAM block to hold the registers to feed in operands.

Single-cycle addition/subtraction unit.

Our addition/subtraction unit implements the carry-compact addition scheme from [31] to achieve a higher frequency for large additions. The carry-compact adder (CCA) greatly shortens the adder's critical path by compacting some bits of the addends which reduces the number of bits entering the carry chains. The parameters in this scheme are H (hierarchy level) and L (length of carry chain).

The first parameter H is the maximum compaction level allowed in the design. At compaction level i, 2^i bits of each addend are compacted together before being pushed into the carry chain. At compaction level 0, the bits are not compacted, and the design looks exactly like a regular ripple-carry adder. Going beyond compaction level 3 will significantly increase the design complexity of the compaction circuit which in turn increases the routing delay. Therefore, we stick to a maximum compaction level of 3.

Table II
COMPARISON OF SYSTOLIC MONTGOMERY MULTIPLIER WITH VARYING RADIX SIZES ON A VIRTEX-7 FPGA. BASED ON THE MULTIPLIER FROM [21]. INCREASING THE RADIX SIZE INCREASES THE SIZE OF MULTIPLICATIONS WITHIN A PROCESSING ELEMENT (PE). A SINGLE DSP48E CAN COMPUTE A 17X17 UNSIGNED MULTIPLICATION, TWO DSPS CAN COMPUTE A 24X24 UNSIGNED MULTIPLICATION, AND FOUR DSPS CAN COMPUTE A 34X34 UNSIGNED MULTIPLICATION. OUR SELECTED RADIX SIZES ARE BOLDED.

Radix t	#PEs w	#DSPs	Mult (cc)	Freq (MHz)	Time (ns)
SIKEp503					
16	32	64	100	207	483
17	30	60	94	198	475
22	23	92	73	169	432
23	**22**	**88**	**70**	**171**	**409**
24	21	84	67	167	401
34	15	120	49	105	467
SIKEp751					
16	48	96	148	202	734
23	33	132	103	166	620
24	**32**	**128**	**100**	**167**	**600**
29	26	208	82	123	667
34	23	184	73	122	600

The second parameter L determines how many bits to reduce the carry chain at each compaction level. On compaction level $i = 0, 1, 2, \ldots, (i+1) \times L$ positions from the MSB side and iL positions from the LSB side are used for the carry chain computation. At $L = 40$ and $H = 2$, for example, 40 MSB bits of the addends are not compacted (compaction level 0) and pushed to the carry chain. Then, the next $2 \times 40 = 80$ MSB bits and 80 LSB bits of the addends are compacted at compaction level 1 and pushed to the carry chain. Finally, the remaining bits of the addends are compacted at compaction level 2 and pushed to the carry chain since that is the maximum compaction level chosen.

According to [31], the CCA is highly susceptible to differences in routing delay depending on the parameter values chosen. Therefore, an optimal (maximum frequency for maximum performance) set of parameters is not possible without a trial and error approach. Preußer *et al.* mention that the optimal value of L is usually around 30 for every input size they have tested which narrows down the set of values to test. Since they do not go beyond 480 bits in their paper, we increased the upper bound of the values of L to test. Over the Virtex-7 FPGAs, we tested $L \in [25, 55]$ and $H \in [1, 3]$ and found $\{L, H\}$ pairs of $\{38, 3\}$, $\{39, 3\}$, $\{41, 3\}$, and $\{43, 3\}$ as the fastest results for `SIKEp434`, `SIKEp503`, `SIKEp610` and `SIKEp751`, respectively.

This addition scheme allowed us to perform a full 751-bit addition or subtraction in a single cycle, in contrast to 3 cycles in [22]. To perform $\mathbb{F}_p$ addition/subtraction, we cascade two adder/subtractor units in our "addition" pipeline and take the correct result modulo $2p$. By separating the addition and conditional subtraction portions for $\mathbb{F}_p$ addition, we isolate their computations rather than scheduling them on the same adder unit. Thus, this results in faster modular additions and less demand for resources from a scheduling point of view.

Similar to hardware architectures for elliptic curve cryptography, the critical field arithmetic unit choice is the modular multiplier. A simple multiplication between two $\mathbb{F}_p$ elements will produce an element that is twice as long as the inputs (i.e. if a, b are $\log_2\ p$ bits long, then $a \times b$ is 2 $\log_2\ p$ bits long), requiring a reduction. The two major algorithms to perform this reduction are Montgomery and Barrett reduction. There have been new proposed Barrett reduction methods for SIKE primes [32], [33], but here we primarily focused on evaluating various architectures for the well-known Montgomery multiplication [34] algorithm. Montgomery multiplication is fast and efficient because it converts expensive division operations to shift operations, which are very cheap in hardware. The only caveat to Montgomery multiplication is that both inputs must be in the Montgomery domain, which requires a few extra multiplications at the beginning and end of an algorithm. For isogeny-based operations, there are many multiplication and addition operations, so this cost is negligible.

Table III
LATENCY OF ARITHMETIC OPERATIONS IN OUR ARCHITECTURE ON A VIRTEX-7 FPGA. WE COMPARE OUR RESULTS WITH THAT OF KOZIEL *et al.* [22]. WE NOTE THAT ALTHOUGH OUR OPERATIONS GENERALLY REQUIRE FEWER CYCLES THAT OUR OPERATING FREQUENCY IS LOWER. SEE THE IMPLEMENTATION RESULTS IN SECTION V-B FOR A FULL COMPARISON.

Prime	Max Freq. (MHz)	Latency (cc)			Latency (ns)		
		$\mathbb{F}_p$ Add	$\mathbb{F}_p$ Multiplication		$\mathbb{F}_p$ Add	$\mathbb{F}_p$ Multiplication	
			Mult.	Interleave		Mult.	Interleave
SIDH Implementation by Koziel *et al.* [22]							
SIKEp503	202.1	4	100	69	19.8	495	341
SIKEp751	203.7	6	148	101	29.5	727	496
Our SIKE implementation with the same primes							
SIKEp503	171.2	2	70	49	11.7	409	286
SIKEp751	167.4		100	69	11.9	597	412

Higher radix and faster Montgomery multiplier. Our modular multiplier follows the interleaved systolic architecture from [22]. This systolic architecture computes the high-radix Montgomery product $S_{m+3} = A \times B \times R^{-1} \bmod M$, where A and B are inputs, R^{-1} is a Montgomery constant, and M is the modulus. We break this down to computing multiple products $S_{i+1} = (S_i + q_i\bar{M})/2^t + a_iB$, where $\bar{M}$ is a Montgomery constant based on the modulus, a_i is the i-th t-bit chunk of A, and $q_i = S_i \bmod 2^t$. However, rather than compute the product a_iB in one go, we compute each of its partial products over many cycles, t-bits at a time. Each processing element has its own t-bits chunk of B, so each processing element computes $S_{i+1,j} = (s_i + q_i\bar{m}_j)/2^k + a_ib_j$. Thus, each processing element is composed of two parallel radix t-bit multiplications followed by four $2t$-bit additions.

