Safe-Error Attacks on SIKE and CSIDH

Fabio Campos*, Juliane Krämer †, and Marcel Müller†

* Max Planck Institute for Security and Privacy, Bochum, Germany
campos@sopmac.de
† QPC, Technische Universität Darmstadt, Germany
{juliane,marcel}@qpc.tu-darmstadt.de

***Abstract*—The isogeny-based post-quantum schemes SIKE (NIST PQC round 3 alternate candidate) and CSIDH (Asiacrypt 2018) have received only little attention with respect to their fault attack resilience so far. We aim to fill this gap and provide a better understanding of their vulnerability by analyzing their resistance towards safe-error attacks. We present four safe-error attacks, two against SIKE and two against a constant-time implementation of CSIDH that uses dummy isogenies. The attacks use targeted bitflips during the respective isogeny-graph traversals. All four attacks lead to full key recovery. By using voltage and clock glitching, we physically carried out two of the attacks - one against each scheme -, thus demonstrate that full key recovery is also possible in practice.**

***Index Terms*—post-quantum cryptography, isogeny-based cryptography, fault attacks**

I. Introduction

The youngest field of post-quantum cryptography that is studied within NIST's standardization process is isogeny-based cryptography. Isogeny-based cryptography was first described in 2006 [1, 2]. De Feo et al. presented a fast cryptographic scheme based on isogenies, named SIDH (Supersingular Isogeny Diffie-Hellman), in 2011 [3]. One of the schemes submitted to NIST's standardization process is SIKE (Supersingular Isogeny Key Encapsulation) [4]. SIKE uses SIDH to create a key encapsulation mechanism. SIKE was selected as NIST round 3 alternate candidate. The alternate candidates are those schemes which are very promising, but need to be further studied before being considered for standardization. In 2018, Castryck et al. presented another isogeny-based system, called CSIDH (Commutative Supersingular Isogeny Diffie-Hellman) [5]. Unlike SIKE, CSIDH is non-interactive, making it a potential drop-in replacement for current Diffie-Hellman schemes. CSIDH has not been submitted to NIST's standardization process because it was designed only after the submission deadline had passed. Although recently the actual security of the suggested CSIDH parameters against quantum attacks was questioned [6, 7], CSIDH still is a promising and widely discussed isogeny-based scheme. The quantum attacks show that the young field of isogeny-based cryptography has not been sufficiently studied with respect to (quantum) cryptanalysis yet. Also, the physical security of isogeny-based schemes has not been sufficiently studied yet.

In this work, we analyze the physical security of SIKE and CSIDH. Physical attacks allow attackers to deduce secret information of an algorithm by observing or modifying the platform it operates on. In a passive (or side-channel) attack, the attacker analyzes physical information that they can measure while cryptographic operations are computed. In an active attack, on the other hand, the attacker directly interacts with the running algorithm, causing a change in its operations through which information is extracted. Hence, active attacks are also called fault attacks.

Analyzing SIKE and CSIDH with respect to a specific fault attack is the focus of this work. We analyze both schemes regarding their vulnerability towards *safe-error attacks*. This method was first published in attacks by Yen and Joye in 2000 [8]. Yen and Joye have proposed attacks on smart cards using a square-and-multiply algorithm. They suggested that by inducing transient faults, an implementation leaks one bit of information by observing if its result is correct. Shortly after, Joye and Yen applied safe-error attacks on the Montgomery ladder, showing that by perturbing the memory of a running Montgomery ladder computation one can deduce one bit of secret information [9]. Safe-error attacks are particularly interesting because even if the algorithm were to detect an error in its operation, it will still leak information. Hence, standard countermeasures, like checking for faults and outputting a random value in case a fault was detected, still provide the attacker with information and therefore are not sufficient to protect an implementation against safe-error attacks.

Unfortunately, safe-error attacks are not taken seriously enough: Some of our attacks are similar to attacks that have long been known in the ECC community, e.g., [8, 9]. Our work shows that recent implementations of isogeny-based schemes nevertheless do not provide protection against them.

Our Contribution. The focus of this work is to analyze SIKE and CSIDH with respect to safe-error attacks. To the best of our knowledge, SIKE has not been studied with respect to these attacks before.

We develop attack scenarios for SIKE and CSIDH and demonstrate the feasibility of the presented safe-error attacks by performing practical experiments. The experiments were performed against C implementations of SIKE and CSIDH on a ChipWhisperer board with an ARM Cortex-M4 processor as target core. The implementation of CSIDH that we attacked is a constant-time implementation based on dummy isogenies. We achieve full key recovery of all n bits of the secret key within $O(n)$ interactions for two of the four attacks laid out

in this paper. We discuss possible countermeasures and their performance impact. The code used for this work is available[1] in the public domain, which includes the modified CSIDH and SIKE Cortex-M4 implementation and all attack scripts.

The attack against SIKE that we carried out practically can analogously be applied to B-SIDH [10].

Related work. Although isogeny-based cryptography provides promising candidates for quantum-resistant public-key schemes, only few results regarding the physical security of isogeny-based cryptography and SIDH [11–14], SIKE [15], and CSIDH [16–18] exist. Galbraith et al. presented the first fault attack on SIDH, together with corresponding countermeasures [12]. In [11], Koziel et al. propose different zero-value attacks on SIDH. Based on loop-abort fault injection, Gélin and Wesolowski presented side-channel and fault attacks against isogeny-based primitives [13]. Ti proposes a fault attack on SIDH by changing the base point to a random point via fault injection [14]. The only published physical attack on SIKE so far is a power side-channel attack exploiting differences in calculations depending on the secret key [15]. CSIDH has been analyzed for potential attacks by Cervantes-Vázquez et al. [17]. A recent work by Campos et al. presents safe-error and further fault attacks, together with countermeasures, on a constant-time CSIDH implementation with dummy isogenies [16]. As another countermeasure, LeGrow and Hutchinson suggest to randomize the order of execution of isogenies [18].

Organization. In Section II, we present necessary background on SIKE, CSIDH, and safe-error attacks. In Sections III and IV, we present safe-error attacks on SIKE and CSIDH, respectively. In Section V, we explain how to perform the described safe-error attacks on a real device and present full key recovery. We discuss possible countermeasures in Section VI and conclude this work in Section VII.

II. Background

We first discuss implementation details of SIKE and CSIDH. For readers not familiar with isogenies, we refer to [19]. Afterwards, the introduction to safe-errors shows the pattern common to the attacks and how they work.

A. SIKE

SIKE (Supersingular Isogeny Key Encapsulation) is an interactive key encapsulation using supersingular elliptic curves [20]. SIKE has passed into the third round of the NIST process[2] as alternate candidate for future standardization. To achieve the goal of becoming standardized it will need to be studied further, especially with respect to efficiency improvements and all aspects of misuse resistance. SIKE uses SIDH internally, and SIDH will be the main target of the attacks presented in the following section. For a detailed overview of SIDH as used in SIKE, we refer to [20].

