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

Brian Koziel[12], A-Bon Ackie[2], Rami El Khatib[2], Reza Azarderakhsh[2], and Mehran Mozaffari-Kermani[3]

[1]Texas Instruments, kozielbrian@gmail.com.
[2]CEECS Dept, FAU, {bkoziel2017, aackie, relkhatib2015, razarderakhsh}@fau.edu.
[3]CSE Dept, USF, mehran20@usf.edu.

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. Since SIKE can support static keys, we propose and implement additional differential power analysis countermeasures. We synthesized this architecture on the Xilinx Virtex-7 and Kintex UltraScale+ FPGA families. Over Virtex-7 FPGA's, our constant-time implementations are roughly 20% 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 SIKE protocol in 15.6 ms. This work continues to improve the speed of isogeny-based computations and also features the first full implementation of SIKE, with results applicable to NIST's post-quantum standardization process.

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

1 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 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. Over 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 over NIST security level 5, a 564 byte public key and 596 byte ciphertext are exchanged and a 24 byte shared secret is computed.

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 over 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 over 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 architecture presented in [22] and present a fast and secure FPGA hardware implementation that continues to push isogeny-based computations faster. We propose a fast isogeny accelerator architecture, over which we utilize fast parallelized isogeny formulas. Even with additional differential power analysis countermeasures, our constant-time implementation performs isogeny-based primitives 20% faster than the previous fastest known FPGA implementation with a better area-time product. We implement the IND-CCA secure `SIKEp503` and `SIKEp751` which conservatively provide NIST security levels 2 and 5 over large finite fields of 503 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 introduce and describe our implemented side-channel countermeasures. In Section 6, we present our FPGA results and compare to the previous state-of-the-art. Lastly, in Section 7, we conclude this paper.

2 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.

2.1 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 [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 $\mathbb{F}_q$, $\phi: E \to E'$ as a non-constant rational map from $E(\mathbb{F}_q)$ to $E'(\mathbb{F}_q)$ that preserves the point at infinity. The j-invariant of a curve acts as its unique identifier for an elliptic curve isomorphism class. An isogeny is a mapping of points from one elliptic curve to another. We can compute a unique isogeny by using Vélu's formulas [25] over a kernel, $\phi: E \to E/\langle ker \rangle$. 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 $p/12$ isomorphism classes.

Isogeny-based cryptography relies on the difficulty to compute isogenies between elliptic curves. For $\phi: E \to E'$ 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. For an unknown isogeny of degree s, the best known classical and quantum attacks to find an isogeny between two distant supersingular elliptic curves have complexity $O(\sqrt{p})$ and $O(\sqrt[3]{p})$, respectively.

2.2 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 key $n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$ and Bob likewise chooses a private 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

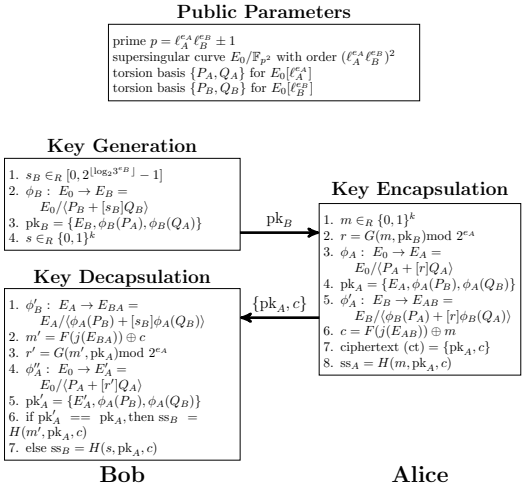

Fig. 1. Supersingular isogeny key encapsulation protocol [14].

a unique isogeny over that kernel $\phi : E \to E/\langle R \rangle$. SIDH is composed of two rounds of isogenies. In the first round, Alice computes the isogeny $\phi_A : E_0 \to E_A = E_0/\langle P_A + [n_A]Q_A \rangle$ 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 $\phi_B : E_0 \to E_B = E_0/\langle P_B + [n_B]Q_B \rangle$ 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 $\phi'_A : E_B \to E_{AB} = E_B/\langle \phi_B(P_A) + [n_A]\phi_B(Q_A) \rangle$ and Bob computes his isogeny $\phi'_B : E_A \to E_{BA} = E_A/\langle \phi_A(P_B) + [n_B]\phi_A(Q_B) \rangle$. The curves E_{AB} and E_{BA} reside in the same isomorphism class, so the j-invariant can be used as the shared secret.

