

University of Technology Darmstadt
Department of Computer Science
Cryptography and Computeralgebra

Bachelor Thesis

November 2007

Side Channel Attacks on NTRUEncrypt



Nikolay Vasilev Vizev

University of Technology Darmstadt
Mathematics with Computer Science

Supervised by | Prof. Dr. Johannes Buchmann,
Richard Lindner,
Daniel Shepers

Contents

1	Introduction to NTRU Encryption Algorithm	1
1.1	Basic terms	1
1.2	Encryption	2
1.3	Decryption	3
1.4	Why Decryption works	3
1.5	Product Form variant(Application to Other Keys)	4
1.6	Security	4
1.7	Choice of N p and q	5
1.8	Generalization	6
2	Side-Channel Attacks	7
2.1	Introduction	7
2.2	Timing Attacks	8
2.3	Power Analysis Attacks	8
2.4	EM Analysis	11
2.5	Template Attacks	12
3	Timing Attacks on NTRUEncryption	13

Introduction

The following paper concerns one of the new public key encryption algorithms (NTRUEncrypt). Along with it, side channel attacks are introduced. At the end of this paper we will see how the side channels effect NTRUEncrypt, and some countermeasures will be introduced too. Let us first of all see what NTRUEncrypt is.

1 Introduction to NTRU Encryption Algorithm

NTRUEncrypt is an asymmetric key encryption algorithm for public key cryptography. It was invented in the mid-1990s and is patented and endorsed by NTRU Cryptosystems Inc., which was founded in 1996 by Joseph H. Silverman, Jeffrey Hoffstein, Jill Pipher and Daniel Lieman, four mathematicians from Brown University. The algorithm relies on the presumed difficulty of factoring certain polynomials in special rings, into a quotient of two polynomials, having very small coefficients.

1.1 Basic terms

We explain how the public and the private NTRU keys are generated. We first need to introduce some basic terms. We start by giving an explanation of the algorithm's name. There are various opinions and versions, about what the abbreviation NTRU stands for. Some say that it does not mean anything. Other say that the name NTRU is short for N -th degree truncated polynomial over the ring R (sometimes referred to as the ring of convolution polynomials), or in mathematical notation

$$R_m := (\mathbb{Z}/m\mathbb{Z})[X] / \langle X^N - 1 \rangle.$$

This became the most common interpretation of NTRU. We define the multiplication on R , to be the $*$ convolution operation. During encryption and decryption, at various stages of these processes, the coefficients of the polynomials are reduced modulo q and/or modulo p , where p and q are relatively prime integers (in some *cases* p is not an integer at all, instead it is a polynomial for example of the form $X + 1$). Reduction is done, so that the polynomial coefficients are in the range from 0 to $q - 1$ (and respectively from 0 to $p - 1$, where p is a number). In general, the reduction operations modulo the relatively prime p and q do not commute. What is meant, is described by the following example:

$$(11 \bmod 7) \bmod 2 = 4 \bmod 2 = 0$$

$$(11 \bmod 2) \bmod 7 = 1 \bmod 7 = 1.$$

Let us also define two particular types of polynomials, to ease notation:

$$\begin{aligned}\beta_N &:= \{d_1X^{N-1} + d_2X^{N-2} + \dots + d_{N-1}X^1 + d_N1 \in R, d_i = \{0, 1\}\} \\ &\quad \text{(the set of binary polynomials)} \\ \beta_N(d) &:= \left\{d_1X^{N-1} + \dots + d_N1 \in R, d_i = \{0, 1\} \text{ and } \sum d_i = d\right\} \\ &\quad \text{(the set of binary polynomials with exactly } d \text{ ones)}\end{aligned}$$

We can now give a proper definition for the NTRU private key. An NTRU private key consists of a pair of (binary) polynomials f and g . The associated public key is the polynomial:

$$h := p * f_q^{-1} * g \pmod{q}$$

where f_q^{-1} denotes the inverse of f modulo q . Similarly we let f_p^{-1} denote the inverse of f modulo p . The polynomial f is often taken in the form $f = 1 + pF$ where $F \in \beta_N$ to speed the process of decryption.