There are $w = \lceil \frac{\log_2 p}{t} \rceil$ processing elements in this systolic architecture. It takes $w+2$ cycles for a single t-bit chunk of the result to be computed as there is an initialization cycle. Since there are additional feedback and computation cycles to compute the reduction, the total computation takes $3w+4$ cycles for the entire modular multiplication to be performed. In this architecture, we shift in t-bits of the input every other cycle. With careful management of the internal registers, we can actually simultaneously fit two modular multiplications in parallel with a single multiplier. We start one multiplication on an "even" cycle and the other on an "odd" cycle. Furthermore, once the inputs are consumed in the systolic architecture's pipeline, we can interleave a new multiplication in the "even" or "odd" cycle slot, this happens after $2w+5$ cycles. With a strong choice of radix t, this scalable multiplier provides high-performance and high throughput.

However, rather than use radix $t = 16$ for each processing element as was done in [22], we experimented with various radices to find the best choice for performance, resulting in performance gains in exchange for more DSPs. The t-bit multiplications were optimized by using Xilinx FPGA's DSP48E1. A single DSP can compute a 17×17 bit unsigned multiplication, but we can also combine multiple DSPs for larger multiplications. In particular, two DSPs can compute a 24×24 bit unsigned multiplication and four DSPs can compute a 34×34 bit unsigned multiplication. In general, the larger the radix t, the fewer number of processing elements in the systolic architecture, resulting in smaller latencies. We focused our efforts on finding the sweet spot where latency and frequency produced the best performance.

In our experiments (SIKEp503 and SIKEp710 shown in Table II), we found that $t = 22$, $t = 23$, $t = 24$, and $t = 24$ were optimal for SIKEp434, SIKEp503, SIKEp610, and SIKEp751 architectures, respectively. We note that for SIKEp503, $t = 23$ is approximately the square root of 503. We chose $t = 23$ instead of $t = 24$ as our subsequent scheduling over those parameters found a 1% improvement in performance for the full SIKE protocol. For SIKEp751, there was a signficant performance hit when moving from 2 DSPs per multiplication to 4 DSPs per multiplication. However, for larger parameters we expect that 4 DSPs per processing unit will feature the greatest performance as there will be a large enough reduction in latency to counteract the critical path hit. We compare the latency of our finite-field architectures with that of Koziel *et al.* [22] in Table III.

The main configuration option for our architecture is how many replicated multipliers to include. Our addition unit is fully pipelined so it can accept a new modular addition or subtraction operation every cycle. However, our multiplier is

Algorithm 1 Right-to-left ladder to compute $x(P + [k]Q)$ [17]. Note that additions indicate differential point addition.

Input: k, a v-bit scalar, $x(P), x(Q), x(Q - P) \in E(\mathbb{F}_q)$
Output: $x(P + [k]Q)$
1. $R_0 = x(Q), R_1 = x(P), R_2 = x(Q - P)$
2. **for** i in 0 to $v - 1$ **do**
  3. **if** $k_i = 1$, **then**
    4. $R_1 = R_0 +_{(R2)} R_1$
  5. **else**
    6.$R_2 = R_0 +_{(R1)} R_2$
  7. **end if**
  8. $R_0 = [2]R_0$
9. **end for**
10. **return** $R_1 = x(P + [k]Q)$

not fully pipelined and takes many more cycles. Based on the finite field scheduling methodology we discuss in the next section, we found the best balance between area and timing results at 3 dual-multipliers for `SIKEp434`, `SIKEp503`, and `SIKEp610` and 4 dual-multipliers for `SIKEp751`.

B. Fast Parallelized Isogeny Formulas

Next comes the design of a controller that can efficiently issue instructions to the field arithmetic unit to perform the isogeny-related computations fast. To achieve this, our controller reads from a program ROM to control our addition and multiplication pipelines. We utilized a dual-port block RAM (BRAM) as our register file. This contains up to 256 registers for parallelized isogeny computations.

Montgomery curve arithmetic. For SIKE and SIDH, the top-level isogeny computations involve generating a secret kernel, $R = P + [n]Q$, (n is the private key) and then performing a large-degree isogeny over that kernel, $\phi : E \rightarrow E/\langle R \rangle$. To perform these computations efficiently, we utilize state-of-the-art formulas over Montgomery curves [35]. Montgomery curves are well-known for their extremely fast differential doubling and addition point arithmetic on the Montgomery powering ladder where the y-coordinate is not needed.

Fast kernel generation. Previously, the best known algorithm for computing $R = P + [n]Q$ was from Jao and De Feo [15] and required two diffierential point additions and one point doubling per bit in n. However, a new algorithm from Hernandez *et al.* [17] sped this up to roughly the same complexity as the Montgomery ladder. This is shown in Algorithm 1. By performing a right-to-left ladder, a step of the three-point differential ladder only requires one differential point addition and one point doubling. This ladder operates in constant-time, so the same number of steps will be taken, regardless of any secret inputs.

