applied sciences

Article

Cold Boot Attacks on the Supersingular Isogeny Key Encapsulation (SIKE) Mechanism

Ricardo Villanueva-Polanco * and **Eduardo Angulo-Madrid**

Computer Science Department, Universidad del Norte, Barranquilla 080001, Colombia; edangulo@uninorte.edu.co
* Correspondence: rpolanco@uninorte.edu.co

Abstract: This research paper evaluates the feasibility of cold boot attacks on the Supersingular Isogeny Key Encapsulation (SIKE) mechanism. This key encapsulation mechanism has been included in the list of alternate candidates of the third round of the National Institute of Standards and Technology (NIST) Post-Quantum Cryptography Standardization Process. To the best of our knowledge, this is the first time this scheme is assessed in the cold boot attacks setting. In particular, our evaluation is focused on the reference implementation of this scheme. Furthermore, we present a dedicated key-recovery algorithm for SIKE in this setting and show that the key recovery algorithm works for all the parameter sets recommended for this scheme. Moreover, we compute the success rates of our key recovery algorithm through simulations and show the key recovery algorithm may reconstruct the SIKE secret key for any SIKE parameters for a fixed and small $\alpha = 0.001$ (the probability of a 0 to 1 bit-flipping) and varying values for β (the probability of a 1 to 0 bit-flipping) in the set $\{0.001, 0.01, \ldots, 0.1\}$. Additionally, we show how to integrate a quantum key enumeration algorithm with our key-recovery algorithm to improve its overall performance.

Keywords: cold boot attacks; key enumeration; key-recovery algorithm; post-quantum cryptography; Supersingular Isogeny Key Encapsulation (SIKE) Mechanism; Side-channel Attacks

check for updates

Citation: Villanueva-Polanco, R.; Angulo-Madrid, E. Cold Boot Attacks on the Supersingular Isogeny Key Encapsulation (SIKE) Mechanism. *Appl. Sci.* **2021**, *11*, 193. https://doi.org/10.3390/app11010193

Received: 2 December 2020
Accepted: 14 December 2020
Published: 28 December 2020

1. Introduction

This research paper assesses the viability of cold boot attacks on the Supersingular Isogeny Key Encapsulation (SIKE) Mechanism [1], which is built upon a key-exchange construction known as Supersingular Isogeny Diffie–Hellman (SIDH) [2]. This key encapsulation mechanism has been included in the list of alternate candidates of the third round of the National Institute of Standards and Technology (NIST) post-quantum cryptography standardization process. According to the status report on the second round of this standardization process (NISTIR 8309) [3], the alternate candidates are considered as potential candidates for future standardization. This report also states the main drawback to SIKE is that its performance is approximately an order of magnitude worse than many of its competitors. So improvements on its implementations have been carried out so far to ameliorate this disadvantage [4–8]. The report additionally states that more research is needed to add more side-channel protection on SIKE's operation. This paper looks into side channel attacks against SIKE and in particular assesses SIKE in the cold boot attack setting. To the best of our knowledge, this is the first time this scheme is assessed in this setting and our evaluation focus on this scheme's reference implementation, in particular version v3.3 [9].

In the cold boot attack setting, an adversary having physical access to a computer may retrieve any content from the computer's main memory by carrying out a series of steps on the target computer, such as shutting it down improperly, booting a lightweight operating system (OS) on it and finally using this OS to dump remaining memory contents to an external device. However, as a consequence of some physical effects on the targeted main memory, the retrieved content is altered [10]. This means that on the condition that an attacker gets possession of a chunk of memory content, this content might be noisy, which means that some 0 bits may have flipped to 1 bits and vice-versa. The next attacker's task is to try to recover secret information stored in the procured memory content, based on what this attacker may learn from the error distribution that occurs during the data acquisition process.

In order to evaluate a public key scheme in this setting, we assume that the attacker procures memory content from a memory region in which the scheme's secret key was stored, and thus such adversary obtains a noisy version of it. On possession of this noisy memory content, the adversary's main task is to try to recover the original secret key from its bit-flipped version. More specifically, the evaluation of a public key encryption scheme in this setting entails three main tasks: (1) the attacker is required to learn the in-memory representations of the scheme's secret key, i.e., the data structures that are used to store the scheme's secret key, (2) the attacker is required to estimate error probability distributions for the bit-flipping, and (3) finally, the attacker is required to devise and develop a key-recovery algorithm for the scheme's secret key.

To deal with the first task, the attacker may either study natural in-memory representations for the scheme's secret key or review scheme's actual implementations to further learn the scheme's secret key memory formats. We follow the latter approach, since it is more realistic, and hence we make a deep review of the SIKE reference implementation (Version v3.3) [9]. Regarding the second task, we assume the attacker estimate α, the probability for a 0 bit of original content to turn to a 1 bit, and β, the probability for a 1 bit of original content to turn to a 0 bit, as per the cold boot model described in Section 2.2. Lastly, we develop our key-recovery algorithm based on a general key-recovery strategy introduced in [11]. Basically, by exploiting and combining the optimal key enumeration algorithm [12], and a non-optimal key enumeration algorithm, as in [11–27], we are able to get a suitable algorithm for the key-recovery task. In particular, this algorithm takes advantage of the SIKE secret key in-memory representation and the estimations for both α and β to generate suitable key candidates for the secret key.

This paper presents two contributions: our first contribution is the evaluation of SIKE against this class of attack. Considering SIKE is one of the alternate candidates, this evaluation represents an important part of the overall assessment of this scheme within the NIST standardization process. Our second contribution is a dedicated key-recovery algorithm for SIKE in the cold boot attack setting. This contribution is in alignment with the current tendency of devising and constructing novel key-recovery algorithms for various schemes in this setting, as our review of the literature, discussed at length in Section 2.3, reveals.

The organization of this research paper is as follows. In Section 2, we discuss the required background on cold boot attacks, detailing on how this attack may be carried out by an attacker. Additionally, we describe the cold boot attack model we assume throughout this paper and also review the literature relevant to our work, focusing on the evaluation of various cryptographic schemes in this setting. Finally, we detail our approach to key-recovery, i.e., the general technique we use to design our key-recovery algorithm. In Section 3, we describe the supersingular isogeny key encapsulation mechanism and its reference implementation, mainly focusing on the key generation algorithm. In Section 4, we give a detailed account of our key-recovery algorithm and our choice for key enumeration algorithms. In Section 5, we evaluate our key-recovery algorithm, focusing on the success rate of our algorithm and how to integrate a quantum key search algorithm with it to improve its overall performance. Finally, in Section 6, we conclude our paper and give insight into a research line to continue with our work.

2. Background

In this section, we give a detailed description of cold boot attacks, the attack model we use through this article and an overview of the previous papers focusing on cold boot attacks on several cryptographic schemes, and finally present a description of the general approach to key recovery we use in this article.

2.1. Cold Boot Attacks

A cold boot attack is a type of data remanence attack in which an attacker could retrieve sensitive data from a computer's main memory after supposedly having been deleted. This attack relies on the data remanence property of Dynamic RAM (DRAM) and allows an attacker to recover memory content that remains readable for a period of time after the computer's power has been turned off. The first time this attack was discussed was about a decade ago in [10], and, since then, it has been studied extensively against multiple cryptographic schemes. In this attack, an adversary, who is assumed to have physical access to a computer, might retrieve portions of memory content from a running operating system via carrying out a cold-rebooting on it. This means the running operating system is forced to shut down improperly, making it to bypass all shut-down-related processes, for example, file system synchronization, that would normally take place during an ordinary shutdown. As a consequence of this improper shutdown, the attacker may use an external disk to start and run a lightweight operating system to copy memory contents of pre-boot DRAM to a file. In an alternative way, such attacker may take the memory modules out of the computer and place them in an adversary-controlled computer, which is then booted, allowing the attacker to have access to the memory content and be able to copy chunks of memory content to an external drive.

As soon as the attacker procures data from the main memory, the adversary may carry out a detailed analysis on that data in order to find sensitive information, such as cryptographic keys, via running several key finding algorithms [10]. Due to some physical effects on the main memory, some bits in the memory experience a process of degradation after the computer's power is turned off. Therefore, the extracted data obtained by the attacker from the target machine's main memory will be recognizably different from the original memory data. In particular, some 0 bits of the original content may have flipped to 1 bits and vice-versa. The authors in [10] remarked that the degradation of bits in memory can be slowed down via spraying some chemical compounds (liquid nitrogen) on the memory modules and thus the original bits may be preserved for a longer period of time. Nevertheless, the attacker has yet to extract the memory content before restoring any important information (or sensible) from the target machine's main memory. To this end, the attacker has to handle several possible issues that may occur. For example, the BIOS of the target computer, on rebooting, may overwrite a fraction of memory with its own code and data, even though the affected fractions are usually small. Moreover, the BIOS during its Power-On Self Test (POST) might execute a destructive memory check, yet this test may be disabled or bypassed in some computers.

To properly handle this problem, some small-scale special-purpose programs, like memory-imaging tools, may be used and are expected to produce correct dumps of memory contents to any external device, as was reported in [10]. These programs generally make use of trivial amounts of RAM, and their memory offsets are usually adjusted so as to guarantee that information of interest remains unchanged. Furthermore, if an attacker cannot force boot memory-imaging tools, the attacker could pull out the memory modules and place them in a compatible machine controlled by the attacker and subsequently dump the content, like that mentioned by the authors of [10]. After extracting memory content, the attacker needs to profile it to learn the memory regions and to estimate the probability for both a 1 flipping to 0 and a 0 flipping to 1. Furthermore, according to the results of the experiment reported in [10], almost all memory bits in a memory region tend to decay to predictable "ground" states, with only a portion flipping in the opposite direction. Additionally, the authors of [10] mention the probability of a bit flipping in the opposite direction stays constant and is very small (circa 0.01) as time elapses, while the probability of decaying to the "ground state" increases over time. These results suggest that the attacker could model the decay in a portion of the memory as a binary asymmetric channel, i.e., we can assume that the probability of a 1 flipping to 0 is a fixed number, and that the probability of a 0 flipping to a 1 is another fixed number in a given time. Note that by reading and counting the number of 0 bits and 1 bits, the attacker can discover the "ground state" of a specific memory region. Additionally, the attacker can estimate the bit-flipping probabilities by comparing the bit count of original content in a memory region with its corresponding noisy version.

Finding encryption keys after procuring memory content is another challenge that the attacker has to address. Such problem has been extensively discussed in [10] for Advanced Encryption Standard (AES) and RSA (Rivest–Shamir–Adleman) keys in memory images. Even though the algorithms presented are scheme-specific, their algorithmic rationale may be easily adapted to devise key-finding algorithms for other schemes. These algorithms simply search for special secret-key-identifying characteristics in the secret key in-memory formats as identifying labels for sequences of bytes. More precisely, these algorithms search for byte sequences with low Hamming distance to these identifying labels and verify that the remaining bytes in a possible sequence satisfy some conditions.

Once the previous problems have been dealt with, the attacker now has access to a version with errors of the original secret key obtained from the memory image. Therefore the attacker's aim is to reconstruct the original secret key from its noisy version, which is a mathematical problem. Moreover, we remark that the attacker may have access to cryptographic data associated with the key to be reconstructed (e.g., ciphertexts for a symmetric key encryption scheme) or the attacker may have access to public parameters of the cryptosystem (e.g., public key for a asymmetric key encryption scheme). The main center of interest of cold boot attacks has been so far to devise, develop and implement key-recovery algorithms to efficiently and effectively reconstruct secret keys from its noisy versions for different cryptographic schemes and testing these algorithms to learn how much noise they can tolerate.

2.2. *Cold Boot Attack Model*

Based on our previous discussion on cold boot attacks, we assume an adversary has knowledge of the defined structures for the storage of the target private key in memory and has possession of the corresponding public parameters without any noise. Moreover, we assume the attacker procures a noisy version of the target private key via a cold boot attack. However, we do not tackle the issue of locating the memory region where the bits of the private key are stored. Such issue would be of great importance to deal with when carrying out this attack in practice and may be tackled via applying several key finding algorithms [10]. As a result of this assumption, the adversary's main objective is to reconstruct the original private key.