The security of SIDH depends on whichever secret isogeny walk is easier to compute. Alice performs an isogeny of degree $\ell_A^{e_A}$ and Bob performs an isogeny of degree $\ell_B^{e_B}$.Thus, assuming $\ell_A^{e_A} \approx \ell_B^{e_B}$, the classical and quantum security of SIDH is approximately $O(\sqrt[4]{p})$ and $O(\sqrt[6]{p})$, respectively.

2.3 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 [26] 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 key $s_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$. The security properties of SIKE allow Bob to safely reuse any private 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 computes 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 encrypted j-invariant, $\text{ct} = \{\text{pk}_A, c\}$.

To decapsulate the key, Bob decrypts the random message by computing his secret isogeny walk over Alice's public key, $\phi'_B : \ E_A \to E_{BA} = E_A/\langle \phi_A(P_B) + [s_B]\phi_A(Q_B)\rangle$. 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 encryption to check if the public keys match. He recalculates Alice's secret scalar, $r' = G(m', \text{pk}_B)$ and recalculates Alice's secret isogeny walk, $\phi''_A : \ E_0 \to E'_A = E_0/\langle P_A + [r']Q_A\rangle$. 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 and Bob computes 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 1 specification lists three sets of public parameters: `SIKEp503`, `SIKEp751`, and `SIKEp964`. These are intended to give a low, medium, and high assurance of quantum security. The public primes are chosen with $\ell_A = 2$ and $\ell_B = 3$ and the hash functions F, G, H are each cSHAKE256, the customizable SHAKE function based on the Keccak sponge construction [28]. We summarize the security levels, key sizes, and ciphertext sizes in Table 1. In

Table 1. Summary of round 1 SIKE public parameters [14]. Each of the NIST security levels are based on the difficulty to break existing cryptosystems proposed in [27]. For instance, AES difficulty is based on exhaustive key search and SHA difficulty is based on collision search. Note that `SIKEp503`= $2^{250}3^{159}-1$, `SIKEp751`= $2^{372}3^{239}-1$, and `SIKEp964`= $2^{486}3^{301}-1$.

Curve: $E_0/\mathbb{F}_{p^2}: y^2 = x^3 + x$ (All sizes in bytes).				
Parameter Set	NIST Security Level	Public Key	Cipher Text	Shared Secret
`SIKEp503`	2 (SHA256)	378	402	16
`SIKEp751`	5 (AES256)	564	596	24
`SIKEp964`	>5 (AES256)	726	766	32

particular, we implement `SIKEp503` and `SIKEp751`, for which the SIKE proposal also provides optimized software implementations [14].

In the NIST PQC competition, each set of parameters is assigned a NIST security level. Each of 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, 3, and 5 for schemes `SIKEp503`, `SIKEp751`, and `SIKEp964`, respectively. More exploration of the problem revealed that this estimate may have been too conservative. A recent paper by Adj *et al.* [29] suggests that a 434-bit prime gives 128-bit classical security (NIST level 1) and subsequent work by Jaques and Schanck [30] supported this proposition and gave further insights into quantum attacks on SIDH/SIKE. Lastly, a new cryptanalysis paper by Costello *et al.* [27] proposes that the parameter set `SIKEp751` is sufficient for NIST security level 5 and provides metrics that `SIKEp503` is approximately NIST security level 2.

3 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.

3.1 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

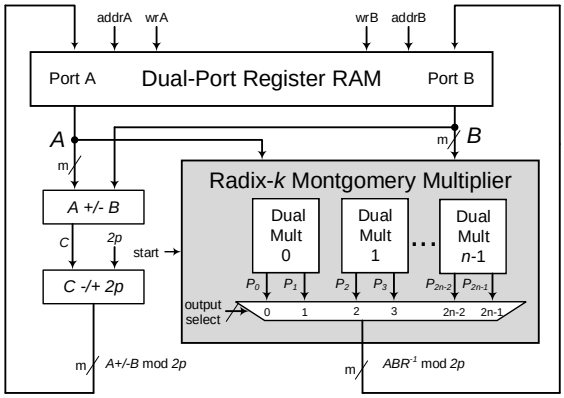

Fig. 2. Proposed field arithmetic unit. This design centers on isolated field addition (left side) and multiplication (right side) pipelines. In this work, we implement over `SIKEp503` and `SIKEp751`.