Fast inversion-free projective isogeny formulas. For isogeny-based computations, the fast differential point arithmetic and absence of y-coordinate also produce extremely fast isogeny formulas. The primary targets for optimization have been for $\ell_A = 2$ and $\ell_B = 3$ since they scale well for exponentially large isogenies. We opted for the fast projective isogeny formulas for degree $\ell = 3, 4$ from Costello and Hisil [36]. These are the fastest formulas in the literature. Since the SIKE parameters have even e_A, we can perform base isogenies of degree $2^2 = 4$ to reduce the number of isogeny computations. These formulas take a point of order $\ell_A^2 = 4$ and $\ell_B = 3$ to compute a mapping from one elliptic curve to a 4- or 3-isogenous curve, respectively. Originally proposed in [16], these formulas change up the "affine" representation of Montgomery curves, $E_{(a,b)} : by^2 = x^3 + ax^2 + x$, to a "projective" representation: $E_{(A:\ B:\ C)} : By^2 = Cx^3 + Ax^2 + Cx$, where C is a "projective" curve coefficient such that $a = A/C$ and $b = B/C$. This is similar to moving from affine curve

coordinates (x, y) to projective curve coefficients $(X : Y : Z)$, where $x = X/Z$ and $y = Y/Z$. In both cases, the goal is to avoid many expensive field inversions. Projective curve coefficients allow us to compute a large number of isogenies in sequence and then compute an expensive inversion at the end of a protocol operation. There are no conditional branches in these formulas, so this architecture's constant-time field arithmetic unit will always finish these formulas in the same number of clock cycles.

Large-Degree Isogeny Computation. The most expensive computation in SIKE and SIDH is the large-degree isogeny computation. Given some large-isogeny of degree ℓ^e, we can chain together isogenies of degree ℓ by computing isogenies over specific representations of the secret kernel point. For a base curve E_0 and kernel point $R_0 = R$ of order ℓ^e we compute e isogenies of degree ℓ as follows:

$$E_{i+1} = E_i/\langle \ell^{e-i-1} R_i \rangle,\ \phi_i : E_i \to E_{i+1},\ R_{i+1} = \phi_i(R_i) \tag{1}$$

This computation can be represented as traversing an acyclic graph in the shape of a triangle starting from the kernel point (R_0) to each of the leaves $(\ell^{e-i-1} R_i)$. To traverse this graph, moving left requires a point multiplication by ℓ and moving right requires an isogeny evaluation of degree ℓ. Two simple strategies to compute this are the multiplication-based and isogeny-based strategies with complexity $O(e^2)$ [6]. A much more efficient strategy comes from the insight that an optimal strategy can be composed from two optimal sub-strategies [15]. Thus, by comparing the cost of a point multiplication by ℓ and an isogeny evaluation of degree ℓ, we can find a traversal path of least cost to efficiently compute the large-degree isogeny with complexity $O(e \log e)$. This requires saving multiple pivot points, but the reduced complexity greatly brings the total computation time of large-degree isogenies down. For this implementation, we used optimal strategies found with the ratio 2:1, where a point multiplication by ℓ is twice as expensive as an isogeny evaluation by ℓ. By forcing the serial point multiplications to be more expensive, we emhasize performing more isogeny evaluations which can be effectively parallelized. Once an optimal strategy is found, this can large-degree isogeny can always be performed in the same order of point multiplications and isogeny evaluations. Since our implementation features constant-time field arithmetic and isogeny formulas, this algorithm operates in constant-time.

We note that for `SIKEp610`, e_A is an odd-power so we cannot directly compute a large-degree $4^{e_A/2}$ isogeny. We must first perform a 2-isogeny by computing $e_A - 1$ point doublings, computing the 2-isogeny, and pushing the original kernel point through the 2-isogeny. Next, we can proceed as normal with an optimized strategy of degree $(e_A - 1)/2$ over base degree $\ell = 4$. This odd power of 2 is the primary reason that encapsulation is slower than decapsulation for `SIKEp610` in our results in Table VI in Section V.

Isogeny evaluation loop unrolling. Similar to [22], we performed isogeny computations (finding a new mapping between curves) serially and isogeny evaluations (pushing a point through the new map) in parallel. By unrolling the isogeny evaluation loop up to twelve times and balancing our resource utilization, we can ensure close to maximum utilization of our memory and arithmetic pipelines. The order in which we perform the unrolled operations is determined by the availability of resources and data dependencies as determined by our scheduler. With our many registers we can balance each of the computations necessary in multiple isogeny evaluations to perform it much faster than a naive serial implementation.

Greedy scheduler. To schedule these isogeny-related computation blocks, we utilized an external scheduling script over our custom $\mathbb{F}_p$ assembly code to generate a program ROM that our hardware would read. We utilized a greedy algorithm that would track available resources each cycle. As soon as data dependencies were fulfilled and the memory and arithmetic pipelines were available we could schedule an instruction. We simply unrolled all $\mathbb{F}_{p^2}$ arithmetic into basic $\mathbb{F}_p$ operations and allowed our compiler to fit each instruction based on available resources. The order of our assembly code dictated which instructions would have priority to our architecture's resources. There are no conditional loops in this assembly code.

Scheduling for all SIKE parameter sets. In this work, we primarily optimized for one particular parameter set per implementation. Each of the isogeny formulas are identical across the parameter sets. The multiplication and addition units are generic and can be used across parameter sets so long as the controls are slightly altered. Thus, by including a large

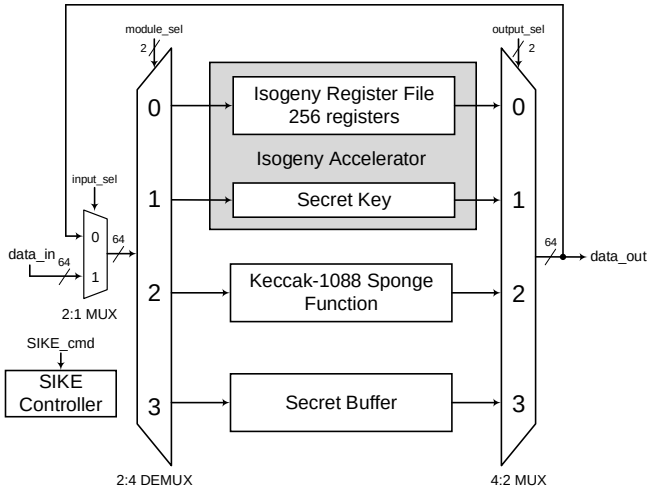

Figure 3. Proposed SIKE architecture. The architecture moves data in chunks of 64-bits to facilitate each SIKE function. The four main components are isogeny register file, secret scalars, Keccak function, and message buffer.

multiplier/adder we can support each of the smaller parameter sizes. To optimize the performance for these parameter sets, we would include a scheduling block rather than a cycle-by-cycle program ROM. This scheduling block would schedule resources on-the-fly, but would most likely not be as optimized.