We denote $\alpha = P(0 \to 1)$ as the probability of a 0 to 1 bit-flipping, i.e., that a 0 bit in the private key changes to a 1 bit in its noisy version. Moreover, we denote $\beta = P(1 \to 0)$ as the probability of a 1 to 0 bit-flipping, i.e., that a 1 bit in the private key changes to a 0 bit in its noisy version. According to our previous discussion on cold boot attacks, one of these values normally may be very small (approximately 0.001) and not liable to variation over time, while the other values do increase over time. Moreover, the adversary may estimate both α and β and they remain unchanged across the memory region where the bits of the private key are stored. As stated by our previous discussion on cold boot attacks, these suppositions are plausible, since an adversary may estimate the error probabilities by comparing original content with its corresponding noisy version (e.g., using the public key), and the memory regions are normally large.

2.3. *Previous Work*

Throughout this section, we present our literature review, describing works on cold boot attacks against multiples cryptographic schemes. These studies can be divided into several categories, namely the RSA setting, the discrete logarithm setting, the symmetric key setting and finally the post-quantum setting.

2.3.1. RSA Setting

The research paper by Heninger and Shacham [28] was the first work dealing with this attack on RSA keys. They presented a key-recovery algorithm, which relies on Hansel lifting, and exploited the redundancy found in the common RSA secret key in-memory format. The research papers by Henecka et al. [29] and Paterson et al. [30] followed the inaugural paper and both research papers further exploited the mathematical structure on which RSA is based. Additionally, the research paper by Paterson et al. [30] further centered on the error channel's asymmetric nature, which is intrinsically connected to the cold boot setting, and also considered the key-recovery problem from an information theoretic perspective.

2.3.2. Discrete Logarithm Setting

The first paper that explored this attack in the discrete logarithm setting was that of Lee et al. [31]. Their key-recovery algorithm is limited due to the cold boot attack model they assumed. In particular, their work focused on recovering the secret key x, given the public key g^x, with g being a field element and x being an integer. Their model, in addition to assuming the attacker has knowledge of the public key g^x and of the noisy version of the private key x, supposes the adversary knows an upper bound on the number of errors found in the noisy version of the secret key. Since knowing such an upper bound may not be practical and small redundancy in the secret key was exploited, their key-recovery algorithm is not expected to recover keys if these are subjected to a high level of noise, or if a bit-flipping model is assumed. A follow-up work by Poettering and Sibborn [32] also studied this attack in the discrete logarithm setting, more concretely in the elliptic curve cryptography setting. Their work was more practical, since they had a deep review of two implementations for elliptic curve cryptography. With such review, they found redundant in-memory formats and focused on two common secret key in-memory formats from two popular ECC implementations from TLS libraries. The first one is the windowed non-adjacent form (wNAF) representation, and the second is the comb-based representation. For each format, they developed dedicated key-recovery algorithms and tested them both in the bit-flipping model.

2.3.3. Symmetric Key Setting

Regarding this attack against symmetric key primitives, there are also a few papers that explored this attack against a few block ciphers. The paper by Albrecht and Cid [33] centered on the recovery of symmetric encryption keys by employing polynomial system solvers. Particularly, they used integer programming techniques to apply them for key-recovery of the Serpent block cipher's secret key. Furthermore, they also introduced a dedicated key-recovery algorithm to Twofish's secret keys. Moreover, the paper by Kamal and Youssef [34] introduced key-recovery algorithms based on SAT-solving techniques to the same problem. We refer the interested reader to [33–35] for more details.

2.3.4. Post-Quantum Setting

Regarding the viability of carrying out this attack against post-quantum cryptographic schemes, there are already several works showing the feasibility of this attack against some post-quantum cryptographic schemes implementations. The first paper that evaluated a post-quantum cryptographic scheme in this setting was that of Paterson et al. [36]. They looked into this attack against NTRU and their work reviewed two existing NTRU implementations, the `ntru-crypto` implementation and the `tbuktu` Java implementation based on `Bouncy Castle`. For each in-memory format found in these implementations, they introduced dedicated key-recovery algorithms and tested them in the bit-flipping model. According to the results reported in [36], their key-recovery algortihm was able to find the private key, when α is fixed and small and β ranges from 1% up to 9%, for one of the private key in-memory representations. This paper was followed upon by that of Villanueva [11], which extended previous results and introduced a general key-recovery strategy via key enumeration. Additionally, they applied their general strategy to BLISS and their results are comparable to the results reported for the NTRU case. Another recent paper by Villanueva [37] adapted and applied this general key-recovery strategy to the signature scheme LUOV, exploiting the fact that LUOV's private key is a 256 bit string, and showed the versatility of this general key-recovery strategy and promising results. Moreover, Albrecht et al. [38] investigated cold boot attacks on post-quantum cryptographic schemes based on the ring—and module—variants of the Learning with Errors (LWE) problem. Their work centered on Kyber key encapsulation mechanism (KEM) and New Hope KEM, where two encodings were considered to store LWE keys (polynomials): coefficient-based in-memory format and number-theoretic-transform (NTT)-based in-memory format. Finally, they presented dedicated key recovery algorithms to tackle both cases in the bit-flipping model.

2.3.5. General Strategy to Key Recovery

According to the work by Villanueva [11,27], we can deal with the key-recovery problem in the cold boot attack setting via key enumeration algorithms.

Let $\texttt{ch} = b_0 b_1 b_2 \dots b_W$ denote a W bit string with the noisy bits of the secret key encoding. Note that $\texttt{ch}$ may be written as a sequence of $\mathcal{N} = W/w$ chunks, with each chunk being a w bit-string, i.e., $\texttt{ch} = \texttt{ch}^0 || \texttt{ch}^1 || \dots || \texttt{ch}^{\mathcal{N}-1}$ with $\texttt{ch}^i = b_{i \cdot w} b_{i \cdot w+1} \dots b_{i \cdot w+(w-1)}$. On the condition that we have access to a suitable key-recovery algorithm, we can generate full key candidates $\texttt{c}$ for the original secret key encoding. Based on Bayes's theorem, the probability of $\texttt{c}$ being the right full key candidate given the noisy version $\texttt{ch}$ is given by $\mathbf{P}(\texttt{c}|\texttt{ch}) = \frac{\mathbf{P}(\texttt{ch}|\texttt{c})\mathbf{P}(\texttt{c})}{\mathbf{P}(\texttt{ch})}$. Therefore, $\texttt{c}$ should be chosen so as to maximize this probability, according to the maximum likelihood estimation method. Note that both $\mathbf{P}(\texttt{ch})$ and $\mathbf{P}(\texttt{c})$ are constants, with the former independent of $\texttt{c}$, and that $\mathbf{P}(\texttt{ch}|\texttt{c}) = (1-\alpha)^{n_{00}} \alpha^{n_{01}} \beta^{n_{10}} (1-\beta)^{n_{11}}$, where n_{00} counts the positions in which both $\texttt{c}$ and $\texttt{ch}$ contain a 0 bit, n_{01} counts the positions in which $\texttt{c}$ contains a 0 bit and $\texttt{ch}$ contains a 1 bit, etc. Hence our optimization problem reduces to find the candidate $\texttt{c}$ to maximize this probability $\mathbf{P}(\texttt{ch}|\texttt{c})$. Note that this problem can be stated equivalently as choosing $\texttt{c}$ that maximizes the log of these probabilities $\log(\mathbf{P}(\texttt{ch}|\texttt{c})) = n_{00} \log(1-\alpha) + n_{01} \log \alpha + n_{10} \log \beta + n_{11} \log(1-\beta)$. Therefore, each candidate can be assigned a score, namely $S(\texttt{c}, \texttt{ch}) := \log(\mathbf{P}(\texttt{ch}|\texttt{c}))$.

If we now assume that the full key candidates $\texttt{c}$ may be written as a sequence of chunks as for $\texttt{ch}$, i.e., $\texttt{c} = \texttt{c}^0 || \texttt{c}^1 || \dots || \texttt{c}^{\mathcal{N}-1}$, with $\texttt{c}^i$ being a sequence of w bits, then we may also assign a score to each of the at most 2^w values for a chunk candidate $\texttt{c}^i$.

$$S\left(\texttt{c}^i, \texttt{ch}^i\right) = n^i_{00} \log(1-\alpha) + n^i_{01} \log \alpha + n^i_{10} \log \beta + n^i_{11} \log(1-\beta) \tag{1}$$

where the n^i_{ab} values count occurrences of bits across the i_{th} chunks, $\texttt{c}^i, \texttt{ch}^i$. Because of $S(\texttt{c}, \texttt{ch}) = \sum_{i=0}^{\mathcal{N}-1} S\left(\texttt{c}^i, \texttt{ch}^i\right)$, $\mathcal{N}$ lists of chunk candidates, containing up to 2^w entries, may be created. More concretely, each list contains at most 2^w 2-tuples of the form $(score, value)$, where the first component $score$ is a real number (candidate score) and the second component $value$ is a w-bit string (candidate value). Now note that the original key-recovery problem reduces to an enumeration problem that consists in traversing the lists of chunk candidates to produce full key candidates $\texttt{c}$ of which total scores are obtained by summation. Fortunately for us, the enumeration problem has been previously studied in the side-channel analysis literature [12–27], and there are many algorithms that may be useful for our key-recovery setting, in particular those producing full key candidates sorted in descending order based on the score component.

More formally, let $L^i = \left[\texttt{c}^i_0, \texttt{c}^i_2, \dots, \texttt{c}^i_{m_i-1}\right]$ be a list of chunk candidates for chunk $\texttt{ch}^i$, with $0 < m_i \leq 2^w$ and $\texttt{c}^i_j = \left(score^i_j, value^i_j\right)$. Given the chunk candidates $\texttt{c}^{i_0}_{j_0}, \dots, \texttt{c}^{i_n}_{j_n}$, with $0 \leq i_0 < \dots < i_n < \mathcal{N}, 0 \leq j_i < m_i$, we define $\texttt{combine}\left(\texttt{c}^{i_0}_{j_0}, \dots, \texttt{c}^{i_n}_{j_n}\right)$ as a function that outputs a new chunk candidate c computed by setting $\mathsf{c} = \left(\mathsf{c}^{i_0}_{j_0}.score + \ldots + \mathsf{c}^{i_n}_{j_n}.score, \mathsf{c}^{i_0}_{j_0}.value \parallel \ldots \parallel \mathsf{c}^{i_n}_{j_n}.value\right)$. When $i_0 = 0, i_1 = 1, \ldots, i_{\mathcal{N}-1} = \mathcal{N} - 1$, c is known as a full key candidate.

The key enumeration problem entails choosing a chunk candidate $\mathsf{c}^{i}_{j_i}$ from each list L^i, $0 \le i < \mathcal{N}$, to form full key candidates $\mathsf{c} = \texttt{combine}\left(\mathsf{c}^{0}_{j_0}, \ldots, \mathsf{c}^{\mathcal{N}-1}_{j_{\mathcal{N}-1}}\right)$ conditioned to a rule on their scores [27]. An algorithm that enumerates full key candidates c is known as a key enumeration algorithm (KEA).

For our key-recovery problem, we use an algorithm that combines two key enumeration algorithms and is based on one introduced in [11,37]. Moreover, we also exploit the in-memory representation of the secret key of SIKE mechanism to improve our results. This key-recovery algorithm will be described in Section 4.2.

3. Supersingular Isogeny Key Encapsulation Mechanism

This section gives a brief account of the Supersingular Isogeny Key Encapsulation mechanism. This mechanism relies on Supersingular Isogeny Diffie–Hellman (SIDH), a key-exchange protocol introduced by Jao and De Feo in 2011 [2]. In particular, we follow the description provided by SIKE's specification submitted to the NIST post-quantum cryptography standardization process.

