Cold Boot Attacks on NTRU

Kenneth G. Paterson and Ricardo Villanueva-Polanco

Information Security Group, Royal Holloway, University of London,
{kenny.paterson,ricardo.villanuevapolanco.2013}@rhul.ac.uk

Abstract. Cold boot attacks target memory remanence effects in hardware to secret key material. Such attacks were first explored in the scientific literature by Halderman *et al.* (USENIX Security Symposium 2008) and, since then, different attacks have been developed against a range of asymmetric key and symmetric key algorithms. Such attacks in general receive as input a noisy version of the secret key as stored in memory, and use redundancy in the key (and possibly knowledge of a public key) to recover the secret key. The challenge is to recover the key as efficiently as possible in the face of increasing levels of noise. For the first time, we explore the vulnerability of lattice-based cryptosystems to this form of analysis, focussing in particular on NTRU, a well-established and attractive public-key encryption scheme that seems likely to be a strong candidate for standardisation in NIST's post-quantum process. We look at two distinct NTRU implementations, showing how the attacks that can be developed depend critically on the in-memory representation of the secret key. We develop, efficient, dedicated key-recovery algorithms for the two implementations and provide the results of an empirical evaluation of our algorithms.

Keywords: cold boot attacks, NTRU, key enumeration.

1 Introduction

Cold boot attacks have received significant attention since they were first described in the literature by Halderman *et al.* nearly a decade ago [7] (see also [8]). This class of attack relies on the fact that computer memory normally keeps information when going through a power-down/power-up cycle, so an adversary might be able to gain access to confidential information such as cryptographic keys after a system reboot. Unluckily for such an adversary, once the power is cut off, the bits in memory will undergo a gradual degradation, meaning that any information retrieved from the computer memory will probably be noisy. Thus, once the location of the key in memory has been discovered (itself a non-trivial task), the adversary's task becomes the mathematical problem of recovering a key from a noisy version of that key. The adversary may have access to reference cryptographic data created using that key (e.g. ciphertexts for a symmetric key encryption scheme) or have a public key available (in the asymmetric setting).

The amount of time for which information is maintained while the power is off depends on the particular memory type and the ambient temperature. Experimental results shown in [7] reveal that, at normal operating temperatures, there is little corruption within the first few seconds, but this phase is then followed by a rapid decay. The period of mild corruption can be prolonged by cooling the memory chips. For instance, according to [7], in an experiment at $-50°C$ (which can be achieved by spraying compressed air onto the memory chips) less than 0.1% of bits decay within the first minute. At temperatures of approximately $-196°C$ (achieved by means of the use of liquid nitrogen) less than 0.17% of bits decay within the first hour. Notably, once power has been switched off, the memory will be partitioned into regions, and each region will have a 'ground state' which is associated with a bit, 0 or 1. In a 0 ground state, the 1 bits will eventually decay to 0 bits, while the probability of a 0 bit switching to a 1 bit is very small, but not vanishing (a common probability is circa 0.001 [7]). When the ground state is 1, the opposite is true. An attacker can determine the ground state of a particular region of memory rather easily in an attack by reading all the bits and determining how many of them are 0 bits and how many are 1 bits.

The main focus of cold boot attacks after the initial work pointing out their feasibility [7] has been to develop algorithms for efficiently recovering keys from noisy versions of those keys for a range of different cryptosystems, whilst exploring the limits of how much noise can be tolerated. Heninger and Shacham [10] focussed on the case of RSA keys, giving an efficient algorithm based on Hensel lifting to exploit redundancy in the typical RSA private key format. This work was followed up by Henecka, May and Meurer [9] and Paterson, Polychroniadou and Sibborn [18], with both papers also focussing on the mathematically highly structured RSA setting. The latter paper in particular pointed out the *asymmetric* nature of the error channel intrinsic to the cold boot setting and recast the problem of key recovery for cold boot attacks in an information theoretic manner. Cold boot attacks in the discrete logarithm setting were considered in [19]. There, the authors emphasise the critical role of the format in which the private key is stored in memory in the development and success of attacks. Several papers have considered cold boot attacks in the symmetric key setting, including Albrecht and Cid [1] who focussed on the recovery of symmetric encryption keys in the cold boot setting by employing polynomial system solvers, and Kamal and Youssef [14] who applied SAT solvers to the same problem. Further research on the development of cold boot attacks for specific schemes can be found in [15,13] Cold boot attacks are also widely cited in the theoretically-oriented literature on leakage-resilient cryptography, but the relevance there is marginal because the cold boot attack scenario (direct access to a noisy version of the whole key) does not really apply in the leakage-resilient setting.

1.1 Our contributions

In this paper, we examine the feasibility of cold boot attacks against the NTRU public key encryption scheme [12,11]. We believe this to be the first time that this has been attempted. Our work can be seen as a continuation of the trend to develop cold boot attacks for different schemes (as evinced by the literature cited above). But it can also be seen as the beginning of the evaluation of the leading post-quantum candidates against this class of attack. Such an evaluation should form a small but important part of the overall assessment of schemes in the soon-to-commence NIST selection process for post-quantum algorithms.[1] In particular, this paper evaluates what seems likely to be a leading candidate and lays the groundwork for the later study of other likely candidates in the same broad family of schemes that operate over polynomial rings (such as NTRUprime [3] and various recently proposed ring-LWE-based schemes [2,5]).

As noted above, the exact format in which the private key is stored is critical to developing key recovery attacks in the cold boot setting. This is because the attack depends on physical effects in memory, represented by bit flips in private key bits, and the main input to the attack is a bit-flipped version of the private key. For this reason, it is necessary to either propose natural ways in which keys would be stored in memory in NTRU implementations or to examine specific implementations of NTRU. We adopt the latter approach, and we study two distinct implementations. The first, `ntru-crypto`, is a pair of C and Java libraries developed by OnBoard Security, a spin off of Security Innovation, the patent-holder for some NTRU technology.[2] The second, `tbuktu` is a pair of libraries developed by "Tim Buktu", and is available in 'C' and Java languages.[3] A fork of the Java implementation is included in the popular Bouncy Castle Java crypto library.[4]

Each of these implementations stores its private keys in memory in slightly different ways. For example, in Java, `tbuktu` supports a number of different formats, including a representation where the key is stored as 6 lists of indices, each index being a 32-bit integer representing a position where a certain polynomial has a coefficient of value $+1$ or -1. Meanwhile, `ntru-crypto`'s C implementation uses a special representation of polynomial coefficients by trits (three-valued bits), and then packs 5 trits at a time into octets using base-3 arithmetic.

Each of these different private key formats therefore requires a different approach to key recovery in the cold boot setting. In this paper, we will focus on just a couple of the more interesting cases, where there is some additional structure

[1] See `http://csrc.nist.gov/groups/ST/post-quantum-crypto/` for details of the NIST process.