1.2 Encryption

In this subsection we will proceed with the encryption algorithm. We will define the four main steps and explain them.

Encryption uses two hash functions. We denote them by G and H . In Practice they are built using either SHA-1 or SHA-256 (Secure Hash Algorithms) in various ways, depending on the desired security level. The encryption process works as follows.

Step 1: We take the padded plain text M (the text we want to encrypt) and encode it into β_N :

$$M \in \beta_N$$

Step 2: With the help of the hash function G we randomize the padded text M , into the binary polynomial with exactly d_r ones:

$$r = G(M) \in \beta_N(d_r)$$

Step 3: In this step we calculate m' , the masked message representation:

$$m' = M \oplus H(r * h \pmod{q})$$

Step 4: At the last step we get the cipher text e :

$$e = (r * h + m') \pmod{q}$$

1.3 Decryption

In this subsection we will explain how to decrypt a given cyphertext, and how to verify, that a given pair (m', e) is valid.

NTRUDecryption algorithm uses the same two hash functions G and H .

Step1: The decryption algorithm first recovers the message representative m' :

$$m' = ((f * e \bmod q) \bmod p) * f_p^{-1} \bmod p$$

Step2: In the second step the decryption algorithm recovers and the plain text M

$$M = m' \oplus H(e - m' \bmod q)$$

Step3: Once we have calculated M and m' we use them to recreate the blind value r

$$r = G(M)$$

Step4: We have to verify, that (m', e) is a valid NTRUEncryption pair:

$$e = r * h + m' \bmod q$$

1.4 Why Decryption works

To understand better why and how the whole algorithm functions, let us explain why Decryption works. The polynomial a that we compute satisfies

$$\begin{aligned} a &= f * e = f * r * p * h + f * m' \bmod q \\ &= f * f_q^{-1} * p * r * g + f * m' \bmod q \\ &= p * r * g + f * m' \bmod q \end{aligned}$$

Decryption works because for all the coefficients of $p * r * g + f * m$ we have chosen q big enough such that these coefficients lie within the range $(0, q]$. If we define reducing mod q as reducing into this range, we leave all the coefficients unchanged. This means that when we reduce the coefficients of $f * e$ modulo q into the interval from 0 to q we recover exactly the polynomial

$$a = p * r * g + f * m' \bmod q.$$

Reducing a modulo p then gives us the polynomial $f * e$ and multiplication by f_p^{-1} retrieves the message $m' \bmod p$. For detailed and further explanation see[2][3].

1.5 Product Form variant(Application to Other Keys)

In this subsection (not going into details) we will show that there is a possibility for NTRU private keys to have a form other than random binary polynomials with d ones. For example they may be of the form

$$f = f_1 * f_2$$

or

$$f = f_1 * f_2 + f_3$$

By using more key spaces we aim to make our private key even more secure. It is even faster when the parameters are chosen according to *ieee* standards. In particular the idea and application are the same as for one private key(except that we have more polynomials in this case).We will briefly mention only the reason why decryption works. In this variant the polynomial, that we want to calculate is the same

$$a = f * m' + r * p * g,$$

. Till now there are no known attacks based on product form structure(at least not in public). Let us focus on the normal NTRUEncryption private key again.

1.6 Security

Once we have seen how encryption and decryption work, let us talk about the security of this algorithm. That is exactly what we do in this subsection — we discuss briefly the main weak point of NTRU. For more focused and detailed discussions, based on the NTRU security, we will spend most of the time in the second and the third sections.

The number of SHA calls required by G depends on the input to G . Thus by measuring decryption time, an attacker may obtain information about the input to G , which in turn reveals information about the private key f .

