Article

Cold Boot Attacks on LUOV

Ricardo Villanueva-Polanco

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

Received: 7 April 2020; Accepted: 25 May 2020; Published: 15 June 2020

Abstract: This research article assesses the feasibility of cold boot attacks on the lifted unbalanced oil and Vinegar (LUOV) scheme, a variant of the UOV signature scheme. This scheme is a member of the family of asymmetric cryptographic primitives based on multivariable polynomials over a finite field $\mathbb{K}$ and has been submitted as candidate to the ongoing National Institute of Standards and Technology (NIST) standardisation process of post-quantum signature schemes. To the best of our knowledge, this is the first time that this scheme is evaluated in this setting. To perform our assessment of the scheme in this setting, we review two implementations of this scheme, the reference implementation and the libpqcrypto implementation, to learn the most common in-memory private key formats and next develop a key recovery algorithm exploiting the structure of this scheme. Since the LUOV's key generation algorithm generates its private components and public components from a 256-bit seed, the key recovery algorithm works for all the parameter sets recommended for this scheme. Additionally, we tested the effectiveness and performance of the key recovery algorithm through simulations and found the key recovery algorithm may retrieve the private seed when $\alpha = 0.001$ (probability that a 0 bit of the original secret key will flip to a 1 bit) and β (probability that a 1 bit of the original private key will flip to a 0 bit) in the range $\{0.001, 0.01, 0.02, \dots, 0.15\}$ by enumerating approximately 2^{40} candidates.

Keywords: cold boot attacks; key enumeration; key recovery; post-quantum cryptography

1. Introduction

This research article evaluates the feasibility of cold boot attacks on the lifted unbalanced oil and vinegar (LUOV) scheme [1,2], which is a variant of the UOV signature scheme [3]. The LUOV scheme is a member of the family of asymmetric cryptographic primitives based on multivariate polynomials over a finite field $\mathbb{K}$ and has been submitted as candidate to the ongoing NIST standardisation process of post-quantum signature schemes [1]. We believe this to be the first time that this scheme is evaluated in this setting; however, there are recent studies evaluating other type of side channel attacks on other related signature schemes [4].

In the cold boot attack setting, an attacker may fetch any content from a machine's main memory after performing a cold-rebooting of the machine. However, this content will likely be perturbed due to physical effects on the main memory. That is, if the attacker obtains a content from memory, this content will likely be noisy, meaning that some 1 bits may have changed to 0 bits and vice versa. Thus, evaluating a public key cryptographic scheme in this setting means analysing the feasibility of an attacker, assumed to both have a noisy version of private key and know the probabilities associated with a 1 bit to changing to 0 and vice versa, to recover the original private key from the bit-flipped version of it.

To this end, the attacker is needed to have knowledge about the in-memory private key representations of the scheme. This can be achieved by either proposing usual in-memory private key representations for the scheme or by reviewing specific implementations of the scheme to learn

more about the actual formats used to store the scheme's private key. We adopt the second approach, and study the reference implementation and libpqcrypto implementation of this scheme, which both are written in C language. In addition, the reference implementation has been included in the package submitted to the NIST process.

Our key recovery algorithm exploits the fact this signature scheme's private key is generated from a 256-bit seed. This allows us to reconstruct it from its noisy version by treating it as a combination of blocks of chunks, each of which has an assigned log-likelihood score obtained from combining candidate values for each of the chunks from their noisy versions. By making use of a general key recovery strategy developed in [5] as a core algorithm, we generate lists of high scoring candidates for each block and then generate complete candidates for the the 256-bit seed, which test to pinpoint the correct one.

Our contribution is twofold: on the one hand, we present a key recovery algorithm for LUOV in the cold boot attack setting. This is in line with the current trend of devising, developing and implementing key recovery algorithms for different schemes, as is evinced by the literature discussed at length in Section 2.3. On the other hand, we further the research on the assessment of the most-important post-quantum candidates against this type of attack. In particular, this work is a small but significant part of the comprehensive assessment of schemes in the NIST selection process for post-quantum algorithms.

This article is organised as follows. Section 2 discusses a background on cold boot attacks, including details on how this attack may be performed, the cold boot attack model we use in this article, the different approaches previously introduced to tackle the key recovery problem for quite a few cryptographic schemes in the cold boot setting and our approach to key recovery, which consists in splitting a byte representation of a private key component key into chunks, and then computing log-likelihood scores for each candidate value for each of the chunks. Section 3 describes LUOV, the reference implementation and the libpqcrypto implementation, mainly focusing on the key generation algorithm. Section 4 describes our key recovery algorithm, while Section 5 describes an experimental evaluation, focusing on an analysis regarding the success rate of our algorithm. In Section 6, we finally draw some important conclusions of our work.

2. Background

In this section, we present a description on cold boot attacks, the cold boot attack model we use throughout this article, a summary of the previous works done to tackle key recovery problem for multiple cryptographic schemes and an account of the general approach to key recovery we use in this article.

2.1. Cold Boot Attacks

In a cold boot attack, an attacker exploits the data remanence property of a Dynamic Random Access Memory (DRAM) to obtain data from a computer's memory that remain readable for a period of time after the device's power has been turned off. Hence, this attacker could read sensitive data from memory using this technique after apparently having been erased [6].

This attack was first described by Halderman et al. [6] about ten years ago and it has been studied significantly since then. In this setting, an attacker with physical access to a computer may retrieve content from a running operating system by cold-rebooting the computer. That is, such attacker turns off the operating system incorrectly, which makes it skip file system synchronisation and other tasks that would happen when an operating system is shutdown correctly, and then leverages a removable drive to start a lightweight operating system, put it into a state of readiness for operation and then use it to dump the content of pre-boot physical memory to a file. If the previous procedure does not work, such attacker could detach the memory modules from the original computer and quickly place them in a compatible computer controlled by the attacker, which is then booted to access the memory content.

Further analysis is performed on the data that were copied off from main memory to search for sensitive information, such as cryptographic keys, by employing key finding algorithms [6]. Unluckily for such an adversary, the bits contained in main memory will undergo a process of deterioration immediately after the computer's power is stopped. Therefore, if such an adversary is able to obtain any data from the computer's main memory while the power is off, the extracted data bits will be dissimilar from the original data bits by having some 0 bits flipped to 1 bits and vice versa.