[1] https://github.com/Safe-Error-Attacks-on-SIKE-and-CSIDH/SEAoSaC

[2] https://csrc.nist.gov/news/2020/pqc-third-round-candidate-announcement

SIDH is constructed as follows: A public prime $p = 2^{e_2}3^{e_3} - 1$ such that $2^{e_2} \approx 3^{e_3}$ is chosen, as well as two points on the torsion group associated to their base: $P, Q \in E_0[2^{e_2}]$ or $E_0[3^{e_3}]$. These represent the respective public generators. The rest of the algorithm is done in $\mathbb{F}_{p^2}$. At the start of the exchange, each party agrees on picking a base of either 2 or 3 as long as they differ between them. Afterwards, each party generates a private key $\mathrm{sk} \in \mathbb{F}_{p^2}$. Of note here is that in the efficient implementation of [20], three points are used. The third point is $R = P - Q$ and is used to speedup the computation through a three-point ladder [21](cf. Algorithms 1 and 4 and Listing 1). Using these generators as well as their private key, each party then computes their public curve E_2 or E_3. This curve is calculated through a chain of e_2 2-isogenies, or e_3 3-isogenies respectively. Each isogeny uses a generator of the form $\langle P + [\mathrm{sk}]Q \rangle$ as the kernel. The projection of the other party basis point and this curve are then sent to the other party, where the same procedure is repeated to arrive at the curve $E_{2/3}$ and $E_{3/2}$. These two curves are isomorphic to each other and thus the parties have arrived at a shared secret: the j-invariant of $E_{2/3}$ and $E_{3/2}$, respectively.

Listing 1. LADDER3PT – SIKE

```
352  // Main loop
353  for (i = 0; i < nbits; i++) {
354      bit = (m[i >> LOG2RADIX] >> (i &
           (RADIX-1))) & 1;
355      swap = bit ^ prevbit;
356      prevbit = bit;
357      mask = 0 - (digit_t)swap;
358
359
360      swap_points(R, R2, mask);
361      xDBLADD(R0, R2, R->X, A24);
362      fp2mul_mont(R2->X, R->Z, R2->X);
363  }
```

The submitted implementation from round 3 is constant-time and already includes several countermeasures against fault attacks. The implementation is secure against the attack presented in Section III-A, but vulnerable to the second one as presented in Section III-B.

B. CSIDH

CSIDH (Commutative Supersingular Isogeny Diffie-Hellman) describes a non-interactive key exchange using supersingular elliptic curves [5]. For a more detailed overview of the key exchange, we refer to [5].

CSIDH is constructed as follows: A prime p is chosen of the form $p = 4 \cdot \ell_1 \cdots \ell_n - 1$, where the ℓ_i are small pairwise distinct odd primes. The rest of the algorithm is computed in $\mathbb{F}_p$. The algorithm uses elliptic curves in Montgomery form: $E_0 : y^2 = x^3 + Ax^2 + x$. To begin, each party generates a secret key $(e_1, \ldots, e_n)$, where each e_i is sampled uniformly random from the interval $[-m, m]$ with $m \in \mathbb{N}$. The key exchange is then prepared by calculating the elliptic curve associated with the secret key: For each e_i a total of $sign(e_i)$ ℓ_i-isogenies

Algorithm 1: xDBLADD

function xDBLADD
- **Input:** $(X_P : Z_P), (X_Q, Z_Q), (X_{Q-P} : Z_{Q-P})$, and $(a_{24}^+ : 1)$ $(A + 2C : 4C)$
- **Output:** $(X_{[2]P} : Z_{[2]P}), (X_{P+Q}, Z_{P+Q})$
- $t_0 \leftarrow X_P + Z_P$
- $t_1 \leftarrow X_P - Z_P$
- $X_{[2]P} \leftarrow t_0^2$
- $t_2 \leftarrow X_Q - Z_Q$
- $x_{P+Q} \leftarrow X_Q + Z_Q$
- $Z_{[2]P} \leftarrow t_1^2$
- $t_1 \leftarrow t_1 \cdot X_{P+Q}$
- $t_2 \leftarrow X_{[2]P} \cdot -Z_{[2]P}$
- $X_{[2]P} \leftarrow X_{[2]P} \cdot Z_{[2]P}$
- $X_{P+Q} \leftarrow a_{24}^+ \cdot t_2$
- $Z_{P+Q} \leftarrow t_0 - t_1$
- $Z_{[2]P} \leftarrow X_{P+Q} + Z_{[2]P}$
- $X_{P+Q} \leftarrow t_0 + t_1$
- $Z_{[2]P} \leftarrow Z_{[2]P} \cdot t_2$
- $Z_{P+Q} \leftarrow Z_{P+Q}^2$
- $X_{P+Q} \leftarrow X_{P+Q}^2$
- $Z_{P+Q} \leftarrow X_{Q-P} \cdot Z_{P+Q}$
- $X_{P+Q} \leftarrow Z_{Q-P} \cdot X_{P+Q}$
- **return** $(X_{[2]P} : Z_{[2]P}), (X_{P+Q}, Z_{P+Q})$

Algorithm 2: CSIDH Algorithm by Onuki et al.

Input: $A \in \mathbb{F}_p, m \in \mathbb{N}$, a list of integers $(e_1, \ldots, e_n) \in [-m, m]^n$ and n distinct odd primes $\ell_1, \ldots, \ell_n$ s.t. $p = 4 \prod_i \ell_i - 1$.

Output: $B \in \mathbb{F}_p, m \in \mathbb{N}$ s.t. $E_B = (\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_2}) * E_A$, where $\mathfrak{l}_i = (\ell_i, \pi - 1)$ for $i = 1, \ldots, n$, and π is the p-th power Frobenius endomorphism of E_A.

Set $e'_i = m - |e_i|$ for $i = 1, \ldots, n$
while *some* $e_i \neq 0$ *or* $e'_i \neq 0$ **do**
- Set $S = \{i | e_i \neq 0 \text{ or } e'_i \neq 0\}$
- Set $k = \prod_{i \in S} \ell_i$
- Generate points $P_0 \in E_A[\pi + 1]$ and $P_1 \in E_A[\pi - 1]$ by Elligator
- Let $P_0 \leftarrow [(p+1)/k]P_0$ and $P_1 \leftarrow [(p+1)/k]P_1$
- **for** $i \in S$ **do**
  - Set s the sign bit of e_i
  - Set $Q = [k/\ell_i]P_s$
  - Let $P_{1-s} \leftarrow [\ell_i]P_{1-s}$.
  - **if** $Q \neq \infty$ **then**
    - **if** $e_i \neq 0$ **then**
      - Compute an isogeny $\phi : E_A \to E_B$ with $\ker \phi = \langle Q \rangle$
      - Let $A \leftarrow B, P_0 \leftarrow \phi(P_0), P_1 \leftarrow \phi(P_1)$, and $e_i \leftarrow e_i - 1 + 2s$
    - **else**
      - Dummy computation
      - Let $A \leftarrow A, P_s \leftarrow [\ell_i]P_s$, and $e'_i \leftarrow e'_i - 1$.
  - Let $k \leftarrow k/\ell_i$
- **return** A

have to be calculated. The sign of e_i represents the direction taken in the respective ℓ_i-isogeny graph. As the composition of isogenies is commutative, each computed curve will be isomorphic no matter in which order they are calculated. The isognies are then chained to compute the public curve associated to the secret key: $E_0 \xrightarrow{(e_1,\ldots,e_n)} E_A$. Bob does the same to calculate E_B. The parameter of the curves E_A and E_B correspond to the public keys and are then exchanged and each party repeats their isogeny calculation using the other's public key as the starting curve: Alice calculates $E_B \xrightarrow{(e_1,\ldots,e_n)} E_{BA}$ and Bob calculates E_{AB} in a similar fashion. The final curves E_{BA} and E_{AB} are the same, and the shared secret is the A parameter of this curve in montgomery form.