The number of hash calls required to create the blinding value r from a message representative/ciphertext pair $(m'; e)$ may be different for different pairs. Each hash call requires a nontrivial amount of time, so an adversary might be able to determine how many hash calls we have used in attempting to decrypt a (possibly bogus) ciphertext e . This gives the chance to an attacker to find our private key.

1.7 Choice of N , p and q

Careful choice of parameters is necessary to avoid some published attacks. For this purpose we will use the research work of the IEEE P1363 (Institute of Electrical and Electronics Engineers standardization project for public key cryptography). To begin we shall say, that there are different levels of security, at which each key encryption algorithm works. The lowest acceptable level of security at present time is the 80-bit security. We will use it to give exact values for N , p and q .

We already know that NTRUEncryption polynomial multiplication is convolution multiplication: polynomial terms of degree $> N$ are reduced $(\text{mod } X)^N - 1$. As already mentioned we use the information, achieved by IEEE P1363. The newest known research says, that for 80-bit security N should be at least = 251. Since we have mentioned, that the discussed security level is the lowest we have to examine how this reacts on the value of N , how it changes. With increasing the security level, N increases roughly linearly (with k for k -bit security.)

In this paragraph we determine the q term in the same manner. All coefficients in a truncated polynomial are reduced mod q . This requires from q to be "big". What does this mean? For example in case of 80-bit security level, again using information from the same research, q should be at least = 193. With increasing security level this parameter increases roughly linear.

The term p is used for reduction step in the NTRUDecryption. It is a "small" modulus. $p = 2, 2 + X$ or 3 for all security levels. In the practice, since all the computations are done by computers, the most suitable value of p is 2. So from now on we set p to be 2 for all security levels.

N	q	p	k bits-security level
251	193	2	$k = 80$
347	$193 + k$	2	$k = 112$
397	$193 + k$	2	$k = 128$
491	$193 + k$	2	$k = 160$
587	$193 + k$	2	$k = 192$
787	$193 + k$	2	$k = 256$

[1]

1.8 Generalization

In this first section we have introduced the NTRU asymmetric key encryption algorithm. We have gone through both encryption and decryption (step by step). We have seen, that both of them use only simple polynomial multiplication. The operations during these two processes (encryption and decryption) are very fast compared to other asymmetric encryption schemes (RSA, El Gamal). NTRU algorithm also requires a low memory. This makes NTRU as a whole a lot faster than other known algorithms.

Despite NTRU's limited practical usage, taking everything into consideration, NTRU is a well-accomplished, widely usable, promising cryptosystem. However, NTRUEncrypt has not yet undergone a comparable (to the already mentioned RSA and El Gamal asymmetric encryption schemes) amount of cryptographic analysis.

2 Side-Channel Attacks

In the following section we will get to know the Side-Channel Attacks[4]. We will introduce four main groups of types attacks, and we will explain them separately.

2.1 Introduction

Maintaining information integrity and confidentiality is a main security requirement in almost all systems. These goals are realized using a combination of authentication, mandatory access control, use of high-assurance trusted components, encryption and secure communication protocols.[4]

In most commercial environments these techniques are applied and analyzed only with respect to an abstract model of the system being protected. That is why gaps between the model and actual implementation can give the option to some attacks to bring a system down and reveal some (if not the whole) private information (for example the pin code of a bank card). Badly designed and implemented systems are full of gaps and weak parts. For example, many abstract models assume a secure operating environment. Such models can easily be compromised by attacking the operating environment. Sometimes the abstract models do not fully specify the error conditions and appropriate responses. This leads to attacks, where sensitive information can be extracted by analyzing the error messages returned by the implementation. In many cases, because of poor quality control, the actual functionality of an implementation differs from its specifications. Part of these errors may permit an attacker to use data driven attacks; carefully chosen data could be used to subvert the implementation via buffer overflows, script injection, and so on.