Even though the content (with errors) of main memory may practically be retained for a interval of time by using cooling techniques [6], an attacker has yet to extract such content from memory before even considering reconstructing any relevant information from it. To perform such task, the attacker first has to handle several possible issues that may happen. For instance, on rebooting the target machine, the Basic Input/Output System (BIOS) may write portions of memory with its own code and data, although the affected portions normally are small. Additionally, the BIOS may perform a destructive memory check during its Power-On Self Test (POST) (however this test might be disabled or bypassed in some machines). To handle these issues, tiny special-purpose programs (memory-imaging tools) may be used, as reported in [6], which are expected to produce accurate dumps of memory contents to some external device. These special-purpose programs normally use small amounts of RAM, and their memory offsets are normally adjusted to some extent to ensure that data structures of interest are unaffected. Additionally, the authors of [6] pointed out that, when an attacker is not able to make a target system to boot memory-imaging tools, such attacker could physically remove the memory modules, place them in a computer controlled by the attacker and then dump their contents. After extracting the memory content, the attacker has to profile this content to gain knowledge about the regions of memory and the probabilities of both a 1 flipping to 0 and a 0 flipping to 1. According to the results of the experiments reported in [6], almost all memory bits tend to decay to predictable "ground" states, with only a tiny fraction flipping in the opposite direction. In addition, the authors pointed out that the probability of decaying to the "ground state" increases as time goes on, while the probability of flipping in the opposite direction remains relatively constant and tiny (e.g., 0.001). These results suggest that the attacker may model the decay in a region as a binary asymmetric channel, i.e., assuming the probability of a 1 flipping to 0 is some fixed number, while the probability of a 0 flipping to a 1 is some other fixed number. Note that an attacker can establish the "ground state" of a particular region of memory very easily in an attack by scanning all the bits and counting the number of 0 bits and the number of 1 bits. Furthermore, the attacker may estimate the probabilities by comparing any original content in such region with its respective noisy version.

Another major challenge presented to the attacker after obtaining memory images is locating encryption keys in them. This problem has already been addressed in previous research to locate AES keys and RSA keys in a memory image [6], and, although those key-finding algorithms are scheme-specific, their rationale may be easily adopted to design key-finding algorithms for other schemes, since they rely on finding identifying features of the formats for storing a scheme-specific key and using such identifying features as markers to identify sequences of bytes, while searching over decayed memory images. More specifically, the algorithm searches for sequences of bytes with low Hamming distance to these markers and checks that the remaining bytes in a candidate sequence satisfy some conditions.

After tackling the previous challenges, the attacker can only aspire to have access to a noisy version of the original key from main memory; therefore, their principal task becomes the mathematical problem of recuperating the original key from a noisy version of the original key, taking into consideration such attacker may have access to reference cryptographic data generated using the secret key, such as cipher-texts for a secret-key encryption scheme, or have access to a public key in the public-key setting. Thus, the main objective in the cold boot attacks setting is design, implement, and evaluate efficient algorithms to obtain the original private key back from its corresponding noisy version for a range of different cryptographic schemes, as well as explore the behaviour of these

key-recovery algorithms for various parameters, in particular establish upper bounds of how much noise such algorithms can tolerate.

2.2. Cold Boot Attack Model

In this paper, we assume that an adversary can obtain a noisy version of a secret key regardless of what format was made use of to store it in memory. Additionally, such adversary is assumed to have access to the corresponding public parameters without noise. Here, we do not make determined efforts to deal with the problem of finding the memory region in which the bits of the secret keys are stored, although this would be a relevant consideration when carrying out this attack in practice (this can be achieved by using the ideas developed in [6]). Thus, the main goal of the adversary is to recover the secret key.

We let $\alpha = P(0 \rightarrow 1)$ be the probability of a 0 bit of the original secret key to change to a 1 bit and $\beta = P(1 \rightarrow 0)$ be the probability of a 1 bit of the original private key to flip to a 0 bit. Note that, according to the results of the experiments reported in [6], one of these values may be assumed to be small (e.g., 0.001) and relatively steady as time goes by, while the other value grows as time elapses. Furthermore, we assume that the attacker knows these probabilities, i.e., α and β , and that these are fixed values across the region of memory in which the private key is located [6]. Note that these assumptions are acceptable in practice, since the memory regions are normally large and the probabilities, α and β , can be easily determined by scanning a memory region where a known content is stored, and then comparing the original known content with its noisy version [6].

2.3. Previous Work

This section explores the previous works to tackle the recovery problem for multiple cryptographic schemes in the cold boot attack setting.

2.3.1. RSA Setting

Heninger and Shacham [7] were the first to explore cold boot attacks on RSA keys. Their introduced algorithm is based on Hensel lifting and exploits redundancy found in the standard in-memory RSA private key representation. This work was improved upon by the research paper by Henecka, May, and Meurer [8] and then by the research paper by Paterson, Polychroniadou, and Sibborn [9], both of which further exploited the rich structure of the RSA setting. The latter in particular identified the inherent features of the error channel, especially its asymmetric essence inherent to the cold boot setting, and redefined the key recovery problem in an information theoretic manner.

2.3.2. Discrete Logarithm Setting

This attack in the discrete logarithm setting was first explored by the work by Lee et al. [10]. In particular, the model assumed in the paper presupposes that the adversary knows the public key, the private-key noisy version and an upper bound on the flips of the bits of the private key. Because the last assumption may not be reasonable and their introduced key recovery algorithm does not use further redundancy, the algorithm might not be able to recover keys affected by relatively high values of noise in a bit-flipping model. A later work by Poettering and Sibborn [11] reviewed two practical elliptic curve cryptography implementations to find exploitable in-memory representations. In particular, they analysed two scenarios obtained from two elliptic curve implementations from TLS libraries: the windowed non-adjacent form (wNAF) representation used in OpenSSL and the comb-based approach used in PolarSSL. The authors exploited the extra redundancy found in the corresponding in-memory private key formats, developed key recovery algorithms and tested them in the real cold boot attack setting.

2.3.3. Symmetric Key Setting

There are several papers considering cold boot attacks on secret-key encryption schemes. Albrecht and Cid [12] analysed the key recovery problem for several secret-key encryption schemes and presented key recovery algorithms based on polynomial system solvers. Kamal and Youssef [13] focused on the same problem and presented algorithmic techniques based on SAT solvers. Additionally, Huang and Lin [14] expanded on the research of key recovery techniques for secret-key schemes. Theoretically-oriented literature on leakage-resilient cryptography also cites cold boot attacks, but the relevance there is minimal because directly accessing a noisy version of the entire key (as in cold boot attacks) does not really apply in the leakage-resilient setting.

2.3.4. Post-Quantum Setting

There are several papers considering cold boot attacks on post-quantum cryptographic schemes. In particular, Paterson et al. [15] studied NTRU in the cold boot attack setting by reviewing two existing NTRU implementations, the ntru-crypto implementation and the tbuktu/Bouncy Castle JAVA implementation, and developed key recovery algorithms for various private key formats found in these implementations. One of their key recovery algorithms was able to tolerate a noise level with $\alpha = 0.001$ and $\beta = 0.09$ for one of the formats when performing a 2^{40} enumeration. This work is followed upon by Villanueva-Polanco [5] who evaluated BLISS in the cold boot attack setting. This paper particularly introduces a general key strategy via key enumeration whose results for various parameters found in BLISS are comparable to the results found in the NTRU case.