The straightforward implementation of the algorithm would be highly variable in time, since different amounts of isogenies need to computed, depending on the secret key. It would be easy for an attacker to trace the amount of isogenies calculated and their degree as isogenies with a larger degree require more computational effort. In 2019 Meyer et al. have presented a constant-time implementation of CSIDH [22]. The authors tackle this issue by making the amount of isogeny evaluations constant, thus only leaking the degree of the isogenies themselves and not the exact number of them. This follows from the aforementioned fact that higher degree isogenies take longer to construct and, e.g., could be recovered through a timing attack. They achieve this by calculating "dummy" isogenies which serve as extra computational time to thwart timing attacks from finding the real amount of isogenies of a given degree. Further, they change the interval from which the secret key parts are sampled from $[-m, m]$ to $[0, 2m]$ so that an attacker cannot tell apart secret keys with unbalanced positive and negative parts. Unfortunately, these dummy calculations have added a new attack vector: loop-abort attacks. Such an attack was first described in passing by Cervantes-Vázquez et al. in [17]. In [16] the approach using dummy isogenies has been further refined. Campos et al. analyzed the constant-time implementation for fault-injection attacks. This resulted, among others, in added safeguards to the point evaluation and codomain curve algorithm. However, these safeguards do not protect against the attack described in Section IV-A, as the attacker assumed in this paper has a different threat model.

Following [22] Onuki et al. proposed to speed-up the implementation by reverting the secret key part interval to $[-m, m]$ and guarding against unbalanced keys by using two points instead of one [23]. This change, however, has introduced a possible new attack vector as described in Section IV-B.

C. Safe-Error Attacks

In [8], Yen and Joye introduce a new category of active attacks, so called safe-error attacks. In this kind of attacks, the adversary uses fault injections to perturb a specific memory location with the intent of not modifying the final result of the computation: the algorithm may overwrite or throw away modified values, making them "safe errors". The presence or absence of an error then gives insight into which codepath the algorithm executed. Two kinds of safe-error attacks exist: in a memory safe-error (M safe-error) attack, the attacker modifies the memory, i.e., in general these attacks focus on specific implementations [9, 24, 25]. In a computational safe-error (C safe-error) attack, however, the computation itself is attacked through, e.g., skipping instructions. Hence, C safe-error attacks rather target algorithmic vulnerabilities [9, 25].

The general construction of a safe-error attack is as follows:

Algorithm 3: A toy algorithm vulnerable to a variable-access attack

Input: S the n-bit secret key
Output: a public message M

$M \leftarrow 1$
$K \leftarrow 0$
$P \leftarrow 0$
for $i \in 0..n$ **do**
  if $S_i = 0$ **then**
    $K \leftarrow \text{calculate}(S_i, P, K)$
  else
    $P \leftarrow \text{calculate}(S_i, K, P)$
  $M \leftarrow M + K * P$
return M

Suppose an algorithm iterates over secret data. It then branches and does slightly different calculations depending on whether a given bit in the secret data is equal to 0 or 1. The algorithm presented in Algorithm 3 has been secured against timing side-channel attacks, and takes the same time in each branch. This predictability, enforced to thwart timing attacks, makes safe-error attacks easier to carry out, as they require timed fault-injections. Such measures, that are intended to add security or efficiency, but enable new attacks, are commonly called footguns[3] and need to be avoided unless their implications are fully understood. In implementations, explicit branching on secret data is usually avoided. However, the different memory access patterns still occur due to the structure of the respective algorithm. As we show in Section III-A, using a constant time swap algorithm instead of condition branching is not sufficient and may even provide an additional attack vector.

Analyzing the read and write patterns of Algorithm 3 and classing them according to the state that they occur in allows to look for differences that could be exploitable. These differences can be rendered in a table, such as Table I. This allows for visual inspection of differences.

TABLE I
ACCESS PATTERNS DEPENDING ON THE i-TH BIT OF THE SECRET KEY

Condition	Read Variables	Written Variables
$S_i = 0$	P, K	K
$S_i = 1$	P, K	P

This representation makes it immediately clear that even though the same method is being called, it affects different data. This allows an attacker to exploit the difference between the two branches by modifying one memory location and checking whether a safe-error occurred.

a) Example:: Let's assume we try to attack the first branch, when $S_i = 0$. During the calculate routine, we modify the memory used by the variable K in such a way that it does not change the result of the computation. This is done by perturbing the memory once the given memory location is not read anymore, but before it is being potentially written to. After the calculate routine has executed, either K or P has been overwritten. If our guess of $S_i = 0$ was correct, due to being overwritten after being perturbed by the fault, K now holds again correct information in context of the algorithm. Letting the algorithm finish leaks the information whether our guess was correct: If it finishes normally, S_i was indeed 0. If we assume that M is known and verifiable, we can check to see if the outcome was wrong, or, simpler, an error occurred. If either happened, then S_i was 1, as the faulted K did not get overwritten and subsequently changed the calculation. This attack needs to be then repeated n times to fully recover the secret key S.

III. ATTACKS ON SIKE

In this section, we analyze the implementation of SIKE submitted to round 3 of NIST's standardization process [20] in the context of safe-error attacks. This implementation is implemented in a constant-time manner. First, we describe a memory safe-error attack in Section III-A, then we describe a computational safe-error attack in Section III-B. For both attacks, we assume that the victim has a static secret key. Both the encapsulator and the decapsulator can be the victim of this attack.

A. M-Safe Attack on SIKE

We first give a high-level overview on how the attack is constructed. Then, we give a more detailed analysis of the individual steps of the attack.

As shown in Section II-A each SIKE participant has their own secret key $m \in \mathbb{F}_{p^2}$. This key is used to calculate the subgroup $\langle P, [m]Q \rangle$ representing the kernel of their secret isogeny. The point multiplication $[m]Q$ is performed through a three-point ladder algorithm as seen in Algorithm 4. Important here is that the **LADDER3PT** function is called with the secret key m as the first argument. The attacker requires the following capabilities: They need to be able to introduce a

[3] See the following thread in the NIST-PQC mailinglist: https://tinyurl.com/yy4m7rud

Algorithm 4: The 3-Point Ladder

1 **function** **LADDER3PT**
  Input: $m = (m_{l-1}, ..., m_0)_2 \in \mathbb{Z}, (x_P, x_Q, x_{Q-P}),$ and $(A : 1)$
  Output: $(X_{P+[m]Q} : Z_{P+[m]Q})$
2 $((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2)) \leftarrow ((x_Q : 1), (x_P : 1), (x_{Q-P} : 1))$
3 $a_{24}^+ \leftarrow (A + 2)/4$
4 **for** $i = 0$ **to** $l - 1$ **do**
5 **if** $m_i = 1$ **then**
6 $((X_0 : Z_0), (X_1 : Z_1)) \leftarrow$ **xDBLADD**$((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2), (a_{24}^+ : 1))$
7 **else**
8 $((X_0 : Z_0), (X_2 : Z_2)) \leftarrow$ **xDBLADD**$((X_0 : Z_0), (X_2 : Z_2), (X_1 : Z_1), (a_{24}^+ : 1))$
9 **return** (X_1, Z_1)

