Timing attacks and the NTRU public-key cryptosystem

Gunter, S.P.

Award date:
2019

Timing Attacks and the NTRU Public-Key Cryptosystem

by

Stijn Gunter

supervised by

prof.dr. Tanja Lange
ir. Leon Groot Bruinderink

TU/e

A thesis submitted in partial fulfilment of the requirements for the degree of

Bachelor of Science

July 2019

Abstract

As we inch ever closer to a future in which quantum computers exist and are capable of breaking many of the cryptographic algorithms that are in use today, more and more research is being done in the field of post-quantum cryptography. One of the proposed post-quantum cryptosystems is NTRUEncrypt.

In this thesis we show that NTRUEncrypt's reference implementation is vulnerable to timing side-channel attacks. This vulnerability is exploited under two assumptions regarding the number of secret coefficients whose side-channel leaks are observed: one in which full information on all coefficients is assumed and one in which information on the first k coefficients is assumed. A mitigation using a constant-time implementation of the center-lift step in the decryption algorithm is proposed.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

This fall, 50 years will have passed since the first messages were sent over the ARPANET. In the decades since, the Internet – ARPANET's spiritual successor – has become an integral part of everyday life. As industries like the healthcare sector and financial-services sector become increasingly digitalized, more and more sensitive information is exchanged over the Internet. In order to ensure the integrity, confidentiality and authenticity of such data, cryptographic protocols are used.

Most of these protocols rely on public-key cryptography for key exchange and authentication. The Transport Layer Security protocol, for example, uses cryptosystems such as RSA and elliptic-cure cryptography [1]. Public-key cryptosystems rely on hard mathematical problems for security: elliptic-curve cryptography is based on the discrete-logarithm problem and RSA is based on the integer-factorization problem. Both problems are considered hard on classical computers – as of yet, no algorithms exist that can solve either in polynomial time.

Shor's algorithm [2], however, is capable of solving both in polynomial time – provided a quantum computer with a sufficient number of qubits and a low enough error rate exists. At the time of writing, the largest integer factored using Shor's algorithm is 21 [3]. While this may suggest that factoring the kinds of integers typically used by RSA is still a long way off, research on quantum computers is developing rapidly. Because of this, the National Institute of Standards and Technology started its Post-Quantum Cryptography Standardization project back in 2016 [4]. Most submissions are based on problems in lattices or coding theory, and are thought to be secure against both classical and quantum computers.

The NTRU public-key cryptosystem, NTRUEncrypt, is one such system. It was first published in 1996 [5] and is based on a hard problem on lattices. Two variations on the original scheme, NTRU (a merger of NTRUEncrypt and NTRU-HRSS-KEM [6]) and NTRUPrime, have made it to the second round of the NIST standardization project [7].

However, even theoretically secure schemes may suffer from vulnerable implementations. So-called side-channel attacks aim to extract secrets from systems by exploiting information that is leaked by the implementation. Paul Kocher's timing attack [8] was one of the first of its kind: by precisely measuring the amount of time required to perform certain cryptographic operations, he was able to recover RSA keys and discrete logarithms, among others. Other side-channel attacks abuse leaked electromagnetic radiation or variable power consumption.

In this thesis, we exploit a timing side-channel in the reference implementation[1] of NTRUEncrypt.

[1] Available at https://github.com/NTRUOpenSourceProject/NTRUEncrypt.

The structure of this thesis is as follows. Chapter 2 discusses the mathematical background for this thesis and provides an overview of commonly used notation. Chapter 3 introduces the NTRU public-key cryptosystem and its component algorithms, as well as some common variations on the initial scheme. Chapter 4 presents a timing attack on the reference implementation of NTRUEncrypt. Finally, Chapter 5 summarizes the content in this thesis, presents the conclusions of this thesis and outlines some opportunities for future research.

Chapter 2

Preliminaries

In this chapter, we consider the mathematical background for this thesis. Section 2.1 provides an overview of commonly used notation. From Section 2.2 onwards, we focus on algebraic structures called lattices, which play an instrumental role in quantifying the security of the NTRU public-key cryptosystem. Unless stated otherwise, the definitions in this chapter are taken from [9].

2.1 Notation

For a prime n and $p, q \in \mathbb{N}$ we let

$$R_n = \mathbb{Z}[x]/(x^n - 1) \tag{2.1}$$
$$R_{n,p} = (\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1) \tag{2.2}$$
$$R_{n,q} = (\mathbb{Z}/q\mathbb{Z})[x]/(x^n - 1) \tag{2.3}$$

be polynomial rings. If n is fixed, we will drop it from the notation. Elements of R are polynomials of degree less than n, and elements of R_p (R_q) are elements of R with coefficients reduced modulo p (q).

We will write elements $f \in R$ either as polynomials or as vectors, i.e.

$$f = \sum_{i=0}^{n-1} f_i x^i = (f_0, \ldots, f_{n-1}). \tag{2.4}$$

Addition of two elements $f, g \in R$ corresponds to the vector addition of their respective coefficient vectors. Multiplication of two elements $f, g \in R$ corresponds to the convolution product

$$f(x) \star g(x) = h(x), \quad h_k = \sum_{\substack{i+j \equiv k \\ (\mathrm{mod}\ n)}} f_i g_j. \tag{2.5}$$

We denote by $\mathcal{T}_n(d^+, d^-)$ the set of all ternary polynomials (polynomials with coefficients in $\{-1, 0, 1\}$) of degree less than n, having exactly d^+ coefficients equal to 1, exactly d^- coefficients equal to -1 and with all other coefficients equal to 0.

Given a set X we denote by $x \leftarrow_{\$} X$ that x is sampled uniformly from X. Similarly, given a probability distribution Y, we denote by $y \leftarrow Y$ that y is sampled from Y.

We will also make use of the rounding function $\lfloor \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ given by

$$\lfloor x \rceil = \lfloor x + 1/2 \rfloor \quad x \in \mathbb{R} \tag{2.6}$$

which rounds to the integer closest to x.

Finally, we let $\| \cdot \| : \mathbb{R}^n \to \mathbb{R}$ be the usual 2-norm on $\mathbb{R}^n$ given by

$$\|x\| = \sqrt{\sum_{i=1}^{n} x_i^2} \tag{2.7}$$

for $x \in \mathbb{R}^n$.

2.2 Lattices

We now turn to algebraic structures called lattices.

Definition 2.1 (Lattice)**.** Let $v_1, \ldots, v_n \in \mathbb{R}^m$ be a set of linearly independent vectors. The *lattice generated by* $v_1, \ldots, v_n$ is the set of linear combinations of $v_1, \ldots, v_n$ with coefficients in $\mathbb{Z}$:

$$L = \{a_1 v_1 + a_2 v_2 + \cdots + a_n v_n \mid a_1, a_2, \ldots, a_n \in \mathbb{Z}\}.$$

Any independent set of vectors that generates L is called a *basis* for L. We call n the *rank* of L and m the *dimension* of L.

We will only concern ourselves with a special class of lattices, the so-called integral lattices.

Definition 2.2 (Integral lattice)**.** An *integral lattice* is a lattice whose vectors have integer coordinates.

We can define determinants of lattices, and relate this concept to determinants of matrices. In order to do this, we first define the fundamental domain of a lattice.

Definition 2.3 (Fundamental domain)**.** Let L be a lattice of rank n and let $v_1, v_2, \ldots, v_n$ be a basis for L. The *fundamental domain* for L corresponding to this basis is the set

$$\mathcal{F}(v_1, \ldots, v_n) = \{t_1 v_1 + t_2 v_2 + \cdots + t_n v_n \mid 0 \leq t_i < 1\}. \tag{2.8}$$

Definition 2.4 (Lattice determinant)**.** Let L be a lattice and let $\mathcal{F}$ be a fundamental domain for L. Then the volume of $\mathcal{F}$ is called the *determinant* of L, and is denoted by $\det(L)$.

Given a basis $v_1, v_2, \ldots, v_n$ of a lattice L, we can define a matrix with the basis vectors as rows:

$$A = \begin{pmatrix} \text{——} & v_1 & \text{——} \\ \text{——} & v_2 & \text{——} \\ & \vdots & \\ \text{——} & v_n & \text{——} \end{pmatrix}. \tag{2.9}$$

Theorem 2.1. *Let L be a lattice of rank and dimension n with basis $v_1, v_2, \ldots, v_n$. Let A be the associated matrix as defined by (2.9). Then*

$$\det(L) = |\det(A)|. \tag{2.10}$$

See [9, p. 393] for a proof.

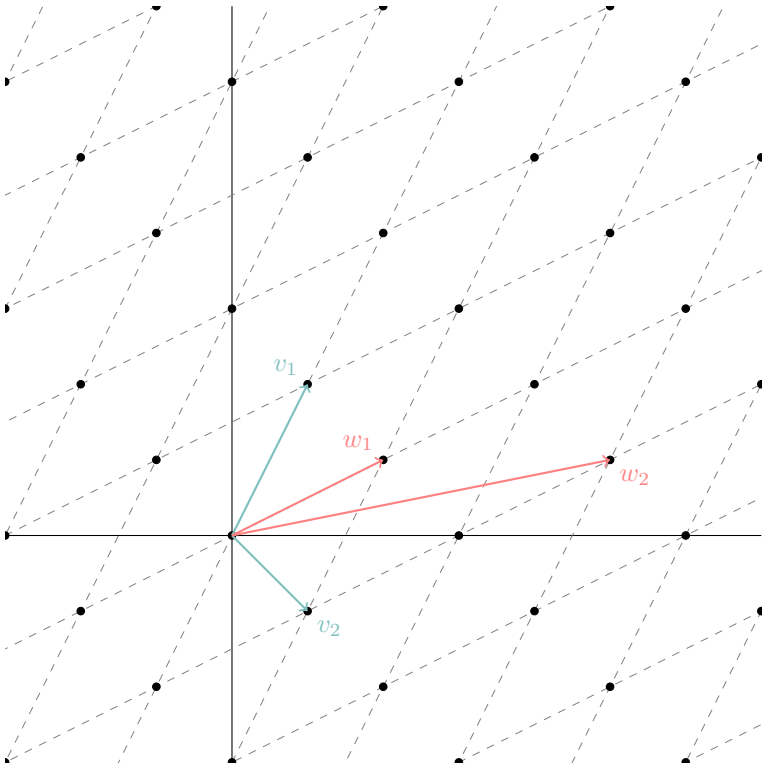

Figure 2.1: A lattice with two generating bases, $\{v_1, v_2\}$ and $\{w_1, w_2\}$.

Note that a lattice can have more than one basis. Figure 2.1, for example, depicts a lattice of dimension 2 and two generating bases. In fact, given a lattice L with a corresponding matrix of basis vectors A, we can left multiply A by a unimodular matrix $U \neq I$ to obtain a new matrix UA whose rows form a new basis for L. In particular, if a lattice has rank greater than 1, then the lattice has infinitely many bases. As $\det(U) = 1$, the lattice determinant is invariant under the chosen basis.

We can distinguish between so-called 'good' bases and 'bad' bases. Generally, we consider a basis 'good' if it is (nearly) orthogonal and 'bad' if it is not.

In this sense, $\{v_1, v_2\}$ is an example of a 'good' basis for the lattice in Figure 2.1 while $\{w_1, w_2\}$ is an example of a 'bad' basis.

It turns out that there exist lattice problems which are easy to solve when you have a 'good' basis, but much harder to solve when you have a 'bad' basis. In the next section, we will describe one such problem.

2.3 The Shortest Vector Problem

Problems that have an asymmetric computational cost – that is, operations or computations that are easy to perform one way, but computationally difficult to invert – have often been used as a basis for cryptosystems. The RSA cryptosystem, for example, derives its security from the assumption that it is much harder to factor (large) numbers into primes than it is to multiply them. Similarly, the ElGamal encryption system relies on the assumed difficulty of computing the discrete logarithm, while exponentiation is relatively easy.

Lattice problems that are easy in certain circumstances but very difficult in other circumstances could therefore prove interesting as well. Two such problems are the Shortest Vector Problem (SVP) and its associated approximate variant (apprSVP or approxSVP).

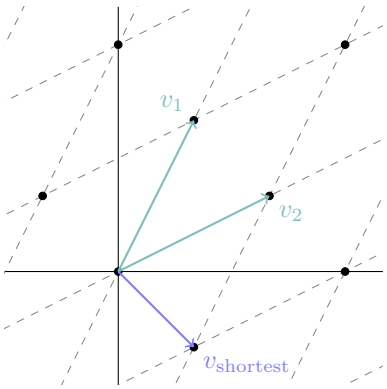

Figure 2.2: A lattice with basis $\{v_1, v_2\}$ and shortest vector v_{shortest}.

Definition 2.5 (Shortest Vector Problem)**.** Find a shortest non-zero vector in a lattice L, i.e. find a non-zero vector $v \in L$ such that $\|v\|$ is minimal.

Definition 2.6 (Approximate Shortest Vector Problem)**.** Let $\psi(n)$ be a function of n. In a lattice L of dimension n, find a non-zero vector that is no more than $\psi(n)$ times longer than the shortest non-zero vector. In other words, if v_{shortest} is a shortest non-zero vector, find a non-zero vector $v \in L$ satisfying

$$\|v\| \leq \psi(n)\|v_{\text{shortest}}\|.$$

There exist various techniques to estimate the length of the shortest vector. One such technique is the Gaussian heuristic.