Poorly designed and implemented systems are expected to be insecure, but most well-designed and implemented systems also have subtle gaps between their abstract models and their physical realization due to the existence of side channels. A side channel is a potential source of information flow from a physical system to an adversary, beyond what is available via its abstract model. These side channels could be as subtle as externally observable timing of certain operations, the electromagnetic (EM) radiation emanating from the computing equipment, the power being consumed by the equipment, the analog signals being propagated on the communication and ground lines and other conducting surfaces attached to it, the acoustic emanation from mechanical components and peripherals, the light and heat emanating from the equipment and peripheral devices, and so on. All these side channels have been demonstrated to provide enough information to break system security.[4]

2.2 Timing Attacks

The idea of using timing of operations as a crypt-analytic tool was proposed by Kocher (1996), who showed how it could be used to attack implementations of Diffie-Hellman, RSA, DSS, and other cryptographic algorithms. Before timing attacks were discovered, designers of cryptographic implementations focused primarily on performance and employed many optimizations to reduce unnecessary computation. These resulted in implementations that executed different sets of instructions and had different timings for different values of the secret key and the input data, even if the data and key sizes were fixed. Timing variation can exist even in the absence of explicit optimizations because different processing sequences can lead to different timing behavior from the hardware and the operating environment. For example, timing variation due to variable cycle instructions, pipeline stalls, hardware cache misses, page faults, and so on are strongly dependent on the processing sequence.

So far we have ignored the timing noise and variations and measurement errors that inevitably arise when measuring the timing of operations on a remote system under varying load. It turns out that it is still possible statistical techniques to overcome the effect of the added timing noise at the cost of requirement many more timing samples. A simple rule of thumb is that if the added noise reduces the signal-to-noise ratio by a factor of n , then the number of samples needed for the attack have to increase by a factor of n^2 . There is a countermeasure against timing attacks. Such one is to ensure that the timing of an operation is fixed. However there are many cases where fixing the operation timing is not feasible or practical. For example the timing leakage from an implementation may depend on the specifics of the hardware, operating system, and so on, and creating tailor-made implementations for every single platform is not commercially viable. In such cases, an alternative is to develop an implementation in which the timing is either independent of secret information such as keys or is dependent only on a limited amount of information about the keys, and disclosure of that limited information is safe.[4]

2.3 Power Analysis Attacks

A few years after the discovery of timing attacks, Kocher, Jaffe and Jun introduced power analysis (1998), a far more devastating type of side-channel attack that has had a profound impact on the entire smart-card industry. This attack extracts information from cryptographic devices by analyzing their instantaneous power consumption. For many large systems using filtered power supplies, this may not be an issue. The filters substantially reduce the information available externally, and bypassing them may require physical access to the system that the adversary may not have. However,

such attacks are a big problem for tamper-resistant devices that are required to withstand physical attacks and an even bigger problem for chip card that are further constrained to draw power from an intrusted terminal without further filtration. Although the first power analysis attacks were mounted against chip cards, there has been recent interest and success in attacking other devices.

Power analysis attacks are easy to mount against chip cards. An attacker can attach a current probe inside the card terminal to tap the power line that feeds the chip card and connect the probe to a digital oscilloscope. In addition, the attacker may need to install additional software and hardware components to trigger the digital oscilloscope to collect samples of the power signal at the right time. The power signal is sampled at a frequency that is typically 10 to 100 times the chip card's clock frequency.[4]

Information Within Power Consumption Signals

Compared with the timing side channel, the power consumption signal from even a single simple operation contains an overwhelming amount of information. For example, a short computation involving a few hundred instructions could generate several tens of thousands of power samples. By plotting these power samples within respect to time the power consumption characteristics of the computation at different times at various levels of granularity can be viewed.

At the macro level, that is, when we plot the power signal over a longer period of time, one can easily discern the structure of the execution sequence, because the shape of the power consumption signal depends on the execution sequence. For instance, iteration of a loop or repeated sequence of similar operations show up as a repeating structure in the power signal.