TABLE II
ACCESS PATTERNS DEPENDING ON THE iTH-BIT OF THE SECRET KEY

Condition	Read Variables	Written Variables
$m[i] = 0$	$(X_0, Z_0), (X_1, Z_1), (X_2, Z_2)$	$(X_0, Z_0), (X_2, Z_2)$
$m[i] = 1$	$(X_0, Z_0), (X_1, Z_1), (X_2, Z_2)$	$(X_0, Z_0), (X_1, Z_1)$

memory fault during a specific point of execution, as well as be able to verify the result of a given SIKE run. Both the shared secret as well as any execution errors need to be known afterwards. The attack proposed in this section then follows three parts:

With the goal of extracting an n-bit secret key, the attacker

1) initiates a SIKE key agreement,
2) introduces a memory fault of any kind (bit-flip, scrambling,...) during the i-th iteration of **LADDER3PT**, and
3) uses the result of the SIKE run to obtain the value of the i-th bit of the secret key.

Steps 1 to 3 have to be repeated n times to reconstruct the complete secret key.

In detail, this means that the attack on this three-point ladder algorithm follows the schema as described in Section II-C. Depending on a given bit of the secret key, different variables are modified. This can be seen in Table II. In this case either (X_1, Z_1) or (X_2, Z_2) are passed to **xDBLADD**. Without loss of generality, let's assume for the rest of this section that we attack m and that the guess for the i-th bit is $m[i] = 1$. By following the general outlines of a safe-error attack one needs to modify (X_1, Z_1) between its last use and the moment it gets written to. Such a moment exists in Algorithm 4 Line 6 (cf. Section II-A): (X_1, Z_1) is passed to the **xDBLADD** subroutine as the *second* argument, thus $(X_Q, Z_Q) = (X_1, Z_1)$ in Algorithm 1 (cf. Section II-A). (X_1, Z_1) is passed as the *third* argument in Line 8, this difference is dependent on the secret key. The **xDBLADD** method (as seen in Line 6 in Algorithm 1) then returns two values, one of which is assigned to (X_1, Z_1) in Algorithm 4. In the **xDBLADD** routine from Line 7 onwards, (X_Q, Z_Q) is no longer read, and thus the value of (X_1, Z_1) stays unused until the function returns. This is where the attacker executes the active attack, by scrambling the values backing (X_Q, Z_Q), i.e., (X_1, Z_1). If the attack on the memory location of (X_1, Z_1) was successful and our guess was correct, the algorithm will, upon return, overwrite our modification and finish without encountering an error. One can thus conclude that $m[i] = 1$. Should our guess of $m[i] = 1$ be incorrect, then the algorithm computes a mismatching shared secret or raises an error. In this case, $m[i] = 0$. Either way, a single bit of information is gained of the secret key. Consequently, all n bits of the static secret key m can be read by this method and the full key can be recovered through n runs of this attack. The complete attack thus consists of these steps:

1) The attacker observes a normal SIKE key agreement.
2) As **xDBLADD** gets called during **LADDER3PT**, overwrite (X_1, Z_1) on the i-th iteration and observe the final result.
3) If the SIKE de/encapsulation fails, we know that (X_1, Z_1) did *not* get overridden. Thus $m[i] = 0$ otherwise $m[i] = 1$.

Repeat steps 1 to 3 n times to recover the complete n-bit secret key.

The SIKE implementation in [20] has several parameter sets, each influencing the range of possible values of the secret key. For example, SIKEp610 has an exponent $e_2 = 305$ with an estimated NIST security level 3 [20]. The private key m is thus sampled from $\{0, ..., 2^{305} - 1\}$, giving the private key 305 bits of total length. Therefore an attacker, trying to attack a SIKEp610 instantiation, would need to repeat the attack at least 305 times to achieve full key recovery.

In the latest version of **xDBLADD**, as published for the third NIST PQC process round [20], the authors have chosen to use a simultaneous double-and-add algorithm. This implementation prevents this particular attack as there is no moment during execution that P or Q is written before it is potientially read. This is also true during compilation: the order of operations in the assembly stays the same. Nonetheless, future implementations have to make sure that they are not vulnerable when using a different algorithm.

B. C-Safe Attack on SIKE

Similar to the M safe-error attack on SIKE described in the previous section, the attack described in this section exploits the difference in memory accesses depending on a bit of the secret key. Again, each party generates their own private key m, used to generate the subgroup $\langle P + [m]Q \rangle$ of their private isogeny (cf. Section II-A). This point multiplication $P + [m]Q$ is done through a three-point ladder as seen in Algorithm 4 and Figure 1 (cf. Section II-A). In the C implementation, published in [20], the authors use a constant-time swapping algorithm to exchange the points R and $R2$ depending on the i-th bit of the secret key (see Line 360 of Listing 1). The function is called `swap_points` and accepts both points and a mask as input. We denote the i-th bit of the secret key as $m[i]$. The mask of the swapping function is calculated as $\text{xor}(m[i], m[i-1])$, with a starting value of 0 for $m[i-1]$ if $i = 0$. If the mask is 1 the points are exchanged, otherwise they are left as is. This behavior can be exploited by meddling with this function call. It could for example simply be skipped, or the computation of the mask be perturbed such that on a 0 mask it stays 0, but on a 1 mask the value is randomized. Assuming $\text{xor}(m[i], m[i-1])$ and an attacker skips this function call using an active attack on the i-th loop, the end result will be unchanged. If the value had been $\text{xor}(m[i], m[i-1])$, then the end result would be wrong, as the wrong point would have been used for the rest of the calculation.[4] Since we know that in the first iteration $m[i-1]$ is forced to 0, the mask is simply set to the value of $\text{xor}(m[0], 0) = m[0]$. The second iteration of attack then knows the value of $m[0]$ and so on. Thus, in general the bit $m[i]$ is leaked through a C safe-error. As the keyspace for m is equal to $[0, ..., 2^{e_2} - 1]$, similar to the attack in Section III-A, the attack needs to be repeated at least 305 times to achieve full key recovery when the parameter set SIKEp610 is used.

IV. Attacks on CSIDH

In this section, we analyse CSIDH with respect to safe-error attacks. We analyse two recent implementations of CSIDH [16, 23]. Both implementations are constant-time implementations, and both implementations achieve this kind of timing attack resistance through dummy isogeny computations. The main difference between both implementations is that in [16], computations are done on one point only, while in [23], two points are used. The analysis of both implementations with respect to safe-error attacks is presented in Section IV-A and Section IV-B, respectively. For both of the presented attacks, it is assumed that the victim uses a static secret key.