$\mathbb{F}_p$. 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]. By combining carry-look ahead and parallel prefix adders towards an FPGA target, we can greatly reduce the critical path. The parameters in this scheme are L (length of carry chain) and H (hierarchy level). For our target Virtex-7 and Ultrascale+ FPGAs , we found that the values $L = 39$, $H = 3$ and $L = 43$, $H = 3$ produce the fastest adders for `SIKEp503` 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, requiring a reduction. To perform modular multiplication efficiently, we compared various architectures for the well-known Montgomery multiplication [32] 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

Table 2. 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	#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

at the beginning and end of an algorithm. For isogeny-based operations, there are many multiplication and addition operations, so this cost is negligible.

Higher radix and faster Montgomery multiplier. Our modular multiplier follows the interleaved systolic architecture from [22]. This multiplier's systolic architecture utilizes many processing elements composed of two parallel radix t-bit multiplications followed by four $2t$-bit additions. Based on the pipeline structure of this systolic architecture, two Montgomery multiplications can be performed in parallel so long as they start on an "even" cycle and an "odd" cycle, respectively. Furthermore, once the inputs are consumed in the pipeline, we can interleave a new multiplication in the "even" or "odd" cycle slot. 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, 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 the 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.

Table 3. Latency of arithmetic operations in our architecture. 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 6.2 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

In our experiments shown in Table 2, we found that $t = 23$ and $t = 24$ were optimal for `SIKEp503` 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 `SIKEp964` or 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 3.

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 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 and 4 dual-multipliers for `SIKEp503` and `SIKEp751`, respectively.

3.2 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, $\phi : E \rightarrow$

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** $R_1 = x(P + [k]Q)$

$E/\langle R \rangle$. To perform these computations efficiently, we utilize state-of-the-art formulas over Montgomery curves [33]. 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. Further, the SIKE instantiation specified public parameters where Q is defined over $\mathbb{F}_p$, resulting in much cheaper $\mathbb{F}_p$ point double operations for public key generation computations.

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 [34]. 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. In these formulas, a projective map between curves implies that an additional curve coefficient, C, is updated after each isogeny. This allows us to compute a large number of isogenies in sequence and then compute an expensive inversion at the end of a protocol operation, similar to how projective coordinates arithmetic greatly sped up scalar point multiplications. To the best of our knowledge, this is the first hardware implementation of isogeny-based cryptography that utilizes these new and faster formulas.

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.

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 assembly code to generate a program ROM. 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.

Scheduling for all SIKE parameter sets. In this work, we primarily optimized for one particular parameter set. 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 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.

4 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.

4.1 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 [28]. We modified it, however, for cSHAKE256, 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, cSHAKE256 provides a collision resistance of $\min(\frac{d}{2}, 256)$ bits. Each value in $\mathbb{F}_p$ was 63 bytes and 94 bytes for `SIKEp503` and `SIKEp751`, respectively. This is an awkward divisor, 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.

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.

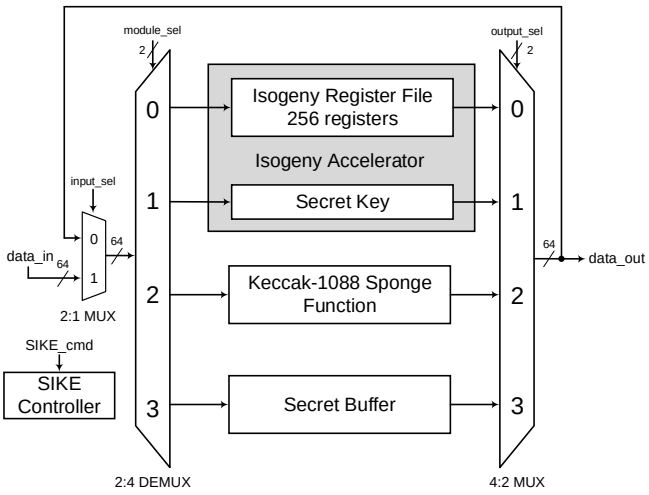

Fig. 3. Proposed SIKE architecture. The architecture moves data in chunks of 64-bits to facilitate each SIKE function. The four main components are isogeny register file, secret scalars, Keccak function, and message buffer.