3.1. Mathematical Foundations

3.1.1. The Finite Field $\mathbb{F}_p$

Let $\mathbb{F}_p$ be a finite field, with p being a prime number. The elements of this field are represented by the integers in the set $\{0, 1, 2, \ldots, p-1\}$. These elements are operated with the two common operations, addition and multiplication modulo p.

3.1.2. The Finite Field $\mathbb{F}_{p^2}$

The quadratic field extension $\mathbb{F}_{p^2}$ can be constructed easily from $\mathbb{F}_p$. In particular, the elements of $\mathbb{F}_{p^2}$ are represented by $e = e_0 + e_1{\cdot}i$, where $e_0, e_1 \in \mathbb{F}_p$, $i \in \mathbb{F}_{p^2}$, with $i^2 = -1$. Note that e can also be represented as a 2-tuple $(e_0, e_1) \in \mathbb{F}_p^2$. The addition and multiplication operations are defined as follows:

- If $a, b \in \mathbb{F}_{p^2}$, then $(a_0 + a_1{\cdot}i) + (b_0 + b_1{\cdot}i) = (a_0 + b_0) + (a_1 + b_1){\cdot}i \in \mathbb{F}_{p^2}$, where the additions $(a_i + b_i)$ carries out in $\mathbb{F}_p$. Equivalently,
$$(a_0, a_1) + (b_0, b_1) = ((a_0 + b_0) \ mod \ p, (a_1 + b_1) \ mod \ p).$$
- If $a, b \in \mathbb{F}_{p^2}$, then $(a_0 + a_1{\cdot}i){\cdot}(b_0 + b_1{\cdot}i) = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0){\cdot}i \in \mathbb{F}_{p^2}$, where the element operations take place in $\mathbb{F}_p$. Equivalently,
$$(a_0, a_1){\cdot}(b_0, b_1) = ((a_0 b_0 - a_1 b_1) \ mod \ p, (a_0 b_1 + a_1 b_0) \ mod \ p).$$

Since $\mathbb{F}_{p^2}$ is a field, we can compute the additive inverse of a, denoted by $-a$, and also the multiplicative inverse of a, denoted by a^{-1}. To see more details on how all field operations are calculated, we refer the reader to [1].

3.1.3. Montgomery Curves

A Montgomery curve is a special form of an elliptic curve. Let $\mathbb{F}_q$ be a finite field, where $q = p^n$, p a prime number with $p \ge 3$, and let $A, B \in \mathbb{F}_q$ be field elements satisfying $B\left(A^2 - 4\right) \ne 0$ in $\mathbb{F}_q$. A Montgomery curve $E_{A,B}$ defined over $\mathbb{F}_q$, denoted by $E_{A,B}/\mathbb{F}_q$, is defined to be the set of points $P = (x_P, y_P)$ of solutions in $\mathbb{F}_q$ to the equation

$$By^2 = x^3 + Ax^2 + x.$$

The set of points of $E_{A,B}/\mathbb{F}_q$ together with the point at infinity, denoted by O, form a finite abelian group under a point addition operation, denoted by $E_{A,B}\left(\mathbb{F}_q\right)$. That is, given the points P and Q in $E_{A,B}(\mathbb{F}_q)$, we can compute $R = P + Q = Q + P$, where $R = (x_R, y_R)$ is a point in $E_{A,B}(\mathbb{F}_q)$. Consequently, we also can compute $[2]P = P + P$, $[3]P = P + P + P$ or in general $[k]P = P + P + P + \cdots + P$ (k times, with $k \geq 0$). These operations are called point doubling, point tripling or point multiplication, respectively. To see more details on how to do these computations, we refer the reader to [1].

The order of an elliptic curve E over a finite field $\mathbb{F}_q$, denoted by $\#E(\mathbb{F}_q)$, is the number of points in E including O. Additionally, the order $ord(P)$ of a point P is the smallest positive integer n such that $[n]P = O$.

Given an abelian group G, we say a set of elements $\{P_1, P_2, \cdots, P_t\} \subseteq H$ forms a basis of the subgroup $H \leq G$ if every element P of H can be represented by a unique expression of the form $P = [k_1]P_1 + [k_2]P_2 + \ldots + [k_t]P_t$, where $0 \leq k_i \leq ord(P_i)$ for $1 \leq i \leq t$.

Consider an elliptic curve $E(\mathbb{F}_q)$ and a positive integer m, we define the group of the m-torsion elements of $E(\mathbb{F}_q)$ as $E[m] = \left\{P \in E\left(\overline{\mathbb{F}_q}\right) \ : \ [m]P = O\right\}$, where $\overline{\mathbb{F}_q}$ is the algebraic closure of $\mathbb{F}_q$. An elliptic curve $E(\mathbb{F}_q)$ over a field of characteristic p is called supersingular if $p \mid (q + 1 - \#E(\mathbb{F}_q))$, and ordinary otherwise. Additionally, the j-invariant of the elliptic curve $E_{A,B}$ can be calculated as $j(E_{A,B}) = \frac{256(A^2-3)^3}{A^2-4}$. The j-invariant of an elliptic curve over a field $\mathbb{F}_q$ is unique up to the isomorphism of the elliptic curve. The SIKE protocol defines a shared secret as a j-invariant of an elliptic curve.

3.1.4. Isogenies

Let E_1 and E_2 be elliptic curves over a finite field $\mathbb{F}_q$. An isogeny $\Phi : E_1 \to E_2$ is a non-constant rational map defined over $\mathbb{F}_q$, with Φ being a group homomorphism from $E_1(\mathbb{F}_q)$ to $E_2(\mathbb{F}_q)$. Additionally, Φ can be written as $\Phi(x, y) = (f(x), y \cdot g(x))$, where f and g are rational maps over $\mathbb{F}_q$. Note that we can write $f(x) = p(x)/q(x)$ with polynomials $p(x)$ and $q(x)$ over $\mathbb{F}_q$, where $p(x)$ and $q(x)$ do not have a common factor, and similarly for $g(x)$. We define the degree $deg(\Phi)$ of the isogeny to be $max\{deg(p(x)), deg(q(x))\}$, where $p(x)$ and $q(x)$ are as above. Given an isogeny $\Phi : E_1 \to E_2$, $ker(\Phi) = \{P \in E_1 \ : \ \Phi(P) = O\}$ denotes the kernel of Φ. Additionally, given a finite subgroup $H \leq E_1(\mathbb{F}_q)$, there is a unique isogeny (up to isomorphism) $\Phi : E_1 \to E_2$ such that $ker(\Phi) = H$ and $deg(\Phi) = |H|$, where the curve $E_2 \cong E_1/H$. We can compute the isogeny Φ and the curve E_1/H for a given a subgroup $H \leq E_1(\mathbb{F}_q)$, using Vélu's formula [39]. However, it is computationally impractical for arbitrary subgroups. SIKE uses isogenies over subgroups that are powers of 2, 3 and 4 [1].

3.2. Key Encapsulation Mechanism

This section describes the components of the supersingular isogeny key encapsulation mechanism as in [1].

3.2.1. Public Parameters

We start off by describing the public parameters that SIKE makes use of:

- A prime number of the form $p = 2^{\mathsf{e}_2}3^{\mathsf{e}_3} - 1$, where e_2 and e_3 are two positive integers.
- A starting supersingular elliptic curve defined as $E_0/\mathbb{F}_{p^2} : y^2 = x^3 + 6x^2 + x$, such that $\#E_0\left(\mathbb{F}_{p^2}\right) = (2^{\mathsf{e}_2}3^{\mathsf{e}_3})^2$ and $j(E_0) = 287{,}496$.
- Two points P_2 and Q_2 in $E_0[2^{\mathsf{e}_2}]$, such that both points have exact order 2^{e_2} and $\{P_2, Q_2\}$ forms a basis for $E_0\left(\mathbb{F}_{p^2}\right)[2^{\mathsf{e}_2}]$.
- Two points P_3 and Q_3 in $E_0[3^{\mathsf{e}_3}]$, such that both points have exact order 3^{e_3} and $\{P_3, Q_3\}$ forms a basis for $E_0\left(\mathbb{F}_{p^2}\right)[3^{\mathsf{e}_3}]$.

These points are normally encoded for efficiency. Particularly, the points P_2 and Q_2 are encoded as a byte array from the three x-coordinates x_{P_2}, x_{Q_2} and x_{R_2}, where $R_2 = P_2 - Q_2$. Similarly, the points P_3 and Q_3 are encoded as a byte array from the three x-coordinates x_{P_3}, x_{Q_3} and x_{R_3}, where $R_3 = P_3 - Q_3$ [1].

3.2.2. Secret Keys

Two secret keys, $\texttt{sk}_2$ and $\texttt{sk}_3$, are used to compute $2^{\texttt{e}_2}$-isogenies and $3^{\texttt{e}_3}$-isogenies, respectively. On the one hand, the secret key $\texttt{sk}_2$ is an integer in the set $\{0, 1, \ldots, 2^{\texttt{e}_2} - 1\}$ and is stored as a byte array of length $\texttt{N}_{\texttt{sk}_2} = \lceil \frac{\texttt{e}_2}{8} \rceil$. The corresponding keyspace is denoted $\mathcal{K}_2$. On the other hand, the secret key $\texttt{sk}_3$ is an integer in the set $\left\{0, 1, \ldots, 2^{\lfloor \log_2 3^{\texttt{e}_3} \rfloor} - 1\right\}$ and is stored as a byte array of length $\texttt{N}_{\texttt{sk}_3} = \left\lceil \frac{\lfloor \log_2 3^{\texttt{e}_3} \rfloor}{8} \right\rceil$. The corresponding keyspace is denoted $\mathcal{K}_3$.

3.2.3. Isogeny Algorithms

Let $l, m \in \{2, 3\}$ with $l \neq m$. The two core isogeny algorithms used in SIKE are $\texttt{isogen}_l$ and $\texttt{isoex}_l$. On the one hand, the algorithm $\texttt{isogen}_l$ takes in the public parameters and a private key $\texttt{sk}_l$ as inputs and returns the corresponding public key $\texttt{pk}_l$. Specifically, it computes $S = (x_S, y_S) = P_l + [\texttt{sk}_l]Q_l$ and then computes the $l^{\texttt{e}_l}$-degree isogeny $\Phi : E_0 \to E_{a',b'}$ via the composition of $\texttt{e}_l$ individual l-degree isogenies, where $E_{a',b'} \cong E_0 / \langle S \rangle$, and returns the points $\Phi(P_m)$ and $\Phi(Q_m)$ encoded as a byte array from three x-coordinates as the public key, i.e., $\texttt{pk}_l = \left(x_{\Phi(P_m)}, x_{\Phi(Q_m)}, x_{R_m}\right)$, where $R_m = \Phi(P_m) - \Phi(Q_m)$.

On the other hand, the algorithm $\texttt{isoex}_l$ receives a secret key $\texttt{sk}_l$ and a public key $\texttt{pk}_m$ as inputs, and then outputs the corresponding shared key. Specifically, on receiving a private key $\texttt{sk}_l$ and an encoded public key $\texttt{pk}_m = \left(x'_{P_l}, x'_{Q_l}, x'_{R_l}\right)$, it then decodes $\texttt{pk}_m$ to get the corresponding curve $E_{a',b'}$ and the points P'_l and Q'_l, then computes $S = (x_S, y_S) = P'_l + [\texttt{sk}_l]Q'_l$ and the $l^{\texttt{e}_l}$-degree isogeny $\Phi : E_{a',b'} \to E_{a,b}$ via the composition of $\texttt{e}_l$ individual l-degree isogenies, where $E_{a,b} \cong E_{a',b'} / \langle S \rangle$, and finally returns $j(E_{a,b})$.