Definition 2.7 (Gaussian heuristic)**.** Let L be a lattice of dimension n. The *Gaussian expected shortest length* is

$$\sigma(L) = \sqrt{\frac{n}{2\pi e}}\,(\det(L))^{\frac{1}{n}}. \tag{2.11}$$

The *Gaussian heuristic* states that for a fixed $\epsilon > 0$ and sufficiently large n, a shortest non-zero vector will satisfy

$$(1-\epsilon)\sigma(L) \leq \|v_{\text{shortest}}\| \leq (1+\epsilon)\sigma(L). \tag{2.12}$$

In essence, this means that we can approximate the length of a shortest non-zero vector by $\sigma(L)$. Because the determinant of a lattice is invariant under the choice of basis, the Gaussian heuristic is too. This ensures the Gaussian heuristic is meaningful.

Now if we have a 'good' basis for L, i.e. if the basis vectors are nearly orthogonal, shortest vectors are comparatively easy to find. If we do not have such a basis, we can still attempt to solve these problems by first finding a better basis.

2.4 The Lenstra-Lenstra-Lovász algorithm

The interesting question now is if it is possible to find a 'good' basis $\{v_1, v_2, \ldots, v_n\}$ given a 'bad' basis $\{w_1, w_2, \ldots, w_n\}$. As it turns out, this is possible. The process of turning a 'bad' basis into a 'good' basis is called *lattice basis reduction*. While there exist various algorithms capable of reducing a lattice basis, most require at least exponential time in the dimension of the lattice. This makes the process infeasible for lattices of larger dimension.

In this section, we will discuss the Lenstra-Lenstra-Lovász algorithm, or the LLL algorithm in short. It was first introduced in [10], and produces a so-called LLL reduced basis.

Definition 2.8 (LLL reduced basis)**.** Let $\mathcal{B} = \{v_1, v_2, \ldots, v_n\}$ be a basis for a lattice L and let $\mathcal{B}^* = \{v_1^*, v_2^*, \ldots, v_n^*\}$ be the associated Gram-Schmidt orthogonal basis. Let $\delta \in \left[\frac{1}{4}, 1\right]$. The basis $\mathcal{B}$ is said to be *LLL reduced* if it satisfies the following two conditions:

(i) **Size Condition**

$$|\mu_{i,j}| = \frac{v_i \cdot v_j^*}{\|v_j^*\|^2} \leq \frac{1}{2} \quad \text{for all } 1 \leq j < i \leq n. \tag{2.13}$$

(ii) **Lovász Condition**

$$\|v_i^*\|^2 \geq \left(\delta - \mu_{i,i-1}^2\right) \|v_{i-1}^*\|^2 \quad \text{for all } 1 < i \leq n. \tag{2.14}$$

A consequence of these properties is the following.

Theorem 2.2. *Let L be a lattice of dimension n. The initial vector in an LLL reduced basis $\{v_1, v_2, \ldots, v_n\}$ satisfies*

$$\|v_1\| \leq 2^{(n-1)/2} \min_{\substack{v \in L \\ v \neq 0}} \|v\|.$$

A proof is provided in [9, p. 441], for example.

In essence, this theorem states that LLL solves apprSVP for $\psi(n) = 2^{(n-1)/2}$. As LLL runs in polynomial time, it might appear as if apprSVP is 'solved'. However, as n increases, the 'approximation factor' increases exponentially and the vector found by LLL can become very large. There do exist other algorithms that can find sufficiently small vectors, such as BKZ, but these have problems of their own. BKZ, for instance, has a subroutine that obtains a short basis of a lattice of dimension $2 < \beta < n$, which is an expensive operation for larger β. If $\psi(n)$ is small, BKZ requires a large β to find sufficiently small vectors, hence BKZ is still not fast enough.

Algorithm 1 provides a brief description of the LLL lattice reduction algorithm. Note that the Gram-Schmidt orthogonal set of vectors $v_1^*, v_2^*, \ldots, v_n^*$ and their associated quantities

$$\mu_{i,j} = \frac{\left(v_i \cdot v_j^*\right)}{\|v_j^*\|^2} \tag{2.15}$$

are recomputed continuously.

Algorithm 1 The LLL lattice basis reduction algorithm.

Require: $\delta \in \left[\frac{1}{4}, 1\right]$

procedure LLL-REDUCE$(v_1, v_2, \ldots, v_n)$
  $k \leftarrow 2$
  $v_1^* \leftarrow v_1$
  while $k \leq n$ **do**
    for $j = k-1, k-2, \ldots, 1$ **do**
      $v_k \leftarrow v_k - \lfloor \mu_{k,j} \rceil \, v_j$ ▷ Size reduction step
    if $\|v_k^*\|^2 \geq \left(\delta - \mu_{k,k-1}^2\right) \|v_{k-1}^*\|^2$ **then** ▷ Lovász Condition
      $k \leftarrow k + 1$
    else
      swap v_{k-1}, v_k
      $k \leftarrow \max(k-1, 2)$
  return $\{v_1, v_2, \ldots, v_n\}$ ▷ This basis is LLL reduced

Chapter 3

The NTRU public-key cryptosystem

In this chapter, we describe the NTRU public-key cryptosystem, which was first introduced in [5]. Section 3.1 provides an overview of public-key cryptography in general. Sections 3.2, 3.3 and 3.4 describe the various component algorithms that make up NTRUEncrypt. Finally, in Section 3.5 we discuss the system's underlying mathematical problem and its resistance to attacks.

3.1 Public-key cryptography

Public-key encryption systems, sometimes called *asymmetric ciphers*, are schemes in which the sender and received have unequal (asymmetric) knowledge and abilities.

The concept is formalized as follows. For spaces of keys $\mathcal{K}$, plaintexts $\mathcal{M}$ and ciphertexts $\mathcal{C}$ we have that every $k \in \mathcal{K}$ is a pair $k = (k_{\text{priv}}, k_{\text{pub}})$. For each $k \in \mathcal{K}$ we have an encryption function

$$e_{k_{\text{pub}}} : \mathcal{M} \to \mathcal{C} \tag{3.1}$$

and a decryption function

$$d_{k_{\text{priv}}} : \mathcal{C} \to \mathcal{M} \tag{3.2}$$

which must satisfy (with high probability) the property

$$d_{k_{\text{priv}}}\left(e_{k_{\text{pub}}}(m)\right) = m \tag{3.3}$$

for all $m \in \mathcal{M}$.

Assume Alice wants to send a message $m \in \mathcal{M}$ to Bob. Bob has a keypair $k = (k_{\text{pub}}, k_{\text{priv}}) \in \mathcal{K}$, of which he published k_{pub}. Alice uses k_{pub} to compute $c = e_{k_{\text{pub}}}(m)$ and sends this ciphertext to Bob. Bob can then compute $m' = d_{k_{\text{priv}}}(c)$ using his private key. If the property (3.3) holds, we have $m' = m$ and Bob has successfully retrieved Alice's plaintext. This process is illustrated in Figure 3.1.

The public key k_{pub} and the ciphertext c are sent over insecure channels. This means that any third party, often called Eve, can intercept these values. Thus, the security of public-key encryption systems hinges on two assumptions, which ensure so-called OW-CPA security:

- Given a ciphertext $c \in \mathcal{C}$, Eve cannot obtain the corresponding plaintext m in any reasonable amount of time, unless she has the private key k_{priv}.

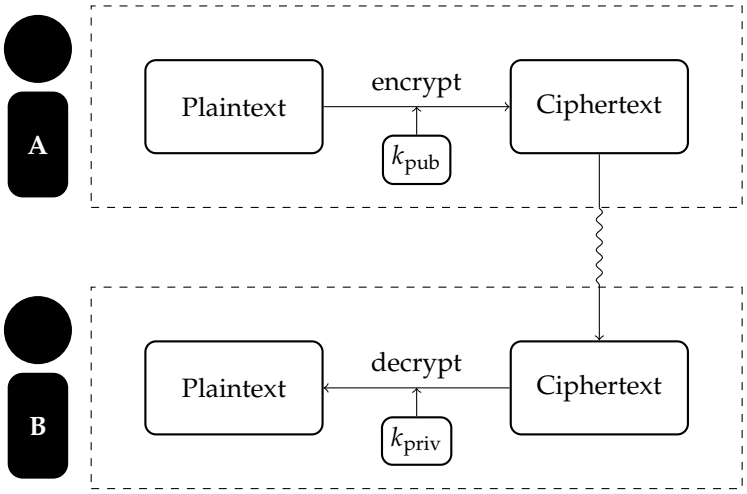

Figure 3.1: Overview of public-key encryption and decryption.

- Given Bob's public key k_{pub}, Eve cannot obtain Bob's private key k_{priv} in any reasonable amount of time.

3.2 Key generation

Keys in NTRUEncrypt consist of elements of R, R_p and R_q. For a set of parameters (n, p, q, d), with n prime and $\gcd(n, p) = \gcd(p, q) = 1$, a key pair consisting of a public key $h(x)$ and a private key $\big(f(x), F_p(x)\big)$ is generated as in Algorithm 2.

Depending on the required security level, the NTRUEncrypt NIST submission recommends parameter sets $(443, 3, 2048, 143)$ or $(743, 3, 2048, 247)$ [11].

Algorithm 2 The NTRU key generation algorithm.

Require: n prime, n and p coprime, p and q coprime
1: **procedure** NTRU-Keygen(n, p, q, d)
2: $g(x) \leftarrow_{\$} \mathcal{T}_n(d, d)$
3: **repeat**
4: $f(x) \leftarrow_{\$} \mathcal{T}_n(d+1, d)$
5: **until** $f(x)$ invertible in R_p and R_q
6: $F_p(x) = f(x)^{-1} \in R_p$
7: $h(x) = f(x)^{-1} \star g(x) \in R_q$
8: **return** private key $\big(f(x), F_p(x)\big)$, public key $h(x)$

We will illustrate this using a running example. As in Section 3.1, we consider a sender, Alice, and a receiver, Bob. Both agree on a parameter set $(n, p, q, d) = (7, 3, 32, 3)$.

Bob then generates a key pair. He starts by choosing

$$g(x) = x^5 - x^3 + x - 1 \in \mathcal{T}_7(2, 2), \tag{3.4}$$
$$f(x) = x^6 + x^5 - x^4 + x^2 - x \in \mathcal{T}_7(3, 2). \tag{3.5}$$

He then computes the inverses of $f(x)$

$$F_p(x) = f(x)^{-1} = x^6 + x^5 + x^4 + x^3 + 2x + 1 \in R_3, \tag{3.6}$$
$$F_q(x) = f(x)^{-1} = 8x^6 + 12x^5 + 26x^4 + 3x^3 + 31x^2 + x + 16 \in R_{32}. \tag{3.7}$$

Finally, he computes his public key

$$h(x) = F_q(x) \star g(x) = 2x^6 + 31x^5 + 16x^4 + 24x^3 + 20x^2 + 6x + 29 \in R_{32}. \tag{3.8}$$

Bob publishes $h(x)$ and keeps $\big(f(x), F_p(x)\big)$ private.

3.2.1 Linearly transformed coefficients

Often, elements of $\mathcal{T}_n(d+1, d)$ do not form the key $f(x)$ directly. Instead, a linear transformation is applied to the key in order to simplify the decryption process.

For a $f'(x) \in \mathcal{T}_n(d+1, d)$ we let

$$f(x) = pf'(x) + 1 \tag{3.9}$$

and compute $h(x)$ as in the original algorithm. Note that after this transformation, $F_p(x) = f^{-1}(x) = 1 \in R_p$, so f is guaranteed to have an inverse in R_p and multiplication by $F_p(x)$ becomes trivial, which leads to more efficient implementations. Furthermore, as $F_p(x)$ no longer needs to be stored, the keysize is reduced.

Algorithm 3 summarizes the complete procedure.

Algorithm 3 The NTRU key generation algorithm using a linear transformation.

Require: n prime, n and p coprime, p and q coprime

procedure NTRU-KEYGEN(n, p, q, d)
  $g(x) \leftarrow_\$ \mathcal{T}_n(d, d)$
  repeat
    $f'(x) \leftarrow_\$ \mathcal{T}_n(d+1, d)$
    $f(x) = pf'(x) + 1$
  until $f(x)$ invertible in R_q
  $F_p(x) = 1 \in R_p$
  $h(x) = f(x)^{-1} \star g(x) \in R_q$
  return private key $(f(x), 1)$, public key $h(x)$

3.2.2 Normally distributed coefficients

There exist other variants of the NTRUEncrypt system. The Stehlé-Steinfeld variant, for example, makes use of a discrete Gaussian distribution to generate a key pair [12]. These and other modifications allow for a provably secure variant of NTRUEncrypt, but come at the cost of decreased performance.

As the name suggests, a discrete Gaussian distribution allows sampling integers proportionally to a Gaussian function. It is more precisely defined as follows.

Definition 3.1 (Discrete Gaussian distribution)**.** Let $c, \sigma \in \mathbb{R}$, $\sigma \neq 0$,

$$S_{c,\sigma} = \sum_{k=-\infty}^{\infty} e^{-(k-c)^2/\left(2\sigma^2\right)}$$

and let $D_{c,\sigma}$ be a random variable on $\mathbb{Z}$ such that, for $x \in \mathbb{Z}$,

$$\mathbb{P}(D_{c,\sigma} = x) = \frac{e^{-(x-c)^2/(2\sigma^2)}}{S_{c,\sigma}}$$

then $D_{c,\sigma}$ denotes the *discrete Gaussian distribution* with centre c and width σ. If $c = 0$, we let $D_\sigma = D_{c,\sigma}$.