Secret buffer. The secret buffer was simply $3k$ registers where $k = 192$ or $k = 256$ bits for `SIKEp503` 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 `SIKEp503` and `SIKEp751` we had a program ROM of 163 and 167 instructions, respectively. This also included support for producing invalid shared secrets on failed key decapsulations.

4.2 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 cSHAKE256 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 3,826 LUTs and 2,703 flip-flops on a Xilinx 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 with less than 170 32-bit instructions, thus fitting well within a 1KB ROM block. Ignoring any isogeny costs, key generation took 7 cycles, key encapsulation took 3,086 cycles, and key decapsulation took 3,089 cycles for `SIKEp503`. Compared to around a million cycles for isogeny computations, these costs are extremely small.

5 Side-Channel Considerations

Here, we propose the side-channel countermeasures that are included in our implementation. Although the SIKE protocol claims IND-CCA security, a real-world implementation of SIKE also emits various side-channels such as power, timing, heat, and electromagnetic radiation. Clever abuses of these side-channels have broken implementations of cryptosystems by recovering a user's private keys. Fault [35], power analysis [36], and exposure [37] attacks have been proposed in the context of isogenies. Luckily, many of the isogeny-based computations resemble elliptic curve computations, so the elliptic curve side-channel literature can be leveraged for isogenies. In what follows, we discuss several common side-channel attacks and our implemented countermeasures.

5.1 Side-Channel Model

In isogeny-based schemes like SIDH and SIKE, the scheme is effectively broken if

1. A party's private key n is recovered;
2. A party's secret kernel point $R = P + nQ$ is recovered; or
3. A party's secret isogeny walk $\phi : \ E \to E' = E/\langle R \rangle$ is recovered.

Any combination of side-channels can be used to recover these pieces of information.

In the SIKE protocol, there are three steps: key generation, key encapsulation, and key decapsulation. In this protocol, Bob starts out by choosing a private key and creates his own public key over a parameter set, which is done only *once* and then broadcast over a public channel. From there, any party can securely exchange keys with Bob by performing key encapsulation and sending Bob the ciphertext. Upon receiving the ciphertext, Bob can perform key decapsulation so that both parties agree on the same shared secret. When considering side-channels, a malicious party Eve can send a number of ciphertexts/public keys to Bob and passively observe Bob's power or timing measurements.

Algorithm 2 Proposed and implemented isogeny differential power analysis countermeasures for Bob's secret key operations in SIKE. By randomizing the representations of our secret key k and public input points, our implementation provides additional resistance against differential power analysis attacks. Note that the double point multiplication scheme will not work if order$(Q) = 2^{e_a}$. In Step 2, we represent points with projective Kummer coordinates where a point $P = (X : Z)$ and $x(P) = X/Z$.

Input: k, a v-bit scalar, $x(P), x(Q), x(Q-P) \in E(\mathbb{F}_{p^2})$
Random inputs $\lambda_0, \lambda_1, \lambda_2 =_R \{0,1\}^p$, $r =_R \{0,1\}^{64}$
Output: $x(P + [k]Q)$
1. $k' = k + r \times \text{order}(Q)$
2. $R_0 = (x(Q) \times \lambda_0 : \lambda_0)$, $R_1 = (x(P) \times \lambda_1 : \lambda_1)$,
$R_2 = (x(Q-P) \times \lambda_2 : \lambda_2)$
3. **for** i in 0 to $v + 63$ **do**
  4. **if** $k'_i = 1$, **then**
    5. $R_1 = R_0 +_{(R2)} R_1$
  6. **else**
    7. $R_2 = R_0 +_{(R1)} R_2$
  8. **end if**
  9. $R_0 = [2]R_0$
10. **end for**
11. **return** $R_1 = x(P + [k']Q) = x(P + [k]Q)$

5.2 SPA and Timing Countermeasures

Variations in timing and power information can be used to pinpoint critical pieces of secret values. For SIDH and SIKE implementations, it is typical to iterate over each bit of the private key n to perform the three-point ladder. A naive implementation may perform a step of this ladder differently if the current iterator bit is '1' or '0'.