Furthermore, Albrecht et al. [16] explored cold boot attacks on cryptographic schemes based on the ring and module variants of the learning with errors (LWE) problem. They particularly focused on two encodings to store LWE keys for two cryptographic schemes: the Kyber key encapsulation mechanism (KEM) and new hope KEM. When the first encoding is used, the polynomial coefficients are stored directly in memory. When the second encoding is used, a number theoretic transform (NTT) on the key is applied just before storing it. The authors showed that, at a 1% bit-flip rate, their algorithmic techniques are able to recover the keys of Kyber KEM with a cost of 2^{43} operations when the second encoding is used for key storage, compared to 2^{70} operations when the first encoding is employed.

2.3.5. General Approach to Key Recovery

The author of [5,17] noted that there is a general strategy to key recovery in the cold boot attack setting. We here follow the description in [17] to give an account of this general strategy.

Let $ns = b_0b_1b_2 \dots b_W$ be the noisy version of the encoding of the secret key and let us assume that ns can be represented as a concatenation of $\mathcal{N} = W/w$ chunks, where each chunk is a sequence of w bits. That is, $ns = ns^0 || ns^1 || \dots || ns^{\mathcal{N}-1}$ with $ns^i = b_{i \cdot w} b_{i \cdot w + 1} \dots b_{i \cdot w + (w-1)}$. Additionally, let us assume there exists a key recovery algorithm that generates complete candidates c for the encoding of the secret key and that these complete candidates c can also be represented by concatenations of chunks in the same way, i.e., $c = c^0 || c^1 || \dots || c^{\mathcal{N}-1}$ with c^i being a sequence of w bits.

Given the bit-flip probabilities α, β , let us define $P(c|ns)$ to be the probability of c being the correct encoding of the secret key given the noisy version of the secret key. Hence, $P(c|ns) = \frac{P(ns|c)P(c)}{P(ns)}$. According to the maximum likelihood estimation method, c should be chosen as the value that maximises $P(c|ns)$. Since $P(ns)$ and $P(c)$ can be seen as constants, then c should be chosen as the value that maximises $P(ns|c) = (1 - \alpha)^{n_{00}} \alpha^{n_{01}} \beta^{n_{10}} (1 - \beta)^{n_{11}}$, where n_{00} denotes the number of positions where both c and ns contain a 0 bit, n_{01} denotes the number of positions where c contains a 0 bit, ns contains a 1 bit, etc. Equivalently, c may be chosen as the value that maximises the log of these probabilities, i.e., $\log(P(ns|c)) = n_{00} \log(1 - \alpha) + n_{01} \log \alpha + n_{10} \log \beta + n_{11} \log(1 - \beta)$. Consequently, we can give a score to a given candidate c , i.e., $S(c, ns) := \log(P(ns|c))$.

Let us further assume that each of the at most 2^w candidate values for chunk c^i ($0 \leq i < W/w$) can be enumerated; then, its own score also can be calculated as

$$S(c^i, ns^i) = n_{00}^i \log(1 - \alpha) + n_{01}^i \log \alpha + n_{10}^i \log \beta + n_{11}^i \log(1 - \beta) \quad (1)$$

where the n_{ab}^i values count occurrences of bits across the i th chunks, c^i, ns^i , resulting in $S(c, ns) = \sum_{i=0}^{N-1} S(c^i, ns^i)$. As a result, we may obtain $\mathcal{N}$ lists of chunk candidates, where each list contains up to 2^w entries and a chunk candidate is defined as a 2-tuple of the form $(score, value)$ such that the first component *score* is a real number (candidate score) and the second component *value* is an array of w -bit strings (candidate value).

From the previous observation, it follows that the problem of key recovery becomes that of developing efficient algorithmic techniques to construct full key candidates c with high accumulated scores calculated by summation, by travelling across the lists of chunk candidates while merging chunk candidates c^i [17]. This problem has already been extensively studied in the side-channel analysis literature [17–32], and there is a great variety of different algorithms able to solve the problem.

More generally, let $L^i = [c_{j_0}^i, c_{j_1}^i, \dots, c_{j_{m_i-1}}^i]$ be the list of chunk candidates for chunk i , with $0 < m_i \leq 2^w$, and let $c_{j_0}^{i_0}, \dots, c_{j_n}^{i_n}$ be chunk candidates, with $0 \leq i_0 < \dots < i_n < \mathcal{N}, 0 \leq j_i < m_i$. Let us define the function $\text{combine}(c_{j_0}^{i_0}, \dots, c_{j_n}^{i_n})$ to return a new chunk candidate c constructed by calculating $c = (c_{j_0}^{i_0}.score + \dots + c_{j_n}^{i_n}.score, c_{j_0}^{i_0}.value \parallel \dots \parallel c_{j_n}^{i_n}.value)$. Note that, when $i_0 = 0, i_1 = 1, \dots, i_{\mathcal{N}-1} = \mathcal{N} - 1$, c will be a full key candidate. Therefore, the key enumeration problem asks for selecting a chunk candidate $c_{j_i}^i$ from each list $L^i, 0 \leq i < \mathcal{N}$, in order to construct full key candidates $c = \text{combine}(c_{j_0}^{i_0}, \dots, c_{j_n}^{i_n})$ subjected to a condition on their scores [17]. Typically, an algorithm that generates full key candidates c is called key enumeration algorithm (KEA).

For our key recovery problem, we adapt a technique combining several key enumeration algorithms that has been recently introduced in [5] and exploit the in-memory representation of the secret key of the LUOV scheme. This adaptation is described in Section 4.

3. Multivariable Polynomial Signature Schemes

In this section, we describe the LUOV scheme, a variant of the UOV scheme.

3.1. UOV Signature Schemes

We first describe the UOV scheme [3]. Let $\mathbb{K}$ be a finite field and $o, v \in \mathbb{N}$. Let us set $n = v + o$. An oil–vinegar polynomial is defined as

$$f(x_1, \dots, x_n) = \sum_{i=1}^v \sum_{j=i}^n \alpha_{i,j} x_i x_j + \sum_{i=1}^n b_i x_i + \gamma,$$

where $\alpha_{i,j}, \beta_i, \gamma \in \mathbb{K}$. Additionally, $x_1, \dots, x_v$ are called vinegar variables, while $x_{v+1}, \dots, x_n$ are called oil variables.

The UOV signature scheme uses a map $\mathcal{F} : \mathbb{K}^n \rightarrow \mathbb{K}^o$ known as the UOV map or the central map. This map consist of oil–vinegar polynomials $f^{(1)}, \dots, f^{(o)}$, i.e.,

$$\mathcal{F}(x_1, x_2, \dots, x_n) = (f^{(1)}(x_1, x_2, \dots, x_n), \dots, f^{(o)}(x_1, x_2, \dots, x_n))$$

where $f^{(k)}(x_1, x_2, \dots, x_n) = \sum_{i=1}^v \sum_{j=i}^n \alpha_{i,j}^{(k)} x_i x_j + \sum_{i=1}^n b_i^{(k)} x_i + \gamma^{(k)}$ for $1 \leq k \leq o$

Note that given $y \in \mathbb{K}^o$ we can efficiently find an $x \in \mathbb{K}^n$ such that $\mathcal{F}(x) = y$ by first randomly choosing values $v_1, v_2, \dots, v_v \in \mathbb{K}$ for the vinegar variables and then solving the resulting linear system of o equations and o variables to find the values of the the oil variables. Additionally, the structure of $\mathcal{F}$ is hidden by composing it with a random invertible linear map $\mathcal{T} : \mathbb{K}^n \rightarrow \mathbb{K}^n$ to get a public map

$\mathcal{P} = \mathcal{F} \circ \mathcal{T}$. To describe the UOV signature scheme, we assume to have access to a cryptographic hash function $\mathcal{H} : \{0, 1\}^* \rightarrow \mathbb{K}^o$.