Algorithm 4 demonstrates how the NTRU-KEYGEN procedure can be adapted to use normally distributed coefficients.

Algorithm 4 The NTRU key generation algorithm using a discrete Gaussian distribution.

Require: n prime, n and p coprime, p and q coprime
procedure NTRU-KEYGEN(n, p, q, σ)
  for $i = 0, \ldots, n-1$ **do**
    $g_i \leftarrow D_\sigma$
  $g(x) = \sum_{i=0}^{n-1} g_i x^i \in R$
  repeat
    for $i = 0, \ldots, n-1$ **do**
      $f_i \leftarrow D_\sigma$
    $f(x) = \sum_{i=0}^{n-1} f_i x^i \in R$
  until $f(x)$ invertible in R_p and R_q
  $F_p(x) = f(x)^{-1} \in R_p$
  $h(x) = f(x)^{-1} \star g(x) \in R_q$
  return private key $\big(f(x), F_p(x)\big)$, public key $h(x)$

3.3 Encryption

The message space $\mathcal{M}$ of NTRUENCRYPT consists of elements of R which are center-lifted elements of R_p, i.e. elements $m \in R$ with coefficients $-\frac{p}{2} < m_i \leq \frac{p}{2}$. Elements $m \in M$ are encrypted using a public key $h \in R_q$ as in Algorithm 5.

Algorithm 5 The NTRU encryption algorithm.

Require: $m(x) \in R$ such that $-\frac{1}{2}p < m_i \leq \frac{1}{2}p$, $h(x)$ a valid public key
procedure NTRU-ENCRYPT($m(x)$, $h(x)$)
  $r(x) \leftarrow_\$ \mathcal{T}_n(d, d)$
  $c(x) \equiv p \cdot h(x) \star r(x) + m(x) \in R_q$
  return ciphertext $c(x)$

The resulting ciphertext c is an element of R_q, hence NTRUENCRYPT has ciphertext-space $\mathcal{C} = R_q$.

Continuing the running example from the previous section, Alice chooses

$$m(x) = x^5 - x^3 \in R, \tag{3.10}$$
$$r(x) = x^6 + x^2 - x - 1 \in \mathcal{T}_7(2, 2). \tag{3.11}$$

She then uses Bob's public key (3.8) to compute the ciphertext

$$\begin{aligned} c(x) &= 3 \cdot h(x) \star r(x) + m(x) \\ &= 3 \cdot (2x^6 + 31x^5 + 16x^4 + 24x^3 + 20x^2 + 6x + 29) \star (x^6 + x^2 - x - 1) + (x^5 - x^3) \\ &= 4x^6 + 2x^5 + x^4 + 29x^3 + 17x^2 + 25x + 18 \in R_{32} \end{aligned} \tag{3.12}$$

and sends it on to Bob.

3.4 Decryption

Ciphertexts $c(x) \in \mathcal{C} = R_q$ are decrypted as in Algorithm 6. Note that in the variant described in Section 3.2.1, step 5 of this algorithm becomes trivial.

Algorithm 6 The NTRU decryption algorithm.

Require: $c(x) \in R_q$ is a valid ciphertext, $\big(f(x), F_p(x)\big)$ is a valid private key

1: **procedure** NTRU-DECRYPT($c(x)$, $f(x)$, $F_p(x)$)
2: $a(x) \equiv f(x) \star c(x) \in R_q$
3: center-lift $a(x)$ to an element of R
4: reduce $a(x)$ modulo p
5: $m(x) \equiv F_p(x) \star a(x) \in R_p$
6: **return** plaintext $m(x)$

Continuing the running example from the previous section, Bob uses his private key (3.4) and the ciphertext he receives from Alice (3.12) to compute

$$\begin{aligned} a(x) &\equiv f(x) \star c(x) \\ &\equiv (x^6 + x^5 - x^4 + x^2 - x) \star (4x^6 + 2x^5 + x^4 + 29x^3 + 17x^2 + 25x + 18) \\ &\equiv 25x^6 + 25x^5 + 8x^4 + 7x^3 + 21x^2 + 31x + 11 \in R_{32} \end{aligned} \tag{3.13}$$

which he center-lifts to obtain

$$a(x) \equiv -7x^6 - 7x^5 + 8x^4 + 7x^3 - 11x^2 - x + 11 \in R. \tag{3.14}$$

Bob now reduces (3.14) modulo $p = 3$

$$a(x) \equiv 2x^6 + 2x^5 + 2x^4 + x^3 + x^2 + 2x + 2 \in R_3 \tag{3.15}$$

and uses his private key (3.6) to obtain

$$\begin{aligned} m(x) &\equiv F_p(x) \star a(x) \\ &\equiv (x^6 + x^5 + x^4 + x^3 + 2x + 1) \star (2x^6 + 2x^5 + 2x^4 + x^3 + x^2 + 2x + 2) \\ &\equiv x^5 - x^3 \in R_3, \end{aligned} \tag{3.16}$$

which is equal to Alice's plaintext (3.10).

3.5 The NTRU Key Recovery Problem

As alluded to in previous chapters, the security assumption described in Section 3.1 is guaranteed by an underlying mathematical problem that is believed to be hard.

We will begin by stating the security assumption in terms of the NTRUENCRYPT keyspace.

Definition 3.2 (NTRU Key Recovery Problem [9]). Given $h(x)$, find ternary polynomials $f(x)$ and $g(x)$ satisfying

$$f(x) \star h(x) = g(x) \in R_q.$$

In essence, an attacker Eve needs to find an element $f(x) \in \mathcal{T}_n(d+1, d)$ such that $f(x) \star h(x)$ is a ternary polynomial. Note that this $f(x)$ is not unique: if $f(x)$ is a solution, then so is $x^k \star f(x)$ for $0 \leq k < n$ (this will produce rotated, but still ternary, polynomials $g(x)$).

n	d	$\lvert\mathcal{T}_n(d+1,d)\rvert \;/\; n$
443	143	2^{684}
743	247	2^{1158}

Table 3.1: Approximate number of tries before finding a decryption key.

3.5.1 Brute-force resistance

The most obvious attack on NTRUEncrypt would be to simply try all possible ternary polynomials $f(x) \in \mathcal{T}_n(d+1,d)$. In order to estimate how resistant the cryptosystem is against such attacks, we compute the number of possible keys $f(x)$ – the size of the set $\mathcal{T}_n(d+1,d)$.

Elements of $\mathcal{T}_n(d_1,d_2)$ can be modelled by first choosing d_1 out of n indices i where $f_i = 1$, and then choosing d_2 out of $n-d_1$ indices i where $f_i = -1$. Then

$$|\mathcal{T}_n(d_1,d_2)| = \binom{n}{d_1}\binom{n-d_1}{d_2} = \frac{n!}{d_1!d_2!(n-d_1-d_2)!} \tag{3.17}$$

However, as the decryption key has n possible rotations (which will also function as decryption keys, but will simply produce rotated plaintexts), the effective number of unique keys becomes $|\mathcal{T}_n(d+1,d)| \;/\; n$.

Taking the parameter set $(n,p,q,d) = (443,3,2048,146)$ from Section 3.2, we expect that Eve will need to try approximately

$$\frac{|\mathcal{T}_{443}(147,146)|}{443} = \frac{1}{443}\binom{443}{147}\binom{296}{146} \approx 2^{684} \tag{3.18}$$

candidates for (rotations of) $f(x)$.

Table 3.1 summarises the expected number of tries for all parameter sets mentioned in Section 3.2.

3.5.2 Reducing to apprSVP

Let

$$h = \sum_{i=0}^{n-1} h_i x^i \tag{3.19}$$

be a NTRUEncrypt public key. Then the rows of the matrix

$$M_h^{\text{NTRU}} = \left(\begin{array}{cccc|cccc} 1 & 0 & \dots & 0 & h_0 & h_1 & \dots & h_{n-1} \\ 0 & 1 & \dots & 0 & h_{n-1} & h_0 & \dots & h_{n-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & h_1 & h_2 & \dots & h_0 \\ \hline 0 & 0 & \dots & 0 & q & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & q & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & q \end{array}\right). \tag{3.20}$$

are basis vectors of the so-called *NTRU lattice* L_h^{NTRU}.

If we represent each pair of polynomials $a(x), b(x) \in R$ given by

$$a(x) = \sum_{i=0}^{n-1} a_i x^i \tag{3.21}$$

$$b(x) = \sum_{i=0}^{n-1} b_i x^i \tag{3.22}$$

as a $2n$-dimensional vector in $\mathbb{Z}^{2n}$

$$(a, b) = (a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{n-1}), \tag{3.23}$$

then the vector (f, g) is in the lattice L_h^{NTRU}. We can see this by choosing $u(x) \in R$ such that

$$f(x) \star h(x) = g(x) + qu(x). \tag{3.24}$$

If we now compute $(f, -u)M_h^{\text{NTRU}}$, then as the leftmost n columns of the NTRU matrix consist of the identity matrix above the zero matrix, we obtain $(f, \cdot)$. The rightmost n columns consist of rotations of h, matching the convolution multiplication by $h(x)$, above qI where I is the identity matrix. This means we compute $f(x) \star h(x) - qu(x) = g(x)$. Combined with (3.24) we obtain (f, g).

Furthermore, we can approximate the length of this vector by

$$\|(f, g)\| \approx \sqrt{4d} \tag{3.25}$$

as f and g both have approximately $2d$ non-zero coefficients, and all other coefficients are equal to 0. If we take $d \approx \lfloor \frac{n}{3} \rfloor$, then this becomes

$$\|(f, g)\| \approx 1.155\sqrt{n}. \tag{3.26}$$

Further, we note that the matrix M_h^{NTRU} is upper triangular. Consequently, the determinant can be obtained by multiplying the diagonal entries, hence

$$\det\left(L_h^{\text{NTRU}}\right) = \left|\det(M_h^{\text{NTRU}})\right| = q^n \tag{3.27}$$

by Theorem 2.1. By the Gaussian heuristic (Definition 2.7) and the Gaussian expected shortest length σ, we obtain

$$\begin{aligned} \|v_{\text{shortest}}\| &\approx \sigma\left(L_h^{\text{NTRU}}\right) \\ &= \sqrt{\frac{2n}{2\pi e}}\left(\det\left(L_h^{\text{NTRU}}\right)\right)^{\frac{1}{2n}} \\ &= \sqrt{\frac{n}{\pi e}}\sqrt{\left(\det\left(L_h^{\text{NTRU}}\right)\right)^{n}} \\ &= \sqrt{\frac{qn}{\pi e}} \\ &\approx 15.486\sqrt{n} \end{aligned} \tag{3.28}$$

for $q = 2048$.

By comparing (3.26) and (3.28), we conclude that (f, g) is very likely to be a shortest vector in the NTRU matrix. Therefore, we say that the NTRU Key Recovery Problem approximately

reduces to the approximate Shortest Vector Problem. As recovering a message from a ciphertext is about as expensive as obtaining the private key itself, an attacker is better off going for the private key. Hence, NTRUEncrypt is considered secure due to the hardness of apprSVP.

Chapter 4

A timing attack on NTRUEncrypt

In this chapter, we describe a timing attack on the reference implementation of NTRUEncrypt. Section 4.1 outlines the vulnerability, which is exploited in Sections 4.2 and 4.3. We end the chapter by discussing how this attack could be mitigated in Section 4.4.

4.1 The side channel

Line 3 of Algorithm 6 (NTRU decryption) requires a polynomial $a(x) \in R_q$ to be center-lifted to an element in R with coefficients in $[-q/2, q/2)$. Algorithm 7 illustrates how this operation has been implemented in NTRUEncrypt's reference implementation[1].

Algorithm 7 Center-lifting a polynomial to R.

```
1: procedure CENTER-LIFT(a(x))
2:     for i = 0, ..., n − 1 do
3:         if a_i ≥ q / 2 then
4:             a_i = a_i − q
5:     return a(x)                    ▷ a(x) has been lifted, i.e. a_i ∈ [−q / 2, q / 2)
```

This implementation branches based on the coefficients of the polynomial $a(x)$. We recall that this polynomial is defined in the decryption step as

$$a(x) \equiv c(x) \star f(x) \in R_q \tag{4.1}$$

so its coefficients depend on the values of $c(x)$ and $f(x)$. As an attacker is able to send arbitrary ciphertexts $c(x)$ to a receiver, he can choose a ciphertext such that the coefficients of $a(x)$ depend entirely on the secret key $f(x)$. Note that this holds even for CCA-secure implementations, because the leakage happens before the validity of $c(x)$ is tested. Therefore, the implementation of the ciphertext branches based on the coefficients of the secret key. Consequently, the running time of the decryption step depends on those coefficients. An attacker can exploit this by carefully measuring how long a decryption operation takes, and deducing from this the coefficients of the secret key.

[1] https://github.com/NTRUOpenSourceProject/NTRUEncrypt/blob/c8ec1d7c81671357ca457ebd95eeb9818115c10f/src/ntru_crypto_ntru_encrypt.c#L641-L651

4.2 Exploiting perfect information

In this section, we assume that we can time the decryption operation with sufficient precision to recover the side-channel information for each a_i, $i = 0, \ldots, n-1$. In other words, we assume that for all a_i we can find out if $a_i \geq \frac{q}{2}$.