IV. Upgrading an Isogeny Accelerator to a SIKE Architecture

Here, we describe how we expand our isogeny accelerator to include all components necessary for supersingular isogeny key encapsulation. The goal of this architecture was to encapsulate all isogeny, hashing, and storage functionalities needed to independently perform SIKE operations.

A. Proposed SIKE Architecture

To implement a SIKE architecture, we require a Keccak hash function and extra registers to handle the encrypted message and hidden key. Thus, we now have four different object entities to interact with: isogeny register file, secret scalars, Keccak function, and message buffer. To handle interactions between these entities, we implemented an addressing mode for the input and output of each entity, which is shown in Figure 3. This approach allowed us to move data in chunks of 64 bits from each object entity to any object entity.

Keccak sponge function. Since our isogeny accelerator already performs all necessary isogeny functions, the emphasis was on interfacing with our Keccak block. For the Keccak implementation, we reused the high-performance module provided by the Keccak team [29]. We modified it, however, for SHAKE256, which required a rate of 1088 bits per permutation. Keccak "absorbs" data in chunks of 1088 bits and then "squeezes" out data also 1088 bits at a time. For an output of size d bits, SHAKE256 provides a collision resistance of $\min(\frac{d}{2}, 256)$ bits. Each value in $\mathbb{F}_p$ was 55, 63, 77, and 94 bytes for `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751`, respectively. These are awkward divisors, so we decided to shift all values into our Keccak input buffer byte by byte. When the buffer was full, an XOR with the current state would trigger a new run of Keccak permutations.

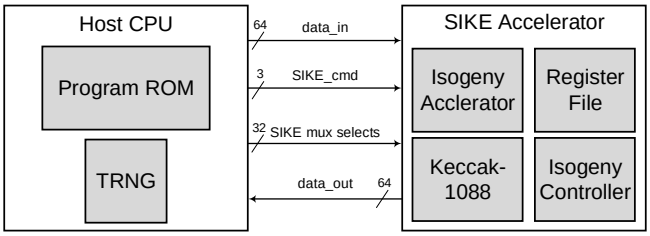

Figure 4. Proposed SIKE input/output architecture. The host initializes certain registers in the register file before initiating an operation. Once a SIKE command is sent from the host, the SIKE accelerator will perform the entire operation to completion.

We utilized two different approaches to pushing data to our Keccak block: (a) specifying a specific byte or (b) loading a 64-bit value from some other module (i.e., from register file or ciphertext) and sequentially shifting each byte. Pushing specific bytes was necessary for customization strings and 64-bit values were pushed to hash the public keys. The results from the Keccak block were then pushed to their consumer, such as the secret scalar for key encapsulation or secret hash to encrypt the random message.

Secret buffer. The secret buffer was simply $3k$ registers where $k = 128$, $k = 192$, $k = 192$, and $k = 256$ bits for `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751`, respectively. We used one chunk of k bits to hold Bob's hidden key if decapsulation failed and the other two k-bit chunks to hold Alice's random message in plaintext and ciphertext form. We converted between plaintext and ciphertext by XORing the first k-bit chunk with the first k-bit chunk from our Keccak block, as is described in SIKE.

SIKE controller. On top of the isogeny accelerator, Keccak, and secret message block, we implemented an additional controller. The primary purpose of this controller is to drive data to the Keccak block and call the isogeny accelerator functions with the correct inputs. Our SIKE control unit included a number of simple functionalities such as reading multiple consecutive words to heavily reduce the total number of instructions. For `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751`, respectively. we had a program ROM of 103, 103, 105, and 107 instructions, respectively. This also included support for producing invalid shared secrets on failed key decapsulations.

B. Additional Complexity for SIKE

To transform an isogeny accelerator to SIKE, we required a Keccak unit, secret buffer registers, and SIKE controller. If this accelerator interacted with a CPU or device that already had a SHAKE256 block, then this overhead would be greatly reduced.

The largest of these additions is the Keccak hash function. Based on the implementation from the Keccak team, our Keccak block required 1,600 registers for the Keccak state, 1,088 registers for shifting in new data, and other logic for the Keccak permutations. When synthesizing on a Virtex-7, we found no problems running the Keccak block at 200 MHz after place and route, despite performing a single Keccak permutation each clock cycle. The Keccak synthesis required 3,747 LUTs and 2,703 flip-flops on a Xilinx Virtex-7 FPGA. If the Keccak block contained the critical path, 1,600 additional registers could be placed after the ρ block, which is approximately half-way through a Keccak permutation.

For the SIKE controller, we fit all SIKE functionality in 107 32-bit instructions, thus fitting well within a 1KB ROM block. Ignoring any isogeny costs, key generation took 7 cycles, key encapsulation took 1,799 cycles, and key decapsulation took 1,813 cycles for `SIKEp503`. Compared to around a million cycles for isogeny computations, these costs are extremely small.

Table IV
AREA RESULTS OF SIKE ROUND 2 ARCHITECTURES ON TARGETED FPGAS. NOTE THAT ARTIX-7 AND VIRTEX-7 PLATFORMS UTILIZE SLICES AND ULTRASCALE+ PLATFORMS UTILIZE CLBS. FPGA RESOURCE UTILIZATION APPEARS ON THE SECOND ROW FOR AN IMPLEMENTATION. SIKEp751 SUPPORTS 8 PARALLEL MULTIPLICATIONS WHILE ALL OTHER IMPLEMENTATIONS SUPPORT 6 PARALLEL MULTIPLICATIONS.