3.2.4. The Reference Implementation

In this section, we review the SIKE's main algorithms as in the reference implementation (see the file sike.c). Reviewing this implementation is of great importance to learn the in-memory representation used to store private keys.

Let $\texttt{n}$ denote the length in bits of the messages to be encapsulated, i.e., $\texttt{n} \in \{128, 192, 256\}$, and let $\texttt{k}$ denote the number of bits of the output shared key, i.e., $\texttt{k} \in \{128, 192, 256\}$. Also, $\texttt{SHAKE}_{256}(\texttt{m}, d)$ denotes hashing the message $\texttt{m}$ and obtaining only d bits from the output, i.e., the output may be regarded as an infinite bit string, of which computation is stopped after the desired number of output bits is produced [40].

The first algorithm to describe is the key generation algorithm. It is the most relevant algorithm for our work, so we describe it in more detail. Algorithm 1 shows a pseudo-code of the key generation algorithm's implementation. It takes in $\texttt{Params}$, that contains $\texttt{n}$, $\texttt{N}_{\texttt{sk}_3}$, and $\texttt{MASK}$, and first generates a random $\lceil \texttt{n}/8 \rceil$ byte array, then generates a secret key $\texttt{sk}_3$ chosen uniformly at random in the keyspace $\mathcal{K}_3$ and finally calls $\texttt{isogen}_3$ to get the corresponding encoded public key. The in-memory representation of the private key is a byte array of length $\lceil \texttt{n}/8 \rceil + \texttt{N}_{\texttt{sk}_3} + \texttt{l}_{\texttt{pk}}$, where $\texttt{l}_{\texttt{pk}}$ is the length in bytes of the encoded public key. The first $\lceil \texttt{n}/8 \rceil$ bytes correspond to $\texttt{s}$, the next $\texttt{N}_{\texttt{sk}_3}$ bytes correspond to the secret key $\texttt{sk}_3$ and finally the last bytes correspond to the encoded public key $\texttt{pk}_3$. Note that $\texttt{pk}_3$ is the only publicly known part stored in the private array.

Algorithm 1 Key generation algorithm for SIKE.

```
function KeyGen(Params)
    // generate a random $\lceil n/8 \rceil$ byte array.
    $s \leftarrow$ randombytes($\lceil$Params.n/8$\rceil$);
    // generate a random $N_{sk_3}$ byte array.
    $sk_3 \leftarrow$ randombytes(Params.$N_{sk_3}$);
    // masking last byte of $sk_3$ to guarantee $sk_3$ is in $\mathcal{K}_3$. MASK depends on the chosen parameters.
    $sk_3$[Params.$N_{sk_3} - 1$] $\leftarrow$ $sk_3$[Params.$N_{sk_3} - 1$] & Params.MASK;
    // generate the corresponding public key $pk_3$.
    $pk_3 \leftarrow$ $isogen_3$(sk3,Params);
    return (s,$sk_3$,$pk_3$);
end function
```

Regarding the encapsulation mechanism for SIKE, as described by Algorithm 2, it basically generates a message in the message space and hashes it along with the given public key to get r, which is passed as key in $\mathcal{K}_2$ to $\mathtt{isogen}_2$ to compute the corresponding public key c_0 and the shared string j. The value j is hashed to get h, which is xored with m to get c_1, and then the shared key K is computed and returned, along with the ciphertext (c_0, c_1).

Algorithm 2 Encapsulation algorithm for SIKE.

```
function Encaps($pk_3$,Params)
    $m \leftarrow$ randombytes($\lceil$Params.n/8$\rceil$);
    $r \leftarrow$ $SHAKE_{256}$(m || $pk_3$,Params.$e_2$);
    $c_0 \leftarrow$ $isogen_2$(r,Params);
    $j \leftarrow$ $isoex_2$($pk_3$,r,Params.$e_2$);
    $h \leftarrow$ $SHAKE_{256}$(j,Params.n);
    $c_1 \leftarrow h \oplus m$;
    $K \leftarrow$ $SHAKE_{256}$(m || ($c_0$,$c_1$),Params.k);
    return (($c_0$,$c_1$),K);
end function
```

Regarding the decapsulation mechanism for SIKE, as described by Algorithm 3, it basically computes the shared string j from c_0 and the secret key sk_3. It then recomputes h, then m and then r', which is used to compute c_0'. At this point, it compares c_0' with the given c_0. If they are equal, then the real shared key K is computed and returned, otherwise a fake shared key is computed and returned.

Algorithm 3 Decapsulation algorithm for SIKE.

```
function Decaps((s,$sk_3$,$pk_3$),($c_0$,$c_1$),Params )
    $j \leftarrow$ $isoex_3$($c_0$,$sk_3$,Params.$e_3$);
    $h \leftarrow$ $SHAKE_{256}$(j,Params.n);
    $m' \leftarrow h \oplus c_1$;
    $r' \leftarrow$ $SHAKE_{256}$(m' || $pk_3$,Params.$e_2$);
    $c_0' \leftarrow$ $isogen_2$(r',Params);
    if $c_0' = c_0$ then
        $K \leftarrow$ $SHAKE_{256}$(m' || ($c_0$,$c_1$),Params.k);
    else
        $K \leftarrow$ $SHAKE_{256}$(s || ($c_0$,$c_1$),Params.k);
    end if
    return K;
end function
```

SIKE Parameters

Table 1 shows relevant entries from each SIKE parameter for our key-recovery attack. For a comprehensive list of all entries defined for each SIKE parameter, such as P_2, Q_2,R_2, P_3, Q_3 and R_3, we refer the reader to [1,9].

Table 1. Relevant constants from Supersingular Isogeny Key Encapsulation (SIKE) parameters.

Name	$\lfloor \log_2 3^{e_3} \rfloor$	MASK	$\mathtt{N}_{\mathtt{sk}_3}$	$\mathtt{e}_2$	$\mathtt{e}_3$
SIKEp434	218	0x01	28	216	137
SIKEp503	253	0x0F	32	250	159
SIKEp610	305	0xFF	38	305	192
SIKEp751	379	0x03	48	372	239

4. Key Recovery

Throughout this section, we give a detailed account of our key-recovery algorithm, which relies on the general key-recovery strategy described in [11,27].

4.1. Assumptions

Based on the model we described in Section 2.2, we suppose that an adversary procures a noisy version of a SIKE's private key encoding as it is stored by the reference implementation. In particular, the adversary procures an array containing $(\mathtt{s}, \mathtt{sk}_3, \mathtt{pk}_3)$. Additionally, such adversary accesses the corresponding original public parameters (i.e., without noise), so such adversary may use them to properly locate the offset from which the secret key bits ($\mathtt{sk}_3$) are stored via running key-finding algorithms exploiting identifying features found in the memory formats for storing the SIKE's private key. These algorithms would be similar to those developed for the RSA setting [10].

Additionally, by $\alpha = P(0 \to 1)$ we denote the probability for a 0 bit of the original secret key to change for a 1 bit, and by $\beta = P(1 \to 0)$ we denote the probability for a 1 bit of the original private key to change for a 0 bit. According to our model, the attacker knows both α and β, since these values are fixed across the memory region in which the secret key is located and can be easily estimated by comparing any noisy content with its corresponding original one (e.g., the public key). Moreover, memory regions are normally large and the size in bytes of $\mathtt{sk}_3$ is at most 48 bytes according to Table 1.

4.2. Our Key-Recovery Algorithm

Throughout this section we give a detailed account of our key-recovery strategy, taking into consideration the notation introduced in Section 2.3.5 and the review of SIKE's reference implementation.

Let $\mathtt{ch} = b_0 b_1 b_2 \ldots b_W$ be W bit string that represents the noisy version of the secret key encoding $\mathtt{sk}_3$. Note that the value for W depends on the SIKE parameters on which SIKE is configured. In particular, $W = 8 \cdot \mathtt{N}_{\mathtt{sk}_3}$ (see the fourth column of Table 1). We denote w as the size in bits of a chunk (it is set to 8), and so $\mathtt{ch}$ can be written as a sequence of $\mathcal{N} = W/w$ chunks

$$\mathtt{ch} = \mathtt{ch}^0 || \mathtt{ch}^1 || \ldots || \mathtt{ch}^{\mathcal{N}-1}$$

with $\mathtt{ch}^i = b_{i \cdot w} b_{i \cdot w+1} \ldots b_{i \cdot w+(w-1)}$. We define a block $\mathtt{B}^k_m = \mathtt{ch}^k || \mathtt{ch}^{k+1} || \cdots || \mathtt{ch}^{k+m-1}$ as a sequence of consecutive chunks, where $0 \le k \le k+m-1 < \mathcal{N}$ and m is the number of chunks in the block. We can now rewrite $\mathtt{ch}$ as a concatenation of n_b blocks, namely

$$\mathtt{ch} = \mathtt{B}^0_{m_0} || \mathtt{B}^{m_0}_{m_1} || \ldots || \mathtt{B}^{m_0+m_1+m_2+\ldots+m_{n_b-2}}_{m_{n_b-1}}$$

with $\sum_{i=0}^{n_b-1} m_i = \mathcal{N}$. Now we are ready to present our key-recovery algorithm.

1. Set the array $\texttt{P} = [m_0, m_1, m_2, \ldots, m_{n_b-1}]$ with $\sum_{j=0}^{n_b-1} m_j = \mathcal{N}$. Each m_j is the number of chunks in the block $\texttt{B}_{m_j}^{m_0+m_1+\ldots+m_{j-1}}$.
2. Set the array $\texttt{M} = [M_0, M_1, M_2, \ldots, M_{n_b-1}]$. Each M_j is the number of high-scoring candidates to generate for the block $\texttt{B}_{m_j}^{m_0+m_1+\ldots+m_{j-1}}$.
3. By making use of Equation (1), our algorithm computes the log-likelihood scores for all possible candidate values for each chunk $\texttt{ch}^i$, with $0 \leq i < \mathcal{N}$. Note that, in the SIKE setting, all candidate values are in the set $\{0, 1, 2, \ldots 255\}$, except for the last chunk, where all candidate values are in the set $\{0, 1, 2, \ldots, 2^{n_s} - 1\}$ with n_s being the number of ones of the binary string `Params.MASK`. Each tuple $(score, value)$ generated for a particular chunk $\texttt{ch}^i$ is inserted into the list $L_{\texttt{ch}^i}$, and this is finally inserted in the corresponding index i of the main list L.
4. For each block $\texttt{B}_{m_j}^{m_0+m_1+\ldots+m_{j-1}}$, with $0 \leq j < n_b$ and $m_{-1} = 0$, then the lists stored in L at indexes $m_0 + m_1 + \ldots + m_{j-1} + k$ for $0 \leq k < m_j$ are given as parameters to an optimal key enumeration algorithm, which we denote as `OKEA`, to produce the list $L_{\texttt{B}^j}$ that contains the M_j candidates with the highest scores for the block $\texttt{B}_{m_j}^{m_0+m_1+\ldots+m_{j-1}}$.
5. All lists $L_{\texttt{B}^j}$ are given as parameters to an instance of a key enumeration algorithm (`KEA`). This instance regards each $L_{\texttt{B}^j}$ as a list of suitable candidates for the corresponding block $\texttt{B}_{m_j}^{m_0+m_1+\ldots+m_{j-1}}$ and outputs high-scoring full key candidates for the target secret key encoding. Upon generating a full key candidate `c`, our algorithm passes it as input to a testing function (`Test` in our case) to verify whether the full key candidate `c` is a valid secret key or not.

Algorithm 4 shows a pseudo-code of our key-recovery algorithm. Note that `OKEA` points to the optimal key enumeration algorithm that our key-recovery algorithm runs at step 4, and `KEA` points to the key enumeration algorithm that our key-recovery algorithm runs at step 5. Additionally, `args` refers to the arguments that the algorithm pointed by `KEA` may take in and `Test` is a pointer to the testing function `Test`, i.e., Algorithm 5. We next expand on our choice of key enumeration algorithms for our key-recovery algorithm.