We further assume the following:

- The parameter set (n, p, q, d) has $p = 3$, $q = 2^\ell$, $\ell \geq 3$. All parameter sets defined in the reference implementation have $p = 3$ and $q = 2048 \geq 2^3$, so this requirement is satisfied.
- The implementation represents ciphertexts with coefficients in $[0, q)$. The reference implementation represents a ciphertext as an array of 16-bit unsigned integers, so this requirement is satisfied.
- The implementation represents private keys with coefficients in $\{-1, 0, 1\}$. The reference implementation represents a private key as two lists, one of indices where the coefficient is equal to 1 and one of indices where the coefficient is equal to -1, so this requirement is satisfied.
- The implementation computes $a(x) = c(x) \star f(x)$ in the ring R_q, i.e. the coefficients of $a(x)$ are reduced to elements of $[0, q)$. The reference implementation performs this multiplication in R_q, so this requirement is satisfied.

Next, we need to model the side-channel. We do this by checking the branching condition ourselves and simply returning a vector of booleans indicating whether we observed a branch 'hit' ($a_i \geq \frac{q}{2}$) or a 'miss' ($a_i < \frac{q}{2}$). Algorithm 8 demonstrates a pseudocode implementation of this model.

Algorithm 8 Observing leaked side-channel information.

```
1: procedure OBSERVE(f(x), c(x))
2:     a(x) ≡ f(x) ⋆ c(x) ∈ R_q
3:     v ∈ 𝔹^n                          ▷ every v_i is initialized to false
4:     for i = 0, ..., n − 1 do
5:         if a_i ≥ q / 2 then
6:             v_i = true
7:     return v
```

We are now ready to describe the attack itself. Let

$$c_1(x) = \frac{1}{8}q, \tag{4.2}$$

$$c_2(x) = \frac{3}{4}q, \tag{4.3}$$

be two ciphertexts in R_q. As we require $q \geq 2^3$, these ciphertexts are well-defined.

The coefficients of the resulting polynomials $a_1(x) = c_1(x) \star f(x)$ and $a_2(x) = c_2(x) \star f(x)$ are given by

$$(c_1 \star f)_i = \frac{1}{8}qf_i, \tag{4.4}$$

$$(c_2 \star f)_i = \frac{3}{4}qf_i, \tag{4.5}$$

for $i = 0, \ldots, n-1$.

We know that $f_i \in \{-1, 0, 1\}$ for $i = 0, \ldots, n-1$, and we consider each case separately.

(i) Case $f_i = 1$. Then $(c_1 \star f)_i = \frac{q}{8} < \frac{q}{2}$ and $(c_2 \star f)_i = \frac{3q}{4} \geq \frac{q}{2}$, so the side-channel reports a 'miss' on the first ciphertext and a 'hit' on the second.

(ii) Case $f_i = 0$. Then $(c_1 \star f)_i = (c_2 \star f)_i = 0 < \frac{q}{2}$ so the side-channel reports a 'miss' for both ciphertexts.

(iii) Case $f_i = -1$. Then $(c_1 \star f)_i = -\frac{q}{8} \equiv \frac{7q}{8} \geq \frac{q}{2} \pmod{q}$ and $(c_2 \star f)_i = -\frac{3q}{4} \equiv \frac{q}{4} < \frac{q}{2} \pmod{q}$ so the side-channel reports a 'hit' on the first ciphertext and a 'miss' on the second.

We can therefore recover the secret coefficient based on which ciphertexts 'hit' or 'miss'. This is illustrated in Figure 4.1. Coefficients for which we observe a 'miss' are coloured green, coefficients for which we observe a 'hit' are coloured red.

Cost. We will analyse the attack cost using the number of decryption queries required to perform the attack. In this case, the attack requires just 2 decryption queries, regardless of the length of the private key. We therefore say that the attack cost is constant in the length of the private key.

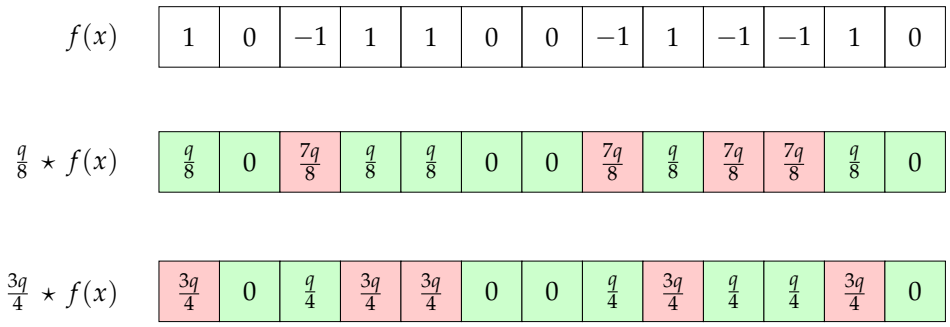

Figure 4.1: Recovering private key coefficients ($n = 13$).

The situation is a little more complicated for the two variants discussed in Sections 3.2.1 and 3.2.2.

4.2.1 Linearly transformed coefficients

We recall that in the linearly transformed variant, the coefficients are transformed as follows

$$f(x) = p \cdot f'(x) + 1 = 3 \cdot f'(x) + 1 \tag{4.6}$$

for some ternary polynomial $f'(x)$.

We use the ciphertext $c_1(x) = \frac{q}{8}$ from the previous section, but instead of $c_2(x) = \frac{3q}{4}$ we let

$$c_2(x) = \frac{q}{4}. \tag{4.7}$$

Because the constant coefficient of $f(x)$ is subjected to a slightly different transformation (by adding 1 to it), the measurements differ for the first coefficient.

We therefore distinguish the following cases.

i	$(\frac{q}{8} \star f)_i \geq \frac{q}{2} \pmod q$?	$(\frac{q}{4} \star f)_i \geq \frac{q}{2} \pmod q$?	f_i
0	Hit	Miss	4
	Miss	Miss	1
	Hit	Hit	-2
$1, 2, \ldots, n-1$	Miss	Hit	3
	Miss	Miss	0
	Hit	Miss	-3

Table 4.1: Determining linearly transformed private key coefficients from side-channel info.

The case $i = 0$. In this case, $f_0 \in \{-2, 1, 4\}$. We consider each case separately.

(i) Case $f_0 = 4$. Then $(c_1 \star f)_0 = \frac{q}{2} \geq \frac{q}{2}$ and $(c_2 \star f)_0 = \frac{4q}{4} \equiv 0 < \frac{q}{2} \pmod q$, so the side-channel reports 'hit' on the first ciphertext and a 'miss' on the second.

(ii) Case $f_0 = 1$. Then $(c_1 \star f)_i = \frac{q}{8} < \frac{q}{2}$ and $(c_2 \star f)_i = \frac{q}{4} < \frac{q}{2}$, so the side-channel reports a 'miss' on both ciphertexts.

(iii) Case $f_0 = -2$. Then $(c_1 \star f)_i = -\frac{q}{4} \equiv \frac{3q}{4} \geq \frac{q}{2} \pmod q$ and $(c_2 \star f)_i = -\frac{q}{2} \equiv \frac{q}{2} \geq \frac{q}{2} \pmod q$ so the side-channel reports a 'hit' on both ciphertexts.

The case $i = 1, \ldots, n-1$. In this case, $f_i \in \{-3, 0, 3\}$. Again, we consider each case separately.

(i) Case $f_i = 3$. Then $(c_1 \star f)_i = \frac{3q}{8} < \frac{q}{2}$ and $(c_2 \star f)_i = \frac{3q}{4} \geq \frac{q}{2}$, so the side-channel reports a 'miss' on the first ciphertext and a 'hit' on the second.

(ii) Case $f_i = 0$. Then $(c_1 \star f)_i = (c_2 \star f)_i = 0 < \frac{q}{2}$ so the side-channel reports a 'miss' for both ciphertexts.

(iii) Case $f_i = -3$. Then $(c_1 \star f)_i = -\frac{3q}{8} \equiv \frac{5q}{8} \geq \frac{q}{2} \pmod q$ and $(c_2 \star f)_i = -\frac{3q}{4} \equiv \frac{q}{4} \leq \frac{q}{2} \pmod q$ so the side-channel reports a 'hit' on the first ciphertext and a 'miss' on the second.

These results are summarized in Table 4.1. Algorithm 9 demonstrates how this attack might be implemented. The resulting polynomial $s(x)$ will equal $f(x)$.

Cost. As in the previous attack, this attack requires just 2 decryption queries, regardless of the length of the private key. Again, we say that the attack cost is constant in the length of the secret key.

4.2.2 Normally distributed coefficients

We will also consider the variant described in Section 3.2.2, in which the private key's coefficients are sampled from a discrete Gaussian distribution.

We can recover these coefficients by performing a binary-search-like attack.

Let

$$c_i(x) = 2^i \tag{4.8}$$

be a set of ciphertexts for $i = 0, \ldots, \log_2(q) - 1$.

Figure 4.2 illustrates how every possible value of f_i is uniquely identified by a sequence of 'hits' or 'misses'. Note that these coefficients are all reduced to $[0, q)$, as per the assumptions in the previous attacks. Algorithm 10 demonstrates how the coefficients can be computed.

Algorithm 9 Exploiting the side-channel when using linearly transformed coefficients.

procedure ExploitLinear($f(x)$)
  $v = \text{Observe}(f(x), c_1(x))$
  $u = \text{Observe}(f(x), c_2(x))$
  if not v_0 **then**
    $s_0 = 1$
  else if u_0 **then**
    $s_0 = -2$
  else
    $s_0 = 4$
  for $i = 1, \ldots, n-1$ **do**
    if v_i **then**
      $s_i = -3$
    else if u_i **then**
      $s_i = 3$
    else
      $s_i = 0$
  return $s(x) \in R_q$

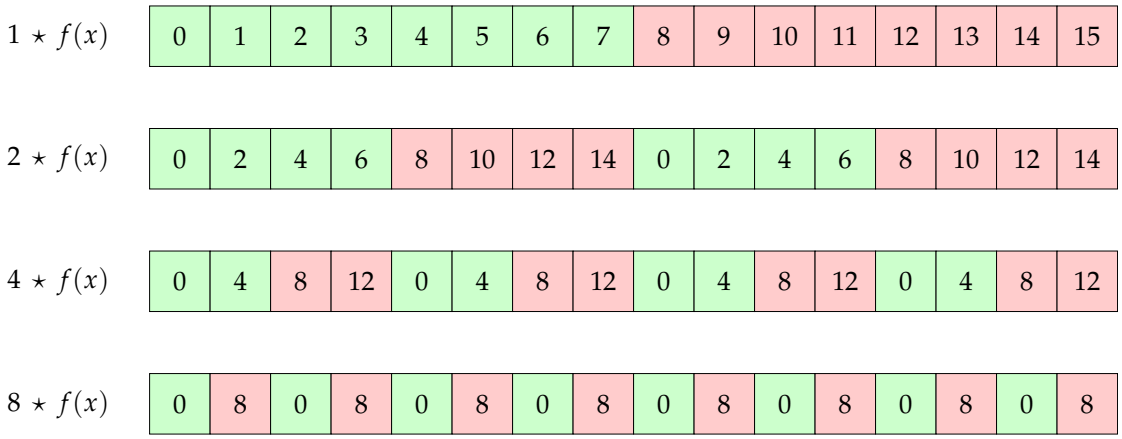

Figure 4.2: Recovering normally distributed coefficients ($q = 16$).

Algorithm 10 Exploiting the side-channel when using normally distributed coefficients.

procedure ExploitNormal($f(x)$)
  $m = \log_2(q) - 1$
  for $i = 0, \ldots, m$ **do**
    $v_i = \text{Observe}(f(x), c_i(x))$
  for $i = 0, \ldots, n$ **do**
    $s_i = \sum_{j=0}^{m} (v_i)_j \cdot 2^{m-j}$ ▷ where $(v_i)_j = 0$ if $(v_i)_j =$ **false**, and 1 otherwise
  return $s(x) \in R_q$

Cost. In this case, the attack requires $\log_2 q$ decryption queries. Consequently, the attack cost is constant in the length of the secret key, but logarithmic in the ring parameter q.

4.3 Exploiting partial information

The central assumption in Section 4.2 is that we can observe the side-channel leakage for every individual coefficient. In a real-world scenario this is unlikely to be the case: the vulnerable operation would most likely be performed so fast that we would not be able to measure it precisely enough. In order to provide a more realistic attack, we now make the assumption that we can only observe the first k coefficients *collectively*. In other words, we can only observe if $a_i < \frac{q}{2}$ for *all* $0 \leq i < k$ or if there is *at least one* $0 \leq i < k$ such that $a_i \geq \frac{q}{2}$.

We make the same assumptions on the representation of coefficients as in Section 4.2. Regarding the parameters, we assume that $p = 3$, $q = 2^\ell$ for some $\ell \geq 4$.

Algorithm 11 shows how we can adapt the side-channel model from Algorithm 8 to these new assumptions.

Algorithm 11 Side-channel model when only the first k coefficients can be observed.

```
1: procedure ObservePartial(f(x), c(x), k)
2:     v = Observe(f(x), c(x))
3:     for i = 0, ..., k − 1 do
4:         if v_i then
5:             return true
6:     return false
```

Finally, in this section, we will only consider the variant of NTRUEncrypt which uses linearly transformed coefficients. This is because the reference implementation does not implement any of the other variants.