[2] See `https://github.com/NTRUOpenSourceProject/ntru-crypto` for the code and `https://www.onboardsecurity.com/products/ntru-crypto/ntru-resources` for a list of useful resources related to NTRU.

[3] See `http://tbuktu.github.io/ntru/`.

[4] See `http://bouncycastle.org/`.

that we can exploit, or where novel approaches are called for. Nevertheless, we will pose the problem of key recovery in a more general way that makes it possible to see how to generalise our ideas to cover other cases. Specifically, each of our analyses involves splitting the (noisy) private key into chunks, and creating log-likelihood estimates for each candidate value for each of the chunks. Each such estimate can be regarded as a per chunk score. A log-likelihood estimate (or score) for a candidate for the complete private key can then be computed by summing the per chunk scores across the different chunks. Our problem then becomes one of efficiently enumerating complete candidates and their scores based on lists of candidates for chunks and per-chunk scores, so that each complete candidate can then be tested for correctness (for example, by trial encryption and decryption). It makes sense to perform the enumeration in decreasing order of score if possible, starting with the most likely candidate. This is a problem that also arises in the side-channel attack literature, cf. [20,4,17,16,6], where, for example, one might obtain scoring information for each byte of an AES key from a power analysis attack and then want to efficiently enumerate and test a large number of complete16-byte candidates in decreasing order of score until the correct key is found. We are able to apply standard algorithms (e.g. depth-first search on a tree with pruning) as well as algorithms from this literature to solve the key recovery problem in our context.

1.2 Paper organisation

This paper is organised as follows. Section 2 describes cold boot attacks and the adversary model in more detail. It also gives a basic statistical approach to recovering private keys in the face of noise, based on maximum likelihood estimation. Section 3 describes the NTRU public key encryption algorithm and details of the two implementations that we target. Section 4 describes our algorithms for attacking these implementations; these are largely based on established key enumeration techniques from the literature on side-channel attacks. Section 5 describes our implementation of the algorithms and the results of our empirical evaluation of the performance of the attacks. We conclude in Section 6

2 Further Background

2.1 Cold boot Attack Model

Our cold boot attack model assumes that the adversary can obtain a noisy version of the original NTRU private key (using whatever format is used to store it in memory). We assume that the corresponding NTRU public key is known exactly (without noise). We do not consider here the important problem of how to locate the appropriate area of memory in which the private key bits are stored, though this would be an important consideration in practical attacks.

Our aim is then recover the private key. Note that it is sufficient to recover a list of key candidates in which the true private key is located, since we can always test a candidate by doing a trial encryption using the known public key and then decryption using the candidate. It is highly likely that a simple test of this type will filter out all wrong candidates (especially when the NTRU variant considered is CCA secure).

We assume throughout that a 0 bit of the original private key will flip to a 1 with probability $\alpha = P(0 \to 1)$ and that a 1 bit of the original private key will flip with probability $\beta = P(1 \to 0)$. We do not assume that $\alpha = \beta$; indeed, in practice, one of these values may be very small (e.g. 0.001) and relatively stable over time, while the other increases over time. Furthermore, we assume that the attacker knows the values of α and β and that they are fixed across the region of memory in which the private key is located. These assumptions are reasonable in practice: one can estimate the error probabilities by looking at a region where the memory stores known values (e.g. where the public key is located), and the regions are typically large. Moreover, our algorithms will be fairly robust to mis-estimations of these parameters.

2.2 Log Likelihood Statistic for Key Candidates

Suppose we have a true private key that is W bits in size, and let $\mathbf{r} = (r_0, \ldots, r_{W-1}$ denote the bits of the noisy key (input to the adversary in the attack). Suppose a key recovery algorithm constructs a candidate for the private key $\mathbf{c} = (c_0, \ldots, c_{W-1})$ by some means (to be determined). Then, given the bit-flip probabilities α, β, we can assign a likelihood score to $\mathbf{c}$ as follows:

$$L[\mathbf{c}; \mathbf{r}] := \Pr[\mathbf{r}|\mathbf{c}] = (1-\alpha)^{n_{00}} \alpha^{n_{01}} \beta^{n_{10}} (1-\beta)^{n_{11}}$$

where n_{00} denotes the number of positions where both $\mathbf{c}$ and $\mathbf{r}$ contain a 0 bit, n_{01} denotes the number of positions where $\mathbf{c}$ contains a 0 bit and $\mathbf{r}$ contains a 1 bit, etc.

The method of maximum likelihood estimation[5] then suggests picking as $\mathbf{c}$ the value that maximises the above expression. It is more convenient to work with log likelihoods, and equivalently to maximise these, viz:

$$\mathcal{L}[\mathbf{c}; \mathbf{r}] := \log \Pr[\mathbf{r}|\mathbf{c}] = n_{00} \log(1-\alpha) + n_{01} \log \alpha + n_{10} \log \beta + n_{11} \log(1-\beta).$$

We will frequently refer to this log likelihood expression as a *score* and seek to maximise its value (or, equally well, minimise its negative).

[5] See for example `https://en.wikipedia.org/wiki/Maximum_likelihood_estimation`.

2.3 Combining Chunks to Build Key Candidates

Now suppose that the true private key **r** can be represented as a concatenation of W/w chunks, each on w bits. As we shall see in the specific analyses of different NTRU implementations, this will be the case in practice. For example, each chunk might arise from the value of a coefficient of some polynomial making up the NTRU private key.

Let us name the chunks $r^0, r^1, \ldots, r^{W/w-1}$ so that $r^0 = r_0 r_1 \ldots r_{w-1}$, $r^1 = r_w r_{w+1} \ldots r_{2w-1}$, etc. Suppose also that candidates **c** can be represented by concatenations of chunks $c^0, c^1, \ldots, c^{W/w-1}$ in the same way.

Suppose further that each of the at most 2^w candidates for chunk c^i $(0 \leq i < W/w)$ can be enumerated and given its own score by some procedure (formally, a sub-algorithm in an overall attack). For example, the above expression for log likelihood across all W bits of private key is easily modified to produce a log likelihood expression for any candidate for chunk i as follows:

$$\mathcal{L}[c^i; r^i] := \log \Pr[r^i | c^i] = n^i_{00} \log(1-\alpha) + n^i_{01} \log \alpha + n^i_{10} \log \beta + n^i_{11} \log(1-\beta) \tag{1}$$

where the n^i_{ab} values count occurrences of bits across the i-the chunks, r^i, c^i.

Thus we can assume that we have access to W/w lists of scores, each list containing up to 2^w entries. Note that W/w scores, one from each of these per-chunk lists, can be added together to create a total score for a complete candidate **c**. Indeed, this total score is statistically meaningful in the case where the per-chunk scores are log likelihoods because of the additive nature of the scoring function in that case.