At the micro level, that is, when we zoom into a small portion of the computation, we can view the cycle and instruction level power function. The power consumption for each clock cycle and instruction has a basic shape that is determined by the current and a limited prior history of values, addresses, and location of the code, and the addresses, values, and location of the operands and results. This basic shape is further perturbed to some extent by noise. In the coming paragraphs we will get acquainted with two power analysis attacks - *simple power analysis* and *differential power analysis*.[4]

Simple Power Analysis (SPA)

SPA uses the shape of the power signal during a single execution of a cryptographic algorithm to extract information about the secret key. The shape of the power signal can leak information about the key in many ways. If an implementation performs key- dependant conditional operations, then the

shape of the power signal can reveal the execution sequence and hence the values of the key-dependant conditions. For example, in the RSA square and multiply algorithm, the sequence of modular squares and multiplies is key-dependant. Because the implementation of modular squaring can be different from modular multiplication, the square-and-multiply sequence and hence the secret exponent may be discernable from the power signal. *SPA* may also be possible if an implementation manipulates key-dependant data using instructions that leak a lot of information into the power channel. For example, in some chip-cards, instructions affecting the carry bit leak its value in the power channel.

Although *SPA* is a very strong attack, it requires some easily observable information leakage in the power signal. This makes it easy to defend against. With a little work, most algorithms can be implemented using low-leakage instructions and a fixed instruction execution sequence. This fixes most of the factors that effect the power signal at any cycle, such as the current and prior values, addresses and location of code. When such implementations are executed with different keys and data, the only variation possible at a cycle level would be due to variation in the prior and current values or addresses of operands and results. Such a variation produces only a minor variation on the shape of the power signal at a cycle, and such minor variations are easily obscured by noise. Therefore, little usable information is available by inspecting a single power sample, and *SPA* becomes ineffective. To overcome such a defence, Kocher proposed *differential power analysis*.^[4]

Differential Power Analysis (DPA)

DPA is a far more sophisticated attack technique based on statistical analysis of a large number of power traces with different data. The technique combines statistical hypothesis testing together with details of the algorithm being attacked to derive information about the secret keys. The basic premise for a *DPA* attack is that the shape of the power signal of an instruction that is processing an operand or result depends on (or correlates with) the individual bits of the value and address of the operand or result. Although such a dependance on a particular bit may be small and easily masked by the values of other bits and by noise, the right statistical test will average out and remove these masking influences to expose the correlation, given enough power samples. It is a fact though, that the *DPA* is based only on the specification of a given algorithm, without requiring any knowledge of the implementation. This property of being major advantage of *DPA*-style attacks and the main reason that such attacks have such a major impact on the chip-card industry.^[4]

2.4 EM Analysis

Among all side channels this is the one with the longest history. The first researches in this area date back to 1982. However the first openly published works on *EM* analysis of cryptographic devices took place in 2001. They were limited to chip cards and required an adversary to be in close proximity to the chips being attacked. In fact, the best attacks required the decapsulation of the chip packaging and the careful positioning of micro antennas on the layer of the chip substrate. Subsequently in 2002 the work of Agrawal, Archambeault, Rao and Rohatgi removed these limitations. They showed that *EM* analysis provides an avenue for attacking cryptographic devices from a distance where the power line is inaccessible and also that *EM* emanations can leak information not readily available from the power side channel.

We notice two broad classes of *EM* emanations: *Direct* and *Unintentional*. The first class are result from *intentional* current flows within circuit. Normally these flows consist of short bursts of current with sharp rising edges that occur during the switching operation and the result in emanations observable over a wide frequency band. Often higher frequency emanations are more useful to the attacker because there is substantial noise and interference in the lower frequency bands. In complex circuits, it may be difficult to isolate direct emanations, because of interference from other signals. Reducing such interference requires tiny probes positioned close to the signal source or special filters to separate the desired signal from other interfering signals.