4.3.1 Locating a zero-box

This attack hinges on the existence of a set of k subsequent coefficients equal to 0 in $f'(x)$ (the pre-transformation key). We denote the index of the first coefficient in this set by i_0, and we require that $0 \leq i_0 \leq n - k$ because for $n - k + 1 \leq i_0 \leq n - 1$ the requirement cannot be satisfied (as one of the 'subsequent' indices would be 0, and $f_0 \neq 0$). If $i_0 \neq 0$, we will further require that $f_{i_0 - 1 \pmod n}$ is invertible modulo q. There are alternative approaches which do not require this; they will be discussed briefly in Section 4.3.4.

We define the sets of ciphertexts

$$c_{1,i}(x) = \frac{q}{8} x^{n-i} \pmod n \tag{4.9}$$

$$c_{2,i}(x) = \frac{q}{4} x^{n-i} \pmod n \tag{4.10}$$

for $i = 0, \ldots, n-1$. In essence, this rotates the coefficients of the polynomial $n - i \pmod n$ places to the right (towards higher-order terms) and then multiplies all coefficients by a constant. This ensures that the constant coefficient in the resulting product $(c_{1,i} \star f)(x)$ will be f_i multiplied by some constant.

From Table 4.1 we gather that the only case in which we observe a 'miss' for both polynomials is if $f_i \in \{0, 1\}$. Under the assumption that we collectively observe the first k coefficients for each of the decryptions, we will only observe a 'miss' for both polynomials if for all $i \leq j < i + k$ we

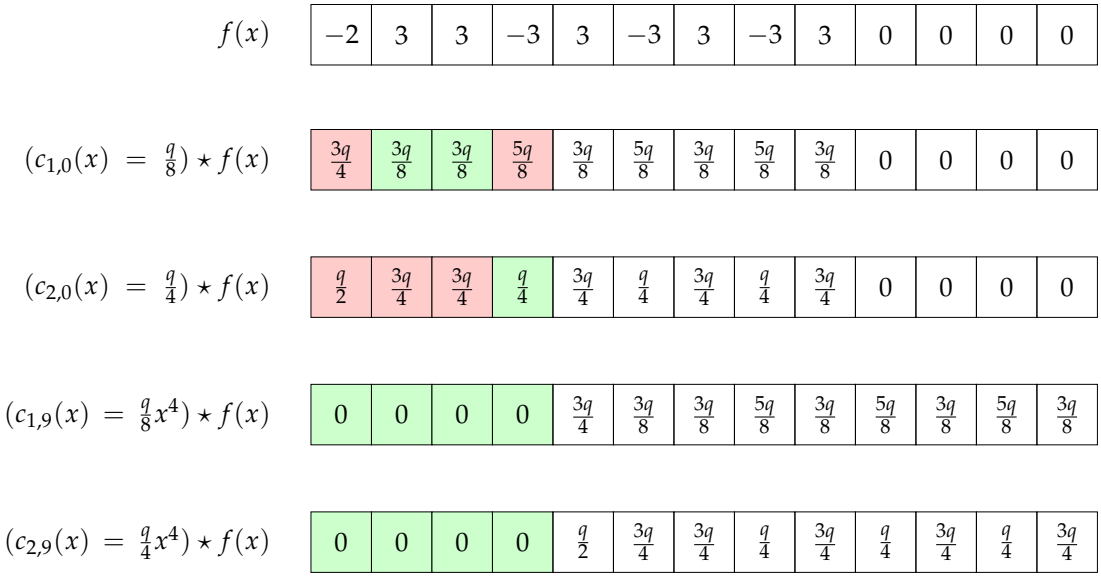

Figure 4.3: Locating a zero-box ($n = 13$, $p = 3$, $k = 4$).

have $f_{j \pmod n} \in \{0, 1\}$. Hence, we can locate a zero-box by observing the ciphertexts $c_{1,i}$ and $c_{2,i}$ for $i = 0, 2, \ldots, n - k$ until we observe a 'miss' on both.

Algorithm 12 demonstrates how i_0 can be found. We define a special case for $i_0 = 1$, because in that case $f_{i_0-1} = f_0$ is not invertible modulo q, which is required by our assumption. As we therefore skip $i_0 = 1$, we can no longer safely assume that we return on the *first* zero-box. This means that there may be a zero-valued coefficient directly to the left of the zero-box, which is in conflict with our assumption. Therefore, if a zero-box starts at $i_0 = 1$, we skip indices until we find a non-zero coefficient. By continuing our search after this coefficient, we can again assume that if we find another zero-box, it is preceded by a non-zero coefficient.

Algorithm 12 Locating a zero-box.

```
1: procedure FindZero(f(x), k)
2:     for i = 0, 1, ..., n − k do
3:         if not ObservePatial(f(x), c_{1,i}(x), k) and not ObservePartial(f(x), c_{2,i}(x), k) then
4:             if i = 1 then
5:                 increment i until either c_{1,i} or c_{2,i} causes a branch 'hit'
6:             else
7:                 return i
8:     return −1                                  ▷ Unable to locate zero-box, attack fails
```

We will consider a running example for $n = 13$, $p = 3$, $k = 4$ and

$$f = (-2, 3, 3, -3, 3, -3, 3, -3, 3, 0, 0, 0, 0). \tag{4.11}$$

This example is illustrated in Figure 4.3. For $i = 0$ we see that we observe a 'hit' on both ciphertexts. This same observation holds for $1 \leq i \leq 8$. However, for $i = 9$ we have $c_{1,9} = \frac{q}{8}x^4$ and $c_{2,9} = \frac{q}{4}x^4$ and we see that for both ciphertexts we observe a 'miss'. This means that we have found the zero-box, and we let $i_0 = 9$.

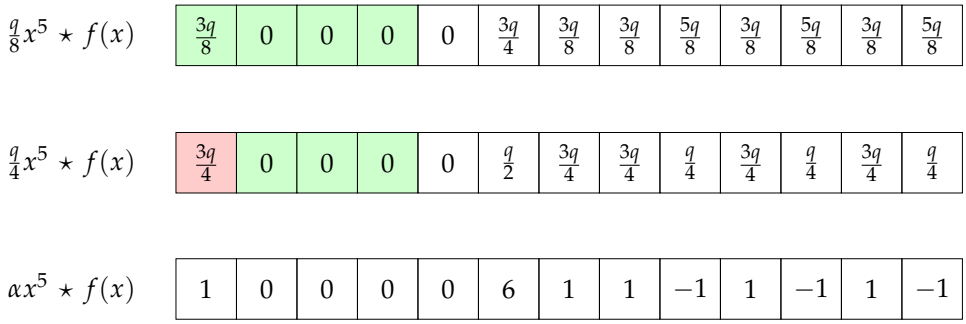

Figure 4.4: Recovering the coefficient adjacent to the zero-box ($n = 13$, $p = 3$, $k = 4$, $q = 16$).

4.3.2 Recovering the adjacent coefficient

We can now start using the zero-box. Our goal in this section is to find an index i_α and a value α such that we obtain the following polynomial for $c(x) = \alpha x^{n-i_\alpha \pmod n}$:

$$(\alpha x^{n-i_\alpha \pmod n} \star f) = (1, \overbrace{0, 0, \ldots, 0}^{k-1 \text{ times}}, \ldots). \tag{4.12}$$

We can use this polynomial to add or subtract arbitrary values at certain indices.

Based on the value of i_0 found in the previous section, we distinguish two cases.

(i) Case $i_0 = 0$. In this case, we let $i_\alpha = i_0 = 0$, and hence $f_{i_\alpha} = 1$ (see Table 4.1). We let $\alpha = 1$ so that $\alpha f_{i_\alpha} = 1$ and $\alpha f_i = 1 \cdot 0 = 0$ for $0 < i \leq k$.

(ii) Case $i_0 \in \{2, 3, \ldots, n-k\}$. In this case, we let $i_\alpha = i_0 - 1 \pmod n$. Then $f_{i_\alpha} \in \{-3, 3\}$ as $f_{i_\alpha} \neq 0$ (see Section 4.3.1). As $f_{i_\alpha+1 \pmod n} = \cdots = f_{i_\alpha+k-2 \pmod n} = 0$, we can simply use c_{1,i_α} and c_{2,i_α} to determine f_{i_α} as we did in Section 4.2.1– the $k-1$ coefficients after f_{i_α} are guaranteed to cause 'misses'. After determining f_{i_α}, we let $\alpha = f_{i_\alpha}^{-1} \pmod q$. As p and q are required to be coprime in NTRUEncrypt, this inverse exists and we have $\alpha f_{i_\alpha} = 1 \pmod q$ and $\alpha f_i = \alpha \cdot 0 = 0$ for $0 < i \leq k$.

The procedure is summarized in Algorithm 13 and applied to the running example in Figure 4.4. We have $i_0 = 9$ so $i_\alpha = 8$ and $c_{1,i_\alpha} = \frac{q}{8}x^5$ and $c_{2,i_\alpha} = \frac{q}{4}x^5$. We recover $f_{i_\alpha} = -3$, hence $\alpha = f_{i_\alpha}^{-1} = 5 \pmod q$ for $q = 16$.

Algorithm 13 Recovering the coefficient adjacent to the zero-box.

Require: $i_0 \in \{0, 2, 3, \ldots, n-k\}$

procedure FindAlpha($f(x), i_0, k$)
  if $i_0 = 0$ **then**
    $i_\alpha = 0$
    $\alpha = 1$
  else
    $i_\alpha = i_0 - 1 \pmod n$
    if ObservePartial($f(x), c_{1,i_\alpha}, k$) **then**
      $f_{i_\alpha} = -3$
    else if ObservePartial($f(x), c_{2,i_\alpha}, k$) **then**
      $f_{i_\alpha} = 3$
    $\alpha = f_{i_\alpha}^{-1} \pmod q$
  return (i_α, α)

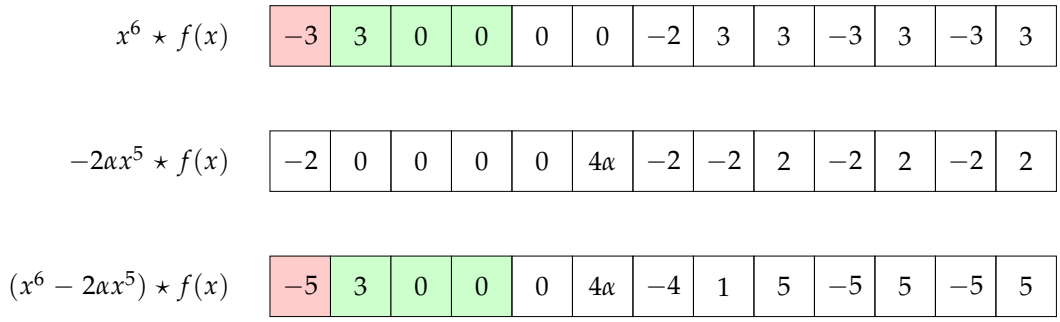

Figure 4.5: Recovering a coefficient without lifting ($n = 13$, $p = 3$, $k = 4$, $i_\alpha = 8$).

4.3.3 Recovering subsequent coefficients

We can now use the buildings blocks defined in the previous section to find the remaining coefficients of f. We define the following ciphertexts

$$c_{1,i}(x) = x^{n-i \pmod n}, \tag{4.13}$$
$$c_{2,i}(x) = x^{n-i \pmod n} - 2\alpha x^{n-i_\alpha \pmod n}. \tag{4.14}$$

Ciphertext $c_{1,i}(x)$ simply rotates $f(x)$ such that $(c_{1,i} \star f)_0 = f_i$. Ciphertext $c_{2,i}(x)$ rotates $f(x)$ and subtracts 2 from the first coefficient, i.e. $(c_{2,i} \star f)_0 = f_i - 2$, but does not touch the $k-1$ coefficients after it.

If we were to know that $(c_{1,i} \star f)_j < \frac{q}{2}$ for $0 < j < k$, then none of these coefficients would cause a 'hit'. Hence, if we would observe a 'hit', we would know it was caused solely because $f_i \geq \frac{q}{2}$ or $f_i - 2 \geq \frac{q}{2}$.

We will first consider how to recover the unknown coefficient f_i from these ciphertexts, assuming that $(c_{1,i} \star f)_j < \frac{q}{2}$ for $0 < j < k$. We distinguish two cases.

Case $i = 0$. In this case, we have $f_0 \in \{-2, 1, 4\}$. We note that we require $q \geq 2^4$ in this attack, and consider each possible value of f_0 separately.

(i) Case $f_0 = 4$. Then $(c_{1,0} \star f)_0 = 4 < \frac{q}{2} \pmod q$ and $(c_{2,0} \star f)_0 = 2 < \frac{q}{2} \pmod q$. Hence we observe two 'misses' for the lifted polynomials.

(ii) Case $f_0 = 1$. Then $(c_{1,0} \star f)_0 = 1 < \frac{q}{2} \pmod q$ and $(c_{2,0} \star f)_0 = q - 1 \geq \frac{q}{2} \pmod q$. Hence we observe a 'miss' for the first polynomial and a 'hit' for the second.

(iii) Case $f_0 = -2$. Then $(c_{1,0} \star f)_0 = q - 2 \geq \frac{q}{2} \pmod q$ and $(c_{2,0} \star f)_0 = q - 4 \geq \frac{q}{2} \pmod q$. Hence we observe two 'hits' for the lifted polynomials.

Case $i = 1, \ldots, n-1$. In this case, we have $f_i \in \{-3, 0, 3\}$. Again, we note that we require $q \geq 2^4$ in this attack, and consider each possible value of f_i separately.