The question then becomes: can we devise efficient algorithms that traverse the lists of scores to combine chunk candidates c^i, obtaining complete key candidates **c** having high total scores (with total scores obtained by summation)? As noted in the introduction, this is a problem that has been previously addressed in the side-channel analysis literature [20,4,17,16,6], with a variety of different algorithmic approaches being possible to solving the problem. We shall return to this question after having described the specific NTRU implementations that we will attack.

3 NTRU Encryption Scheme and Private Key Formats

In this section we briefly describe the NTRU public key encryption scheme and explore the various private key formats in the two implementations we will be working with.

Let $N, p, q \in \mathbb{Z}^+$. We define three polynomial rings:

$$R = \mathbb{Z}[x]/(X^N - 1), \quad R_p = \mathbb{Z}_p[x](X^N - 1), \quad R_q = \mathbb{Z}_q[x](X^N - 1).$$

Thus, for example, elements of R_p can be represented as polynomials of degree at most $N-1$ with coefficients from $\mathbb{Z}_p$. They can also be represented as vectors of dimension N over $\mathbb{Z}_p$ in the natural way, and we will switch between representations at will.

Definition 1. *Let* $\mathbf{a} \in R_q$ *The centred lift of* $\mathbf{a}$ *to* R *is the unique polynomial* $\mathbf{a}' \in R$ *satisfying* $\mathbf{a}' \mod q = \mathbf{a}$ *whose coefficients all lie in the interval* $-\frac{q}{2} < a'_i \leq \frac{q}{2}$.

Definition 2. *Let* r *be a fixed integer and let* C_r *be the function that, given* $\mathbf{a} \in R$, *outputs the number of coefficients of* $\mathbf{a}$ *equal to* r. *Let* $d_1, d_2 \in \mathbb{Z}^+$. *We define* $T(d_1, d_2) = \{\mathbf{a} \in R \mid C_1(\mathbf{a}) = d_1, C_{-1}(\mathbf{a}) = d_2, C_0(\mathbf{a}) = N - d_1 - d_2\}$. *Note that* $|T(d_1, d_2)| = \binom{N}{d_1}\binom{N-d_1}{d_2}$. *An element* $\mathbf{a} \in R$ *is called a ternary polynomial if and only if* $\mathbf{a} \in T(d_1, d_2)$ *for some* $d_1, d_2 \in \mathbb{Z}^+$.

3.1 NTRU Public Key Encryption Scheme

The NTRU public key encryption scheme is a lattice-based alternative to RSA and ECC with security that is (informally) based on the problem of finding the shortest vector in a particular class of lattices. The scheme exists in several different versions, offering different forms of security (IND-CPA, IND-CCA). The details of the scheme's operation matter less to us than the format of private keys in implementations. However, for completeness, we give an overview of NTRU. We follow the description in [11].

The scheme relies on public parameters (N, p, q, d) with N and p prime, $gcd(p, q) = gcd(N, q) = 1$ and $q > (6d + 1)p$.

Key generation:

1. Choose $\mathbf{f} \in T(d + 1, d)$ that is invertible in R_q and R_p.
2. Choose $\mathbf{g} \in T(d_1, d_2)$ for some $d_1, d_2 \in \mathbb{Z}^+$.
3. Compute $\mathbf{f_p}$, the inverse of $\mathbf{f}$ in R_p.
4. Compute $\mathbf{f_q}$, the inverse of $\mathbf{f}$ in R_p.
5. The public key is $\mathbf{h} = p\mathbf{f_q} \cdot \mathbf{g} \in R_q$; the private key is the pair $(\mathbf{f}, \mathbf{f_p})$.

Encryption: On input message $\mathbf{m}$, which we assume to be a centre-lifted version of an element of R_p, and public key $\mathbf{h}$:

1. Choose a random $\mathbf{r}$ with small coefficients, in particular $\mathbf{r}$ can be chosen such that $\mathbf{r} \in T(d, d)$.
2. Compute the ciphertext $\mathbf{e}$ as $\mathbf{e} = \mathbf{r} \cdot \mathbf{h} + \mathbf{m} \in R_q$.

Decryption: On input ciphertext $\mathbf{e}$ and private key $(\mathbf{f}, \mathbf{f_p})$:

1. Compute $\mathbf{b} = \mathbf{f} \cdot \mathbf{e}$ in R_q. (Note that this yields $\mathbf{b} = p\mathbf{g} \cdot \mathbf{r} + \mathbf{f} \cdot \mathbf{m}$ over R_q.)
2. Centre-lift $\mathbf{b}$ modulo q to obtain $\mathbf{a}$, and then compute $\mathbf{f_p} \cdot \mathbf{a} \in R_p$. Centre-lift the result modulo p to obtain $\mathbf{m}'$.

We omit the correctness proof for this description of the NTRU scheme. Note that $\mathbf{f_p}$ can be computed from $\mathbf{f}$) on the fly, and so some implementations may only store $\mathbf{f}$ as the private key.

3.2 Private Key Formats for NTRU Implementations

NTRU at first was only available as a proprietary, paid-for library. It was not until 2011 that the first open-source implementation, `tbuktu`, appeared under a BSD licence.[6] A fork of this first implementation forms the basis of the NTRU code in the Bouncy Castle library. Two years later Security Innovation exempted open source projects from having to obtain a patent license for their `ntru-crypto` implementation[7] and released an NTRU reference implementation under the GPL v2 licence. Each of these two implementations is available in both Java and C. We examine the private key formats for each of these implementations in turn.

3.2.1 The `tbuktu`/Bouncy Castle Java Implementation. In this implementation, there are four pieces of information that determine how the private key is stored:

1. A variable `t` that points to a polynomial and from which variables corresponding to the private key components $\mathbf{f}$ and $\mathbf{f_p}$ are constructed.
2. A variable `polyType` that indicates the type of polynomial to use. This can hold two values: `SIMPLE` or `PRODUCT`.
3. A boolean `sparse` that indicates if `t` is an sparse or dense polynomial. This variable applies only if `polyType` has value `SIMPLE`.
4. A boolean `fastFp` that indicates the manner in which $\mathbf{f}$ is built from `t`. If `fastFp` = `true`, then $p = 3$, $\mathbf{f} = 1 + 3$`t` and $\mathbf{f_p} = 1$; otherwise $\mathbf{f} =$ `t` and $\mathbf{f_p} = \mathtt{t}^{-1} \bmod p$. This relates to an implementation trick for the case $p = 3$.