Our implementation performs the SIKE protocol in *constant-time*. A constant set of operations are performed over `SIKEp503` and `SIKEp751`, regardless of the private keys. This implies that there will be a single timing signature from running a SIKE operation, reducing the attack surface of a malicious observer. Likewise, the power consumption signatures will also be extremely similar among runs as the same set of operations will be performed regardless of secret keys, providing an effective countermeasure against simple power analysis (SPA).

5.3 DPA Countermeasures

Differential power analysis (DPA), on the other hand, analyzes the power signatures of many different runs to find correlations in key bits and other secret information [38]. In this case, Eve poses as different identities and sends Bob many different public keys. By passively observing the power levels over many

different runs, Eve can statistically determine bits of the private key n even if the implementation is constant-time.

The primary countermeasure to DPA is to include some randomness in sensitive operations so that key correlations are much more difficult. In the SIKE protocol, the only private key is generated by Bob. Bob only needs to generate his public key once with his secret key, but he must also use this private key multiple times for each key decapsulation. In particular, we attempt to mask both the double-point multiplication and large-degree isogeny in Bob's key generation and decapsulation steps.

Since all arithmetic is defined over $\mathbb{F}_{p^2}$, we can create a random representation of the secret kernel point (in projective form) which will then help randomize the power consumption over secret isogeny walk computations. For elliptic curve scalar multiplication, Coron *et al.* [39] proposed blinding the private scalar, blinding the base point, and randomly projectivizing the base points as effective countermeasures for DPA attacks. Blinding the base point is not realistic since we are working with Montgomery differential point arithmetic, but we consider the other two further in the context of SIKE.

Scalar randomization. To generate the secret kernel in isogeny-related computations, we perform the double-point multiplication $R = P + nQ$, where n is the secret scalar. Interestingly, since points on an elliptic curve form an abelian group under addition, there are multiple possibilities for n that will still produce the same result. The general goal of scalar randomization is to blind the scalar n by adding some random multiple, r, of the curve of Q, i.e., $n' = n + r \cdot \text{order}(Q)$. For SIKE, the order of a valid Q changes based on if we are performing an isogeny over $\ell_A = 2$ over $\ell_B = 3$. For the former, the order of Q is 2^{e_A}. The latter will have order 3^{e_B}. Unfortunately, our choice of double-point multiplication in Algorithm 1 will not work with scalar randomization for 2^{e_A}. Notice that point Q has order 2^{e_A} and we double this point at each step of the ladder. Thus, for keys that are larger than 2^{e_A}, Q will be the point at infinity which will break the laddering algorithm. Luckily, the SIKE specification uses Bob's private keys over points of order 3^{e_B} and we can utilize this countermeasure. One more note is that the order 3^{e_B} does not have a simple representation, so random multiples of this order will not produce a large bias [40]. For Bob's decapsulation, we utilized scalar randomization with a random 64-bit scalar.

Projective point randomization. Another countermeasure is to randomly generate a different representation of the base point. For a base point $P = (x, y)$, we can perform a projective point randomization by finding a random $\lambda \neq 0$ and scaling $P = (\lambda x, \lambda y, \lambda)$. For isogeny-based computations, we have three input points: $P, Q, Q - P$. Thus, to be conservative, we scale each point by a random scalar. We generate three random scalars, $\lambda_0, \lambda_1, \lambda_2 \in \mathbb{F}_p$, and $\lambda_0, \lambda_1, \lambda_2 \neq 0$ and randomly project $P, Q, Q - P$, respectively. With our choice of double-point multiplication, this scaling does not affect the complexity of kernel generation as the cost of each ladder step remains the same. The main cost is generating three random numbers in $\mathbb{F}_p$.