Key Generation Algorithm.

1. Choose all the coefficients of each oil–vinegar polynomial $f^{(k)}$, for $1 \leq k \leq o$, at random from $\mathbb{K}$ to construct the central map $\mathcal{F}$.
2. Choose an invertible matrix $\mathbb{T} \in \mathbb{K}^{n \times n}$ at random to represent the linear map $\mathcal{T}$.
3. Compute $\mathcal{P} = \mathcal{F} \circ \mathcal{T}$.
4. Output $\mathcal{T}, \mathcal{F}, \mathcal{P}$, where $\mathcal{P}$ is the public key, while $(\mathcal{T}, \mathcal{F})$ is the private key.

Signature Generation Algorithm. Given a message m and the private key $(\mathcal{T}, \mathcal{F})$, then

1. Compute $y = \mathcal{H}(m)$.
2. Compute a preimage $x \in \mathbb{K}^n$ of y under the central map $\mathcal{F}$.
3. Compute the signature $s \in \mathbb{K}^n$ by $s = \mathcal{T}^{-1}(x)$ and return it.

Signature Verification Algorithm. Given a message m , a signature $s \in \mathbb{K}^n$ and the public key $\mathcal{P}$, then

1. If $\mathcal{H}(m) = \mathcal{P}(s)$, return accept. Otherwise, return reject.

In a practical setting, $\mathbb{K}$ is normally set to $\mathbb{F}_2$ for the UOV scheme. The LUOV scheme [1] is basically a variant of the UOV signature scheme, and is described next.

3.2. LUOV Scheme

As mentioned above, LUOV is based on UOV and basically differs from it in the key generation algorithm and in the way a preimage of y is computed under the central map $\mathcal{F}$. More specifically, its key generation algorithm generates the linear map $\mathcal{T}$ and a large part of the public key from the output of a pseudo-random generator on input a 256-bit seed. Additionally, the public key $\mathcal{P}$ is ‘lifted’ to an extension field $\mathbb{F}_2 \subset \mathbb{F}_{2^r}$, $r \in \mathbb{N}$ by considering the map $\mathcal{P} : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^o$ as a map $\mathcal{P} : \mathbb{F}_{2^r}^n \rightarrow \mathbb{F}_{2^r}^o$ (hence its name). Note that this field lifting makes the public key remain small (since the coefficients of the public key are 0 or 1), as well as makes solving the system $\mathcal{P}(x) = y$ for some y in $\mathbb{F}_{2^r}$ more difficult compared to the case when y is in $\mathbb{F}_2$ [2]. Finally, the matrix $\mathbb{T}$ that represents the linear map $\mathcal{T}$ is chosen of the form

$$\begin{pmatrix} I_v & -T \\ O & I_o \end{pmatrix} \quad (2)$$

where T is a v -by- o matrix generated from a private seed. Note that $\mathbb{T}$ is determined by T and its inverse matrix is itself. This choice is made for optimisation, i.e., to make both the key generation algorithm and the signing algorithm run much more quickly, although it does not affect the security of the scheme [1]. We next expand more on the LUOV scheme, the reference implementation and libpqcrypto implementation.

3.2.1. The Reference Implementation

Algorithm 1 shows a pseudo-code representation of the C code of the key generation algorithm from the reference implementation for LUOV [1]. First, it randomly generates `private_seed` that is stored in a byte array of length 32. The subsequent calls uses `private_seed` as input to an extendable function $\mathcal{H}$, which is either SHAKE128 or SHAKE256, to generate `public_seed`, which is stored as a byte array of length 32, as well as the matrix $T \in \mathbb{F}_2^{v \times o}$ that determines the linear map $\mathcal{T}$, which is stored as a byte array of length $\lceil \frac{o}{8} \rceil v$. The entries of such array whose indices are in the range $\{(i-1)\lceil \frac{o}{8} \rceil + 1, \dots, i\lceil \frac{o}{8} \rceil\}$ represent the i th row of T for $1 \leq i \leq v$. If o is not divisible by 8, the most significant bits of the last byte (i.e., $i\lceil \frac{o}{8} \rceil$ -th byte in the sequence) are ignored.

Algorithm 1 Key Generation Algorithm for LUOV.

```

function KEYGEN()
    // generate a random secret key (32 byte array).
    private_seed ← randombytes(32);
    // generate a 32 byte array (the private sponge) from private_seed.
    private_sponge ← InitializeAndAbsorb(private_seed, 32);
    // generate a 32 byte array (the public seed) from private_sponge.
    public_seed ← SqueezePublicSeed(private_sponge, 32);
    // generate the matrix  $T$  from the output of a pseudo-random generator on input private_sponge.
    T ← SqueezeT(private_sponge);
    // generate a 32 byte array (the public sponge) from public_seed.
    public_sponge ← InitializeAndAbsorb(public_seed, 32);
    // generate the matrices  $C, L, Q_1$  from public_sponge.
    C, L,  $Q_1$  ← SqueezePublicMap(public_sponge);
    // calculate  $Q_2$ , the part of the public map that cannot be generated from the public seed.
     $Q_2$  ← CalculateQ2( $Q_1, T$ );
    // return the public key (public_seed,  $Q_2$ ) and the secret key private_seed.
    return (public_seed,  $Q_2$ ) and private_seed;
end function

```

By making $\lceil \frac{o}{16} \rceil$ calls to an extendable function $\mathcal{G}$, which is either SHAKE128 or ChaCha8, with public_seed and a salt $0x0i$ with $i \in \{0, \dots, \lceil \frac{o}{16} \rceil - 1\}$ as inputs, the function SqueezePublicMap generates C, L and Q_1 , where

- $C \in \mathbb{F}_2^o$ is the constant part of the public map $\mathcal{P}$ and stored in a byte array of length $\lceil \frac{o}{8} \rceil$.
- $L \in \mathbb{F}_2^{o \times n}$ is the linear part of $\mathcal{P}$ and stored in a byte array of length $\lceil \frac{o}{8} \rceil n$
- $Q_1 \in \mathbb{F}_2^{o \times (\frac{v \cdot (v+1)}{2} + v \cdot o)}$ is the first $\frac{v \cdot (v+1)}{2} + v \cdot o$ columns of the Macaulay matrix of the quadratic part of $\mathcal{P}$ in the lexicographic ordering and stored in a byte array of length $\lceil \frac{o}{8} \rceil (\frac{v \cdot (v+1)}{2} + v \cdot o)$.