When `polyType` has the value `SIMPLE`, `t` will be either a dense ternary polynomial or a sparse ternary polynomial, as determined by the value of `sparse`. In the dense case, `t` is represented as an `int` array of length N whose entries have values from $\{-1, 0, 1\}$. In memory, each entry is stored as a 32-bit signed integer, using two's complement, i.e, $+1$ is stored as the 32-bit string $000\ldots01$,

[6] See `http://tbuktu.github.io/ntru/`.
[7] See `https://github.com/NTRUOpenSourceProject/ntru-crypto/blob/master/FOSS%20Exception.md`.

0 is stored as $000\dots00$ and -1 is stored as $111\dots11$. Meanwhile, in the sparse case, `t` is represented as two `int` arrays, `ones` and `negOnes`, where:

1. The array `ones` contains the indices of the $+1$ coefficients of `t` in increasing order (so that the entries in the area are 32-bit representations of integers in the range $[0, N-1]$).
2. The array `negOnes` contains the indices of the -1 coefficients of `t` in increasing order (with entries having the same bit representation as the entries of `ones`).

When `polyType` has the value `PRODUCT`, `t` will be a product form polynomial. In this case, `t` is represented by three different sparse ternary polynomials $\mathbf{f_1}, \mathbf{f_2}, \mathbf{f_3}$ such that $\mathbf{t} = \mathbf{f_1}\mathbf{f_2} + \mathbf{f_3}$. All three of $\mathbf{f_1}, \mathbf{f_2}, \mathbf{f_3}$ are stored in memory separately in sparse form. This means that, when `polyType` has the value `PRODUCT`, then the private key is represented in memory by a total of 6 `int` arrays $\mathbf{f_i}$`.ones`, $\mathbf{f_i}$`.negOnes`, $1 \leq i \leq 3$.

Note that the private key formats for the `tbuktu` C implementation are largely the same as for the Java one, and so we do not detail them further here.

3.2.2 Reference Parameters for `tbuktu`. The `tbuktu` implementation includes 10 named reference parameter sets with a range of choices for N and q, targeting different security levels and optimisations. These 10 sets are detailed in the `EncryptionParameters` class.

For example, both `APR2011_439` and `APR2011_439_FAST` parameter sets target 128 bits of security. If the former set is selected, $\mathbf{f} = \mathbf{t}$, with $\mathbf{t}$ being represented as a sparse ternary polynomial with `df` $= 146$ coefficients set to $+1$ and with `df` $-\ 1$ of them set to -1. If the latter set is selected, then $\mathbf{f} = 1 + 3\mathbf{t}$, with $\mathbf{t} = \mathbf{f_1}\mathbf{f_2} + \mathbf{f_3}$; moreover $\mathbf{f_1}$ has `df1` $= 9$ coefficients set to each of $+1$ and -1, so $\mathbf{f_1} \in T(9, 9)$ (while `df2` $= 8$ and `df3` $= 5$ for $\mathbf{f_2}$ and $\mathbf{f_3}$, respectively).

3.2.3 The `ntru-crypto` Java Implementation. Here, the private key $\mathbf{f}$ is always of the form $1 + 3\mathbf{t}$ where $\mathbf{t}$ is a ternary polynomial and we have $p = 3$ (so that $\mathbf{f_p} = 1$). In this implementation, $\mathbf{f}$ is stored directly in memory as an array of short integers. That is, the coefficients $f_0, f_1, \dots, f_{N-1}$ of $\mathbf{f}$ are stored as a sequence of 16-bit signed two's complement integers with $f_0 \in \{-2, 1, 4\}$ and $f_i \in \{-3, 0, 3\}$, for $1 \leq i < N$.

3.2.4 The `ntru-crypto` C Implementation. Here, key generation is carried out by the function `ntru_crypto_ntru_encrypt_keygen`. During its execution, $\mathbf{f}$ (the private key) is initially generated as either a product of polynomials ($\mathbf{f} = \mathbf{f_1} \cdot \mathbf{f_2} + \mathbf{f_3}$) or as a single polynomial. Either way, $\mathbf{f}$ is represented internally as a list of the indices of the $+1$ coefficients followed by a list of the indices of the -1 coefficients, where each index is stored in an unsigned 16-bit integer. This data is then used to construct a packed private key blob following one of two formats. The information-dense nature of these formats makes it harder to mount cold boot key recovery attacks that perform significantly better than a combinatorial search based on searching over low-weight error patterns. For this reason we do not consider this format any further in this paper.

3.2.5 Reference Parameters for `ntru-crypto`. The `ntru-crypto` implementation includes 12 named reference parameter sets with a range of choices for N and q, targeting different security levels and optimisations. For the Java implementation, these parameter sets are defined in the `KeyParams` class. The values of N range from 401 to 1499, with $p = 3$ and $q = 2048$ throughout; the number of $+1$'s and -1's in $\mathbf{f}$ (resp. $\mathbf{g}$), denoted `df` (resp. `dg`) depends on N; for example, for the parameter set `ees449ep1`, we have $N = 449$, `df` $= 134$, and `dg` $= 149$.

4 Mounting Cold Boot Key Recovery Attacks

In this section, we present our cold boot key recovery attacks on the implementations and corresponding private key formats introduced in the previous section. First, because of its simplicity, we consider the `ntru-crypto` Java Implementation (in which $\mathbf{f}$ is stored directly in memory as an array of short integers). We will then consider the `tbuktu` Java Implementation in which the `PRODUCT` form of private key is used and such that `fastFp` $=$ `true`, so that $\mathbf{f} = 1 + 3\mathbf{t}$ with $\mathbf{t} = \mathbf{f_1}\mathbf{f_2} + \mathbf{f_3}$ where all three of $\mathbf{f_1}, \mathbf{f_2}, \mathbf{f_3}$ are stored in memory in sparse form.

We continue to make the assumptions outlined in Section 2. We additionally assume that all relevant public parameters and private key formatting information are known to the adversary.

4.1 The `ntru-crypto` Java Implementation

Recall from Section 3.2.3 that the coefficients of $\mathbf{f}$ are stored directly in memory as an array of 16-bit, signed two's complement integers, with $f_0 \in \{-2, 1, 4\}$ and $f_i \in \{-3, 0, 3\}$, for $1 \leq i < N$. For simplicity, we assume that f_0 is known (there are only 3 possible values for f_0 and the attack can be repeated for each possible value). The attacker then receives a noisy version $\mathbf{r} = (r_0, \ldots, r_{W-1})$ of the array with entries $f_1, \ldots, f_{N-1}$ which is $W = 16(N-1)$ bits in size. In the terminology of Section 2, we set $w = 16$ and partition the noisy key into $W/w = N - 1$ chunks r^i, each chunk corresponding to a single, 16-bit encoded coefficient f_{i+1}.