On Our Choice of an Optimal Key Enumeration Algorithm

Our key-recovery algorithm requires an optimal key enumeration algorithm run at step 4 to enumerate full key candidate `c` sorted by their scores in descending order. There are several algorithms suitable for this task [12,27,41], and we use the algorithm (`OKEA`) from [12], since it is flexible and easy to implement. Additionally, note that `OKEA` is an inherently sequential algorithm, and so its optimality property is fulfilled only if it is not run in parallel. In addition, its memory consumption may be high if used to enumerate many candidates [27]. In our setting, we select M_j to be a value in the set $\{256, 512, 1024\}$ for all $0 \leq j < n_b$.

On Our Choice of Non-Optimal Key Enumeration Algorithms

Regarding the key enumeration algorithm (`KEA`) that our key-recovery algorithm runs at step 5, this algorithm is required to be parallelizable and memory-efficient, since this step of our approach requires a considerably high amount of computational resources. Here we explore the use of two different key enumeration algorithms. On the one hand, we use an algorithm called `NOKEA`, which combines several good characteristics from multiples algorithms (`KEA`*s*) [11,37]. This particular algorithm may be configured to search over a well-defined range $[a, b]$, where $a \leq b \leq max$ and max is the maximum total score that a full key candidate may be given to. In this configuration, this algorithm enumerates each possible full key candidate of which total score lies in the chosen interval $[a, b]$ [11]. We expand on the use of this algorithm in Section 5.1, in which we also present the success rates of our key-recovery algorithm. On the other hand, we also explore how to combine our key-recovery algorithm with a quantum key enumeration algorithm [42], which we denote by QKEA, that combines a non-optimal key enumeration algorithm and the Grover algorithm [43]. We expand on this in Section 5.2.

Algorithm 4 Key-recovery algorithm.

```
function KeyRecovery(ch, Params, P, M, $\alpha$, $\beta$, OKEA, KEA, args, Test)
    for ($i \leftarrow 0, i < \texttt{Params.N}_{\texttt{sk}_3} - 1, i \leftarrow i+1$) do
        for ($\texttt{c} \leftarrow 0, \texttt{c} < 256, \texttt{c} \leftarrow \texttt{c}+1$) do
            //with Equation (1), getScore computes a score for each candidate c.
            score $\leftarrow$ getScore(ch[$i$], c, $\alpha$, $\beta$);
            $L_{\texttt{ch}^i}$.add((score, c));// adds the tuple (score, c) to the list.
        end for
        $L_{\texttt{ch}^i}$.sort()// this sorts the list $L_{\texttt{ch}^i}$ by score in descending order.
        $L$.add($L_{\texttt{ch}^i}$)// this appends the list $L_{\texttt{ch}^i}$ to the index $i$ of $L$.
    end for
    $i \leftarrow \texttt{Params.N}_{\texttt{sk}_3} - 1$;
    for $\left(\texttt{c} \leftarrow 0, \texttt{c} < 2^{\texttt{HammingWeight(Params.MASK)}}, \texttt{c} \leftarrow \texttt{c}+1\right)$ do
        score $\leftarrow$ getScore(ch[$i$], c, $\alpha$, $\beta$);// with Equation (1), getScore computes a score for each candidate c.
        $L_{\texttt{ch}^i}$.add((score, c));// adds the tuple (score, c) to the list.
    end for
    $L_{\texttt{ch}^i}$.sort()// this sorts the list $L_{\texttt{ch}^i}$ by score in descending order.
    $L$.add($L_{\texttt{ch}^i}$)// this appends the list $L_{\texttt{ch}^i}$ to the index $i$ of $L$.
    $n_{\texttt{B}} \leftarrow$ len(P);
    $s \leftarrow 0$;
    for ($j \leftarrow 0, j < n_{\texttt{B}}, j \leftarrow j+1$) do
        // The next instruction inits OKEA passing it the lists $L_{\texttt{ch}^s}, \ldots, L_{\texttt{ch}^{s+\texttt{P}[j]-1}}$ obtained from $L$.
        OKEA.init($L$, $s$, P[$j$]);
        for ($i \leftarrow 1, i \leq \texttt{M}[j], i \leftarrow i+1$) do
            // The next instruction returns the best block candidate for block $j$ and appends it to $L_{\texttt{B}^j}$.
            $L_{\texttt{B}^j}$.add(OKEA.getNextCandidate());
        end for
        $L_{\texttt{B}}$.add($L_{\texttt{B}^j}$); // $L_{\texttt{B}}$ is an accessory list to store lists $L_{\texttt{B}^j}$.
        $s \leftarrow s + \texttt{P}[j]$; // this updates $s$, the starting index for the next block.
    end for
    // The next instruction inits KEA passing it the lists in $L_{\texttt{B}}$ and args.
    KEA.init($L_{\texttt{B}}$, args);
    // The next instruction runs KEA passing it Params and a pointer to the function Test, i.e., Algorithm 5.
    KEA.run(Test, Params);
end function
```

Algorithm 5 Test Function.

```
function Test(c, $\texttt{pk}_3$, Params )
    // generate a public key $\texttt{pk}_3'$ from the candidate c.
    $\texttt{pk}_3' \leftarrow \texttt{isogen}_3$(c, Params);
    if Arrays.equal($\texttt{pk}_3$, $\texttt{pk}_3'$) then// verify whether $\texttt{pk}_3$ and $\texttt{pk}_3'$ are equal.
        return 1;
    end if
    return 0;
end function
```

5. Evaluation of Our Key-Recovery Algorithm

Throughout this section we assess our key-recovery algorithm. We first give a detailed account of how we compute the success rates for our key-recovery algorithm considering several scenarios, each with a different set of SIKE parameters, and finally show how to integrate a quantum key search algorithm (QKEA) to our key-recovery algorithm to improve its overall performance.

5.1. Performance and Success Rates of Our Key-Recovery Algorithm

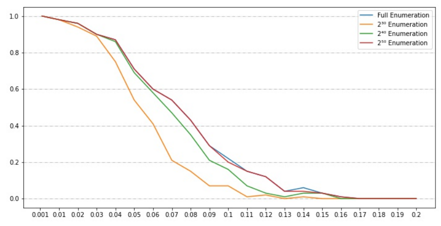
This section reports success rates of our key-recovery algorithm. We first note that in this setting the attacker procures a noisy version ch of $\mathtt{sk}_3$ via a cold boot attack, as stated in Section 4.1, and so our key-recovery algorithm tries to find a full key candidate for $\mathtt{sk}_3$ from the noisy version. In order to compute estimates for success rates of our key-recovery algorithm, we analyze the following scenarios. For a given set of instance parameters, we calculate the success rate for our key-recovery algorithm with NOKEA if it is set to (1) enumerate all the possible full key candidates; (2) enumerate an interval with roughly 2^{30} full key candidates; (3) enumerate an interval with roughly 2^{40} full key candidates; (4) enumerate an interval with roughly 2^{50} full key candidates. In this context, by instance parameters we mean setting the array P, the array M, w (normally 8), α, β, together with the selected SIKE parameters.

Note that given the noisy version ch and the instance parameters, our key-recovery algorithm creates the lists $L_{\mathtt{B}j}$ at the end of step 4, then these are passed to NOKEA at the beginning of the step 5. Here is where we exploit several features of this algorithm in order to compute estimates for the success rates without actually running an enumeration with NOKEA. In particular, we exploit the fact that NOKEA can enumerate full key candidates of which total scores belong to a given interval in order from the highest to the lowest total score and that many scores of block candidates for a given block are repeated [11,36,44]. In particular, once the lists $L_{\mathtt{B}j}$ are created, our tweaked NOKEA creates factorized lists, one per list $L_{\mathtt{B}j}$, that contain entries of the form $(score, count)$ where *score* represents a score and *count* is the number of block candidates having *score* as score in the corresponding list $L_{\mathtt{B}j}$. These factorized lists are passed to an alternative version of OKEA, where *count* is seen as a candidate value, and it generates another factorized list L with entries of the same form $(score, count)$, which is sorted by score in descending order, i.e., from the highest to the lowest score. An entry $(score, count)$ may be interpreted as there are *count* full key candidates of which total score is equal to *score*.

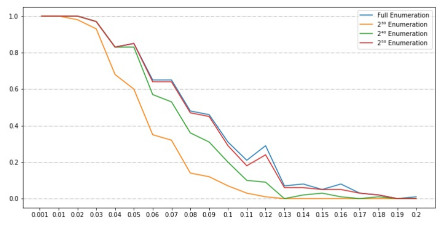
To calculate estimates for success rates, we run our tweaked key-recovery algorithm with all required parameters a fixed number of times (100 times) and then compute the corresponding success rate. Specifically, at each trial the SIKE key generation algorithm is run to obtain $(\mathtt{s}, \mathtt{sk}_3, \mathtt{pk}_3)$ according to the selected set of SIKE parameters, and then $\mathtt{sk}_3$ is perturbed as per α and β and then our tweaked key-recovery algorithm is called by passing the noisy version ch, the instance parameters and the original key as parameters. It then runs until generating the lists $L_{\mathtt{B}j}$ and, at such point, checks that each block of the original key is found in the corresponding list $L_{\mathtt{B}j}$. If so, it means that a complete enumeration, carried out by NOKEA, would not fail in finding the secret key (it would fail otherwise). On the condition that the complete enumeration will not fail, the tweaked NOKEA is run to create the factorized list L, which is used to determine if the original key may be found by the instance of NOKEA if it enumerates a range having roughly 2^{30} or 2^{40} or 2^{50} full key candidates. In particular, say, we want to determine if the original key can be found via enumerating a range with approximately 2^a full key candidates, $a \in \{30, 40, 50\}$, then the algorithm finds an entry at index e in L such that e is the smallest integer satisfying $2^a < \sum_{k=0}^{e} L[k].count$. If the score calculated for the original key (from the noisy version) lies in the interval $[L[e].score, L[0].score]$, then it signifies that an enumeration in such interval (which contains at least 2^a candidates) will find the original key. We next describe the instance parameters we used to run our trials.

SIKEp434

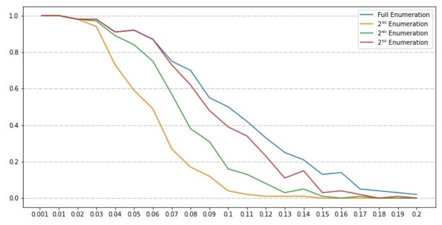
For this set of SIKE parameters, $\mathtt{sk}_3$ has a length of 28 bytes, $\mathtt{Params.MASK} = \mathtt{0x01}$, which means that only the first bit of the last byte of $\mathtt{sk}_3$ is used. Also, we set w to 8, P to $[4, 4, 4, 4, 4, 4, 4]$ and M_j to r, for $0 \leq j < 7$, where $r \in \{256, 512, 1024\}$. Note that the number of chunks is 28 and the number of blocks is 7. Figures 1 and 2 show the obtained results for this set of parameters.



(**a**) $M_j = 256\ for\ 0 \le j < 7$

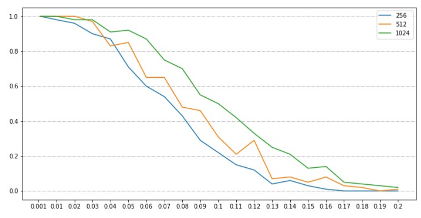


(**b**) $M_j = 512\ for\ 0 \le j < 7$

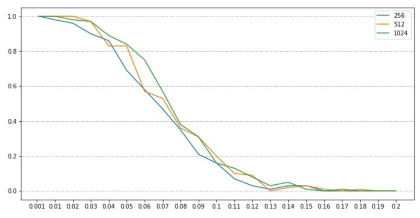


(**c**) $M_j = 1024\ for\ 0 \le j < 7$

Figure 1. Success rates for the SIKE parameters `SIKEp434` for $\alpha = 0.001$. The y-axis denotes the success rate, while the x-axis denotes $\beta \in \{0.001, 0.01, 0.02, \ldots, 0.2\}$.