Prime	Area				
	# FFs	# LUTs	# Slices	# DSPs	# BRAMs
Xilinx Artix-7					
SIKEp434	24,328 9.09%	21,946 16.40%	8,006 23.93%	240 32.43%	26.5 7.26%
SIKEp503	27,759 10.37%	24,610 18.39%	9,186 27.46%	264 35.68%	33.5 9.18%
SIKEp610	33,198 12.41%	29,447 22.01%	10,843 32.42%	312 42.16%	39.5 10.82%
SIKEp751	49,982 18.68%	40,792 30.49%	15,794 47.22%	512 69.19%	43.5 11.92%
Xilinx Virtex-7					
SIKEp434	23,819 2.75%	21,059 4.86%	8,121 7.50%	240 6.67%	26.5 1.80%
SIKEp503	27,609 3.19%	23,746 5.48%	8,907 8.22%	264 7.33%	33.5 2.28%
SIKEp610	33,297 3.84%	28,217 6.51%	10,675 9.86%	312 8.67%	39.5 2.69%
SIKEp751	50,079 5.78%	39,953 9.22%	15,834 14.62%	512 14.22%	43.5 2.96%
Xilinx Kintex UltraScale+					
SIKEp434	23,881 3.50%	21,657 6.35%	5,237 12.28%	240 6.80%	26.5 3.56%
SIKEp503	27,783 4.07%	24,255 7.11%	4,601 10.79%	264 7.48%	33.5 4.50%
SIKEp610	33,193 4.86%	28,758 8.43%	5,404 12.67%	312 8.84%	39.5 5.31%
SIKEp751	50,143 7.35%	40,700 11.93%	7,726 18.11%	512 14.51%	43.5 5.85%

V. SIKE ON FPGA RESULTS

In this section, we describe our SIKE implementation results on Artix-7, Virtex-7, and Kintex UltraScale+ FPGAs. We synthesized the SIKE core with Xilinx Vivado 2019.2 to a Xilinx Artix-7 xc7a200tffg1156-3 device, Xilinx Virtex-7 xc7vx690tffg1157-3 device, and Xilinx Kintex UltraScale+ xcku13p-ffve900-3-e device. All results were obtained after place-and-route.

Our accelerator interacts with a host as is shown in Figure 4. The host initializes any isogeny inputs (values $x(Q)$, $x(Q-P)$, $x(P)$, key k) which are directly accessible from the SIKE multiplexer selects when the SIKE accelerator is not running. We verified our architecture by using the Known Answer Tests (KATs) from the SIKE submission team [14]. Specifically, our testbench acted as the host CPU, initialized the input registers, ran a command, and checked that the public key, ciphertext, and shared secret matched the KAT specifications.

A. Implementation Results

Our area results are shown in Table IV and our timing results are shown in Tables V and VI. To achieve a high performance with a reasonable amount of resources, we targeted 6 and 8 multipliers in our arithmetic unit (3 and 4 dual-multipliers, respectively). More multipliers means that more multiplications can be performed in parallel, but at diminishing returns as data dependencies become the bottleneck. We chose 6 multipliers for SIKEp434, SIKEp503, SIKEp610 and

Table V
TIMING RESULTS OF SIKE ROUND 2 ARCHITECTURES ON TARGETED FPGAS. NOTE THAT SIKE TOTAL TIME INCLUDES KEY ENCAPSULATION AND DECAPSULATION.

Prime	# Mults.	Time			SIKE/s
		Freq. (MHz)	Latency ($cc \times 10^6$)	Total time (ms)	
Xilinx Artix-7					
SIKEp434	6	132.2	1.91	14.4	69.3
SIKEp503		129.9	2.35	18.1	55.4
SIKEp610		125.3	3.59	28.6	34.9
SIKEp751	8	127.0	4.55	35.8	28.0
Xilinx Virtex-7					
SIKEp434	6	168.4	1.91	11.3	88.3
SIKEp503		165.9	2.35	14.1	70.7
SIKEp610		165.8	3.59	21.6	46.2
SIKEp751	8	163.1	4.55	27.8	35.9
Xilinx Kintex UltraScale+					
SIKEp434	6	299.4	1.91	6.4	157.0
SIKEp503		305.3	2.35	7.7	130.2
SIKEp610		300.1	3.59	12.0	83.6
SIKEp751	8	296.9	4.55	15.3	65.4

Table VI
LATENCY OF SIKE OPERATIONS ON OUR SIKE ACCELERATOR. NOTE THAT THE LATENCY IS IDENTICAL FOR ARTIX-7, VIRTEX-7, AND KINTEX ULTRASCALE+. TABLE V CONTAINS THE CORRESPONDING FPGA FREQUENCIES.

Scheme	Latency ($cc \times 10^6$)			
	KeyGen	Encaps	Decaps	**total** (Encaps + Decaps)
SIKEp434	0.53	0.93	0.98	1.91
SIKEp503	0.64	1.14	1.20	2.35
SIKEp610	0.90	1.81	1.78	3.59
SIKEp751	1.25	2.21	2.34	4.55

8 multipliers for SIKEp751 as our best balance point between performance and area. As our timing results indicate, we only require a few million cycles for the SIKE scheme. For our SIKEp751 Artix-7 results, the DSP was the most utilized resource at 69.19%.

When analyzing the scaling of our architecture, the area and timing results appear to scale quadratically with the prime size. Moving from SIKEp503 with 6 multipliers to SIKEp751 with 8 multipliers (approximately 1.5 times larger public parameters) almost doubles all resources except for BRAMs. This area scaling can primarily be attributed to the quadratic area scaling of the Montgomery multiplier. Likewise, the latency increase can be derived from the superlinear latency scaling of the Montgomery multiplier ($O(m \log m)$, where $m = \log_2 p$) as well as the superlinear scaling of the large-degree isogeny ($O(e \log e)$).

In Table VII, we include a breakdown of the area consumption of our major components for our SIKEp503 implementation with 3 dual-multipliers (6 total multipliers). As can be seen, the field arithmetic unit consumes about 78% of total flip-flops and 71% of total LUTs. This is to be expected as there are multiple large multipliers that accelerate the many needed finite-field multiplications. The Keccak-1088 block was added to support fast hashing for SIKE and consumed about 10% of total flip-flops and 16% of total LUTs. In terms of memory elements, the isogeny program ROM and register file consumed about 96% of total block RAM units.

In our latency results shown in Table VI, the SIKE total latency is simply the sum of the key encapsulation and decapsulation operations. This is consistent with the methodology in the SIKE proposal [14]. Generally, for a set of SIKE

Table VII
AREA BREAKDOWN OF MAJOR COMPONENTS IN OUR SIKEp503 IMPLEMENTATION. THE CRITICAL PATH OF THIS IMPLEMENTATION IS THE MULTIPLIER.