Using equation (1), we can compute log-likelihood scores $\mathcal{L}[c^i; r^i]$ for each chunk i and each candidate c^i for that chunk. Note that in each chunk, there are only

3 possible candidates c^i, since $f_i \in \{-3, 0, 3\}$. (In the general formulation with $w = 16$ there could be up to 2^{16} candidates per chunk.)

Hence we obtain $N-1$ lists of scores (log-likelihood values), each list containing 3 values. Alternatively, we can think of this as being an array of size $3 \times (N-1)$. Our task now is to combine candidates, one per chunk, to generate complete private key candidates $\mathbf{c}$ with high log-likelihoods, which can then be tested via trial encryption and decryption.

In order to generate complete private key candidates $\mathbf{c}$ with high scores, we employ an algorithm that is closely based on that of [17] from the side-channel attack literature. Specifically, as in [17], we use a standard depth-first search across the chunk counter i to enumerate candidates. This employs a stack, with partial cumulative scores for candidates at "depth" i in the search being computed by adding the chunk score at depth i to a cumulative score for the candidates at "depth" $i-1$. Once "depth" $N-1$ is reached, and a complete candidate is generated, the candidate can be filtered and then tested. (In fact, the known restrictions on the number of $+3$ and -3 coefficients in private keys for the standard parameters that we are attacking can be used to perform early aborts on partial candidates.)

As in [17], we can restrict the search space to certain intervals of scores by appropriate pruning of partial solutions. By representing a complete search space as a union of intervals, a degree of parallelisation can be achieved (but this may involve repeated computation). We can also adapt the approaches of [17,16] to perform enumeration rather than generating candidates – computing *how many* candidates have scores in a given interval. This is useful for estimating the likely performance of the search algorithm. However, a significant difference with [17,16] arises from the parameters involved – there, typically there are 16 chunks with 256 candidates per chunk (corresponding to AES key bytes), whereas here we will have on the order of a few hundred chunks and only 3 candidates per chunk.

4.2 The `tbuktu` Java Implementation

Now we turn our attention to the `tbuktu` Java implementation in some of its more interesting cases. Recall from Section 3.2.1 that when the `PRODUCT` form of private key is used and when `fastFp` = `true`, then we have $\mathbf{f} = 1 + 3\mathbf{t}$ with $\mathbf{t} = \mathbf{f_1 f_2} + \mathbf{f_3}$ where all three of $\mathbf{f_1}, \mathbf{f_2}, \mathbf{f_3}$ are stored in memory in sparse form.

This means that we have 6 arrays of indices in memory $\mathbf{f_i}$.`ones`, $\mathbf{f_i}$.`negOnes`, $1 \leq i \leq 3$. Each array is of type `int` and each entry in each array stores the position of either a $+1$ or a -1 coefficient in one of the polynomials $\mathbf{f_i}$; moreover the entries should be in increasing order. We assume the starting positions in memory, total sizes, and ranges of possible values in each of these tables is known. We also know that for any pair $\mathbf{f_i}$.`ones`, $\mathbf{f_i}$.`negOnes`, the two tables of values should be non-intersecting. We let L_i denote the common length of the two arrays

$\mathbf{f_i}$.`ones`, $\mathbf{f_i}$.`negOnes` (this is determined by the parameters used to generate the private key).

We now present a two-phase attack to generate complete private key candidates.

4.2.1 Phase 1.

In the first phase, we apply a modified version of the *Optimal Key Enumeration Algorithm (OKEA)* of [20]. As in the description in Section 2, this algorithm takes as input a collection of W/w lists of candidates, one list per chunk, and produces as output a list of `lsize` complete candidates, each across all W bits. It uses a dynamic programming version of a list merging strategy to generate complete candidates in decreasing order of score. The OKEA algorithm has the property that it is guaranteed to output the `lsize` highest scoring (i.e. most likely) candidates across all the chunks (hence its optimality). It seems to be particularly effective when W/w, the number of chunks being considered, is moderate – [20] applied it in the case of reconstructing 16-byte AES keys from their bytes, with 16 chunks.

We perform this step for each of our 6 arrays as follows: we build W/w lists of candidates, setting $w = 32$ and $W = wL_i$ so that we have L_i chunks. Each chunk corresponds to one `int` entry in the array, and each list is of size N (since, at the outset, every chunk could take on any value between 0 and $N - 1$, these being the possible indices of a $+1$ or -1 coefficient). The score for each entry in each list is obtained using our per-chunk log-likelihood expression (1). We modify the OKEA algorithm in such a way that it is guaranteed to output the top `lsize` candidates by score which additionally respect our ordering requirement – that is, the entries in a candidate should be in increasing order of size. This modification is done by adding an extra filtering step in each merge phase of OKEA which removes candidates that do not respect the ordering constraint.

At the end of this step, then, we obtain 6 lists $\mathcal{C}_i$, $1 \leq i \leq 6$, the entries of each list comprising `lsize` high-scoring candidates for one of the 6 arrays $\mathbf{f_i}$.`ones`, $\mathbf{f_i}$.`negOnes`, $1 \leq i \leq 3$.

4.2.2 Phase 2.

In the second phase of the attack, we present these 6 lists as inputs to the algorithm described in Section 4.1 above – that is, we perform a stack-based, depth-first search on the lists, regarding each list as giving a set of candidates on one of 6 chunks. Each complete candidate (on 6 chunks) now gives a candidate for $\mathbf{f_i}$.`ones`, $\mathbf{f_i}$.`negOnes`, $1 \leq i \leq 3$, these being tables of the indices where the component polynomials $\mathbf{f_1}, \mathbf{f_2}, \mathbf{f_3}$ have coefficients $+1$ and -1. We then apply the constraint that the pairs of tables be non-intersecting (applying it earlier in the process is not very efficient, since the probability of a collision of indices is small for the parameters of interest). If a candidate survives this filter, we can construct the full private key $\mathbf{f} = 1 + 3\mathbf{t}$ with $\mathbf{t} = \mathbf{f_1}\mathbf{f_2} + \mathbf{f_3}$ and test it for correctness.

As before, this second phase is amenable to parallelisation and to searching over restricted score intervals. Now the parameters are more akin to those studied in the prior work [17,16] – we have 6 chunks, and `lsize` candidates per chunk, with typical values for `lsize` in our experiments being 256, 512 and 1024.

5 Experimental Evaluation

5.1 Implementation

All of the algorithms discussed in this paper were implemented in Java. We choose Java for several reasons. First, the two implementations that we have studied in this paper were written in Java (as well as C). Second, the Java platform provides the Java Collections Framework to handle data structures, which reduces programming effort, and increases program speed and quality. Finally, the Java platform also easily supports concurrent programming, providing high-level concurrency APIs.