(i) Case $f_i = 3$. Then $(c_{1,i} \star f)_0 = 3 < \frac{q}{2} \pmod q$ and $(c_{2,i} \star f)_0 = 1 < \frac{q}{2} \pmod q$. Hence we observe two 'misses' for the lifted polynomials.

(ii) Case $f_i = 0$. Then $(c_{1,i} \star f)_0 = 0 < \frac{q}{2} \pmod q$ and $(c_{2,i} \star f)_0 = q - 2 \geq \frac{q}{2} \pmod q$. Hence we observe a 'miss' for the first polynomial and a 'hit' for the second.

(iii) Case $f_i = -3$. Then $(c_{1,i} \star f)_0 = q - 3 \geq \frac{q}{2} \pmod q$ and $(c_{2,i} \star f)_0 = q - 5 \geq \frac{q}{2} \pmod q$. Hence we observe two 'hits' for the lifted polynomials.

Next, we consider how to modify a ciphertext $c(x)$ such that in the resulting polynomial $(c \star f)(x)$ we have $(c \star f)_i < \frac{q}{2} \pmod q$ for $0 < j < k$. Algorithm 14 provides a procedure that achieves this. The algorithm assumes that the coefficients $f_{i_{\text{next}}+1}, f_{i_{\text{next}}+2}, \ldots, f_{i_\alpha+k \pmod n}$ are known. If we start by finding a zero-box, then recover the coefficient immediately to the left of f_{i_0}, and then recover the coefficients to the left of $f_{i\alpha}$ one by one, this is guaranteed to be the case.

We will give a line-by-line explanation of how the algorithm works. Line 2 begins the outer loop. In lines 3 and 4, we then compute the index which we may need to reduce and we predict the current value at that index, i.e. $v_{\text{current}} = (c \star f)_i$. This prediction is based on the original value of f at that index, which is known, and on $e_{i_{\text{current}}}$ which contains the value we have added to or subtracted from the coefficient at index i_{current} during the course of the algorithm.

In line 5, we examine if the predicted coefficient needs to be reduced. If not, we move on to the next index to check. If the coefficient does need to be reduced, we proceed by using (4.12), shifted i places to the right (towards higher-order terms), to subtract v_{current} in line 6. As the k coefficients to the right of i_α are zero (see (4.12)), the predicted values at the next k indices higher than i_{current} will remain unchanged. The coefficients to the left of i_α could be anything, however, so these will impact our predictions. As we know by assumption what these coefficients are, we can compute what we have added or subtracted and keep track of that in the e_i values (lines 7–9) before moving to the next iteration of the outer loop.

Finally, in line 10 we ensure that the first coefficient of the resulting polynomial, which is used to determine its value, remains unchanged.

Algorithm 14 Lift a ciphertext $c(x)$ such that $(c \star f)_i < \frac{q}{2} \pmod q$ for $0 < i < k$.

Require: $f_{i_\alpha} > 0$ and $f_{i \pmod n}$ known for $i_{\text{next}} < i < i_\alpha + k$
Ensure: $(c \star f)_{i \pmod n} < \frac{q}{2}$ for $0 < i < k$ and $(c \star f)_0$ invariant

1: **procedure** LIFT($f(x), c(x), i_{\text{next}}, i_\alpha, \alpha, k$)
2: **for** $i = k-1, \ldots, 1$ **do**
3: $i_{\text{current}} = i_{\text{next}} + i \pmod n$
4: $v_{\text{current}} = f_{i_{\text{current}}} + e_{i_{\text{current}}} \pmod q$ ▷ Virtual value at index i_{current}
5: **if** $v_{\text{current}} \geq \frac{q}{2}$ **then**
6: $c_{n-i_\alpha+i \pmod n} = c_{n-i_\alpha+i \pmod n} - \alpha v_{\text{current}} \pmod q$ ▷ Get v_{current} to 0
7: **for** $j = 1, \ldots, i$ **do**
8: $i_{\text{extra}} = i_\alpha - j \pmod n$
9: $e_{i_{\text{current}}-j \pmod n} = e_{i_{\text{current}}-j \pmod n} - \alpha v_{\text{current}} f_{i_{\text{extra}}} \pmod q$
10: $c_{n-i_\alpha \pmod n} = c_{n-i_\alpha \pmod n} - \alpha e_{i_{\text{current}}} \pmod q$

Algorithm 15 combines the lifting step and the coefficient recovery method.

Figures 4.5 and 4.6 illustrate how this algorithm can be used to recover the next two coefficients in the running example. In Figure 4.5, the lift operation checks the vector $(3, 0, 0)$. None of these need to be modified, so we can apply the coefficient recovery method to find $f_7 = -3$.

In Figure 4.6, the lift operation checks the vector $(-3, 3, 0)$. This last value needs to be modified, so we subtract $-3\alpha x^6 \star f(x)$. Because this modifies the unknown value, we need to compensate by subtracting $-3\alpha x^5 \star f(x)$. We can now apply the coefficient recovery method to find $f_6 = 3$.

Cost. In the worst-case scenario, finding a zero-box requires $n - k + 1$ tries, so $2n - 2k + 2$ decryption queries. After locating a zero-box, there are $n - k$ unknown coefficients remaining. For each of those coefficients, at most 2 decryption queries are required. This brings the total

Algorithm 15 Recovering the coefficient at index i_{next}.

Require: $f_{i_\alpha} > 0$ and $f_{i \pmod n}$ known for $i_{\text{next}} < i < i_\alpha + k$

procedure FindNext($f(x), i_{\text{next}}, i_\alpha, \alpha, k$)
  Lift($f(x), c_{1,i_{\text{next}}}, i_{\text{next}}, i_\alpha, \alpha, k$)
  Lift($f(x), c_{2,i_{\text{next}}}, i_{\text{next}}, i_\alpha, \alpha, k$)
  if ObservePartial($f(x), c_{1,i_{\text{next}}}(x), k$) **then**
    $f_{i_{\text{next}}} = -3$
  else if ObservePartial($f(x), c_{2,i_{\text{next}}}(x), k$) **then**
    $f_{i_{\text{next}}} = 0$
  else
    $f_{i_{\text{next}}} = 3$
  if $i_{\text{next}} = 0$ **then**
    $f_{i_{\text{next}}} = f_{i_{\text{next}}} + 1$

$x^7 \star f(x)$	3	-3	3	0	0	0	0	-2	3	3	-3	3	-3
$3\alpha x^6 \star f(x)$	-3	3	0	0	0	0	-6α	3	3	-3	3	-3	3
$3\alpha x^5 \star f(x)$	3	0	0	0	0	-6α	3	3	-3	3	-3	3	-3
$(x^7 + 3\alpha x^6 + 3\alpha x^5) \star f(x)$	3	0	3	0	0	-6α	$3-6\alpha$	4	3	3	-3	3	-3
$(x^7 + 3\alpha x^6 + \alpha x^5) \star f(x)$	1	0	3	0	0	-2α	$1-6\alpha$	2	5	1	-1	1	-1

Figure 4.6: Recovering a coefficient with lifting ($n = 13$, $p = 3$, $k = 4$, $i_\alpha = 8$).

number of required queries to $2n - 2k + 2 + 2(n - k) = 4n - 4k + 2$. Hence, the attack cost is linear in the length of the secret key.

4.3.4 Implementation and attack success probability

An implementation of this attack on the NTRUEncrypt reference implementation is provided in Appendix A. It uses a slightly modified version of the original reference implementation: the decryption method is modified to take an integer array as parameter, which will be populated by ones and zeroes depending on if the branch was 'hit' for a coefficient.

This implementation relies on the assumption stated in Section 4.3.1, which can potentially be quite stringent. Therefore, we will approximate the expected value for the length of the largest zero-box present in a secret key and the probability of finding a zero-box of length k in order to give an idea of how applicable this attack is.

We note that for some n and d, an element $f' \in \mathcal{T}_n(d+1, d)$ contains precisely $n - 2d - 1$ coefficients equal to 0, hence the ratio of zero-valued coefficients to the total number of coefficients is $\frac{n-2d-1}{n}$.

We will use this ratio to approximate the probability of any individual coefficient being equal to 0. Let F_i be a random variable describing the outcome of a Bernoulli trial. We say that the trial succeeds if the coefficient at index i is equal to 0, and fails otherwise. This means we approximate the probability of success at $P = \frac{n-2d-1}{n}$.

We then model the secret key f' (i.e. the key before applying a linear transformation) as a random vector

$$\mathbf{F} = (F_0, \ldots, F_{n-1}) \tag{4.15}$$

with $F_i \sim \text{Bernoulli}\left(\frac{n-2d-1}{n}\right)$. Of course, this model only makes sense if n and P are large – in reality, the F_i are not i.i.d. Bernoulli random variables as the number of coefficients that have certain values are known beforehand.

Next, we let R_n be a random variable indicating the length of longest run of successes in a series of n such Bernoulli trials. We observe that

$$\mathbb{P}(R_n \geq k) \quad = \quad \mathbb{P}(\text{there is a zero-box of length } k) \tag{4.16}$$

hence we can use R_n to estimate the probability that the attack is successful.

The expected value, variation and probability distribution of R_n are then given by

$$\mathbb{E}(R_n) \quad = \quad \log_{\frac{1}{P}}(n(1-P)) + \frac{\gamma}{\ln\left(\frac{1}{P}\right)} - \frac{1}{2} + r_1(n) + \epsilon_1(n) \tag{4.17}$$

$$\text{Var}(R_n) \quad = \quad \frac{\pi^2}{6\ln^2\left(\frac{1}{P}\right)} + \frac{1}{12} + r_2(n) + \epsilon_2(n) \tag{4.18}$$

$$\mathbb{P}(R_n \geq k) \quad = \quad \sum_{i=1}^{\lfloor \frac{n}{k} \rfloor} (-1)^{i+1} \left(P + \frac{n-ik+1}{i}(1-P)\right) \binom{n-ik}{i-1} P^{ki}(1-P)^{i-1} \tag{4.19}$$

where γ is the Euler-Mascheroni constant and $r_i(n), \epsilon_i(n)$ vanish for large enough n [13], [14].

Table 4.2 and Figure 4.7 provide an overview of the expected value $\mathbb{E}(R_n)$, the standard deviation $\sigma_{R_n} = \sqrt{\text{Var}(R_n)}$ and the probability that there is at least one zero-box of length k for common NTRU parameter sets and $k \in \{4, 8, 16, 32\}$. The EESxxxxEPy parameter sets are defined in [15] and are a subset of the parameter sets supported in the reference implementation. The

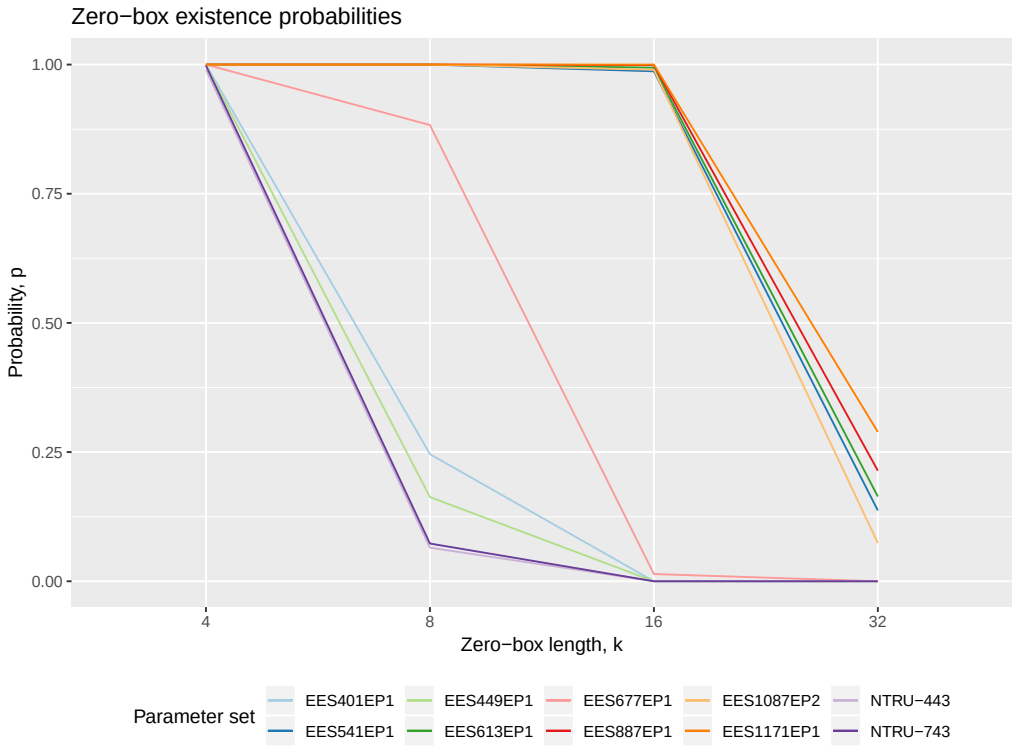

Figure 4.7: Probability of finding a zero-box of length k for common NTRU parameter sets.

NTRU-xxx parameter sets are taken from [11].

In order to verify these probabilities, we simulate them (see Appendix B) by generating 10^6 random private keys and checking the number of keys which have at least one zero-box of length $k \in \{4, 8, 16, 32\}$. By dividing the number of such keys by 10^6, we obtain the probabilities displayed in Table 4.3. The probabilities on the right correspond to cases where all zero-boxes starting at $i_0 = 1$ were excluded. Note that these probabilities are similar to those on the left, but not always equal.