Component	#FFs	#LUTs	#DSPs	#BRAMS
Field Arithmetic Unit	21,504	16,905	264	0
Dual-Multiplier	5,660	2,966	88	0
Adder/Subtractor	1,520	2,453	0	0
Keccak-1088	2,703	3,747	0	0
Isogeny Program ROM	0	0	0	18
Register File	0	0	0	14
Top-Level Interface	2,978	3,094	0	1
Total	27,609	23,746	264	33.5

Table VIII
AREA COMPARISON OF ISOGENY ARCHITECTURES ON A VIRTEX-7 AT APPROXIMATELY NIST SECURITY LEVEL 5 (SIKEp751).

Work	Scheme	# FFs	# LUTs	# Slices	# DSPs	# BRAMs
Koziel *et al.* [18]	SIDH	46,857	32,726	15,224	376	45.5
Koziel *et al.* [22]	SIDH	48,688	34,742	14,447	384	58.5
Jao *et al.* [14]	SIKE R1	51,914	44,822	16,756	376	56.5
Roy *et al.* [37]	SIKE R1	62,124	49,099	18,711	294	22.5
Massolino *et al.* [38] (128)	SIKE R2	7,132	10,937	3,415	57	21
Massolino *et al.* [38] (256)	SIKE R2	13,657	21,210	7,408	162	38
This work	SIKE R2	50,079	40,700	15,834	512	43.5

parameters, Bob needs to only generate a public key once. Any parties that want to exchange secrets with Bob can transmit their encapsulated keys and Bob can decapsulate them with his private key.

We chose to include the Kintex UltraScale+ results as a way to showcase progress in FPGA products for our architecture. The Virtex-7 family was released in 2010 and the Kintex UltraScale+ family first appeared in 2016 as Xilinx's mid-range family. Despite being a mid-range family, our SIKE implementation is approximately 70% faster in the Kintex UltraScale+ FPGA. Since the Kintex UltraScale+ family was released at about the same time as the NIST Round 2 optimized SW platform, Intel i7-6700, we chose to compare the total time to perform SIKE Round 2 key encapsulation and decapsulation in Table X. This FPGA implementation is about the same performance over the NIST security level 1 parameter set SIKEp434, but achieves a speedup of 1.69 over the NIST security level 5 parameter set SIKEp751. This is to be expected as the Intel computer architecture is limited to its internal register size and arithmetic units, whereas a full hardware implementation can expand its internal registers and arithmetic modules as needed.

B. Comparison to Other Isogeny Works

In Tables VIII and IX, we compare our architecture results to other Virtex-7 FPGA implementations [22], [18], [37], [38] as well as the SIKE submission [14]. Each of these works except for [38] target high-performance FPGA architectures. We note that the SIKE submission hardware implementation [14] is similar to the Koziel *et al.* [22] SIDH implementation, but includes the additional Keccak hash function and registers. Our implementation is about 14% faster than the other fastest architecture from Roy and Mukhopadhyay [37]. Our architecture also achieves the best area-time product of any of the implementations by optimizing finite field addition and multiplication, simplifying the field arithmetic unit architecture, and incorporating faster elliptic curve and isogeny formulas.

At face value, the architecture presented in this work utilizes a similar amount of resources as the SIKE submission [14] (as large multipliers dominate the area), but features a lower latency, resulting in 19.5% faster computation times despite

Table IX
TIMING COMPARISON OF ISOGENY ARCHITECTURES ON A VIRTEX-7 AT APPROXIMATELY NIST SECURITY LEVEL 5 (SIKEp751). NOTE THAT SIKE TOTAL TIME INCLUDES KEY ENCAPSULATION AND DECAPSULATION. THE AREA-TIME (AT) PRODUCT IS INCLUDED BASED ON THE NUMBER OF SLICES AND PROTOCOL TIME FOR EACH IMPLEMENTATION.

Work	Scheme	Freq. (MHz)	Cycles ($cc \times 10^6$)	Total Time (ms)	AT Product (#Slices$\times s$)
Koziel *et al.* [18]	SIDH	182.1	7.74	42.5	647
Koziel *et al.* [22]	SIDH	203.7	6.86	33.7	487
Jao *et al.* [14]	SIKE R1	198	6.60	33.4	560
Roy *et al.* [37]	SIKE R1	225.7	7.12	31.6	590
Massolino *et al.* [38] (128)	SIKE R2	152.2	27.3	179.6	613
Massolino *et al.* [38] (256)	SIKE R2	142.2	8.6	60.8	450
This work	**SIKE R2**	**163.1**	**4.54**	**27.8**	**440**

Table X
TIMING COMPARISON OF THIS WORK TO SIKE SUBMISSION'S OPTIMIZED SOFTWARE IMPLEMENTATION.

Work	Platform	SIKE (encap + decap) time (ms)			
		SIKEp434	SIKEp503	SIKEp610	SIKEp751
Jao *et al.* [14]	Intel i7-6700	6.3	9.0	16.8	25.8
This work	Kintex UltraScale+	6.4	7.7	12.0	15.3
Speedup	-	0.98	1.17	1.40	1.69

the lower maximum frequency. In terms of area, this work requires about 36% more DSPs, but also requires 23% fewer block RAMs. We emphasize that these are all a result of our design choices. By performing a higher radix Montgomery multiplication, we can perform a modular multiplication with less latency for a small hit to the frequency. Our simplification of the finite field arithmetic unit to a single $\mathbb{F}_p$ addition and $\mathbb{F}_p$ multiplication pipeline simplifies the greedy scheduling algorithm as there are only two simple operations that also reduces the number of memory calls (rather than addition, subtraction, reduction, etc. over a single adder as is the case in [14], [22]). Simpler scheduling brings the 23% reduction of RAM.

When comparing the area results of the implementations in [14] and [22], we note that the number of flip-flops increased by about 6.6% and the number of LUTs increased by 29%. We can attribute this uptick in LUTs primarily to the Keccak and multiplexer interface created by converting SIDH to SIKE, as this is a great amount of combinatorial logic.

The implementation by Massolino, Longa, Renes, and Batina [38] features a compact and scalable design. This hardware/software co-design does not target performance at all. Rather, this uses a software co-processor to issue field arithmetic and features the flexibility to run any of the four NIST SIKE Round 2 parameter sets. Since the design methodology is vastly different than our performance optimization target, a fair comparison is difficult. Our implementation does feature a better area-time product, but these implementation targets are on different ends of the spectrum.