(**a**)



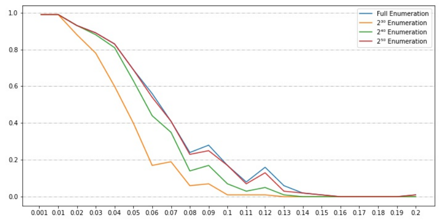
(**b**)

Figure 2. Success rates for the SIKE parameters `SIKEp434` for $\alpha = 0.001$. The y-axis denotes the success rate, while the x-axis denotes $\beta \in \{0.001, 0.01, 0.02, \ldots, 0.2\}$. (**a**) Comparing success rates for $M_j \in \{256, 512, 1024\}$ in a full enumeration. (**b**) Comparing success rates for $M_j \in \{256, 512, 1024\}$ in a 2^{40} enumeration.

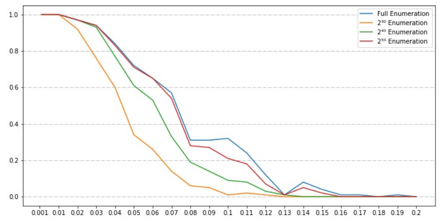
Additionally, the function `Test` for this set of parameters takes about 5,141,167 cycles to run in a virtual machine type e2-highcpu-16 with 16 vCPUs and 16 GB of memory deployed in the Google Cloud Platform.

SIKEp503

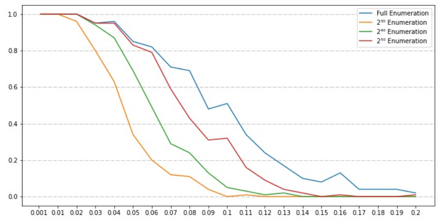
For this set of SIKE parameters, $\texttt{sk}_3$ has a length of 32 bytes, `Params.MASK` $=$ `0x0F`, which means that only the four first bits of the last byte of $\texttt{sk}_3$ are used. Also, we set w to 8, `P` to $[4, 4, 4, 4, 4, 4, 4, 4]$ and M_j to r, for $0 \le j < 8$, where $r \in \{256, 512, 1024\}$. Note that the number of chunks is 32 and the number of blocks is 8. Figures 3 and 4 show the obtained results for this set of parameters.



(**a**) $M_j = 256\ for\ 0 \le j < 8$

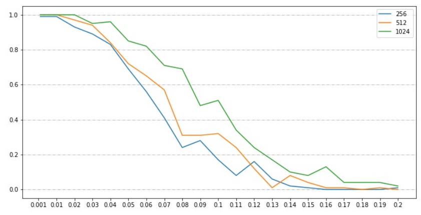


(**b**) $M_j = 512\ for\ 0 \le j < 8$

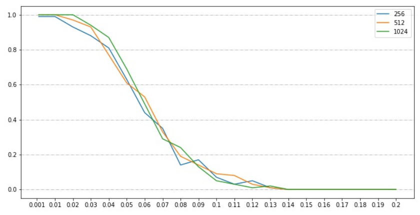


(**c**) $M_j = 1024\ for\ 0 \le j < 8$

Figure 3. Success rates for the SIKE parameters `SIKEp503` for $\alpha = 0.001$. The y-axis denotes the success rate, while the x-axis denotes $\beta \in \{0.001, 0.01, 0.02, \ldots, 0.2\}$.



(**a**)



(**b**)

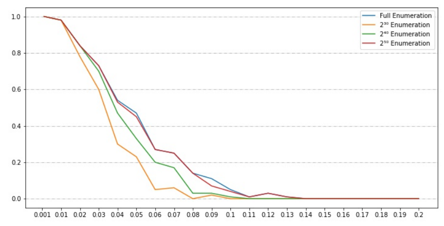
Figure 4. Success rates for the SIKE parameters `SIKEp503` for $\alpha = 0.001$. The y-axis denotes the success rate, while the x-axis denotes $\beta \in \{0.001, 0.01, 0.02, \ldots, 0.2\}$. (**a**) Comparing success rates for $M_j \in \{256, 512, 1024\}$ in a full enumeration. (**b**) Comparing success rates for $M_j \in \{256, 512, 1024\}$ in a 2^{40} enumeration.

Additionally, the function `Test` for this set of parameters takes about 7,069,164 cycles to run in a virtual machine type e2-highcpu-16 with 16 vCPUs and 16 GB of memory deployed in the Google Cloud Platform.

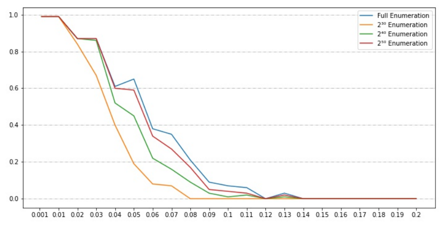
We remark that the success rates for this set of parameters and those obtained for LUOV [37] are essentially similar. This is expected since both LUOV's secret key and SIKE's secret key for this case are of length 32 bytes.

SIKEp610

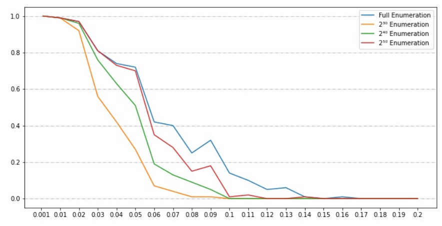
For this set of SIKE parameters, $\mathtt{sk}_3$ has a length of 38 bytes, `Params.MASK` $=$ `0xFF`, which means that all the bits of the last byte of $\mathtt{sk}_3$ are used. We also set w to 8, `P` to $[5, 5, 5, 5, 5, 5, 5, 3]$ and M_j to r, for $0 \le j < 8$, where $r \in \{256, 512, 1024\}$. Note that the number of chunks is 38 and the number of blocks is 8. Figures 5 and 6 show the obtained results for this set of parameters.



(**a**) $M_j = 256\ for\ 0 \leq j < 8$

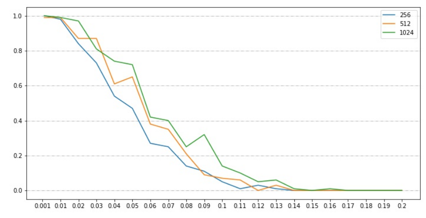


(**b**) $M_j = 512\ for\ 0 \leq j < 8$

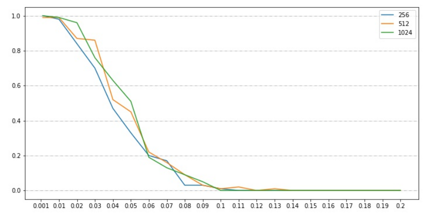


(**c**) $M_j = 1024\ for\ 0 \leq j < 8$

Figure 5. Success rates for the SIKE parameters `SIKEp610` for $\alpha = 0.001$. The y-axis denotes the success rate, while the x-axis denotes $\beta \in \{0.001, 0.01, 0.02, \ldots, 0.2\}$.



(**a**)



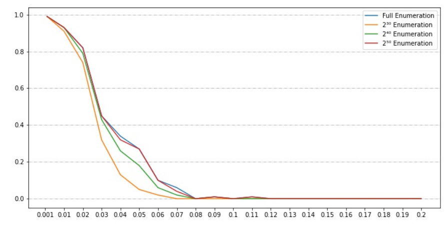
(**b**)

Figure 6. Success rates for the SIKE parameters `SIKEp610` for $\alpha = 0.001$. The y-axis denotes the success rate, while the x-axis denotes $\beta \in \{0.001, 0.01, 0.02, \ldots, 0.2\}$. (**a**) Comparing success rates for $M_j \in \{256, 512, 1024\}$ in a full enumeration. (**b**) Comparing success rates for $M_j \in \{256, 512, 1024\}$ in a 2^{40} enumeration.

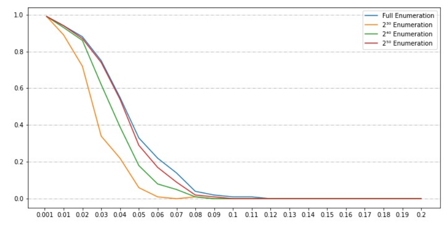
Additionally, the function `Test` for this set of parameters takes about $12,724,569$ cycles to run in a virtual machine type e2-highcpu-16 with 16 vCPUs and 16 GB of memory deployed in the Google Cloud Platform.

SIKEp751

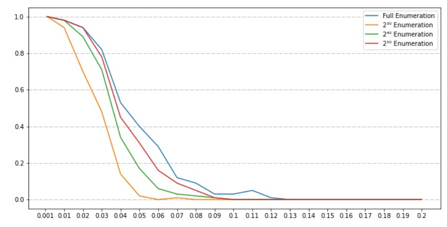
For this set of SIKE parameters, $\mathtt{sk}_3$ has a length of 48 bytes, `Params.MASK` $=$ `0x03`, which means that only the two first bits of the last byte of $\mathtt{sk}_3$ are used. Also, we set w to 8, `P` to $[6, 6, 6, 6, 6, 6, 6, 6]$ and M_j to r, for $0 \leq j < 8$, where $r \in \{256, 512, 1024\}$. Note that the number of chunks is 48 and the number of blocks is 8. Figures 7 and 8 show the obtained results for this set of parameters.



(**a**) $M_j = 256\ for\ 0 \leq j < 8$

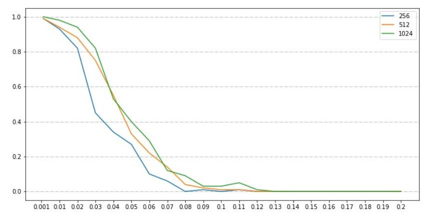


(**b**) $M_j = 512\ for\ 0 \leq j < 8$

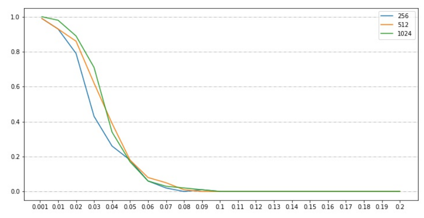


(**c**) $M_j = 1024\ for\ 0 \leq j < 8$

Figure 7. Success rate for the SIKE parameters `SIKEp751` for $\alpha = 0.001$. The y-axis denotes the success rate, while the x-axis denotes $\beta \in \{0.001, 0.01, 0.02, \ldots, 0.2\}$.



(**a**)



(**b**)

Figure 8. Success rates for the SIKE parameters `SIKEp751` for $\alpha = 0.001$. The y-axis denotes the success rate, while the x-axis denotes $\beta \in \{0.001, 0.01, 0.02, \ldots, 0.2\}$. (**a**) Comparing success rates for $M_j \in \{256, 512, 1024\}$ in a full enumeration. (**b**) Comparing success rates for $M_j \in \{256, 512, 1024\}$ in a 2^{40} enumeration.

Additionally, the function `Test` for this set of parameters takes about 21,266,903 cycles to run in a virtual machine type e2-highcpu-16 with 16 vCPUs and 16 GB of memory deployed in the Google Cloud Platform.

Results Discussion

Note that all plots depicted by Figure 1a–c, Figure 3a–c, Figures 5a–c and 7a–c, show a similar trend, namely, for all set of instance parameters and $M_j \in \{256, 512, 1024\}$, the success rate for a complete enumeration dominates the success rate for a 2^{50} enumeration, which in turn dominates the success rate for a 2^{40} enumeration, and which in turn dominates the success rate for a 2^{30} enumeration. This is in alignment with what we expected. Moreover, we remark that the success rate decreases as long as β increases, although there are a few cases where, given β_1 and β_2 with $\beta_1 > \beta_2$, the success rate for β_1 is a bit larger than the success rate for β_2. This is due to the manner in which an interval is constructed by the `NOKEA` algorithm (and hence its tweaked version). However, the general trend is that the success rate decreases while β increases, which is also in line with what is expected. Additionally, Figures 2a, 4a, 6a and 8a show similar trends when comparing success rates for $M_j \in \{256, 512, 1024\}$ in a full enumeration. Analogously, Figures 2b, 4b, 6b and 8b also show similar trends when comparing success rates for $M_j \in \{256, 512, 1024\}$ in a 2^{40} enumeration.