In total, we require 1,573 and 2,317 bits of randomness for private key operations used by Bob in SIKEp503 and SIKEp751, respectively. We illustrate the use of these two countermeasures in Algorithm 2. We note that our architecture does not include a TRNG to generate these random bits. Our architecture expects this randomness to be initialized in the SIKE register file before SIKE key generation or SIKE key decapsulation as these operations will use Bob's private key. One such FPGA TRNG in the literature [41] operates over flip-flop metastability to pass the NIST statistical test suite with only 128 LUTs, 10 FFs, and 2 block ROMs.

5.4 Fault Attacks

Malicious third-parties can also perform active attacks to divulge key information from an implementation. This can be done by sending invalid public information or forcing the device into an error state with a laser. In the SIDH/SIKE setting, a malicious third party can send malicious public keys or force an accelerator to fail, potentially revealing critical internal information.

Based on the SIKE protocol, any active attack that attempts to disrupt Bob's key decapsulation is almost certain to cause a failed key decapsulation as there is an integrity check to check that the other party's public key was honestly generated. In this case, Bob will know that the key exchange failed and act accordingly. The primary fault attack that we choose to protect against is the loop-abort fault proposed in [35]. This loop-abort attack reveals portions of Bob's secret isogeny walk by forcing Bob to prematurely end the large-degree isogeny walk. If Bob unknowingly uses the shared secret after loop-aborting, then Bob unknowingly reveals a portion of his secret isogeny walk.

Since the large-degree isogeny is an iterative computation, a loop counter keeps track of how many isogenies have been computed. This loop-abort attack faults the loop counter so that the isogeny computation ends prematurely, and a partial isogeny walk public key is handed out. Again, this attack will produce a failed decapsulation and be protected, but the countermeasure is to include redundant loop counters to make the attack significantly harder. Such a countermeasure is very cheap in hardware gates, so this was implemented.

6 SIKE on FPGA Results

In this section, we describe our SIKE implementation results on a Virtex-7 FPGA and Kintex UltraScale+ FPGA. We synthesized the SIKE core with Xilinx Vivado 2015.4 to a Xilinx Virtex-7 xc7vx690tffg1157-3 device and Xilinx Vivado 2018.2 to a Xilinx Kintex UltraScale+ xcku13p-ffve900-3-e. 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) as well as any randomness inputs $(\lambda_0, \lambda_1, \lambda_2)$ 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

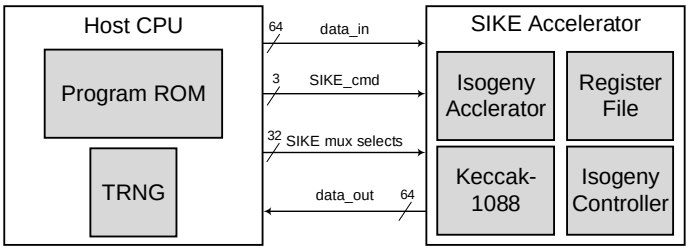

Fig. 4. Proposed SIKE input/output architecture. The host initializes certain registers in the register file before initiating an operation. Once a SIKE command is sent from the host, the SIKE accelerator will perform the entire operation to completion. In order to provide differential power analysis countermeasures, the host must initialize specific SIKE registers with some aspect of randomness.

Table 4. Area results of SIKE architectures on targeted FPGAs. Note that slices are utilized on Virtex-7 platforms and CLBs are utilized on UltraScale+ platforms. FPGA resource utilization appears on the second row for an implementation.

Prime	# Mults.	Area				
		# FFs	# LUTs	# Slices	# DSPs	# BRAMs
Xilinx Virtex-7						
`SIKEp503`	6	26,971	25,094	9,514	264	34
		3.11%	5.79%	8.78%	7.33%	2.31%
`SIKEp751`	8	50,390	45,893	17,530	512	43.5
		5.82%	10.59%	16.19%	14.22%	2.96%
Kintex UltraScale+						
`SIKEp503`	6	27,122	25,041	4,461	264	34
		3.97%	7.34%	10.35%	7.48%	4.57%
`SIKEp751`	8	49,016	39,810	7,541	512	43.5
		7.18%	11.66%	17.68%	14.51%	5.85%

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.