5.1.1 Parallelisation We made extensive use of parallelisation in our implementations, particularly for the stack-based, depth-first search that is at the core of both attacks. The first parallelisation method we used comes directly from [17] and involves splitting up the range of scores of interest into n disjoint, equal-sized sub-intervals. The second method involves splitting the list of candidates for the first chunk in our algorithm into m equal-sized sub-lists, and running the algorithm as a separate task for each sub-list, thereby constraining solutions from each task to begin with a chunk from the specified sublist for that task. These two approaches can be combined, to execute mn threads in parallel. Of course, as soon as one of the threads completes and successfully finds the private key, the others can all be aborted.

5.1.2 Search Intervals Defining appropriate search intervals on which to run our algorithms is important in guaranteeing the success of our attacks within a reasonable amount of running time. Recall that, given a collection of lists as input, each list containing candidate for chunks and their scores, our algorithms will consider all possible candidates with total scores in any specified interval $[a, b]$. We considered two distinct classes of search interval:

1. Class I intervals are the form $[\mu - W, \mu + W]$, where μ is the average score of the correct key and W is some real number that is tuned to the maximum running time available. Here μ can be computed empirically by generating many private keys, flipping their bits according to the error probabilities α, β, and then using the usual log-likelihood scoring function. Using such intervals capture the intuition that it might be better to examine key candidates that are situated around the average score, since these are more likely to

be correct. This of course violates the principle of the maximum likelihood approach.

2. Class II intervals are of the form $[\max - W, \max]$, where max is the maximum possible score and W is again a real number that can be tuned. Here, the value of max is easily calculated by summing across the highest scoring entries in each list. Searching in such intervals better matches the approach of maximum likelihood estimation.

5.1.3 Simulations

To simulate the performance of our algorithms, we generate a private key (according to some chosen format), flip its bits according to the error probabilities α, β, and then run our chosen algorithm with selected parallelisation parameters m, n and interval definition $[a, b]$. We refer to such a run attempting to recover a single private key as a *simulation*.

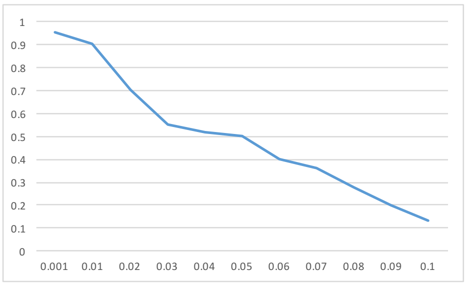
For our experiments, we ran our simulations on a machine with Intel Xeon CPU E5-2667 v2 cores running at 3.30GHz; we used up to 16 cores. In order to run our simulations concurrently, a pool of threads is initialised with a maximum number of threads given as a parameter. When a simulation is to be run and tested, it generates its various tasks according to the given parameters, each of which then is submitted to the main pool in order. After it has finished, a thread outputs either the recovered private key or null value (indicating failure to find the key) along with some statistics. Note that having a pool created with a defined number of threads helps to avoid exhausting and reusing computational resources, in contrast to creating a new thread per task.

5.2 Results for the `ntru-crypto` Java Implementation

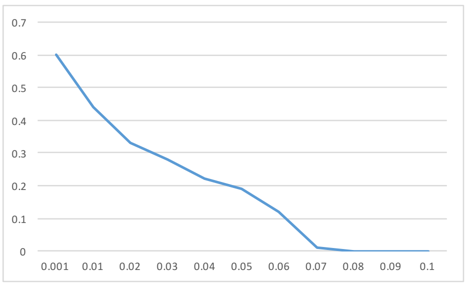
Here, we only considered Class II intervals, i.e. intervals of the form $[\max - W, \max]$. To calculate suitable values for W, we used random sampling from the set of possible candidates (by choosing chunks at random from each list) in order to estimate σ, the standard deviation of the candidate scores. We then set W as $r\sigma$ and experimented with different values of r, the idea being that larger values of r would correspond to bigger intervals, including more candidates and giving a higher chance of success at the cost of more computation. We used 2^{20} candidates in sampling to estimate σ.

After manual tuning, the number of tasks was set to 3, r was set to 0.01 and the number of subintervals m was set to 1. Hence in our experiments, searches were conducted over the interval $[\max - 0.01\sigma, \max]$ with 3 tasks.

Figure 1a shows the success rate of our attack for the `ees449ep1` parameters ($N = 449$, $\texttt{df} = 134$, $\texttt{dg} = 149$, $p = 3$, and $q = 2048$). Figure 1b shows the success rate for the `ees677ep1` parameters($N = 677$, $\texttt{df} = 157$, $\texttt{dg} = 225$, $p = 3$, and $q = 2048$).



(a) `ees449ep1` parameters.



(b) `ees667ep1` parameters.

Fig. 1: Success rate of our algorithm (y-axis) against β (x-axis) for a fixed $\alpha = 0.001$, using Class II intervals.

It can be seen from the two figures that the success rate is acceptably high for small values of β, but rapidly reduces as β is increased in size. Increasing the size of r (and therefore the search interval $[\max - r\sigma, \max]$) would improve the success rate at the cost of increased running time. For $r = 0.01$, we saw running times on the order of minutes to hours. There were a few simulations with very high running times; these were aborted after 1 day of computation. We observed this behaviour in particular for high values of β. In this case, the number of tasks, 3, the number of chunks, 400, and the nature itself of what was considered a suitable candidate (number of 1's and -1's) made it hard to predict the number of candidates in a given interval. So searching over a given interval is done somewhat "blindly", in the sense that searching over the interval $[\max - 0.01\sigma, \max]$ will not behave in a consistent manner in terms of the number of candidates found (and hence the running time needed).

5.3 Results for the `tbuktu` Java Implementation

Due to the additional structure of private keys compared to the `ntru-crypto` implementation, we focussed a greater experimental effort on the `tbuktu` Java implementation.

5.3.1 Counting candidates and estimating running times. Because of the nature of the log-likelihood function employed to calculate scores, each of the six lists $\mathcal{C}_i$ output by Phase I of the attack will have many repeated score values. This enables us to efficiently compute the *number* of candidates that Phase II of the attack will consider in any given interval $[a, b]$. To do this, we run a modified version of Phase II in which the lists $\mathcal{C}_i$ are replaced by "reduced" lists which eliminate chunk candidates having repeated score values, and include the counts (numbers) of such candidates along with their common score. By simultaneously computing the sums of scores and *products* of counts on these reduced lists, we can compute the total number of candidates that will have a given score, over all possible scores in any chosen interval.