The second class of emanations are collected by devices, which pack a large number of circuits and components in a vary small area and suffer from numerous unintentional electrical and electromagnetic couplings between components. The vast majority of those couplings are small, do not effect the functioning of these devices and are typically ignored by circuit designers. Such couplings, however, are rich source of compromising emanations. These emanations manifest themselves as *modulations* of the carrier signal generated, present, or introduced within the device.

While comparing both emanation classes we see, that because of the difficulty of isolating direct emanations, by unlocking the power of the *EM* side channel requires the exploitation of unintentional emanations rather than the direct emanations. This is because some modulated carriers are much stronger and propagate much further than direct emanations. This enables attacks to be carried out at a distance without resorting to any invasive techniques.

In addition to the *EM* analysis we give some equipment needed for the attack without going into further details concerning the *EM* side channel. The *critical* Peace of equipment for *EM* is a tunable receiver-demodulator that can be tuned to various modulated carriers and can perform demodulation

to extract the sensitive signal. Once the best signal to attack is identified , a custom, non tubal receiver demodulator for the attack can be built quiet cheaply. Picking up *EM* signals also requires the use of *EM* field probes and antennas appropriate for the band being considered.[4]

2.5 Template Attacks

Template Attacks attempt to extract the maximum amount of information from each side channel sample to attack implementations in which the adversary is limited to one or only a few compromising samples. A key requirement for template attacks is that the adversary should have a programmable experimental device identical to the device being attacked. Although such an assumption is limiting, it is practical in many cases, especially if an adversary has access to another device from the sam manufacturing run or if the adversary has shared access to the same device but is limited to execute algorithms with only his own key.

The side channel attacks are a powerful weapon, which once known how to be applied and once triggered could make enormous damages. The side channels have their weaknesses though. In the section above we have seen some of them, but still if we are not careful enough, we can be a victim of one of these tricky attacks. To illustrate that we will see in the following section how the timing attack is used to reveal to an adversary the private key f . [4]

3 Timing Attacks on NTRUEncryption

This part concerns timing attacks through side channels applied on the NTRUEncryption key algorithm. We saw briefly what the side channel attacks do, and how they are applied. We also know how NTRUEncryption works. By using its weak fragments we can find the private information, that this key algorithm keeps unknown, with the use of already defined side channels, namely the timing side channel attack.[1]

Time trail of a Cipher text

Let us define first of all a term called *time trail*(to ease explaining and understanding how the timing attack functions). From section 1 we recall the blinding value r . For creating it we have used hash calls on message representative pair (m', e) , the number of which (calls) may be different for a different message representative. The time required for a hash call though is nontrivial. So at this point an adversary might be able to determine how many hash calls are used in attempting to decrypt a cipher text e . In practice there exists a number K , such that the numbers of hash calls needed for computing r is either K or $K + 1$ [1]. We also define for each message representative pair (m', e) an output $r(m', e)$ from the decryption algorithm.

$$r(m', e) = G((m' + H(e - m' \bmod q)) \bmod 2).$$

We define

$$\beta(m', e) \in \{0, 1\}$$

. We set it to be 0 in case that it takes at most K hashes to create $r(m', e)$, and one if more than K hashes are needed. We also look at the rotations $(X^i m', X^i e)$ for $i = 0, 1 \dots$. We are ready now to give a full definition of what *time trail* is.

It is a binary vector of the following form[1]:

$$T(m'e) = ((\beta(m', e), \beta(Xm', Xe), \beta(X^2m', X^2e), \dots, \beta(X^{N-1}m', X^{N-1}e))$$