6.1 Implementation Results

Our area results are shown in Table 4 and our timing results are shown in Tables 5 and 6. 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 `SIKEp503` and 8 multipliers for `SIKEp751` as our best balance point between performance and area. As our tim-

Table 5. Timing results of SIKE architectures on targeted FPGAs. Note that SIKE total time includes key encapsulation and decapsulation.

Prime	# Mults.	Time			
		Freq. (MHz)	Latency ($cc \times 10^6$)	Total time (ms)	SIKE/s
Xilinx Virtex-7					
SIKEp503	6	171.2	2.33	13.6	73.5
SIKEp751	8	167.4	4.51	26.9	37.1
Kintex UltraScale+					
SIKEp503	6	288.0	2.33	8.1	123.5
SIKEp751	8	289.3	4.51	15.6	64.1

Table 6. Latency of SIKE operations on our SIKE accelerator. Note that the latency is identical for Virtex-7 and Kintex UltraScale+. Table 5 contains the corresponding FPGA frequencies.

	Latency ($cc \times 10^6$)			
Scheme	KeyGen	Encaps	Decaps	**total**(Encaps + Decaps)
SIKEp503	0.64	1.12	1.21	2.33
SIKEp751	1.24	2.17	2.33	4.51

ing results indicate, we only require a few million cycles for the SIKE scheme. For our SIKEp751 Virtex-7 and Ultrascale+ results, the DSP was the most utilized resource at 14.5%.

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 extrapolating these results, we expect the total time of public parameters of 434 bits (which [27] estimates is NIST security level 1) to be around 10 ms and 6 ms for our Virtex-7 and Kintex UltraScale+ implementations, respectively, by using the ratio $(\frac{434}{503})^2 \approx 0.75$.

In Table 7, 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 81% of total flip-flops and 65% 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 15% of total LUTs. In terms of memory

Table 7. 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	21,779	16,777	264	0
Dual-Multiplier	5,752	2,952	88	0
Adder/Subtractor	1,520	2,453	0	0
Keccak-1088	2,703	3,826	0	0
Isogeny Program ROM	0	0	0	18
Register File	0	0	0	14.5
Total	26,971	25,893	264	34

Table 8. Area comparison of isogeny architectures on a Virtex-7 at approximately NIST security level 5 (`SIKEp751`). Our implementation includes DPA countermeasures.

Work	Scheme	# FFs	# LUTs	# Slices	# DSPs	# BRAMs
[18]	SIDH	46,857	32,726	15,224	376	45.5
[22]	SIDH	48,688	34,742	14,447	384	58.5
[14]	SIKE	51,914	44,822	16,756	376	56.5
This work	**SIKE**	**50,390**	**45,893**	**17,530**	**512**	43.5

elements, the isogeny program ROM and register file consumed about 96% of total block RAM units.

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.

In our latency results shown in Table 6, 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 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.

6.2 Comparison to Other Isogeny Works

In Tables 8 and 9, we compare our architecture results to the previous state-of-the-art on the Virtex-7 FPGA [22,18] as well as the SIKE submission [14]. Each of these works 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 architecture achieves about 20% faster results with a better area-time product by optimizing finite field addition and multiplication, simplifying the field arithmetic unit architecture, and incorporating faster elliptic curve and

Table 9. Timing comparison of isogeny architectures on a Virtex-7 at approximately NIST security level 5 (`SIKEp751`). Our implementation includes DPA countermeasures. 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$\times s$)
[18]	SIDH	182.1	7.74	42.5	647
[22]	SIDH	203.7	6.86	33.7	487
[14]	SIKE	198	6.60	33.4	560
This work	**SIKE**	**167.4**	**4.51**	**26.9**	**472**

Table 10. 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. BIKE is on an Artix-7 FPGA and all other implementations are on a Virtex-7 FPGA

PQC Submission	NIST Security Level	Public Key Size (Bytes)	Area			Total Time (ms)
			# FFs	# LUTs	# Slices	
BIKE[1] [42]	1	2,541	-	-	1,559	10.2
McEliece[2] [43]	5	1,044,992	111,299	66,615	-	1.4[3]
`SIKEp751` [14]	5	**564**	51,914	44,822	16,756	33.4
`SIKEp503` (this work)	2	**378**	26,971	25,893	9,514	13.6
`SIKEp751` (this work)	5	**564**	50,390	45,893	17,530	26.9

1. BIKE-1 (CPA security) with 2 optimization levels + hash
2. mceliece6688128 with area and time balanced
3. Key generation takes about 120 ms

isogeny formulas. Since we implemented SIKE, we also included timing attack, differential power analysis, and fault attack countermeasures.