A. M Safe-Error Attack on an Implementation Using One Point

In [16] Campos et al. have evaluated possible physical attack vectors for CSIDH implementations using dummy isogenies. One threat model they did not consider, is one that can introduce memory faults. This will be the focus of the attack in this section. The attacker only needs to be able to change a single bit in a certain byte range. In [16], during the execution of a dummy isogeny, the curve parameter A is not modified. If however a non-dummy isogeny is calculated, then the A parameter is changed corresponding to the newly calculated curve. This leads to a possible attack vector: assume without loss of generality that the algorithm is currently calculating isogenies of degree ℓ_i. If it is currently calculating a dummy isogeny, a new parameter A is computed, but *directly discarded*. If a real isogeny is calculated, that result is then used further. A fault injected with the intent of modifying the parameter A can now discern if a real or dummy isogeny is being calculated: if one attacks a real isogeny, the modified value will be propagated and cause a mismatch of the final shared secret. If it was a dummy isogeny however, the modified A was discarded and the shared secret is not impacted. This is now repeated for each possible value of e_i, so as to find out the first time a dummy isogeny is calculated. The value of e_i is then the amount of real isogenies that have been calculated for ℓ_i. In the implementation in [16] e_i is sampled from the range $[0, 10]$, therefore one needs on average 5 attacks per e_i to recover its value. In CSIDH-512 of [16] the secret key has 74 components, thus on average, an attacker would need to run $5 \times 74 = 370$ attacks to recover the full key.

TABLE III
Access pattern depending on the secret key e during the key exchange

Condition	Read Variables	Written Variables
$e_i \neq 0$	P_0, P_1	P_0, P_1
$e_i = 0$	P_s where s is the sign bit of e_i	P_s

B. M Safe-Error Attacks on an Implementation Using Two Points

In [23], Onuki et al. have introduced a new algorithm that uses two points to calculate the CSIDH action. This version has an issue similar to the one described in Section IV-A, where the parameter A is discarded when calculating a dummy isogeny. Thus it has also the potential for an M safe-error attack by attacking the A parameter assignment. Unlike the implementation in [16], in [23] the range $[-5, 5]$ is used for each e_i. Even though an attacker additionally needs to recover the sign of e_i now, this reduces the amount of overall attacks required to recover a single e_i.

Further, the CSIDH action as described in [23] has another M safe-error attack vector that will be explained in this section. Table III shows the access patterns of two different variables depending on a part of the secret key: only one point is overwritten when e_i equals 0 during the CSIDH action calculation at Line 17 in Algorithm 2 (cf. Section II-B). This opens up the potential of perturbing a given P_0 or P_1 and finding out if this had any effect on the calculation. If there was no effect, then the sign of e_i is equal to the index of the point that was overwritten: 0 if positive, 1 if negative. This allows the attacker to find the sign of a specific e_i since the

[4] Wrong shared secret or an error raised from the algorithm.

dependency between isogenies of degree ℓ_i and its running allows for attacking a specific degree ℓ_i [5]. Now let s_i be the sign of e_i. In total, Algorithm 2 does e_i calculations of isogenies of order ℓ_i. After each calculation, it decrements e_i to keep track of how many more real isogenies need to be computed. Once $e_i = 0$, only dummy operations are executed. The task is thus, to find out how many real isogenies are calculated. One can run the following procedure to find the value of e_i: Start with $n = 0$. Modify P_{s_i} after n iterations just before it is potentially overwritten, and check the final result. If the shared secret is correct or n is larger than the maximal possible value for e_i, we know $e_i < n$ at that point and we can stop the process, otherwise $e_i > n$, increment n and retry. Once this procedure terminates, e_i equals the amount of calculated real isogenies. Applying this procedure repeatedly, one can deduce the whole secret key $(e_1, \ldots, e_n)$. As [23] uses an instantiation where the private key elements can range from -5 to 5, in total $2.5 + 1 = 3.5$ attacks are required per e_i, as well as finding s_i. In that instantiation, 74 elements are used per secret key, therefore an attacker would need to run $74 \times 3.5 = 259$ attacks on average for the signs and the full key recovery in total. The attack can be summarised as follows:

1) Reveal which ℓ_i is currently being computed from the length of computation.
2) On Line 17 in Algorithm 2 only P_{s_i} is being assigned. Thus, perturbing the memory of P_{s_i} while $[\ell_i]P_{s_i}$ is being calculated will allow to deduce whether $i = 0$, or $i = 1$. From now on, we assume that s_i is known for each e_i.
3) Knowing the sign allows us to now explicitly attack either P_0 or P_1 and thus find out whether a real or dummy isogeny is being calculated.

If the final shared secret is correct, it was a real isogeny, otherwise it was a dummy. The value of e_i is equal to the count of real isogenies. Once all e_i and their signs s_i have been recovered, the full private key $(e_1, \ldots, e_n)$ can be reconstructed.

V. Practical Experiments

In this section, we explain how to perform the described attacks on a ChipWhisperer board and present the achieved security impact. In the case of SIKE, we present full key recovery. In the case of CSIDH, due to the relatively long runtime on the target architecture (≈ 7 seconds for the reduced version of CSIDH), we calculated the maximum number of possible runs in advance and determined further attack parameters accordingly.

All practical attacks were implemented using the ChipWhisperer tool chain[5] (version 5.3.0) in Python (version 3.8.2) and performed on a ChipWhisperer-Lite board with a 32-bit STM32F303 ARM Cortex-M4 processor as target core. Based on available implementations, we wrote slightly modified ARM implementations of SIKEp434 and CSIDH512 to make them suitable for our setup. Security-critical spots remained unchanged. All binaries were build using the GNU Tools for ARM Embedded Processors 9-2019-q4-major[6] (gcc version 9.2.1 20191025 (release) [ARM/arm-9-branch revision 277599]) using the flags: `-Os -mthumb -mcpu=cortex-m4 -mfloat-abi=soft`.

In all attack models the adversary aims to attack the calculation of the shared secret in order to learn parts of the private key. The shared secrets are calculated without randomness, i.e., points and private keys used were computed in advance. Both in the case of SIKE and CSIDH, the adversary is able to randomise variables or skip instructions by injecting one fault per run. Furthermore, we assume that the attacker is able to trigger and attack the computation of the shared secret multiple times using the same pre-computed private keys. However, in a real environment the attacker is limited to observe the impact of a fault injection (whether both shared secrets are equal or not), by noticing possible unexpected behaviour in the protocol.

A. Attacks on SIKE

Since the current implementation [20] is immune to the attack described in Section III-A, we focus on the attack explained in Section III-B. As described, the adversary deploys safe-error analysis to recover the private key during the computation of the three-point ladder. Since the attacked algorithm runs in constant time, an attacker can easily locate the critical spot, which in our case represents the main loop within the ladder computation. Thus, an attacker who can accurately induce any kind of computational fault inside that spot at the i-th iteration, may be able to deduce if the i-th bit of the private key is set or not, i.e, $\mathrm{sk}_i = 0$ or $\mathrm{sk}_i = 1$ according to whether the resulting shared secret is incorrect or not. Thus, in this model the required number of injections for a full key recovery only depends on the length of the private key. In this setup, the fault is injected by suddenly modifying the clock (clock glitching), thus, forcing the target core to skip an instruction.

The SIKEp434 Cortex-M4 implementation[7] available at the pqm4 project [26] provided the basis for our implementation. However, this attack can be applied to all available software implementations of SIKE[8] including the round-3 submission [20] to NISTs standardisation process. More precisely, the code part that represents this vulnerability remains the same across all available implementations.

Results. Assuming that the attacker knows critical spots within the attacked loop (cf. Listing 1) which reveal one bit of the private key after a single fault injection with high accuracy. As shown in this work, such spots and the corresponding suitable parameters for the injection (e.g., width and internal offset of the clock glitch) can be empirically determined in advance with manageable effort.