5.2. Integrating a Quantum Key Search Algorithm with Our Key-Recovery Approach

Throughout this subsection we give a detailed account of a quantum key enumeration algorithm, denoted as `QKEA` [42], and show how to combine it with our key-recovery algorithm to improve its overall performance. We additionally derive the running time of step 5 of our key recovery algorithm (worst case) if `QKEA` is run. Recall that at step 5, the lists $L_{\mathtt{B}^j}$, for $0 \le j < n_b$, will be given as parameters to an instance of the quantum key enumeration algorithm (`QKEA`).

`QKEA` relies on a non-optimal key enumeration algorithm developed from a rank algorithm (also known as a counting algorithm) by Martin et al. [17]. The core idea is to leverage the rank algorithm to return a single candidate key with a particular rank within a specific range. Additionally, we remark that this algorithm uses a map that associates each score with a weight (a positive integer), where the smallest weight is regarded as the likeliest one [17].

We assume the scores from each block candidate in the list $L_{\mathtt{B}^j}$, for $0 \le j < n_b$, are mapped to weights, as shown in [27,42]. Given the range $[B_1, B_2)$, the rank algorithm constructs a two dimensional array `B` with $n_b \times B_2$ entries, and then uses it to compute the number of full key candidates in the range. Algorithm 6 constructs the two dimensional array `B` and its running time is given by

$$\begin{aligned} T_1(n_b, M, B_2) &= c_1 + \sum_{b=0}^{B_2-1} \sum_{j=0}^{M-1} c_2 + \sum_{i=0}^{n_b-2} \sum_{b=0}^{B_2-1} \sum_{j=0}^{M-1} c_3 \\ &= c_1 + c_2 \cdot M \cdot B_2 + c_3 \cdot M \cdot (n_b - 1) \cdot B_2 \end{aligned}$$

where $M_j = M \in \{256, 512, 1024\}$ for $0 \le j \le n_b - 1$, $n_b \in \{7, 8\}$ and c_1, c_2, c_3 are constants, i.e., upper bounds on the running time consumed by primitive operations, such as return operations, addition operations, comparison operations.

Algorithm 6 constructs the two dimensional array `B`.

```
1:  function create($B_1, B_2$)
2:    $i \leftarrow n_b - 1$;
3:    B $\leftarrow \left[ [0]^{B_2} \right]^{n_b}$
4:    for $b = 0$ to $B_2 - 1$ do
5:      for $j = 0$ to $M - 1$ do
6:        if $B_1 - b \le \mathtt{c}^i_j.score < B_2 - b$ then
7:          $\mathtt{B}_{i,b} \leftarrow \mathtt{B}_{i,b} + 1$
8:        end if
9:      end for
10:   end for
11:   for $i = n_b - 2$ to $0$ do
12:     for $b = 0$ to $B_2 - 1$ do
13:       for $j = 0$ to $M - 1$ do
14:         if $b + \mathtt{c}^i_j.score < B_2$ then
15:           $\mathtt{B}_{i,b} \leftarrow \mathtt{B}_{i,b} + \mathtt{B}_{i+1, b+\mathtt{c}^i_j.score}$;
16:         end if
17:       end for
18:     end for
19:   end for
20:   return B;
21: end function
```

Note that, by construction, $\mathtt{B}_{0,0}$ is the number of full key candidates of which total scores are in the interval $[B_1, B_2)$. So the rank algorithm simply returns $\mathtt{B}_{0,0}$ and is described by Algorithm 7. Concerning its running time, it is given by $T_2(n_b, M, B_2) = c_4 + T_1(n_b, M, B_2)$, where c_4 is a constant, i.e., an upper bound on the running time of primitive operations.

Algorithm 7 computes the number of full key candidates in $[B_1, B_2)$.

```
1:  function rank($B_1, B_2$)
2:      B ← create($B_1, B_2$);
3:      return $\mathtt{B}_{0,0}$
4:  end function
```

By using Algorithms 7 and 8 returns the full key candidate with rank r in the given range $[B_1, B_2)$ [42]. We remark that Algorithm 8 is deterministic and that the full key candidate returned by it is not necessarily the r^{th} highest-scoring full key candidate in the given range. Regarding Algorithm 8's running time, it is given by

$$\begin{aligned} T_3(n_b, M) &= c_5 + \sum_{i=0}^{n_b-2} \sum_{j=0}^{M-1} c_6 + \sum_{j=0}^{M-1} c_7 \\ &= c_5 + c_6 \cdot M \cdot (n_b - 1) + c_7 \cdot M \end{aligned}$$

where c_5, c_6, c_7 are constants, i.e., upper bounds on the running time consumed by primitive operations.

Algorithm 8 returns the r^{th} full key candidate with weight in the interval $[B_1, B_2)$.

```
1:  function getKey(B, $B_1, B_2, r$)
2:      if $r > \mathtt{B}_{0,0}$ then
3:          return $\perp$
4:      end if
5:      k ← $[0]^{n_b}$;
6:      $b \leftarrow 0$;
7:      for $i = 0$ to $n_b - 2$ do
8:          for $j = 0$ to $M - 1$ do
9:              if $r \leq \mathtt{B}_{i+1, b+\mathtt{c}^i_j.score}$ then
10:                 $\mathtt{k}_i \leftarrow j$;
11:                 $b \leftarrow b + \mathtt{c}^i_j.score$;
12:                 break $j$;
13:             end if
14:             $r \leftarrow r - \mathtt{B}_{i+1, b+\mathtt{c}^i_j.score}$;
15:         end for
16:     end for
17:     $i \leftarrow n_b - 1$;
18:     for $j = 0$ to $M - 1$ do
19:         $x \leftarrow \left(B_1 - b \leq \mathtt{c}^i_j.score < B_2 - b\right) ? 1 : 0$;
20:         if $r \leq x$ then
21:             $\mathtt{k}_i \leftarrow j$;
22:             break $j$;
23:         end if
24:         $r \leftarrow r - x$;
25:     end for
26:     return k
27: end function
```

By calling the `getKey` function, Algorithm 9 tries to enumerate and test all full key candidates in the given interval [42]. Note that `Test` is pointer to the testing function (Algorithm 5). Concerning Algorithm 9's running time, it is given by $T_4(n_b, M, B_2, e) = c_8 + T_1(n_b, M, B_2) + e\cdot(T_3(n_b, M) + c_9 + T(\texttt{Params}))$, where e is the number of full key candidates in the range $[B_1, B_2)$, c_8 and c_9 are constants and upper bounds on the running time consumed by primitive operations, and $T(\texttt{Params})$ is the running time of the testing function (Algorithm 5).

Algorithm 9 enumerates and tests all full key candidates with weight in the interval $[B_1, B_2)$.

```
1:  function keySearch($B_1$, $B_2$, Test)
2:      B ← create($B_1$, $B_2$);
3:      $r \leftarrow 1$;
4:      while True do
5:          k ← getKey(B, $B_1$, $B_2$, $r$);
6:          if k = $\perp$ then
7:              break;
8:          end if
9:          if Test(k) = 1 then
10:             break;
11:         end if
12:         $r \leftarrow r + 1$;
13:     end while
14:     return k;
15: end function
```

Algorithm 10 calls the function `keySearch` to search over partitions selected independently with approximately a^s full key candidates for $s = 0, 1, \ldots$ [42]. Regarding Algorithm 10's running time, this is given by

$$T_5(n_b, m, B_e, B_2, e, a) = c_{10} + \log_2(e)\cdot T_2(n_b, M, B_e) + \sum_{s=0}^{p-1}(T_4(n_b, M, B_2, a^s) + \log_2(a^s)\cdot T_2(n_b, M, B_2) \tag{6}$$

where $p = \log_a(e\cdot(a-1)+1)$ and $e \in \{2^{30}, 2^{40}, 2^{50}\}, c_{10}, c_{11}$ are upper bounds on the running time consumed by primitive operations, such as return operations, addition operations, comparison operations. The terms $\log_2(e)\cdot T_2(n_b, M, B_e)$ and $\log_2(a^s)\cdot T_2(n_b, M, B_2)$ are the running times of steps 5 and 13 of Algorithm 10. Note that this algorithm is similar to `NOKEA` in the sense that it also enumerates full key candidates of which total scores belongs to the given range. Furthermore, the technique of partitioning the entire interval helps in improving the overall performance of Algorithm 10 when it is searching over an initial range with a large number of full key candidates.

Given the computational burden of Algorithm 10 lies on the execution of the function `keySearch` (at step 7), then any improvement on this search algorithm implies a speed-up on Algorithm 10's overall performance. The authors of [42] show how this part may be modified by replacing it for a Grover's oracle [43], and so improving Algorithm 10's overall performance. Algorithm 11 results from adjusting Algorithm 10 and relies on a Grover's oracle, giving a quadratic speed-up on searches over partitions. The function $\texttt{f}(r) = \texttt{Test}(\texttt{getKey}(\texttt{B}, B_1, B_2, r))$ returns 1, on the condition that r is the rank of the real secret key; otherwise, it returns 0. Grover's algorithm [43] shows that r can be found on a quantum computer in only $\mathcal{O}\left(\left|a^{s/2}\right|\cdot T_f\right)$ steps, where T_f is the time to evaluate $\texttt{f}(r)$, i.e., $T(\texttt{Params}) + T_3(n_b, M)$. Therefore Algorithm 11's overall running time is given by replacing $T_4(n_b, M, B_2, a^s)$ in Equation (6) for $T_1(n_b, M, B_2)$ (step 7 of Algorithm 11) plus the running time for Grover's algorithm with $\texttt{f}(r)$ (steps from 8 to 9 of Algorithm 11).

Algorithm 10 performs a non-optimal enumeration over an interval with roughly *e* full key candidates.

```
1:  function KS($e$, Test)
2:      $B_1 \leftarrow B_{min}$;
3:      $B_2 \leftarrow B_{min} + 1$;
4:      $s \leftarrow 0$;
5:      Find $B_e$ such that rank($B_1, B_e$) is roughly equals to $e$;
6:      while $B_1 \le B_e$ do
7:          k $\leftarrow$ keySearch($B_1, B_2$, Test);
8:          if k $\neq \perp$ then
9:              return k;
10:         end if
11:         $s \leftarrow s + 1$;
12:         $B_1 \leftarrow B_2$;
13:         Find $B_2$ such that rank($B_1, B_2$) is roughly equals to $a^s$;
14:     end while
15:     return $\perp$;
16: end function
```

Algorithm 11 performs a quantum key enumeration over an interval with roughly *e* full key candidates.

```
1:  function QKS($e$, Test)
2:      $B_1 \leftarrow B_{min}$;
3:      $B_2 \leftarrow B_{min} + 1$;
4:      $s \leftarrow 0$;
5:      Find $B_e$ such that rank($B_1, B_e$) is roughly equals to $e$;
6:      while $B_1 \le B_e$ do
7:          B $\leftarrow$ create($B_1, B_2$);
8:          f($\cdot$) $\leftarrow$ Test(getKey(B, $B_1, B_2, \cdot$));
9:          Call Grover's algorithm with testing function f
10:         if a marked element $r$ is found then
11:             return getKey(B, $B_1, B_2, r$);
12:         end if
13:         $s \leftarrow s + 1$;
14:         $B_1 \leftarrow B_2$;
15:         Find $B_2$ such that rank($B_1, B_2$) is roughly equals to $a^s$;
16:     end while
17:     return $\perp$;
18: end function
```

6. Conclusions

This research paper addressed the question of the viability of cold boot attacks on SIKE. To this end, we reviewed SIKE's reference implementation as it was submitted to the NIST Post-Quantum Cryptography Standardization Process. Furthermore, we presented a dedicated key-recovery algorithm for SIKE in this setting and showed, through simulations, that our algorithm can reconstruct the secret key for SIKE, configured with any SIKE parameters, for varying values $\beta \in \{0.001, 0.01, \dots, 0.1\}$ and $\alpha = 0.001$, by only performing a 2^{30} enumeration. We stress these success rates from our algorithm can be improved as long as there are more available resources to run it. Additionally, we showed that our algorithm could be sped-up by integrating a quantum key search algorithm with it, which brings the computation power of quantum computing into the post-processing phase of a side channel attack. Moreover, as a future work, we may extend our work to include a resource estimation of quantum gates resulting from running our quantum key-recovery algorithm against SIKE or other cryptographic primitives in the post-processing phase.