It then calculates $Q_2 \in \mathbb{F}_2^{o \times (\frac{o \cdot (o+1)}{2})}$, the remaining part of the Macaulay matrix of the quadratic part of $\mathcal{P}$, and finally returns the public key (public_seed, Q_2) and the private key private_seed. On the one hand, the public key gets to be encoded as a byte array of length $32 + \lceil \frac{o^2(o+1)}{16} \rceil$, where the first 32 bytes represent public_seed and the remaining bytes, seen as a long sequence of bits, represent Q_2 as follows: the first o bits represent the first column, the next o bits represent the second column, and so on up to complete the $\frac{o(o+1)}{2}$ th column. On the other hand, the private key is the 32 byte array private_seed. For all parameter choices, as shown in Table 1, the secret key consists of a 32-byte seed.

Table 1. Recommended parameters for LUOV and lengths in bytes of the arrays generated during the key generation algorithm.

Parameter Name	r	o	v	Length of private_seed	Lengths of T, C, L, Q_1	Lengths of public_seed, Q_2	$\mathcal{H}$	$\mathcal{G}$
LUOV-7-57-197	7	57	197	32B	(1576B, 2032B, 8B, 245856B)	(32B, 11778B)	SHAKE128	SHAKE128 ChaCha8
LUOV-7-83-283	7	83	283	32B	(3113B, 4026B, 11B, 700425B)	(32B, 36168B)	SHAKE256	SHAKE128 ChaCha8
LUOV-7-110-374	7	110	374	32B	(5236B, 6776B, 14B, 1557710B)	(32B, 83944B)	SHAKE256	SHAKE128 ChaCha8
LUOV-47-42-182	47	42	182	32B	(1092B, 1344B, 6B, 145782B)	(32B, 4741B)	SHAKE128	SHAKE128 ChaCha8
LUOV-61-60-261	61	60	261	32B	(2088B, 2568B, 8B, 398808B)	(32B, 13725B)	SHAKE256	SHAKE128 ChaCha8
LUOV-79-76-341	79	76	341	32B	(3410B, 4170B, 10B, 842270B)	(32B, 27797B)	SHAKE256	SHAKE128 ChaCha8

The signature generation algorithm, on input m and private_seed , uses private_seed to recalculate public_seed and T, C, L and Q_1 , as done in the key generation algorithm. It then uses them to compute the augmented matrix $A \in \mathbb{F}_2^{o \times (o+1)}$ that represents the linear system with o equations and o variables obtained from $\mathcal{F}((v_1, v_2, \dots, v_v, x_{v+1}, \dots, x_n)) = \mathcal{H}(m || 0x00 || \text{salt})$, where $v_1, v_2, \dots, v_v$ are randomly chosen values from $\mathbb{F}_2^r$, represented as a byte array of length $\lceil \frac{vr}{8} \rceil$; m is a byte array representing the message; salt is a random 16-byte array; $0x00$ is a byte (0); and $\mathcal{H}(m || 0x00 || \text{salt}) \in \mathbb{F}_2^o$ is represented as a byte array of length $\lceil \frac{or}{8} \rceil$. Once $x = (v_1, v_2, \dots, v_v, x_{v+1}, \dots, x_n)$ is found, the signature s is obtained by calculating $\begin{pmatrix} I_v & -T \\ O & I_o \end{pmatrix} x$ and finally (s, salt) is returned as a byte array of length of length $16 + \lceil \frac{nr}{8} \rceil$.

Similarly, the signature verification algorithm, on input $(\text{public_seed}, Q_2)$, (s, salt) and m , uses public_seed to recalculate C, L and Q_1 . It then computes the matrix $Q \in \mathbb{F}_2^{o \times (\frac{n \cdot (n+1)}{2})}$, the Macaulay matrix of the quadratic part of $\mathcal{P}$, by concatenating Q_1 and Q_2 , and finally it makes use of Q, C, L to calculate $\mathcal{P}(s)$ and compare it with $\mathcal{H}(m || 0x00 || \text{salt})$.

3.2.2. The Libpqcrypto Implementation

libpqcrypto is a new cryptographic software library produced by the PQCRYPTO project [33], from which multiple proposals have been submitted to NIST's ongoing post-quantum standardisation project.

Regarding the LUOV's key generation algorithm implemented in this library, it is very similar to that of the reference Implementation. In particular, this implementation differs from the previous one in that it stores both the private seed and T in an instance of a C struct `SecretKey` that defines two components: a 32 unsigned char array to store the private seed and a data structure called `bitcontainer` to store T . For all parameter choices, as shown in Table 1, the private seed consists of a 32-byte array.

4. Key Recovery

In this section, we describe our approach to key recovery based on a recent approach to key recovery introduced in [5].

4.1. Assumptions

We continue to make the assumptions outlined in Section 2.2. In particular, we assume that an adversary knows the values r, n, o, v as well as the functions used by the LUOV's key generation algorithm and the corresponding public key $(\text{public_seed}, Q_2)$. This assumption is plausible since all these parameters are public. In addition, the adversary obtains a noisy version of private_seed , which is denoted by ns , and knows $\alpha = P(0 \rightarrow 1)$ and $\beta = P(1 \rightarrow 0)$. Thus, the adversary's goal is to recover the real 32 byte array private_seed and hence the corresponding private components (determined by private_seed).

4.2. Key Recovery Algorithm

In this section we present our key recovery strategy, which is an adaptation of a general key recovery strategy introduced in [5].

Using the notation outlined in Section 2.3.5, let $\text{ns} = (b_0, \dots, b_{W-1})$ denote the bits of the noisy encoding of the private seed and let us name the chunks $\text{ns}^0, \text{ns}^1, \dots, \text{ns}^{\mathcal{N}-1}$ so that $\text{ns}^i = b_{i \cdot w} b_{i \cdot w + 1} \dots b_{i \cdot w + (w-1)}$, where $\mathcal{N} = W/w$. In our setting, we set $W = 256$ bits and $w = 8$, i.e., each chunk is a byte, and thus $\mathcal{N} = 32$. Furthermore, we partition ns into n_B blocks, where each block consists of the concatenation of n_B^j , with $n_B^j > 0$, consecutive chunks, such that $\mathcal{N} = \sum_{j=0}^{n_B-1} n_B^j$. Therefore,

$$\text{ns} = B^0 || B^1 || B^2 || \dots || B^{n_B-1}$$

where

$$B^j = ns^{i_j} || ns^{i_j+1} || \dots || ns^{i_j+n_b^j-1}$$

for $0 \leq j < n_B$ and some $0 \leq i_j < \mathcal{N}$. We now proceed with Phase I as follows.