C. Comparison to Other NIST Round 2 Candidates

Since SIKE is a Round 2 candidate in NIST's post-quantum standardization process, we compare our hardware results with the results of other candidates in Table XI. For code-based cryptography, we included hardware results for the BIKE [39] and Classic McEliece [40] schemes. For lattice-based schemes, we included hardware results for FrodoKEM [42], [41] and NTRU HPS [42] schemes.

Table XI only gives a rough estimate of hardware complexity for these quantum-resistant schemes. Different foundational problems, design rationales, FPGA platforms, and target NIST security level make a fair comparison among these hardware architectures difficult. The performance of isogeny-based key encapsulation and decapsulation operations are a few times slower than the BIKE schemes with several times more area (disregarding the difference in FPGA platforms). Classic

Table XI
HARDWARE COMPARISON OF ROUND 1 PQC SUBMISSIONS THAT HAVE MOVED ON TO ROUND 2. BIKE MEASURES TOTAL TIME WITH KEY GENERATION AND ENCAPSULATION. ALL OTHERS MEASURE TOTAL TIME WITH KEY ENCAPSULATION AND DECAPSULATION.

PQC Submission	Device	NIST Security Level	Public Key Size (Bytes)	Area					Total Time (ms)
				# FFs	# LUTs	# Slices	# DSPs	# BRAMs	
BIKE[1] [39]	Artix-7	1	2,541	2,886	5,465	1,559	0	13	10.2
McEliece[2] [40]	Virtex-7	5	1,044,992	111,299	66,615	-	0	492	1.4[3]
FrodoKEM[4][41]	Artix-7	5	21,520	5,335	14,528	4,020	16	0	1.3
FrodoKEM[5][42]	Zynq UltraScale+	5	21,520	6,610	7,015	1,215	32	17.5	4.6
NTRU HPS[5][42]	Zynq UltraScale+	3	1,230	21,410	29,389	5,913	821	2.5	0.59
SIKEp434 (this work)	Virtex-7	1	**330**	23,819	21,059	8,121	240	26.5	11.3
SIKEp503 (this work)		2	**378**	27,609	23,746	8,907	264	33.5	14.1
SIKEp610 (this work)		3	**462**	33,297	28,217	10,675	312	39.5	21.6
SIKEp751 (this work)		5	**564**	50,079	39,953	15,834	512	43.5	27.8

1. BIKE-1 (CPA security) with 2 optimization levels + hash
2. mceliece6688128 with area and time balanced
3. Key generation takes about 120 ms
4. Hardware and timing results are for decapsulation only
5. Implementation uses a hardware-software design approach

McEliece is much faster for encapsulation and decapsulation, but the key generation takes about 120 ms, public keys are over a MB, and much more FPGA area is consumed. Lattice-based schemes appear to require a quarter of the hardware resources and are about 5 to 30 times faster, with a moderate increase to public key size.

SIKE's primary advantage is that it has the smallest public keys, which we highlight in Table XI. Considering that currently deployed public keys are 32 bytes for X25519, any of these quantum schemes will raise the bar for establishing secure communications on the internet. Minimizing public key sizes are critical for reducing transmission and storage requirements. There is no clear winner among any of the NIST PQC candidates, but having significantly smaller public keys while still having competitive performance are compelling arguments for the SIKE scheme.

VI. CONCLUSION

In this work, we presented a fast and secure implementation of the SIKE protocol. Our design choices push isogeny-based computations 14% faster than the previous fastest FPGA results. We implemented fast field arithmetic units for large SIKE primes, fast isogeny arithmetic over Montgomery curves, and optimized our controller for fast isogeny formulas. Our SIKE architecture features a Keccak-centered methodology to perform key encapsulation and decapsulation in constant-time. The SIKE protocol is an IND-CCA key encapsulation mechanism featuring the smallest keys in the NIST post-quantum standardization project. This work continues to push the total time of isogeny-based computations lower in hopes that it will be standardized in the future.

One major goal of this research has been to find methods to accelerate hardware-based implementations of isogeny computations. The optimizations and architectures showcased in this work serve as a baseline for later ASIC implementations of SIKE which have the potential to save large amounts of power and energy.

REFERENCES

[1] P. W. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," in *35th Annual Symposium on Foundations of Computer Science (FOCS 1994)*, 1994, pp. 124–134.
[2] J.-M. Couveignes, "Hard Homogeneous Spaces," Cryptology ePrint Archive, Report 2006/291, 2006.
[3] A. Rostovtsev and A. Stolbunov, "Public-Key Cryptosystem Based on Isogenies," Cryptology ePrint Archive, Report 2006/145, 2006.
[4] D. X. Charles, K. E. Lauter, and E. Z. Goren, "Cryptographic Hash Functions from Expander Graphs," *Journal of Cryptology*, vol. 22, no. 1, pp. 93–113, Jan 2009.

[5] A. M. Childs, D. Jao, and V. Soukharev, "Constructing Elliptic Curve Isogenies in Quantum Subexponential Time," *Journal of Mathematical Cryptology*, vol. 8, no. 1, pp. 1–29, 2014.

[6] D. Jao and L. De Feo, "Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies," in *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011*, 2011, pp. 19–34.

[7] Y. Yoo, R. Azarderakhsh, A. Jalali, D. Jao, and V. Soukharev, "A Post-quantum Digital Signature Scheme Based on Supersingular Isogenies," in *Financial Cryptography and Data Security: 21st International Conference, FC 2017*. Cham: Springer International Publishing, 2017, pp. 163–181.

[8] S. D. Galbraith, C. Petit, and J. Silva, "Identification Protocols and Signature Schemes Based on Supersingular Isogeny Problems," in *Advances in Cryptology – ASIACRYPT 2017*, Cham, 2017, pp. 3–33.

[9] D. Jao and V. Soukharev, "Isogeny-Based Quantum-Resistant Undeniable Signatures," in *Post-Quantum Cryptography: 6th International Workshop, PQCrypto 2014*, 2014, pp. 160–179.

[10] L. Chen, S. P. Jordan, Y.-K. Liu, D. Moody, R. C. Peralta, R. A. Perlner, and D. Smith-Tone, "Report on Post-Quantum Cryptography," 2016, NIST IR 8105.