However, as alluded to in Section 4.3.1, there are other ways to solve the issue in the case $i_0 = 1$. For example, it is not strictly necessary to obtain the polynomial (4.12). In particular, the attack could be modified to work with constant coefficients other than 1. Instead, we could add and subtract multiples of values in $\{3, -3, 4, 1, -2\}$ in order to ensure that coefficients are in the range $[0, \frac{q}{2})$. This would allow us to drop the invertibility requirement from the assumption in Section 4.3.1.

Moreover, if we were to use a polynomial of the form

$$(\alpha x^{n-i_\alpha \pmod n} \star f) = (\overbrace{0, 0, \ldots, 0}^{k-1 \text{ times}}, 1, \ldots) \tag{4.20}$$

instead, we could also perform this attack by shifting coefficients towards the opposite direction. In that case, instead of considering polynomials $(c \star f)$ where $(c \star f)_0$ is unknown and $0 \leq (c \star f)_i < \frac{q}{2}$ for $0 < i < k$, we would consider polynomials $(c \star f)$ where $(c \star f)_{k-1}$ is unknown and $0 \leq (c \star f)_i < \frac{q}{2}$ for $0 \leq i < k - 1$.

					$\mathbb{P}(R_n \geq k)$			
Parameter set	n	d	$\mathbb{E}(R_n)$	σ_{R_n}	$k = 4$	$k = 8$	$k = 16$	$k = 32$
EES401EP1	401	113	6.689	1.563	1.000	0.246	0.000	0.000
EES541EP1	541	49	25.092	6.352	1.000	1.000	0.987	0.137
EES449EP1	449	134	6.252	1.433	0.999	0.163	0.000	0.000
EES613EP1	613	55	25.965	6.427	1.000	1.000	0.994	0.164
EES677EP1	677	157	9.611	2.069	1.000	0.883	0.014	0.000
EES887EP1	887	81	27.428	6.323	1.000	1.000	0.999	0.214
EES1087EP2	1087	120	23.684	5.125	1.000	1.000	0.990	0.074
EES1171EP1	1171	106	29.079	6.395	1.000	1.000	1.000	0.289
NTRU-443	443	143	5.475	1.262	0.990	0.065	0.000	0.000
NTRU-743	743	247	5.681	1.204	0.998	0.073	0.000	0.000

Table 4.2: Approximate probabilities $\mathbb{P}(R_n \geq k)$ and expected values $\mathbb{E}(R_n)$ for common NTRU parameter sets.

	Total				Without $i_0 = 1$			
Parameter set	$k = 4$	$k = 8$	$k = 16$	$k = 32$	$k = 4$	$k = 8$	$k = 16$	$k = 32$
EES401EP1	1.0000	0.2351	0.0002	0	1.0000	0.2345	0.0002	0
EES541EP1	1	1	0.9941	0.1205	1	1	0.9940	0.1203
EES449EP1	0.9999	0.1533	0.0001	0	0.9999	0.1530	0.0001	0
EES613EP1	1	1	0.9976	0.1479	1	1	0.9976	0.1476
EES677EP1	1	0.8941	0.0119	0	1	0.8937	0.0119	0
EES887EP1	1	1	0.9998	0.1999	1	1	0.9998	0.1997
EES1087EP2	1	1	0.9941	0.0681	1	1	0.9941	0.0680
EES1171EP1	1	1	1.0000	0.2778	1	1	1.0000	0.2776
NTRU-443	0.9946	0.0591	0.0000	0	0.9946	0.0590	0.0000	0
NTRU-743	0.9992	0.0685	0.0000	0	0.9992	0.0685	0.0000	0

Table 4.3: Simulated probabilities after 10^6 rounds.

4.4 Mitigation

We now turn to the question of how this attack might be mitigated. The obvious solution would be to ensure that the running time of the lift operation no longer depends on any secret values.

Listing 1 recalls the vulnerable implementation of the lift step for one coefficient a and the x86 assembly generated by the compiler[2]. The branch is clearly visible.

Listing 1 Vulnerable center-lift implementation and compiler output.

```
int32_t lift(int32_t a, uint32_t q)
{
    if (a >= q / 2)
    {
        a -= q;
    }
    return a;
}
```

```
lift:
    push    rbp
    mov     rbp, rsp
    mov     dword [rbp+0x10], ecx
    mov     dword [rbp+0x18], edx
    mov     eax, dword [rbp+0x18]
    shr     eax, 0x1
    mov     edx, eax
    mov     eax, dword [rbp+0x10]
    cmp     edx, eax
    ja      0x401571

    mov     eax, dword [rbp+0x10]
    sub     eax, dword [rbp+0x18]
    mov     dword [rbp+0x10], eax

    mov     eax, dword [rbp+0x10]
    pop     rbp
    retn
```

In order to remove the dependence on the secret key, we remove the branching step entirely. We will define an integer b that equals 1 whenever $a \geq \frac{q}{2}$ and 0 otherwise. This integer will be computed in constant time. If we then subtract $b \cdot q$ from a, the constant-time implementation will be functionally equivalent to the previous implementations.

Let $a, v = \frac{q}{2}$ two 32-bit unsigned integers. We define a 32-bit unsigned integer b as follows

$$b = 1 \oplus ((a - v) \gg 31), \tag{4.21}$$

where the XOR operation is denoted by $\oplus$, and the right shift operation is denoted by $\gg$.

We will evaluate the value of b for both the case $a \geq \frac{q}{2}$ and the case $a < \frac{q}{2}$.

1. Case $a \geq \frac{q}{2}$. In this case $a - v \geq 0$. Furthermore, as $a < q$ we also have $0 \leq a - v < q$. Hence, the most significant bit of $a - v$ will be 0, so $(a - v) \gg 31 = 0$. Finally, we have $b = 1 \oplus (a - v \gg 31) = 1 \oplus 0 = 1$.
2. Case $a < \frac{q}{2}$. In this case $a - v < 0$. As a result, the most significant bit of $a - v$ will be 1, so $(a - v) \gg 31 = 1$. Finally, we have $b = 1 \oplus (a - v \gg 31) = 1 \oplus 1 = 0$.

So $bq = 0$ if $a < \frac{q}{2}$ and $bq = q$ if $a \geq \frac{q}{2}$. Consequently, $a - bq = a$ if $a < \frac{q}{2}$ and $a - bq = a - q$ if $a \geq \frac{q}{2}$. This makes the implementation in Listing 2 equivalent to that in Listing 1.

Listing 2 demonstrates the constant-time implementation. Note that the corresponding disassembly contains no conditional operators.

[2] All samples were built with MinGW GCC 8.1.0 using its default settings.

Listing 2 Constant-time center-lift implementation and compiler output.

```
int32_t lift_ct_good(uint32_t a,
    uint32_t q)
{
    uint32_t v = q / 2;
    uint32_t b = 1u ^
        ((a - v) >> 31);

    return a - b * q;
}
```

```
lift_ct:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 0x10
    mov     dword [rbp+0x10], ecx
    mov     dword [rbp+0x18], edx
    mov     eax, dword [rbp+0x18]
    shr     eax, 0x1
    mov     dword [rbp-0x4], eax
    mov     eax, dword [rbp+0x10]
    sub     eax, dword [rbp-0x4]
    shr     eax, 0x1f
    xor     eax, 0x1
    mov     dword [rbp-0x8], eax
    mov     eax, dword [rbp-0x8]
    imul    eax, dword [rbp+0x18]
    mov     edx, dword [rbp+0x10]
    sub     edx, eax
    mov     eax, edx
    add     rsp, 0x10
    pop     rbp
    retn
```

Chapter 5

Conclusions and future work

In this chapter, we summarize the content and conclusions presented in this thesis. We will also outline some opportunities for improvement and future research.

5.1 Summary and conclusions

We began this thesis by providing a brief overview of lattices and hard problems on lattices in Chapter 2. We then introduced the NTRU public-key cryptosystem as well as some common variants in Chapter 3. Using the theory of lattices introduced in the previous chapter, we discussed why NTRUEncrypt is considered theoretically secure.

In Chapter 4 we presented several ways to exploit a timing side-channel vulnerability in the reference implementation of NTRUEncrypt under varying circumstances. Three of these attacks, on 'schoolbook' NTRU, a variant using linearly transformed coefficients and a variant using normally distributed coefficients, assume that we can observe the side-channel leak for each coefficient individually.

We then noted that this assumption is generally unrealistic, and presented a fourth attack that assumes that we can only observe the first k coefficients collectively. We examined for which value of k the attack is likely to work, and found that $k = 4$ works with probability greater than 0.99 for all parameter sets, and that $k = 8$ and $k = 16$ work with probability greater than 0.98 for the parameter sets which use sparse keys. Finally, we examined how the center-lift operation can be implemented in constant time. This mitigates all attacks presented in this thesis.

We conclude, therefore, that the reference implementation of NTRUEncrypt is indeed vulnerable to timing side-channel attacks. In practice, however, this side-channel could be hard to exploit if an attacker is unable to measure the timings with high accuracy and the parameter set does not result in sparse keys.

5.2 Future work

There are several parts of this thesis that could be expanded upon. First of all, the alternative approaches to non-invertible coefficients discussed in Section 4.3.4 could be worked out in further detail. This would slightly increase the probability of a successful attack.

A second possible opportunity would be to attempt to generalize the attack using partial information in such a way that it does not rely on the existence of a zero-box per se. Instead, it might be possible to construct such a zero-box by searching for two subsequences in a private key that are either equal or each other's inverse. Adding (in the case of equal subsequences)

or subtracting (in the case of inverse subsequences) these two subsequences would result in a zero-box in the resulting polynomial.

Finally, it might be interesting to implement the side-channel attack on other implementations of NTRUEncrypt. The first-round submission to the NIST project, for example, has the same vulnerability[1].

[1] Available at https://github.com/NTRUOpenSourceProject/ntruencrypt_nist_submission/blob/b2fba3a0d365d08210e5f300eb7ec99a013c313a/Reference_Implementation/ntru-kem-1024/NTRUEncrypt.c#L436-L441.

Appendix A

Implementation source code

In this appendix, we include (parts of) the implementation of the attack discussed in Section 4.3.

This snippet shows how the side-channel was modelled in the decrypt method of NTRUEncrypt's reference implementation.

```
uint32_t
ntru_crypto_ntru_decrypt(
    uint16_t       privkey_blob_len, /*     in - no. of octets in private key
                                             blob */
    uint8_t const *privkey_blob,     /*     in - pointer to private key */
    uint16_t       ct_len,           /*     in - no. of octets in ciphertext */
    uint8_t const *ct,               /*     in - pointer to ciphertext */
    uint16_t      *pt_len,           /* in/out - no. of octets in pt, addr for
                                             no. of octets in plaintext */
    uint8_t       *pt,               /*    out - address for plaintext */
    int           *leak)             /*    out - side-channel leak */
{
    /* lines omitted for brevity */

    for (i = 0; i < params->N; i++)
    {
        ringel_buf1[i] = (ringel_buf2[i] + 3 * ringel_buf1[i]) & mod_q_mask;

        if (ringel_buf1[i] >= (params->q >> 1))
        {
            ringel_buf1[i] = ringel_buf1[i] - q_mod_p;
            /* this is the information that leaks */
            leak[i] = 1;
        }

        Mtrin_buf[i] = (uint8_t)(ringel_buf1[i] % 3);
    }

    /* lines omitted for brevity */

    return result;
}
```

The following two snippets show how the leaks produced by the side-channel model were observed.

```
std::vector<bool> observe(params params,
        const std::vector<uint16_t>& c,
        const std::vector<uint8_t>& pk)
{
```

```
    // Pack ciphertext.
    uint16_t c_packed_len = (params->N * params->q_bits + 7) / 8;
    uint8_t c_packed[c_packed_len];

    ntru_elements_2_octets(params->N, &c[0], params->q_bits, c_packed);

    // Send decryption query.
    std::vector<int> observation(params->N);
    uint16_t message_len = params->m_len_max;
    uint8_t message[message_len];

    ntru_crypto_ntru_decrypt(pk.size(), &pk[0], c_packed_len, c_packed,
            &message_len, message, &observation[0]);

    return std::vector<bool>(observation.begin(), observation.end());
}

bool observe(params params,
        const std::vector<uint16_t>& c,
        const std::vector<uint8_t>& pk,
        int k)
{
    // Obtain leaked data.
    auto leaks = observe(params, c, pk);

    // If none of the first k coefficients of c * pk were >= q/2, return false.
    bool result = false;

    for (int i = 0; i < k; ++i)
    {
        if (leaks[i])
        {
            // If at least one was >= q/2, return true.
            result = true;
        }
    }

    spdlog::trace("observed {}, {}!", to_string(leaks, k),
            result ? "hit" : "miss");

    return result;
}
```