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 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. Lastly, we emphasize that our implementation includes two different differential power analysis countermeasures.

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 the Keccak and multiplexer interface created by converting SIDH to SIKE, as this is a great amount of combinatorial logic.

6.3 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 10. Unfortunately, the majority of Round 2 candidates do not include a hardware implementation. We included hardware results for the BIKE [42] and Classic McEliece [43] schemes, both based on code-based cryptography.

Table 10 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 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. Although no FPGA results were included in any lattice-based schemes, it is expected that the performance and area will be an order of magnitude better than SIKE.

SIKE's primary advantage is that it has the smallest public keys, which we highlight in Table 10. 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.

7 Conclusion

In this work, we presented a fast and secure implementation of the SIKE protocol. Our design choices push isogeny-based computations 20% faster than the previous fastest FPGA results. We implemented 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. Our constant-time implementation utilizes DPA countermeasures to help protect static keys. 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.

Our future works focus on further characterization of the performance and security of this architecture. Depending on the results of the first round of NIST's standardization project, the public parameters of SIKE may be revised. In particular, if the security analysis of Adj *et al. [29]* and Costello *et al. [27]* holds, then 434-bit primes would provide NIST level 1 security (AES128). Such a change would further reduce the area and timing results of our architecture (751-bit prime down to 434-bit prime). After that, designing, implementing, and testing our device over ASIC devices is a next step. With DPA equipment at hand, our goal is to evaluate the leakage of naive and countermeasure-included versions of our architecture. Comparing such results to past elliptic curve cryptography implementations serves to give confidence in the leakage of isogeny-based cryptography. Lastly, we plan to apply our architecture to CSIDH [44], which utilizes a variant of the supersingular isogeny problem for smaller keys at the cost of longer latency.

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. D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A Modular Analysis of the Fujisaki-Okamoto Transformation," in *Theory of Cryptography*, 2017, pp. 341–371.
27. 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.

28. 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
29. 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
30. 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.
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. P. L. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.
33. ——, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Mathematics of Computation*, pp. 243–264, 1987.
34. 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.
35. A. Gélin and B. Wesolowski, "Loop-Abort Faults on Supersingular Isogeny Cryptosystems," in *Post-Quantum Cryptography : 8th International Workshop, PQCrypto 2017*, 2017, pp. 93–106.
36. B. Koziel, R. Azarderakhsh, and D. Jao, "Side-Channel Attacks on Quantum-Resistant Supersingular Isogeny Diffie-Hellman," in *Selected Areas in Cryptography – SAC 2017, 24th International Conference*, 2018, pp. 64–81.
37. ——, "An Exposure Model for Supersingular Isogeny Diffie-Hellman Key Exchange," in *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018*, 2018, pp. 452–469. [Online]. Available: https://doi.org/10.1007/978-3-319-76953-0_24
38. P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology — CRYPTO' 99*, 1999, pp. 388–397.
39. J.-S. Coron, "Resistance Against Differential Power Analysis For Elliptic Curve Cryptosystems," in *Cryptographic Hardware and Embedded Systems: First International Workshop*, 1999, pp. 292–302.
40. K. Okeya and K. Sakurai, "Power Analysis Breaks Elliptic Curve Cryptosystems Even Secure against the Timing Attack," in *Progress in Cryptology — INDOCRYPT 2000*, 2000, pp. 178–190.
41. M. Majzoobi, F. Koushanfar, and S. Devadas, "FPGA-Based True Random Number Generation Using Circuit Metastability with Adaptive Feedback Control," in *Cryptographic Hardware and Embedded Systems – CHES 2011*, 2011, pp. 17–32.
42. 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/
43. 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/
44. W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes, "CSIDH: An Efficient Post-Quantum Commutative Group Action," Cryptology ePrint Archive, Report 2018/383, 2018.