1. For each chunk ns^i , with $0 \leq i < \mathcal{N}$, we use Equation (1) to compute a log-likelihood score for each candidate c^i for the chunk ns^i . Hence, lists containing up to 2^w chunk candidates, L_{ns^i} , may be produced. In our case, the candidates are the 2^8 possible values, hence each list is of size 256.
2. For each block B^j , the lists $L_{ns^{i_j}}, L_{ns^{i_j+1}}, \dots, L_{ns^{i_j+n_b^j-1}}$ are passed as inputs to an instance of an optimal key enumeration algorithm (denoted as OKEA), in order to create a list containing the M_j candidates having the highest scores for the block B^j , L_{B^j} . Regarding OKEA used in this phase, it is introduced in [24] and enumerates complete candidates c in a decreasing order based on their scores. Expressed in a different way, it first generates the complete candidate that has the highest score, then the one having the second highest score and so on. Therefore, it allows us to find the top M_j candidates with the highest scores in decreasing order, where $1 \leq M_j \ll 2^W$. In our setting, M_j is chosen to be the same value for each block and may take a value in the set $\{256, 512, 1024\}$.

Once Phase I is completed, we proceed with Phase II as follows.

1. The lists L_{B^j} are passed as inputs to an instance of a key enumeration algorithm, which considers each list L_{B^j} as a set of candidates for the block B^j . This instance will generate candidates with a high score for the encoding of the key. Each candidate c generated by the enumeration algorithm is presented as input to a verification function V to determine whether such complete candidate may be regarded as valid or not. As a verification function, we adapt the LUOV key generation algorithm, as shown in Algorithm 2. Note that s is a 32-byte array that represents a full key candidate generated by the Phase II key enumeration algorithm, while $(\text{public_seed}, Q_2)$ is a byte array that represents the public key (without noise). Additionally, note that this test is expected to run very quickly, since the condition of the first **if** is very likely to be **true** only when the enumeration has found the real private seed, since $\mathcal{H}$ is a collision resistant cryptographic hash function. In other words, only the first two instructions (evaluations of $\mathcal{H}$) plus a few instructions of the comparison `Arrays.equal(public_s, public_seed)` will likely be executed for almost all complete candidates generated by the key enumeration algorithm.

Algorithm 2 Verification Function.

```

function TEST( $s$ , ( $\text{public\_seed}, Q_2$ ))
   $\text{private\_sp} \leftarrow \text{InitializeAndAbsorb}(s)$ ; // generate a private sponge candidate from  $s$ .
   $\text{public\_s} \leftarrow \text{SqueezePublicSeed}(\text{private\_sp}, 32)$ ; // compute a public seed candidate from  $\text{private\_sp}$ .
  if Arrays.equal( $\text{public\_s}, \text{public\_seed}$ ) then // check whether  $\text{public\_s}$  and  $\text{public\_seed}$  are equal.
     $T \leftarrow \text{SqueezeT}(\text{private\_sp})$ ; // generate the matrix candidate  $T$  from  $\text{private\_sp}$ .
     $\text{public\_sp} \leftarrow \text{InitializeAndAbsorb}(\text{public\_s})$ ; // generate  $\text{public\_sp}$  from  $\text{public\_s}$ .
     $C, L, Q_1 \leftarrow \text{SqueezePublicMap}(\text{public\_sp})$ ; // generate the matrix candidates  $C, L, Q_1$  from  $\text{public\_sp}$ .
     $Q'_2 \leftarrow \text{CalculateQ2}(Q_1, T)$ ; // compute the matrix  $Q'_2$  from  $Q_1$  and  $T$ .
    if Arrays.equal( $Q'_2, Q_2$ ) then // verify whether  $Q'_2$  and  $Q_2$  are equal.
      return true;
    end if
  end if
  return false;
end function

```

Note that the use of non-optimal key enumeration algorithm would suit better in Phase II [17], since they are parallelisable and memory-efficient. In particular, we make use of a non-optimal key enumeration algorithm combining several good features from various key enumeration algorithms [5],

which we denote as NOKEA. This algorithm can be adjusted to be run over an interval $[a, b]$, with $a \leq b \leq \max$, where $\max$ is the maximum total accumulated score that can be assigned to a complete candidate, such that it enumerates all complete candidates of which total accumulated scores lie in the interval [5]. More specifically, it exploits the fact that many complete candidates will have a same total accumulated score, i.e., only a few total accumulated scores can be attained in the interval. Thus, it constructs a list of all possible attainable total accumulated scores and then, for each total accumulated score s_i in such list, it enumerates all complete candidates of which their total accumulated scores are equal to s_i [5].

Additionally, NOKEA can also be adjusted to run over an interval of the form $[\max - \sigma, \max]$ such that the number of candidates in the interval is the smallest upper bound for a parameter `limit` given to the algorithm, and can also be run with t tasks in parallel. If it is run with a single task, the enumeration performed by it may be near-optimal, since it can be adjusted to enumerate all the complete candidates of which their total accumulated score are orderly taken out of the set of all possible total accumulated scores within the interval (i.e., from the highest score to the lowest one) [5]. However, this good property cannot be attained when it is run using more than one task, since each such task is run independently from one another and over a sub-interval of the interval [5]. Additionally, before running this algorithm, it should be parameterised to use a verification function V to test if a complete candidate c generated while running is a valid candidate [5].

Algorithm 3 shows a pseudo-code representation of our key recovery algorithm. Note that `ns` denotes a 32 byte array that represents the noisy version of `private_seed`. Additionally, `partition` denotes a $(n_B + 1)$ integer array that contains the indices of the limits for each block, i.e., the block j , with $0 \leq j < n_B$, consists of all the chunks i for all $i \in \{\text{partition}[j], \text{partition}[j] + 1, \dots, \text{partition}[j + 1] - 1\}$, while `M` represents a n_B integer array that contains the values M_j for each block. The variables `limit` and `t` represent the number of key candidates to construct the respective interval of the form $[\max - \sigma, \max]$ and the number of tasks in parallel respectively (these are used by the NOKEA algorithm).