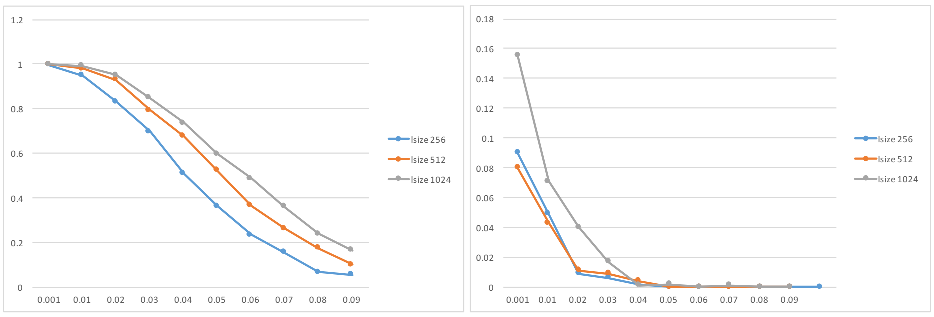
Because the size of each reduced list is less than 10 on average in our experiments (when `lisze` is up to 1024), we obtain a very efficient algorithm for counting the number of candidates in any given interval that our Phase II search algorithm would need to consider. We can combine this counting algorithm with the average time needed to generate and consider each candidate to get estimates for the total running time that our algorithm would encounter for a given choice of interval. We can then also compute the expected success probability and estimated running time (for the given number of candidates or given interval considered) without actually running the full Phase II search algorithm.

5.3.2 Parameters. The encryption parameters used for running the simulations are `APR2011_439_FAST` ($N = 439$, $p = 3$, $q = 2048$, `df1` $= 9$, `df2` $= 8$, `df3` $= 5$, `sparse` $=$ `true`, `fastP` $=$ `true` so that $\mathbf{t} = \mathbf{f_1}\mathbf{f_2} + \mathbf{f_3}$, and $\mathbf{f_1} \in T(9,9)$, $\mathbf{f_2} \in T(8,8)$, $\mathbf{f_3} \in T(5,5)$).

5.3.3 Results – complete enumeration. In our experiments, we set `lsize` to 2^r for Phase I, for $r = 8, 9, 10$. Thus six candidate lists each of size 2^r will be obtained from Phase I. Let p_i denote the probability that the correct candidate is actually found in the i-list; p_i will be a function of r. It follows that the probability that our Phase II algorithm outputs the correct private key when performing a *complete* enumeration over all 2^{6r} candidate keys is given by $p = \prod_{i=1}^{6} p_i$. This simple calculation gives us a way to perform simulations to estimate the expected success rate of our overall algorithm (Phase I and Phase II) without actually executing the expensive Phase II. We simply run many simulations of Phase I for the given value of `lsize` (each simulation generating a fresh private key and perturbing it according to α, β), and, after each simulation, test whether the correct chunks of the private keys are to be found in the lists.

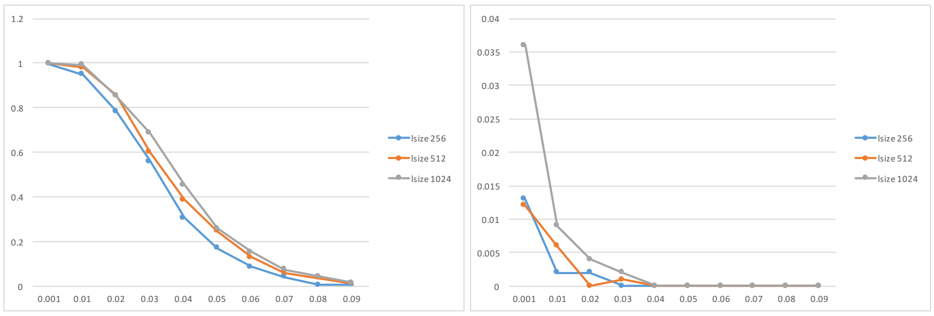
Figure 2 shows the success rates for complete enumeration for values of `lsize` $= 2^r$ for $r \in \{8, 9, 10\}$. As expected, the greater the value of `lsize`, the higher the success rate for a fixed α and β. Also note that when the noise is high (for example $\alpha = 0.09$ and $\beta = 0.09$), the success rate drops to zero. This is expected since it is likely that at least one chunk of the private key will not be included in the corresponding list coming out of Phase 1 when the noise levels are high, at which point Phase II inevitably fails.

Note that each data point in this figure (and all figures in section) were obtained using 100 simulations. Note that the running times for Phase 1 are very low in average (≤ 50 ms), since that phase consists of calling the OKEA for each one of the six lists with `lsize` in the set $\{256, 512, 1024\}$.



(a) $\alpha = 0.001$. (b) $\alpha = 0.09$

Fig. 2: Expected success rate for a full enumeration for $\alpha = 0.001, 0.09$. The y-axis represents the success rate, while the x-axis represents β.



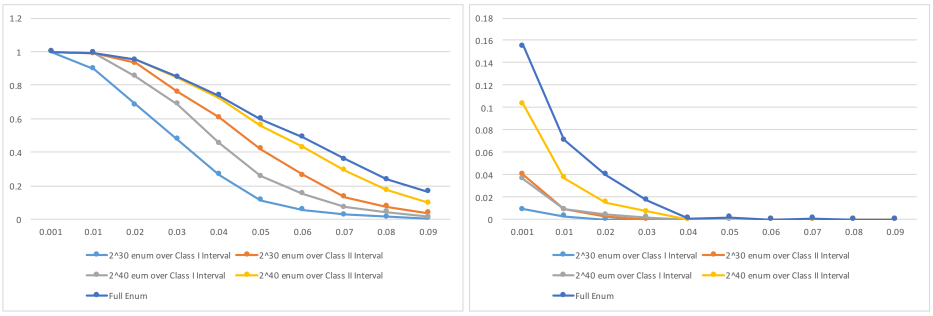
(a) $\alpha = 0.001$. (b) $\alpha = 0.09$

Fig. 3: Success rate for enumeration with 2^{40} keys over a Class I interval for $\alpha = 0.001, 0.09$, for different values of `lsize`. The y-axis represents the success rate, while the x-axis represents β.

5.3.4 Results – partial enumeration. Here, we exploit our counting algorithm to estimate success rates as a function of the total number of keys considered, K. Specifically, given a value K, and an interval type (I or II), we can set W accordingly so that the right number of keys will be considered. Since we can easily estimate the speed at which individual keys can be assessed, we can also use this approach to control the total running time of our algorithms.

Figure 3 shows how the success rate of our algorithm varies for different values of `lsize`, focussing on Class I intervals. We observe the same trends as for full enumeration, i.e. the greater is `lsize`, the higher is the success rate for a fixed α and β. Also, for larger values of (α, β), the success rate drops rapidly to zero.