The following two snippets show how the zero-box was located and how the coefficient directly adjacent to the zero-box was recovered.

```
int find_zero(params params,
        const std::vector<uint8_t>& key,
        int k)
{
    std::vector<uint16_t> lower(params->N);
    std::vector<uint16_t> upper(params->N);

    for (int i = 0; i < params->N; ++i)
    {
        lower[modN(params->N - i)] = params->q / 8;
        upper[modN(params->N - i)] = params->q / 4;

        if (!observe(params, lower, key, k) && !observe(params, upper, key, k))
        {
            // We are looking for a sequence of coefficients starting with a 1,
            // followed by k - 1 zeroes, which we can use in the compensation
            // step.
            if (i == 0)
```

```
            {
                // In this case we know that the first k coefficients of the
                // transformed key are
                //  [1, 0, 0, ..., 0],
                // which suffices for our purposes.
                return i;
            }
            else if (i == 1)
            {
                // In this case the first k+1 coefficients of the transformed
                // key are
                //  [ 4, 0, 0, ..., 0]
                // or
                //  [-2, 0, 0, ..., 0].
                // Neither leading coefficient can be inverted mod q, so we
                // will skip this index.
                spdlog::info("skipping zero-box at index 1");

                // In order to ensure we meet the criteria specified above, we
                // will now skip indices until we encounter a nonzero value. We
                // can then start looking for new 0-boxes after that value to
                // ensure the 0-box is preceded by a nonzero value.
                for (int j = 1; j < params->N - 1; ++j)
                {
                    lower[modN(params->N - i - j)] = params->q / 8;
                    upper[modN(params->N - i - j)] = 5 * params->q / 8;

                    if (observe(params, lower, key, k)
                            || observe(params, upper, key, k))
                    {
                        lower[modN(params->N - i - j)] = 0;
                        upper[modN(params->N - i - j)] = 0;

                        // The coefficient at index j + k is unequal 0. We will
                        // continue searching for 0-boxes starting from index
                        // j + k + 1. As i will be incremented in the outer for
                        // loop, we set i = j + k.
                        i = j + k;
                        spdlog::info(
                                "skipping {} indices, restarting at index {}",
                                i, i + 1);
                        break;
                    }

                    lower[modN(params->N - i - j)] = 0;
                    upper[modN(params->N - i - j)] = 0;
                }
            }
            else
            {
                // In this case we know that the coefficient at index i - 1 is
                // unequal 0.
                return i;
            }
        }

        lower[modN(params->N - i)] = 0;
        upper[modN(params->N - i)] = 0;
    }

    // Unable to find any zero-box.
    return -1;
}
```

```
int recover_alpha(params params,
        int index,
        const std::vector<uint8_t>& key,
        int k)
{
    // Prepare ciphertext.
    std::vector<uint16_t> lower(params->N);
    lower[modN(params->N - index)] = params->q / 8;

    // Recover data.
    if (observe(params, lower, key, k))
    {
        return -1;
    }

    return 1;
}
```

The following two snippets show how all other coefficients were recovered.

```
int recover(params params,
        int index,
        int alpha_index,
        int alpha,
        const std::vector<int>& recovered_key_transformed,
        const std::vector<uint8_t>& key,
        int k)
{
    std::vector<uint16_t> upper(params->N);
    std::vector<uint16_t> lower(params->N);

    lift(params, lower, index, alpha_index, alpha, recovered_key_transformed,
            k);
    lift(params, upper, index, alpha_index, alpha, recovered_key_transformed,
            k);

    bool swap = recovered_key_transformed[alpha_index] < 0;

    // Add base values.
    upper[modN(params->N - index)] = modQ(
            upper[modN(params->N - index)] + (swap ? -1 : 1));
    lower[modN(params->N - index)] = modQ(
            lower[modN(params->N - index)] + (swap ? -1 : 1));

    if (swap)
    {
        upper[modN(params->N - alpha_index)] = modQ(
                upper[modN(params->N - alpha_index)] + 2 * alpha);
    }

    lower[modN(params->N - alpha_index)] = modQ(
            lower[modN(params->N - alpha_index)] - 2 * alpha);

    // Resuling poly is structured as follows:
    //   [X-2 + + + + + + +] lower
    //   [X   + + + + + + +] upper

    // Recover key.
    if (observe(params, upper, key, k))
    {
        return swap ? 1 : -1;
    }
    else if (observe(params, lower, key, k))
    {
```

```
        return 0;
    }
    else
    {
        return swap ? -1 : 1;
    }
}

void lift(params params,
        std::vector<uint16_t>& c,
        int index,
        int alpha_index,
        int alpha,
        const std::vector<int>& recovered_key_transformed,
        int k)
{
    // Goal: ensure (f*c)(i) in [0, q/2) for i = index+1, ..., index + k - 1.
    std::vector<int> extra(params->N);
    int b = recovered_key_transformed[alpha_index] > 0 ? 1 : -1;

    for (int i = k - 1; i > 0; --i)
    {
        int current_index = modN(index + i);
        int current_value = modQ(b * recovered_key_transformed[current_index]
                + extra[current_index]);

        if (current_value >= params->q / 2)
        {
            // Get current_value to zero.
            // We know alpha * key[alpha_index] = 1, so we subtract
            // alpha * key[alpha_index] * current_value.
            c[modN(params->N - alpha_index + i)] = modQ(
                    c[modN(params->N - alpha_index + i)]
                            - alpha * current_value);

            // This has a side-effect: the values to right of alpha_index are
            // zero, but the values to the left are not. We keep track of
            // what we add so we can compensate for it later.
            for (int j = 1; j <= i; ++j)
            {
                int extra_index = modN(alpha_index - j);
                int extra_value = modQ(extra[modN(current_index - j)]
                        - alpha * current_value
                                * recovered_key_transformed[extra_index]);
                extra[modN(current_index - j)] = extra_value;
            }
        }
    }

    // Finally, ensure that the constant coefficient in c * f remains the same
    // after lifting.
    c[modN(params->N - alpha_index)] = modQ(
            c[modN(params->N - alpha_index)] - alpha * extra[index]);
}
```

The following snippet shows how the previous methods are combined into a complete attack.

```
std::function<int(int)> modN;
std::function<int(int)> modQ;

std::optional<std::vector<int>> execute(params params,
        const std::vector<uint8_t>& key,
        int k)
{
```

```
    // Helper functions.
    modN = [=](int i)
    {
        return mod(i, params->N);
    };

    modQ = [=](int i)
    {
        return mod(i, params->q);
    };

    std::vector<int> recovered_key(params->N);
    std::vector<int> recovered_key_transformed(params->N);
    int alpha, alpha_index;

    int zero_index = find_zero(params, key, k);

    if (zero_index < 0)
    {
        spdlog::error("Unable to find a zero-box of length {}.", k);
        return {};
    }
    else if (zero_index == 0)
    {
        alpha_index = 0;
        alpha = 1;
        recovered_key_transformed[alpha_index] = 1;
    }
    else
    {
        alpha_index = modN(zero_index - 1);
        recovered_key[alpha_index] = recover_alpha(params, alpha_index, key, k);
        recovered_key_transformed[alpha_index] = 3 * recovered_key[alpha_index];

        // Compute alpha such that alpha * f[alpha_index] = 1 (mod q).
        // Works because f[alpha_index] = +/- 3 and gcd(3, q) = 1 by NTRU requirement.
        alpha = inv(modQ(recovered_key_transformed[alpha_index]), params->q);
    }

    // Recover other coefficients.
    for (int i = modN(alpha_index - 1); i != modN(zero_index + k - 1); --i)
    {
        i = modN(i);

        recovered_key[i] = recover(params, i, alpha_index, alpha,
                recovered_key_transformed, key, k);
        recovered_key_transformed[i] = 3 * recovered_key[i];
        if (i == 0)
        {
            recovered_key_transformed[i] += 1;
        }
    }

    return recovered_key_transformed;
}
```

Appendix B

Simulation source code

In this appendix, we provide the source code for the simulations of the zero-box probabilities which were discussed in Section 4.3.4.

```python
import random
from typing import List, Tuple


def generate_key(n: int, d: int) -> List[int]:
    key = [0] * n
    indices = [j for j in range(0, n)]

    # Choose d + 1 random indices where the key will equal 1.
    for j in random.sample(indices, d + 1):
        key[j] = 1
        indices.remove(j)

    # Choose d random indices where the key will equal -1.
    for j in random.sample(indices, d):
        key[j] = -1
        indices.remove(j)

    # For all other indices, the key will equal 0.
    return key


def count_boxes(key: List[int]) -> Tuple[List[int], int]:
    current_length = 0
    lengths = [0] * len(key)
    started_at_one = False
    length_at_one = 0

    # Loop over all coefficients and keep track of the length of the zero-box
    # we're currently in.
    for i in range(0, len(key)):
        if key[i] == 0:
            if i == 1 and current_length == 0:
                started_at_one = True

            # The current zero-box extends to this index.
            current_length += 1
            lengths[current_length] += 1
        else:
            if started_at_one:
                length_at_one = current_length
                started_at_one = False
```

```
            # The current zero-box ends at the previous index.
            current_length = 0

    return lengths, length_at_one


if __name__ == '__main__':
    trials = 1000000
    params = [(401, 113), (541, 49), (449, 134), (613, 55), (677, 157),
              (887, 81), (1087, 120), (1171, 106), (443, 143), (743, 247)]

    for n, d in params:
        print(f"checking parameters (n, d) = ({n}, {d}):")
        counts = [0] * n
        counts_at_one = [0] * n

        for i in range(0, trials):
            key = generate_key(n, d)
            lengths, length_at_one = count_boxes(key)

            if i == 0 or (i + 1) % (trials // 1000) == 0:
                print(
                    f"\rfinished {i + 1:>9} of {trials} iterations,"
                    f" {100 * i / trials:.1f}%", end='\r')
            elif i == trials - 1:
                print()

            for k in [4, 8, 16, 32]:
                # Check if there is at least one zero-box of length k.
                if lengths[k] > 0:
                    counts[k] += 1

                    # Check if the only zero-box of length k starts at index 1.
                    if lengths[k] - int(length_at_one >= k) > 0:
                        counts_at_one[k] += 1

        for k in [4, 8, 16, 32]:
            print(f" k = {k:>2} : {counts[k]:>9} {counts[k] / trials:.10f} i=/="
                  f"1: {counts_at_one[k]:>9} {counts_at_one[k] / trials:.10f} ")
```

Bibliography

[1] E. Rescorla, 'The Transport Layer Security (TLS) Protocol Version 1.3', RFC Editor, RFC 8446, 2018. [Online]. Available: https://tools.ietf.org/html/rfc8446.

[2] P. W. Shor, 'Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer', *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997. DOI: 10.1137/S0097539795293172.

[3] E. Martín-López, A. Laing, T. Lawson, R. Alvarez, X.-Q. Zhou and J. L. O'Brien, 'Experimental realization of Shor's quantum factoring algorithm using qubit recycling', *Nature Photonics*, vol. 6, pp. 773–776, 2012. DOI: 10.1038/nphoton.2012.259.

[4] National Institute of Standards and Technology. (2017). Call for Proposals, [Online]. Available: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals (visited on 23/07/2019).

[5] J. Hoffstein, J. Pipher and J. H. Silverman, 'NTRU: A ring-based public key cryptosystem', Algorithmic Number Theory Symposium 1998, in *Lecture Notes in Computer Science*, vol. 1423, Springer, Berlin, Heidelberg, 1998, pp. 267–288. DOI: 10.1007/BFb0054868.

[6] A. Hülsing, J. Rijneveld, J. M. Schanck and P. Schwabe, *NTRU-HRSS-KEM, Algorithm Specifications And Supporting Documents*, 2017. [Online]. Available: https://ntru-hrss.org/data/ntrukem.pdf (visited on 24/07/2019).

[7] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone and Y.-K. Liu, 'Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process', National Institute of Standards and Technology, Interagency/Internal Report 8240, 2019. DOI: 10.6028/NIST.IR.8240.

[8] P. C. Kocher, 'Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems', Advances in Cryptology – CRYPTO '96, in *Lecture Notes in Computer Science*, vol. 1109, Springer, Berlin, Heidelberg, 1996, pp. 104–113. DOI: 10.1007/3-540-68697-2.

[9] J. Hoffstein, J. Pipher and J. H. Silverman, *An Introduction to Mathematical Cryptography*, 2nd ed., ser. Undergraduate Texts in Mathematics. Springer-Verlag New York, 2014. DOI: 10.1007/978-1-4939-1711-2.

[10] A. K. Lenstra, H. W. Lenstra and L. Lovász, 'Factoring polynomials with rational coefficients', *Mathematische Annalen*, vol. 261, no. 4, pp. 515–534, 1982. DOI: 10.1007/BF01457454.

[11] C. Chen, J. Hoffstein, W. Whyte and Z. Zhang, *NIST PQ Submission: NTRUEncrypt, A lattice based encryption algorithm*, 2018. [Online]. Available: https://assets.onboardsecurity.com/static/downloads/NTRUEncrypt-may-20-18.zip (visited on 20/07/2019).

[12] D. Stehlé and R. Steinfeld, 'Making NTRU as Secure as Worst-Case Problems over Ideal Lattices', Advances in Cryptology – EUROCRYPT 2011, in *Lecture Notes in Computer Science*, vol. 6632, Springer, Berlin, Heidelberg, 2011, pp. 27–47. DOI: 10.1007/978-3-642-20465-4_4.

[13] M. F. Schilling, 'The Longest Run of Heads', *The College Mathematics Journal*, vol. 21, no. 3, pp. 196–207, 1990. DOI: 10.2307/2686886.

[14] L. Gordon, M. F. Schilling and M. S. Waterman, 'An extreme value theory for long head runs', *Probability Theory and Related Fields*, vol. 72, no. 2, pp. 279–287, 1986. DOI: 10.1007/BF00699107.

[15] 'IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices ', Institute of Electrical and Electronics Engineers, IEEE Standard 1363.1-2008, 2009. DOI: 10.1109/IEEESTD.2009.4800404.