Algorithm 3 Key Recovery Algorithm.

```

function KEYRECOVERY(ns, partition, M, limit, t,  $\alpha$ ,  $\beta$ )
  //Phase I
  for ( $i \leftarrow 0, i < 32, i \leftarrow i + 1$ ) do
    for ( $j \leftarrow 0, j < 256, j \leftarrow j + 1$ ) do
       $c \leftarrow j$ ;
       $\text{score} \leftarrow \text{getScore}(\text{ns}[i], c, \alpha, \beta)$ ; // use Equation (1) to compute an score.
       $L_{\text{ns}^i}.\text{add}((\text{score}, c))$ ;
    end for
     $\text{sort}(L_{\text{ns}^i})$  //sort list  $L_{\text{ns}^i}$  in decreasing order based on the score component.
  end for
   $n_B \leftarrow \text{length}(\text{partition}) - 1$ ;
  for ( $j \leftarrow 0, j < n_B, j \leftarrow j + 1$ ) do
    for ( $i \leftarrow \text{partition}[j], i < \text{partition}[j + 1], i \leftarrow i + 1$ ) do
       $L.\text{add}(L_{\text{ns}^i})$ ; //  $L$  is an auxiliary list used to store lists.
    end for
    OKEA.init( $L$ );
    for ( $i \leftarrow 1, i \leq M[j], i \leftarrow i + 1$ ) do
      //returns the next highest-scoring key candidate for block  $j$  and adds it to  $L_{Bj}$ .
       $L_{Bj}.\text{add}(\text{OKEA.getNextCandidate}());$ 
    end for
     $L_B.\text{add}(L_{Bj})$ ; //  $L_B$  is an auxiliary list used to store lists  $L_{Bj}$ .
  end for
  //Phase II
  NOKEA.init( $L_B$ , limit, t);
  NOKEA.run(Test); // a reference to the function Test, i.e., Algorithm 2, is passed as a parameter.
end function

```

5. Experimental Evaluation

In this section, we describe an experimental analysis of our algorithm, focusing on the success rate and performance of our key recovery algorithm.

Success Rate and Performance of Our Key Recovery Algorithm

To run simulations of our algorithm, we set `partition` to $[0, 4, 8, 12, 16, 20, 24, 28, 32]$, meaning the number of blocks is 8, and `M` to $[M, M, M, M, M, M, M, M]$ with $M \in \{256, 512, 1024\}$ and `limit` $\in \{2^{30}, 2^{40}, 2^{50}\}$. Note that, since the number of blocks is 8, a full enumeration during the second phase would enumerate $2^{8 \cdot \log_2 M}$ complete candidates. In addition, according to our cold boot attack model, one of the probabilities is normally a fixed value (circa 0.001), while the other probability values vary in a memory region. Thus, in particular, α was set to 0.001, while $\beta \in \{0.001, 0.01, 0.02, 0.03, \dots, 0.19, 0.2\}$. Additionally, we made some small changes to our key recovery algorithm so that we could estimate success rates without running either a prohibitively costly full enumeration or partial enumerations during the second phase. To do so, we exploited a function of NOKEA that returns an interval of the form $[max - \sigma, max]$ such that the number of candidates in the interval is the smallest upper bound for a given parameter `limit`. We denote it as `getInterval( $\cdot$ )`.

Our results were obtained by running our tweaked algorithm 100 times for each combination of the parameters. Also, the success rate for such combination is calculated as the number of successful trials over the number of trials. In particular, a simulation was run as follows. First, a 256-bit seed representing the real private seed, `private_seed`, was randomly generated and stored in a 32 byte array, and then perturbed according to α and β to obtain its noisy version, `ns`. The tweaked algorithm was then called along with the following parameters `ns`, `private_seed`, `partition`, `M`, α and β . It ran Phase I, thus obtaining the lists L_{Bj} for $j = 1, 2, \dots, 8$, and then checked if each corresponding block j of `private_seed` was found as value of some chunk candidate in the list L_{Bj} . If such check was not satisfied for some j , then it means that, if our recovery algorithm ran a full enumeration in Phase II, then the real private key would not be found. Therefore, the tweaked algorithm set output to $[0, 0, 0, 0]$, returned it and exited. However, if such check was satisfied for all j , then it means that a full enumeration run in Phase II would find the real private key. Therefore, the tweaked algorithm set output to $[1, 0, 0, 0]$, calculated the score `sc` of the real private seed using `private_seed`, `ns`, α and β , and then retrieved three intervals $[max - \sigma_1, max]$, $[max - \sigma_2, max]$, $[max - \sigma_3, max]$ for `limit` $\in \{2^{30}, 2^{40}, 2^{50}\}$, respectively, by calling the function `getInterval( $\cdot$ )` from NOKEA and updated the corresponding entries of output if `sc` is in the corresponding interval. It finally returned output.

Figure 1a–c show a comparison between the success rates of a complete enumeration and partial enumerations with `limit` $\in \{2^{30}, 2^{40}, 2^{50}\}$ for each value assumed by `M`, respectively. As expected, the success rate of a full enumeration is greater than the success rate of a 2^{50} enumeration, which in turn is greater than the success rate of a 2^{40} enumeration, and this in turn greater than the success rate of a 2^{30} enumeration. In addition, each plot shows the same trend, i.e., as β increases, the success rate declines until completely vanishing when $\beta \geq 0.15$.

Figure 2a,b show a comparison between the success rates of complete enumerations and between the success rates of 2^{40} partial enumerations for each value assumed by `M`, respectively. As expected, Figure 2a shows that, as `M` increases, the success rate for a complete enumeration also increases, i.e., the graphs for $M \in \{256, 512\}$ are dominated by the graph for $M = 1024$. Similarly, the same observation holds for 2^{40} enumerations, as Figure 2b shows. Additionally, each graph show the same trend, i.e., as β increases, the success rate declines until completely vanishing when $\beta \geq 0.15$.

Note that our key recovery algorithm essentially searches for a 32-byte array and that the running times of both Phase I and the verification function test can be regarded as negligible. Hence, to estimate the number of key candidates enumerated per millisecond per core during Phase II, we ran it several times for some specific values for α and β and a fixed number of key candidates. For each run of our key recovery algorithm, we measured the elapsed time to enumerate a fixed number of key candidates per core. These runs were executed on a machine with an Intel Xeon CPU

E5-2667 v2 running at 3.30 GHz with 8 cores. We found our key recovery algorithm was able to enumerate between 3000 and 3500 full key candidates per millisecond per core.

Figure 1. Comparison between the success rate of a complete enumeration and partial enumerations with $\text{limit} \in \{2^{30}, 2^{40}, 2^{50}\}$ for $M \in \{256, 512, 1024\}$ and $\alpha = 0.001$. The x -axis represents values for β , while the y -axis represents the success rate.

Figure 2. Comparison between the success rates of complete enumerations and between the success rates of 2^{40} enumerations for $\alpha = 0.001$ and $M \in \{256, 512, 1024\}$, respectively. The x -axis represents values for β , while the y -axis represents the success rate.

6. Conclusions

This research article assessed the feasibility of cold boot attacks on the lifted unbalanced oil and vinegar (LUOV) scheme, a candidate in the ongoing NIST standardisation process of post-quantum signature schemes. Our assessment entailed reviewing two implementations of this scheme, the reference implementation and the libpqcrypto implementation, to learn the most common in-memory private key formats, and developing a key recovery algorithm exploiting the structure of this scheme. Since the LUOV's key generation algorithm generates its private components

and public components from a 256-bit seed, our key recovery algorithm in the cold boot attack setting worked for all the parameter sets recommended for this scheme. Additionally, we tested the effectiveness and performance of our key recovery algorithm through simulations for a wide range of input parameters and found our key recovery algorithm may recuperate the private seed with $\alpha = 0.001$ and β in the range $\{0.001, 0.01, 0.02, \dots, 0.15\}$ via an enumeration of approximately 2^{40} candidates. Thus, this work furthers the research trend of developing key recovery algorithms for different schemes, as evinced by the literature discussed at length in Section 2.3, and also furthers the research line of evaluating the leading post-quantum candidates against this class of attack. As future research work, we believe this work might be extended to other multivariable polynomial signature schemes, e.g., Rainbow, which is also a strong candidate in the NIST standardisation process of post-quantum signature schemes.