The *time trail* gives us how many hashes are required for each of the rotations of the pair (m', e) At the beginning of the paragraph we said that the number of the required hash calls may be different for different pairs. At this point we can tell with what probability two pairs have the same *time trail* and respectively how many hash calls. Let P be the probability for a random pair (m'_1, e_1) to require at most K hash calls. Let on the other hand $1 - P$ be the probability of other random pair (m'_2, e_2) that requires at least $K + 1$ hash calls. If P is big enough then the probability that those two pairs have the same *time trails* is quite small. More precisely the probability is represented by the formula[1]:

$$Prob(T(m'_1, e_1) = T(m'_2, e_2)) = (1 - 2P + 2P^2)^N$$

The formula is derived in the following way. We have seen that the *time trail* is a binary vector of length N . Let P is the probability that at the i -th position of the binary vector we have 0. We set $1 - P$ to be the probability for a 1 at the j -th position. Then the probability, that the first position of two random *time trails* are the same is

$$Prob(both = 0) + Prob(both = 1) = P^2 + (1 - P)^2 = 1 - 2P + 2P^2$$

In order two entire *time trails* to be identical, they must have at all N positions the same number (0 or 1). Thus we have[1]:

$$Prob(T(m'_1, e_1) = T(m'_2, e_2)) = (1 - 2P + 2P^2)^N$$

The *time trail* is a key term in timing attacks concerning the NTRU-Encryption algorithm. In the following subsection we will explain how and when such an event occurs.

A timing attack based on variable number of hash calls

To understand easily how the timing attack functions imagine the following situation. Let take three characters A and B . They exchange information between themselves until one of them decides to act like an adversary and tries to get to know the private key of the other character. This is done, using the timing attack. Let say that the adversary is B . He chooses a collection of cipher texts ε . This is nothing but a collection of polynomials mod q . He also chooses a set of message representatives M . He chooses M in a way that it contains many of the message representatives that A creates, namely the following set :

$$\{(f * e \text{ mod } q) \text{ mod } 2 * (f^{-1} \text{ mod } 2) : e \in \varepsilon\}$$

Further more we assume that the probability, that a message representative, created by A to be found in the set M is quite big. Before starting the active part of the attack, B creates a table with the *time trails* of every pair in $M \times \varepsilon$. In other words, he creates a searchable list of binary vectors:

$$(T(m', e)) : m \in M \text{ and } e \in \varepsilon$$

The attack starts when B sends to A some random $e \in \varepsilon$. Once the cipher text is sent, B starts to record how long A needs to decipher it. The main idea at this point is that the adversary does not care if he has sent a bogus cipher text which of course after failure check will be rejected. The only thing that matters is to obtain timing information. The information gained gives him the opportunity to determine the number of hash calls. In other words he now knows the value $m'(e)$ which is defined to be:

$$((f * e \text{ mod } q) \text{ mod } 2) * (f^{-1} \text{ mod } 2).$$

The adversary does not know the value of $m'(e)$. B only knows at this point the value of $\beta(m'(e), e)$, which he will use afterwards, when comparing the values obtained from A with those pre-computed and already stored in the table, that B has already built. For the purpose B needs $N - 1$ more entries that he will put in the time trail. Those values he obtains after sending one after the other the following polynomials :

$$Xe, X^2e, X^3e, \dots, X^{N-1}e$$

After each cipher text sent the adversary obtains the values $\beta(m'(X^ie), X^ie)$ for $i = 0, 1, 2 \dots, N - 1$.

Let us take one random term $m'(X^ie)$. Since we know to what $m'(e)$ is equal to, we can give proper expression for $m'(X^ie)$ too :

$$m'(X^ie) = ((f * X^ie \text{ mod } q) \text{ mod } 2) * (f^{-1} \text{ mod } 2)$$