[11] R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi, "Key Compression for Isogeny-Based Cryptosystems," in *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography*, 2016, pp. 1–10.

[12] C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes, and D. Urbanik, "Efficient Compression of SIDH Public Keys," in *Advances in Cryptology – EUROCRYPT 2017: 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2017, pp. 679–706.

[13] S. D. Galbraith, C. Petit, B. Shani, and Y. B. Ti, "On the Security of Supersingular Isogeny Cryptosystems," in *Advances in Cryptology - ASIACRYPT 2016*, 2016, pp. 63–91.

[14] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, and D. Urbanik, "Supersingular Isogeny Key Encapsulation," Submission to the NIST Post-Quantum Standardization Project, 2017. [Online]. Available: https://sike.org/

[15] L. De Feo, D. Jao, and J. Plût, "Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies," *Journal of Mathematical Cryptology*, vol. 8, no. 3, pp. 209–247, Sep. 2014.

[16] C. Costello, P. Longa, and M. Naehrig, "Efficient Algorithms for Supersingular Isogeny Diffie-Hellman," in *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference*, 2016, pp. 572–601.

[17] A. Faz-Hernández, J. López, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez, "A Faster Software Implementation of the Supersingular Isogeny Diffie-Hellman Key Exchange Protocol," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1622–1636, Nov 2018.

[18] B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani, "Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA," in *Progress in Cryptology – INDOCRYPT 2016: 17th International Conference on Cryptology in India*, 2016, pp. 191–206.

[19] 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 *Cryptology and Network Security: 15th International Conference, CANS 2016*, 2016, pp. 88–103.

[20] A. Jalali, R. Azarderakhsh, and M. Mozaffari-Kermani, "Efficient Post-Quantum Undeniable Signature on 64-Bit ARM," in *Selected Areas in Cryptography – SAC 2017, 24th International Conference*, 2018, pp. 281–298.

[21] B. Koziel, R. Azarderakhsh, M. Mozaffari-Kermani, and D. Jao, "Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 1, pp. 86–99, Jan 2017.

[22] B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani, "A High-Performance and Scalable Hardware Architecture for Isogeny-Based Cryptography," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1594–1609, Nov 2018.

[23] L. D. Feo, "Mathematics of Isogeny Based Cryptography," *CoRR*, vol. abs/1711.04062, 2017. [Online]. Available: http://arxiv.org/abs/1711.04062

[24] J. H. Silverman, *The Arithmetic of Elliptic Curves*, ser. GTM. New York: Springer, 1992, vol. 106.

[25] J. Vélu, "Isogénies Entre Courbes Elliptiques," *Comptes Rendus de l'Académie des Sciences Paris Séries A-B*, vol. 273, pp. A238–A241, 1971.

[26] C. Costello, P. Longa, M. Naehrig, J. Renes, and F. Virdia, "Improved classical cryptanalysis of the computational supersingular isogeny problem," Cryptology ePrint Archive, Report 2019/298, https://eprint.iacr.org/2019/298.

[27] S. Jaques and J. M. Schanck, "Quantum cryptanalysis in the ram model: Claw-finding attacks on sike," Cryptology ePrint Archive, Report 2019/103, https://eprint.iacr.org/2019/103.

[28] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A Modular Analysis of the Fujisaki-Okamoto Transformation," in *Theory of Cryptography*, 2017, pp. 341–371.

[29] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer. (2012, May) Keccak Implementation Overview. [Online]. Available: https://keccak.team/files/Keccak-implementation-3.2.pdf

[30] G. Adj, D. Cervantes-Vázquez, J. Chi-Domínguez, A. Menezes, and F. Rodríguez-Henríquez, "On the Cost of Computing Isogenies Between Supersingular Elliptic Curves," Cryptology ePrint Archive, Report 2018/313, 2018. [Online]. Available: https://eprint.iacr.org/2018/313

[31] T. B. Preußer, M. Zabel, and R. G. Spallek, "Accelerating Computations on FPGA Carry Chains by Operand Compaction," in *2011 IEEE 20th Symposium on Computer Arithmetic*, July 2011, pp. 95–102.

[32] W. Liu, J. Ni, Z. Liu, C. Liu, and M. ONeill, "Optimized Modular Multiplication for Supersingular Isogeny Diffie-Hellman," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1249–1255, 2019.

[33] W. Liu, Z. Ni, J. Ni, C. Rafferty, and M. ONeill, "High Performance Modular Multiplication for SIDH," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2019.

[34] P. L. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[35] ——, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Mathematics of Computation*, pp. 243–264, 1987.

[36] C. Costello and H. Hisil, "A Simple and Compact Algorithm for SIDH with Arbitrary Degree Isogenies," in *Advances in Cryptology – ASIACRYPT 2017 - 23rd International Conference on the Theory and Application of Cryptology and Information Security*, 2017, pp. 303–329.

[37] D. B. Roy and D. Mukhopadhyay, "Post Quantum ECC on FPGA Platform," Cryptology ePrint Archive, Report 2019/568, 2019, https://eprint.iacr.org/2019/568.
[38] P. M. C. Massolino, P. Longa, J. Renes, and L. Batina, "A Compact and Scalable Hardware/Software Co-design of SIKE," Cryptology ePrint Archive, Report 2020/040, 2020, https://eprint.iacr.org/2020/040.
[39] N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, V. Vasseur, and G. Zémor, "Bit Flipping Key Encapsulation," Submission to the NIST Post-Quantum Standardization Project, 2017. [Online]. Available: https://bikesuite.org/
[40] D. J. Bernstein, T. Chou, T. Lange, I. v. Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, and W. Wang, "Classic McEliece: conservative code-based cryptography," Submission to the NIST Post-Quantum Standardization Project, 2017. [Online]. Available: https://classic.mceliece.org/
[41] J. Howe, M. Martinoli, E. Oswald, and F. Regazzoni, "Optimised Lattice-Based Key Encapsulation in Hardware," in *NIST Second PQC Standardization Conference*, 2019.
[42] F. Farahmand, V. B. Dang, M. Andrzejczak, and K. Gaj, "Implementing and Benchmarking Seven Round 2 Lattice-Based Key Encapsulation Mechanisms Using a Software/Hardware Codesign Approach," in *NIST Second PQC Standardization Conference*, 2019.