Author Contributions: Conceptualization, R.V.-P.; methodology, R.V.-P.; software, R.V.-P. and E.A.-M.; validation, R.V.-P. and E.A.-M.; formal analysis, R.V.-P. and E.A.-M.; investigation, R.V.-P.; resources, R.V.-P.; data curation, R.V.-P. and E.A.-M.; writing—original draft preparation, R.V.-P. and E.A.-M.; writing—review and editing, R.V.-P. and E.A.-M.; visualization, R.V.-P. and E.A.-M.; supervision, R.V.-P.; project administration, R.V.-P.; funding acquisition, R.V.-P.; All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Universidad del Norte grant number 2019-029 and the APC was funded by Universidad del Norte.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Costello, C.; De Feo, L.; Jao, D.; Longa, P.; Naehrig, M.; Renes, J. Supersingular Isogeny Key Encapsulation. Post-Quantum Cryptography Standardization. 2020. Available online: https://sike.org/files/SIDH-spec.pdf (accessed on 2 December 2020).
2. Jao, D.; De Feo, L. *Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies*; Post-Quantum Cryptography; Yang, B.Y., Ed.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 19–34.
3. Alagic, G.; Alperin-Sheriff, J.; Aponn, D.; Cooper, D.; Dang, Q.; Kelsey, J.; Liu, Y.K.; Miller, C.; Moody, D.; Peralta, R.; et al. Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. NIST Post-Quantum Cryptography Standardization Process; 2020. Available online: https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf (accessed on 9 December 2020).
4. Faz-Hernández, A.; López, J.; Ochoa-Jiménez, E.; Rodríguez-Henríquez, F. A Faster Software Implementation of the Supersingular Isogeny Diffie–Hellman Key Exchange Protocol. *IEEE Trans. Comput.* **2017**, *67*, 1622–1636. [CrossRef]
5. Seo, H.; Jalali, A.; Azarderakhsh, R. *Optimized SIKE Round 2 on 64-bit ARM*; Information Security Applications–WISA 2019; You, I., Ed.; Springer: Cham, Switzerland, 2019.
6. Naehrig, M.; Renes, J. *Dual Isogenies and Their Application to Public-Key Compression for Isogeny-Based Cryptography*; Advances in Cryptology–ASIACRYPT 2019; Galbraith, S.D., Moriai, S., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 243–272.
7. Massolino, P.M.C.; Longa, P.; Renes, J.; Batina, L. A Compact and Scalable Hardware/Software Co-design of SIKE. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**, 245–271. [CrossRef]
8. Elkhatib, R.; Azarderakhsh, R.; Mozaffari-Kermani, M. Efficient and Fast Hardware Architectures for SIKE Round 2 on FPGA. Cryptology ePrint Archive, Report 2020/611. 2020. Available online: https://eprint.iacr.org/2020/611.pdf (accessed on 9 December 2020).
9. Costello, C.; De Feo, L.; Jao, D.; Longa, P.; Naehrig, M.; Renes, J. Supersingular Isogeny Key Encapsulation: Reference Implementation. Post-Quantum Cryptography Standardization. 2020. Available online: https://github.com/microsoft/PQCrypto-SIDH/releases/tag/v3.3 (accessed on 9 December 2020).
10. Halderman, J.A.; Schoen, S.D.; Heninger, N.; Clarkson, W.; Paul, W.; Calandrino, J.A.; Feldman, A.J.; Appelbaum, J.; Felten, E.W. Lest We Remember: Cold Boot Attacks on Encryption Keys. *Commun. ACM* **2009**, *52*, 91–98. [CrossRef]
11. Villanueva-Polanco, R. *Cold Boot Attacks on Bliss*; Springer: Cham, Switzerland, 2019; pp. 40–61. [CrossRef]
12. Veyrat-Charvillon, N.; Gérard, B.; Renauld, M.; Standaert, F.X. *An Optimal Key Enumeration Algorithm and Its Application to Side-Channel Attacks*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 390–406. [CrossRef]
13. Bogdanov, A.; Kizhvatov, I.; Manzoor, K.; Tischhauser, E.; Witteman, M. *Fast and Memory-Efficient Key Recovery in Side-Channel Attacks*; Springer: Cham, Switzerland, 2016; pp. 310–327. [CrossRef]
14. David, L.; Wool, A. *A Bounded-Space Near-Optimal Key Enumeration Algorithm for Multi-subkey Side-Channel Attacks*; Springer: Cham, Switzerland, 2017; pp. 311–327. [CrossRef]
15. Longo, J.; Martin, D.P.; Mather, L.; Oswald, E.; Sach, B.; Stam, M. How Low Can You Go? Using Side-Channel Data to Enhance Brute-Force Key Recovery. Cryptology ePrint Archive, Report 2016/609. 2016. Available online: http://eprint.iacr.org/2016/609 (accessed on 20 November 2020).
16. Martin, D.P.; Mather, L.; Oswald, E.; Stam, M. *Characterisation and Estimation of the Key Rank Distribution in the Context of Side Channel Evaluations*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 548–572. [CrossRef]
17. Martin, D.P.; O'Connell, J.F.; Oswald, E.; Stam, M. *Counting Keys in Parallel After a Side Channel Attack*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 313–337. [CrossRef]
18. Poussier, R.; Standaert, F.X.; Grosso, V. *Simple Key Enumeration (and Rank Estimation) Using Histograms: An Integrated Approach*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 61–81. [CrossRef]
19. Veyrat-Charvillon, N.; Gérard, B.; Standaert, F.X. *Security Evaluations beyond Computing Power*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 126–141. [CrossRef]
20. Bernstein, D.J.; Lange, T.; van Vredendaal, C. Tighter, Faster, Simpler Side-Channel Security Evaluations Beyond Computing Power. Cryptology ePrint Archive, Report 2015/221. 2015. Available online: http://eprint.iacr.org/2015/221 (accessed on 9 November 2020).

21. Ye, X.; Eisenbarth, T.; Martin, W. Bounded, yet Sufficient? How to Determine Whether Limited Side Channel Information Enables Key Recovery. In *Smart Card Research and Advanced Applications*; Joye, M., Moradi, A., Eds.; Springer International Publishing: Cham, Switzerland, 2015; pp. 215–232.
22. Choudary, M.O.; Popescu, P.G. *Back to Massey: Impressively Fast, Scalable and Tight Security Evaluation Tools*; Springer: Cham, Switzerland, 2017; pp. 367–386. [CrossRef]
23. Choudary, M.O.; Poussier, R.; Standaert, F.X. *Core-Based vs. Probability-Based Enumeration- A Cautionary Note*; Springer: Cham, Switzerland, 2016; pp. 137–152. [CrossRef]
24. Glowacz, C.; Grosso, V.; Poussier, R.; Schüth, J.; Standaert, F.X. *Simpler and More Efficient Rank Estimation for Side-Channel Security Assessment*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 117–129. [CrossRef]
25. Poussier, R.; Grosso, V.; Standaert, F.X. Comparing Approaches to Rank Estimation for Side-Channel Security Evaluations. In *Smart Card Research and Advanced Applications*; Homma, N., Medwed, M., Eds.; Springer International Publishing: Cham, Switzerland, 2016; pp. 125–142.
26. Grosso, V. Scalable Key Rank Estimation (and Key Enumeration) Algorithm for Large Keys. In *Smart Card Research and Advanced Applications*; Bilgin, B., Fischer, J.B., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 80–94.
27. Villanueva-Polanco, R. A Comprehensive Study of the Key Enumeration Problem. *Entropy* **2019**, *21*, 972. [CrossRef]
28. Heninger, N.; Shacham, H. *Reconstructing RSA Private Keys from Random Key Bits*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 1–17. [CrossRef]
29. Henecka, W.; May, A.; Meurer, A. *Correcting Errors in RSA Private Keys*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 351–369. [CrossRef]
30. Paterson, K.G.; Polychroniadou, A.; Sibborn, D.L. *A Coding-Theoretic Approach to Recovering Noisy RSA Keys*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 386–403. [CrossRef]
31. Lee, H.T.; Kim, H.; Baek, Y.J.; Cheon, J.H. *Correcting Errors in Private Keys Obtained from Cold Boot Attacks*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 74–87.
32. Poettering, B.; Sibborn, D.L. *Cold Boot Attacks in the Discrete Logarithm Setting*; Springer: Cham, Switzerland, 2015; pp. 449–465. [CrossRef]
33. Albrecht, M.; Cid, C. *Cold Boot Key Recovery by Solving Polynomial Systems with Noise*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 57–72. [CrossRef]
34. Kamal, A.A.; Youssef, A.M. Applications of SAT Solvers to AES Key Recovery from Decayed Key Schedule Images. In Proceedings of the 2010 Fourth International Conference on Emerging Security Information, Systems and Technologies, SECURWARE '10, Venice, Italy, 18–25 July 2010; IEEE Computer Society: Washington, DC, USA, 2010; pp. 216–220. [CrossRef]
35. Huang, Z.; Lin, D. A New Method for Solving Polynomial Systems with Noise over F_2 and Its Applications in Cold Boot Key Recovery. In *Selected Areas in Cryptography*; Knudsen, L.R., Wu, H., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 16–33.
36. Paterson, K.G.; Villanueva-Polanco, R. Cold Boot Attacks on NTRU. In *Progress in Cryptology—INDOCRYPT 2017*; Patra, A., Smart, N.P., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 107–125.
37. Villanueva-Polanco, R. Cold Boot Attacks on LUOV. *Appl. Sci.* **2020**, *10*, 4106. [CrossRef]
38. Albrecht, M.R.; Deo, A.; Paterson, K.G. Cold Boot Attacks on Ring and Module LWE Keys Under the NTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2018**, *2018*, 173–213. [CrossRef]
39. Vélu, J. Isogénies entre courbes elliptiques. *Comptes Rendus De L'Académie Des Sci. De Paris* **1971**, *273*, 238–241.
40. Dworkin, M.J. Federal Inf. Process. Stds. (NIST FIPS). 2015. Available online: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf (accessed on 9 November 2020).
41. Seshadri, N.; Sundberg, C.W. List Viterbi decoding algorithms with applications. *IEEE Trans. Commun.* **1994**, *42*, 313–323. [CrossRef]
42. Martin, D.P.; Montanaro, A.; Oswald, E.; Shepherd, D.J. *Quantum Key Search with Side Channel Advice*; Springer: Cham, Switzerland, 2017; pp. 407–422. [CrossRef]
43. Grover, L.K. A Fast Quantum Mechanical Algorithm for Database Search. In *STOC '96: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*; Association for Computing Machinery: New York, NY, USA, July 1996; pp. 212–219. [CrossRef]
44. Beullen, W.; Preneel, B.; Szepieniec, A.; Tjhai, C.; Vercauteren, F. LUOV: Signature Scheme proposal for NIST PQC Project (Round 2 version). Submission to NIST's Post-Quantum Cryptography Standardization Project. 2018. Available online: https://www.esat.kuleuven.be/cosic/pqcrypto/luov/ (accessed on 29 November 2020).