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. Beullen, W.; Preneel, B.; Szeponiec, 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; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2018. Available online: <https://www.esat.kuleuven.be/cosic/pqcrypto/luov/> (accessed on 2 February 2020).
2. Beullens, W.; Preneel, B. Field Lifting for Smaller UOV Public Keys. In *Progress in Cryptology—INDOCRYPT 2017*; Patra, A., Smart, N.P., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 227–246. [CrossRef]
3. Kipnis, A.; Patarin, J.; Goubin, L. Unbalanced Oil and Vinegar Signature Schemes. In *Advances in Cryptology—EUROCRYPT '99*; Stern, J., Ed.; Springer: Berlin/Heidelberg, Germany, 1999; pp. 206–222. [CrossRef]
4. Krämer, J.; Loiero, M. Fault Attacks on UOV and Rainbow. In *Constructive Side-Channel Analysis and Secure Design*; Polian, I., Stöttinger, M., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 193–214. [CrossRef]
5. Villanueva-Polanco, R. Cold Boot Attacks on Bliss; In *Progress in Cryptology—LATINCRYPT 2019*. Schwabe, P., Thériault, N., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 40–61. [CrossRef]
6. 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. In *Proceedings of the 17th USENIX Security Symposium, San Jose, CA, USA, 28 July 28–1 August 2008*; pp. 45–60.
7. Heninger, N.; Shacham, H. Reconstructing RSA Private Keys from Random Key Bits. In *Advances in Cryptology—CRYPTO 2009*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 1–17. [CrossRef]
8. Henecka, W.; May, A.; Meurer, A. Correcting Errors in RSA Private Keys. In *Advances in Cryptology—CRYPTO 2010*; Rabin, T., Ed.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 351–369. [CrossRef]
9. Paterson, K.G.; Polychroniadou, A.; Sibborn, D.L. A Coding-Theoretic Approach to Recovering Noisy RSA Keys. In *Advances in Cryptology—ASIACRYPT 2012*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 386–403. [CrossRef]
10. Lee, H.T.; Kim, H.; Baek, Y.J.; Cheon, J.H. Correcting Errors in Private Keys Obtained from Cold Boot Attacks. In *Information Security and Cryptology—ICISC 2011*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 74–87. [CrossRef]
11. Poettering, B.; Sibborn, D.L. Cold Boot Attacks in the Discrete Logarithm Setting. In *Topics in Cryptology—CT-RSA 2015*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 449–465. [CrossRef]
12. Albrecht, M.; Cid, C. Cold Boot Key Recovery by Solving Polynomial Systems with Noise. In *Applied Cryptography and Network Security*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 57–72. [CrossRef]

13. 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, Venice, Italy, 18–25 July 2010; IEEE Computer Society: Washington, DC, USA, 2010; pp. 216–220. [\[CrossRef\]](#)
14. Huang, Z.; Lin, D. A New Method for Solving Polynomial Systems with Noise over $\mathbb{F}_2$ 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. [\[CrossRef\]](#)
15. 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. [\[CrossRef\]](#)
16. 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. [\[CrossRef\]](#)
17. Villanueva-Polanco, R. A Comprehensive Study of the Key Enumeration Problem. *Entropy* **2019**, *21*, 972. [\[CrossRef\]](#)
18. Bogdanov, A.; Kizhvatov, I.; Manzoor, K.; Tischhauser, E.; Witteman, M. Fast and Memory-Efficient Key Recovery in Side-Channel Attacks. In *Selected Areas in Cryptography—SAC 2015*; Springer: Cham, Switzerland, 2016; pp. 310–327. [\[CrossRef\]](#)
19. David, L.; Wool, A. A Bounded-Space Near-Optimal Key Enumeration Algorithm for Multi-subkey Side-Channel Attacks. In *Topics in Cryptology—CT-RSA 2017*; Springer: Cham, Switzerland, 2017; pp. 311–327. [\[CrossRef\]](#)
20. 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 15 January 2020).
21. Martin, D.P.; Mather, L.; Oswald, E.; Stam, M. Characterisation and Estimation of the Key Rank Distribution in the Context of Side Channel Evaluations. In *Advances in Cryptology—ASIACRYPT 2016*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 548–572. [\[CrossRef\]](#)
22. Martin, D.P.; O’Connell, J.F.; Oswald, E.; Stam, M. Counting Keys in Parallel After a Side Channel Attack. In *Advances in Cryptology—ASIACRYPT 2015*; Iwata, T., Cheon, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2015; pp. 313–337. [\[CrossRef\]](#)
23. Poussier, R.; Standaert, F.X.; Grosso, V. Simple Key Enumeration (and Rank Estimation) Using Histograms: An Integrated Approach. In *Advances in Cryptology—ASIACRYPT 2015*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 61–81. [\[CrossRef\]](#)
24. Veyrat-Charvillat, N.; Gérard, B.; Renaud, M.; Standaert, F.X. An Optimal Key Enumeration Algorithm and Its Application to Side-Channel Attacks. In *Selected Areas in Cryptography—SAC 2012*; Knudsen, L.R., Wu, H., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 390–406. [\[CrossRef\]](#)
25. Veyrat-Charvillat, N.; Gérard, B.; Standaert, F.X. Security Evaluations beyond Computing Power. In *Advances in Cryptology—EUROCRYPT 2013*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 126–141. [\[CrossRef\]](#)
26. 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 20 November 2019).
27. 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.
28. Choudary, M.O.; Popescu, P.G. Back to Massey: Impressively Fast, Scalable and Tight Security Evaluation Tools. In *Cryptographic Hardware and Embedded Systems—CHES 2017*; Springer: Cham, Switzerland, 2017; pp. 367–386. [\[CrossRef\]](#)
29. Choudary, M.O.; Poussier, R.; Standaert, F.X. Score-Based vs. Probability-Based Enumeration—A Cautionary Note. In *Progress in Cryptology—INDOCRYPT 2016*; Springer: Cham, Switzerland, 2016; pp. 137–152. [\[CrossRef\]](#)
30. Glowacz, C.; Grosso, V.; Poussier, R.; Schüth, J.; Standaert, F.X. Simpler and More Efficient Rank Estimation for Side-Channel Security Assessment. In *Fast Software Encryption*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 117–129. [\[CrossRef\]](#)

31. 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. [CrossRef]
32. 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. [CrossRef]
33. The Libpqcrypto Implementation. Available online: <https://libpqcrypto.org/index.html> (accessed on 20 February 2020).

© 2020 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).