Figure 4 shows the success rates for a complete enumeration and partial enumerations with 2^{30} keys and 2^{40} keys, for both Class I and Class II intervals. As expected, the success rate for a full enumeration is greater than for the partial



(a) $\alpha = 0.001$. (b) $\alpha = 0.09$

Fig. 4: Success rates for full enumeration, and partial enumeration with 2^{30} keys, 2^{40} keys for $\alpha = 0.001, 0.09$ and with `lsize` $= 1024$. The y-axis represents the success rate, while the x-axis represents β.

enumerations (but note that a full enumeration here would require the testing of up to 2^{60} keys, which may be a prohibitive cost). Note that the closest success rate to the success rate of a full enumeration is achieved with partial enumerations with 2^{40} keys over a Class II interval, and that partial enumerations over Class I intervals perform poorly, in the sense that their success rates are even dominated by the success rate of enumerations with 2^{30} keys over Class II intervals. The superiority of Class II invervals is in-line with the intuition that testing high log-likelihood candidates for correctness is better than examining average log-likelihood ones.

5.3.5 Running times. From our experiments, we find that our code is able to test up to 1200 candidates per millisecond per core during Phase 2. This value may vary in the range 700–1200 when there are multiples tasks running. The reason for this variation may be the cost associated with the Java virtual machine (particularly, its garbage collector). Using only a single core, an enumeration of 2^{30} (2^{40}) candidate keys will take about 14 minutes (10 days, respectively).

6 Conclusions

We have initiated the study of cold boot attacks for the NTRU public key encryption scheme, likely to be an important candidate in NIST's forthcoming post-quantum standardisation process. We have proposed algorithms for this problem, with particular emphasis on two existing NTRU implementations and two private key formats. We have experimented with the algorithms to explore their performance for a range of parameters, showing how algorithms developed for enumerating keys in side-channel attacks can be successfully applied to the problem.

Our attacks do not exploit the underlying mathematical structure of the NTRU scheme. It would be interesting to explore whether our techniques can be combined with other approaches, such as lattice-reduction, to further improve performance. We also focussed mainly on the two available Java implementations. It would be interesting to extend our work to consider the `ntru-encrypt` C implementation which uses packing techniques to reduce the private key size. This seems challenging because of the corresponding increase in information density for these formats; however, there is still some redundancy in the second of the two formats because of the ordering of indices. It is an interesting open problem to find ways to exploit this redundancy. Implementations of NTRU may also compute additional private key values, for example the inverse of $\mathbf{f} \bmod p$, in order to speed up decryption operations. Thus these extra values might be available in a cold boot attack. Finding methods for exploiting this additional redundancy would be of interest.

Acknowledgements

The research of Villanueva-Polanco was supported by Colciencias.

References

1. M. Albrecht and C. Cid. Cold boot key recovery by solving polynomial systems with noise. In J. Lopez and G. Tsudik, editors, *ACNS 11*, volume 6715 of *LNCS*, pages 57–72. Springer, Heidelberg, June 2011.
2. M. R. Albrecht, E. Orsini, K. G. Paterson, G. Peer, and N. P. Smart. Tightly secure ring-LWE based key encapsulation with short ciphertexts. Cryptology ePrint Archive, Report 2017/354, 2017. `http://eprint.iacr.org/2017/354`.
3. D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. NTRU prime. Cryptology ePrint Archive, Report 2016/461, 2016. `http://eprint.iacr.org/2016/461`.
4. A. Bogdanov, I. Kizhvatov, K. Manzoor, E. Tischhauser, and M. Witteman. Fast and memory-efficient key recovery in side-channel attacks. In O. Dunkelman and L. Keliher, editors, *SAC 2015*, volume 9566 of *LNCS*, pages 310–327. Springer, Heidelberg, Aug. 2016.
5. J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé. CRYSTALS – kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634, 2017. `http://eprint.iacr.org/2017/634`.
6. L. David and A. Wool. A bounded-space near-optimal key enumeration algorithm for multi-subkey side-channel attacks. In H. Handschuh, editor, *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, volume 10159 of *Lecture Notes in Computer Science*, pages 311–327. Springer, 2017.
7. J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold

boot attacks on encryption keys. In P. C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 45–60. USENIX Association, 2008.
8. J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.
9. W. Henecka, A. May, and A. Meurer. Correcting errors in RSA private keys. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 351–369. Springer, Heidelberg, Aug. 2010.
10. N. Heninger and H. Shacham. Reconstructing RSA private keys from random key bits. In S. Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 1–17. Springer, Heidelberg, Aug. 2009.
11. J. Hoffstein, N. Howgrave-Graham, J. Pipher, and W. Whyte. Practical lattice-based cryptography: NTRUEncrypt and NTRUSign. In P. Q. Nguyen and B. Vallée, editors, *The LLL Algorithm - Survey and Applications*, Information Security and Cryptography, pages 349–390. Springer, 2010.
12. J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In J. Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998.
13. Z. Huang and D. Lin. A new method for solving polynomial systems with noise over $\mathbb{F}_2$ and its applications in cold boot key recovery. In L. R. Knudsen and H. Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 16–33. Springer, Heidelberg, Aug. 2013.
14. A. A. Kamal and A. M. Youssef. Applications of SAT solvers to AES key recovery from decayed key schedule images. In R. Savola, M. Takesue, R. Falk, and M. Popescu, editors, *Fourth International Conference on Emerging Security Information Systems and Technologies, SECURWARE 2010, Venice, Italy, July 18-25, 2010*, pages 216–220. IEEE Computer Society, 2010.
15. H. T. Lee, H. Kim, Y.-J. Baek, and J. H. Cheon. Correcting errors in private keys obtained from cold boot attacks. In H. Kim, editor, *ICISC 11*, volume 7259 of *LNCS*, pages 74–87. Springer, Heidelberg, Nov. / Dec. 2012.
16. D. P. Martin, L. Mather, E. Oswald, and M. Stam. Characterisation and estimation of the key rank distribution in the context of side channel evaluations. In J. H. Cheon and T. Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 548–572. Springer, Heidelberg, Dec. 2016.
17. D. P. Martin, J. F. O'Connell, E. Oswald, and M. Stam. Counting keys in parallel after a side channel attack. In T. Iwata and J. H. Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 313–337. Springer, Heidelberg, Nov. / Dec. 2015.
18. K. G. Paterson, A. Polychroniadou, and D. L. Sibborn. A coding-theoretic approach to recovering noisy RSA keys. In X. Wang and K. Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 386–403. Springer, Heidelberg, Dec. 2012.
19. B. Poettering and D. L. Sibborn. Cold boot attacks in the discrete logarithm setting. In K. Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 449–465. Springer, Heidelberg, Apr. 2015.
20. N. Veyrat-Charvillon, B. Gérard, M. Renauld, and F.-X. Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. In L. R. Knudsen and H. Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 390–406. Springer, Heidelberg, Aug. 2013.