[5] https://github.com/newaetech/chipwhisperer, commit fa00c1f

[6] https://developer.arm.com/

[7] https://github.com/mupq/pqm4, commit 20bcf68

[8] https://sike.org/#implementation

In order to determine the success rate for each individual of the 218 bits of the private key, we performed 21,800 fault injections (100 injections for each bit) and achieved a relatively high accuracy. More precisely, we obtained on average over all bits 100% (leading to an error probability $p_0 = 0$, as denoted in Fig. 1) accuracy for the case $\mathrm{sk}_i = 0$ and an accuracy of over 86% (denoted as p_1 in Fig. 1) for the case $\mathrm{sk}_i = 1$. As shown in Fig. 1, only 5 fault injections are required for each bit, thus 1,090 injections in total to achieve a success rate above 99% for full key recovery. Since in our inexpensive setup a single run takes about 12 seconds, full key recovery requires about 4 hours.

B. Attacks on CSIDH

Since the practical implementation is similar for both attacks, we show without loss of generality how we realised the attack described in Section IV-A. The attacker aims to distinguish a real from a dummy isogeny. For this, they inject a fault during the computation of an isogeny and observe if it impacts the resulting shared secret. In this attacker model the adversary can target isogeny computations at positions of their choice and is further able to trace the faulty isogeny computation to determine its degree. Due to non-constant time computation within the calculation of the isogeny (e.g., a square-and-multiply exponentiation based on the degree [16, 22, 23]), the degree of a given isogeny might be recovered with manageable effort, e.g., using Simple Power Analysis [27].

In our setup, the fault is injected by temporarily underpowering the target core, i.e., by reducing for some clock cycles the value of the supply voltage of the attacked device below the minimum value the device is specified for. Such an attack might lead to an unpredictable state in the target variable during an assignment and can therefore be applied to attack the vulnerable spot regarding the co-domain curve A, as defined in Section IV-B. Without loss of generality, the attacks occur during the calculation of the first isogeny. The implemented attacks are based on the implementation from [16].

Results.

As suggested in [16], in order to increase the number of attempts by reducing the time required for a single run, we reduced the key space in CSIDH512 from 11^{74} to 3^2. Further, all required values, e.g. points of corresponding order, were calculated in advance, leading in total to a reduction from 15,721M to 115M clock cycles for a single run. Due to the reduced key space, private keys are of the form $S = (e_0, e_1)$, where $e_i \in [-1, 1]$. To obtain results for both cases (dummy and real), we performed experiments using different private keys. In the first case, the private key $S_1 = (-1, 1)$ consists of real isogenies only. Thus, attacks should not impact the computation of the shared secret. As expected, after 2,500 attempts, there is no faulty shared secret, achieving an accuracy of 100% (leading to an error probability $p_0 = 0$, as denoted in Fig. 1). In the second case, however, the selected private key $S_2 = (0, 1)$ implies the calculation of a dummy isogeny since $e_0 = 0$. Hence, fault injections should lead to a faulty shared secret. Here, we achieved an accuracy of over 92% (denoted as p_1 in Fig. 1). Table IV shows the achieved results of the applied attacks in our setup. Hence, based on these numbers, we assume an attacker can distinguish real from dummy isogenies with a single injection with high accuracy.

TABLE IV
RESULTS FOR CSIDH ATTACKING THE FIRST ISOGENY

key	# of trials	faulty shared secret	accuracy
$S_1 = (-1, 1)$	2500	0.0%	100.0%
$S_2 = (0, 1)$	2500	92.4%	92.4%

Since in dummy-based constant-time implementations of CSIDH (e.g., Meyer, Campos, and Reith (MCR) [22] or Onuki, Aikawa, Yamazaki, and Takagi (OAYT) [23]), the private key vector $(e_1, \ldots, e_n)$ is sampled from an interval defined by a bound vector $\mathtt{m} = (m_1, m_2, \ldots, m_n)$, the number of fault injections required to obtain the absolute value of a certain e_i strongly depends on the corresponding bound vector. More precisely, since the computation of a given degree ℓ_i occurs deterministically (real-then-dummy), the attacker performs a binary search through the corresponding m_i to identify the computation of the first dummy isogeny. Thus, the number of attacks required to obtain the absolute value of a certain e_i depends only on the corresponding bound m_i.

The achieved key space reductions are due to the fact that an attacker after a certain number of attacks knows the absolute values for the private key vector $(e_1, \ldots, e_n)$. In the case of the OAYT implementation of CSIDH512 (where $-m_i \leqslant e_i \leqslant m_i, m_i = 5$ for $i = 0, \ldots, 73$), our approach leads to a private key space reduction from 2^{256} to 2^{74} in the worst case ($e_i \neq 0$ for $i = 0, \ldots, 73$) and to $2^{67.06}$ in the average case after at least $222 \cdot 4 = 888$ fault injections for a success rate over 99%. The remaining key space can be further reduced by a meet-in-the-middle approach [5] to about $2^{34.5}$ in the average case. For achieving a success rate over 99%, when attacking the MCR implementation (where $0 \leqslant e_i \leqslant m_i, m_i \in [1, 10]$ for $i = 0, \ldots, 73$), at least $296 \cdot 4 = 1184$ injections are required for full key recovery (cf. Fig. 1) since only positive values are allowed for the private key vector. Considering the running time of the non-optimised implementation of CSIDH512 of about 5 minutes for a single run in our setup, full key recovery would require about 98 hours in the case of the MCR implementation and about 74 hours to achieve the mentioned key space reduction in the case of the OAYT version.

Since recent works [6, 7] suggests that CSIDH-512 may not reach the post-quantum security as initially considered [5], some works recommend to increase the size of the CSIDH prime p [6, 7, 28]. However, from a classical perspective, since the classical security only depends on the size of the private key space, the number of prime factors ℓ_i remains unchanged. Thus, apart from the longer running time due to possibly larger prime factors, increasing the quantum security has no further

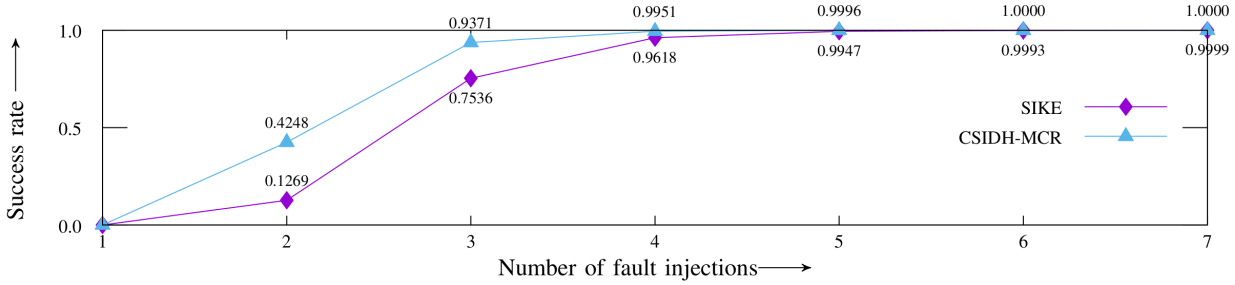

Fig. 1. Success rate for full key recovery as a function of the number of fault injections per bit (SIKE) or isogeny (CSIDH), respectively. Let α be the number of injections for each bit/isogeny. Since a single faulty shared secret is sufficient to distinguish the cases, the success rate for full key recovery can be calculated by $P(\alpha) = [(0.5 \cdot (1 - B(0, \alpha, p_1))) + (0.5 \cdot B(0, \alpha, p_0)))]^{\lambda}$, where λ equals the number of bits in the case of SIKE and equals $\sum_{i=1}^{n} \lceil \log_2(m_i) \rceil$ for all m_i of the corresponding bound vector $\mathtt{m} = (m_1, m_2, \ldots, m_n)$ in the case of CSIDH, $B(k, n, p) = \binom{n}{k} \cdot p^k (1-p)^{n-k}$, and p_0, p_1 correspond to the respective probabilities.

influence on the effectiveness of the presented attack.

VI. Countermeasures

In this section we discuss general countermeasures against safe-error attacks and then present concrete countermeasures for SIKE and CSIDH.

In safe-error attacks, a simple check of the final result before transmitting can still leak one bit. This can be easily seen in the attack on SIKE in Section III-B. If the attacker successfully executes an attack, even if the result is checked for correctness, the implementation will leak one bit: either the algorithm fails or it returns an unusable result, or the induced error is overwritten, both of which represent a successful attack. This makes efficient generic countermeasures hard to design, as, for instance, simply repeating a calculation after a fault has been detected can be detected, too: an algorithm that suddenly takes twice as long shows that the attack was successful. Such foot guns need to be avoided so as not to introduce new attacks while mitigating others.

Using infective computation [29], a succesfully induced fault directly, i.e., without the necessity of checking, modifies the output value such that the faulty output does not allow to reveal secret values. In case of safe-error attacks, this is also not a solution, since any faulty output shows that the fault was successful. This is all an attacker needs to know in case of safe-error attacks.

An effective countermeasure consists in redundant computation with consistency check, i.e., calculating the susceptible operations repeatedly and then choose the value to be output by majority vote. However, this is costly, since, assuming that an attacker can realize a fault n times within a single computation of the algorithm, the susceptible operations have to be computed $2 \cdot n + 1$ times. Since second-order faults, i.e., two faults within one computation, are practical [30], this would require at least a fivefold repetition of the susceptible operations.

Another route, which is not in the hands of the implementer, is the selection of hardware the algorithm executes on. Hardware-based detection of fault attacks through, for instance, voltage sensing or intrusion detection, are possible ways of shutting down the execution - independent of the effect of the fault on the computation - before any information could have been leaked [31].

It is important to note that the attacks presented in this paper exploit secret-dependent memory access. Implementations and future optimizations should thus take special care to eliminate any such occurrence and treat them with the same rigour as secret-depending timings. This also extends to "branchless" versions of algorithms, where, for instance, a pointer is swapped depending on the bit of a secret key; this does not remove the secret dependence of the underlying memory.

The discussion shows that to prevent safe-error attacks, the susceptible functions have to be adjusted, as in [8].

A. Securing SIKE

As explained in Section III-A, by using a simultaneous double-and-add algorithm within **xDBLADD** [20], the particular M safe-error attack on SIKE can be prevented.

A possible countermeasure against the key recovery presented in Section III-B is to add an additional check to the **LADDER3PT** algorithm. The attack relies on skipping the `swap_points` method. Hence, a relatively inexpensive way of detecting an attack is to verify whether the swap actually took place. Thus, in each loop the implementation would save the current points, run the swap operation, and eventually check if the calculated mask had the intended effect.

Although the proposed countermeasure to conditional point swaps from [16] could be adapted to SIKE, the described approach (cf. [16], Section VI, paragraph C, point 1) represents no real countermeasure. An attack in the case where no swap takes place (decision bit = 0) does not lead to a false result (wrong point order), while attacking the conditional swap in the case of a swap (decision bit = 1) the order check of the resulting point should fail.

B. Securing CSIDH

Since the current CSIDH action algorithms branch on the secret key, it is a prime target for exploitation. One possible way of making attacks more difficult is shown in [18]. Here, LeGrow and Hutchinson show that using a binary decision vector to interleave the different ℓ_i-isogenies, an attacker has

to do more than 8x as many attacks to gain the same amount of information.

Another approach is to choose an implementation that is dummy-free. So far however, dummy-free implementations have come at the cost of being twice as slow [17]. Further research might be able to close this performance gap and thereby completely eliminate attacks based on dummy isogenies.

Securing CSIDH against physical attacks is clearly difficult. Moreover, care has to be taken to not accidentally introduce a foot-gun in the form of a novel attack vector. One such occurrence are dummy isogenies, introduced as timing attack countermeasures in [22], which allow an attacker to learn secret information through fault injections.

VII. Conclusion

This work shows how safe-error attacks can be applied to recent isogeny-based cryptographic schemes. We presented four different attacks on the SIKE and CSIDH cryptosystems. It is important to note that the resilience of SIKE against the attack described in Section III-A solely depends on the structure of the actual implementation. As such, any further implemententations need to make sure to not introduce the possibility of this safe-error attack. We have shown how to practically realize two of these attacks and how to achieve full key recovery in a static key context on both SIKE and CSIDH.

We discussed that securing cryptosystems against safe-error attacks is non-trivial. This also partially explains why some of the attacks that we applied to isogeny-based cryptographic schemes have similarly been known in the ECC community for a long time, and yet have not been prevented in current implementations of SIKE and CSIDH. As safe-errors exploit differences of computation and memory access depending on the secret key, a simple check is not sufficient. It is equally important, that countermeasures against certain attacks do not open ways for further safe-error attacks [24]. This can be the case for example when implementing a simple consistency check, which might not trigger on all injections, thus inadvertently leaking data. The same holds true for constant-time implementations, which are designed to thwart timing attacks. The implementations of CSIDH that we attacked in this work are constant-time, but based on dummy isogenies, which enable our attack. Dummy-free implementations, which do also exist, are probably not vulnerable to the attacks presented in this paper; however, they are prone to timing attacks. Future research therefore needs to find a way to secure CSIDH at the same time against timing and safe-error attacks.

References

[1] J.-M. Couveignes, "Hard homogeneous spaces," Cryptology ePrint Archive, Report 2006/291, 2006, http://eprint.iacr.org/2006/291. 1

[2] A. Rostovtsev and A. Stolbunov, "Public-Key Cryptosystem Based On Isogenies," Cryptology ePrint Archive, Report 2006/145, 2006, http://eprint.iacr.org/2006/145. 1

[3] L. De Feo, D. Jao, and J. Plût, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," Cryptology ePrint Archive, Report 2011/506, 2011, http://eprint.iacr.org/2011/506. 1

[4] 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, "SIKE," National Institute of Standards and Technology, Tech. Rep., 2017, available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions. 1

[5] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes, "CSIDH: An efficient post-quantum commutative group action," in *ASIACRYPT 2018, Part III*, ser. LNCS, T. Peyrin and S. Galbraith, Eds., vol. 11274. Brisbane, Queensland, Australia: Springer, Heidelberg, Germany, Dec. 2–6, 2018, pp. 395–427. 1, 2, 7, 8

[6] C. Peikert, "He gives C-sieves on the CSIDH," in *EUROCRYPT 2020, Part II*, ser. LNCS, A. Canteaut and Y. Ishai, Eds., vol. 12106. Zagreb, Croatia: Springer, Heidelberg, Germany, May 10–14, 2020, pp. 463–492. 1, 8

[7] X. Bonnetain and A. Schrottenloher, "Quantum security analysis of CSIDH," in *EUROCRYPT 2020, Part II*, ser. LNCS, A. Canteaut and Y. Ishai, Eds., vol. 12106. Zagreb, Croatia: Springer, Heidelberg, Germany, May 10–14, 2020, pp. 493–522. 1, 8

[8] S.-M. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *IEEE Trans. Comput.*, vol. 49, no. 9, pp. 967 – 970, Sep. 2000. [Online]. Available: https://doi.org/10.1109/12.869328 1, 4, 9

[9] M. Joye and S.-M. Yen, "The Montgomery powering ladder," in *CHES 2002*, ser. LNCS, B. S. Kaliski Jr., Çetin Kaya. Koç, and C. Paar, Eds., vol. 2523. Redwood Shores, CA, USA: Springer, Heidelberg, Germany, Aug. 13–15, 2003, pp. 291–302. 1, 4

[10] C. Costello, "B-SIDH: supersingular isogeny Diffie-Hellman using twisted torsion," Cryptology ePrint Archive, Report 2019/1145, 2019, https://eprint.iacr.org/2019/1145. 2

[11] B. Koziel, R. Azarderakhsh, and D. Jao, "Side-channel attacks on quantum-resistant supersingular isogeny Diffie-Hellman," in *SAC 2017*, ser. LNCS, C. Adams and J. Camenisch, Eds., vol. 10719. Ottawa, ON, Canada: Springer, Heidelberg, Germany, Aug. 16–18, 2017, pp. 64–81. 2

[12] S. D. Galbraith, C. Petit, B. Shani, and Y. B. Ti, "On the security of supersingular isogeny cryptosystems," in *ASIACRYPT 2016, Part I*, ser. LNCS, J. H. Cheon and T. Takagi, Eds., vol. 10031. Hanoi, Vietnam: Springer, Heidelberg, Germany, Dec. 4–8, 2016, pp. 63–91. 2

[13] A. Gélin and B. Wesolowski, "Loop-abort faults on supersingular isogeny cryptosystems," in *Post-Quantum Cryptography - 8th International Workshop, PQCrypto*

2017, T. Lange and T. Takagi, Eds. Utrecht, The Netherlands: Springer, Heidelberg, Germany, Jun. 26–28 2017, pp. 93–106. 2

[14] Y. B. Ti, "Fault attack on supersingular isogeny cryptosystems," in *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, T. Lange and T. Takagi, Eds. Utrecht, The Netherlands: Springer, Heidelberg, Germany, Jun. 26–28 2017, pp. 107–122. 2

[15] F. Zhang, B. Yang, X. Dong, S. Guilley, Z. Liu, W. He, F. Zhang, and K. Ren, "Side-channel analysis and countermeasure design on arm-based quantum-resistant SIKE," *IEEE Trans. Computers*, vol. 69, no. 11, pp. 1681–1693, 2020. [Online]. Available: https://doi.org/10.1109/TC.2020.3020407 2

[16] F. Campos, M. J. Kannwischer, M. Meyer, H. Onuki, and M. Stöttinger, "Trouble at the CSIDH: Protecting CSIDH with Dummy-Operations against Fault Injection Attacks," in *2020 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2020. 2, 3, 6, 8, 9

[17] D. Cervantes-Vázquez, M. Chenu, J.-J. Chi-Domínguez, L. De Feo, F. Rodríguez-Henríquez, and B. Smith, "Stronger and faster side-channel protections for CSIDH," in *LATINCRYPT 2019*, ser. LNCS, P. Schwabe and N. Thériault, Eds., vol. 11774. Springer, Heidelberg, Germany, 2019, pp. 173–193. 2, 3, 10

[18] J. LeGrow and A. Hutchinson, "An analysis of fault attacks on CSIDH," Cryptology ePrint Archive, Report 2020/1006, 2020, https://eprint.iacr.org/2020/1006. 2, 9

[19] C. Costello, "Supersingular isogeny key exchange for beginners," Cryptology ePrint Archive, Report 2019/1321, 2019, https://eprint.iacr.org/2019/1321. 2

[20] 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, D. Urbanik, G. Pereira, K. Karabina, and A. Hutchinson, "SIKE," National Institute of Standards and Technology, Tech. Rep., 2020, available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. 2, 4, 5, 6, 7, 9

[21] A. Faz-Hernández, J. C. López-Hernández, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez, "A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol," *IEEE Trans. Computers*, vol. 67, no. 11, pp. 1622–1636, 2018. [Online]. Available: https://doi.org/10.1109/TC.2017.2771535 2

[23] H. Onuki, Y. Aikawa, T. Yamazaki, and T. Takagi, "(Short paper) A faster constant-time algorithm of CSIDH keeping two points," in *IWSEC 19*, ser. LNCS, N. Attrapadung and T. Yagi, Eds., vol. 11689. Tokyo, Japan: Springer, Heidelberg, Germany, Aug. 28–30, 2019, pp. 23–33. 3, 6, 7, 8

[22] M. Meyer, F. Campos, and S. Reith, "On lions and elligators: An efficient constant-time implementation of CSIDH," in *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019*, J. Ding and R. Steinwandt, Eds. Chongqing, China: Springer, Heidelberg, Germany, May 8–10 2019, pp. 307–325. 3, 8, 10

[24] S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon, "A countermeasure against one physical cryptanalysis may benefit another attack," in *ICISC 01*, ser. LNCS, K. Kim, Ed., vol. 2288. Seoul, Korea: Springer, Heidelberg, Germany, Dec. 6–7, 2002, pp. 414–427. 4, 10

[25] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sang-Jae Moon, "Rsa speedup with chinese remainder theorem immune against hardware fault cryptanalysis," *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 461–472, 2003. 4

[26] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Testing and benchmarking NIST PQC on ARM cortex-M4," Cryptology ePrint Archive, Report 2019/844, 2019, https://eprint.iacr.org/2019/844. 7

[27] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *CRYPTO '99*. Springer, 1999, pp. 388–397. 8

[28] J. Chávez-Saab, J.-J. Chi-Domínguez, S. Jaques, and F. Rodríguez-Henríquez, "The SQALE of CSIDH: Square-root vélu Quantum-resistant isogeny Action with Low Exponents," Cryptology ePrint Archive, Report 2020/1520, 2020, https://eprint.iacr.org/2020/1520. 8

[29] B. Gierlichs, J. Schmidt, and M. Tunstall, "Infective computation and dummy rounds: Fault protection for block ciphers without check-before-output," in *LATINCRYPT*, 2012. 9

[30] J. Blömer, R. G. d. Silva, P. Günther, J. Krämer, and J.-P. Seifert, "A practical second-order fault attack against a real-world pairing implementation," in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2014, pp. 123–136. 9

[31] B. Yuce, N. F. Ghalaty, C. Deshpande, C. Patrick, L. Nazhandali, and P. Schaumont, "FAME: fault-attack aware microprocessor extensions for hardware fault detection and software fault response," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*. ACM, 2016, pp. 8:1–8:8. [Online]. Available: https://doi.org/10.1145/2948618.2948626 9