Since all X^i 's are 0 or 1 we can take the X^i term at the front and what we get is :

$$X^i * ((f * e \text{ mod } q) \text{ mod } 2) * (f^{-1} \text{ mod } 2)$$

which in fact is:

$$X^i m'(e).$$

B now has a time trail $T(m'(e), e)$ of $(m'(e), e)$. The next step to be done is to search into the pre-computed list in which with a high probability finds a small amount of possible pairs for the pair he had obtained. In fact he has something more. B got two polynomials e and m' , that when A decrypted e got the second, m' polynomial. From here the attacker has only to calculate $m' * f \equiv (f * e \text{ mod } q) \text{ mod } 2$ where e and m' are known. In the following subsection we will see how the attacker can gain some advantage for the private key of A . The future exploiting the private key f will depend on the specific form of e . For example, if the elements of ε consist of polynomials with very few nonzero coefficients, then the equation $m' * f \equiv (f * e \text{ mod } q) \text{ mod } 2$ may give information concerning the spacing between the nonzero coefficients of f . In the following subsection we describe a specific collection ε that leads to a practical hash timing attack when the key f has the form $f = 1 + 2F$.

Timing attack for $f = 1 + 2F$

We know from section 1 that in most cases the parameter p is $2[3]$. This means that q is odd, and further more the private key f is of the form:

$$f = 1 + 2F \text{ for some binary polynomial } F \in \beta_N(d_F)$$

For later computations, we write $F = \sum F_i X^i$ with $F_i \in \{0, 1\}$. We define a new term $\lambda = \lceil q/4 \rceil$. In this particular case the starting point is again when B sends to A a cipher text e , but in this case we have the set $\varepsilon =$

$\{\lambda + \lambda X^i : 1 \leq i \leq N\}$. This means that a random e is a polynomial too, but in this case with two coefficients equal to λ and all other coefficients are 0's. Let see briefly how B is conducting the attack against A 's private key[1]:

1. Choose a value δ .
2. Let $\varepsilon = \{e_i = \lambda + \lambda X^i : 0 \leq i \leq (N - 1/2)\}$ and $M = \beta_N(0 \leq d \leq \delta)$
3. Pre-compute and store all the *time trails* $T(m', e)$
4. To A are sent $X^j e_i$ for $j = 0, 1, \dots, N - 1$ and use the decryption times to determine the *time trail* $T(m'(e_i), e_i)$.
5. Search for candidates
6. Use the resulting value $m'(e_i)$ to reconstruct F

At this point we need to find out the possible values of $m'(e)$ from step (2). For the purpose let go through the decryption that A makes. At the beginning A first computes[1]:

$$a = f * e \text{ mod } q \equiv (1 + 2F) * (\lambda + \lambda X^i) \text{ (mod } q) \equiv \lambda + \lambda X^i + \sum_{j=0}^{N-1} 2\lambda(F_j + F_{j-i})X^i$$

Thus for the j^{th} coefficient of a we have the following function[1]:

$$a_j = \begin{cases} \lambda(1 + 2F_0 + F_{-i} \text{ mod } q) & \text{if } j = 0, \\ \lambda(1 + 2F_i + 2F_0) \text{ mod } q & \text{if } j = i, \\ \lambda(2F_j + 2F_{j-i}) \text{ mod } q & \text{if } j \neq 0, i \end{cases}$$

While looking at the above function, we shall make clear, that since $\lambda = 2\lceil q/8 \rceil$ this means that λ is slightly bigger than $q/4$, and therefore the right side of the above function takes values smaller than $q - 1$. Thus we do not need to reduce $a \text{ mod } q$, except when we have the case $F_j = F_{j-i} = 1$. Then we apply a nontrivial reduction after which we have new values for a_j [1]:

$$a_j = \begin{cases} \lambda, 2\lambda \text{ or } 3\lambda & \text{if } F_j = 0 \text{ or } F_{j-i} = 0 \\ 4\lambda - q \text{ or } 5\lambda - q & \text{if } F_j = F_{j-i} = 1 \end{cases}$$

After reducing $a \text{ mod } 2$ we got 0's and 1's, and more precisely since λ is even and q odd, after reduction step we get[1]:

$$a_j = \begin{cases} 0 & \text{if } F_j = 0 \text{ or } F_{j-i} = 0 \\ 1 & \text{if } F_j = F_{j-i} = 1 \end{cases}$$

Now we can define explicitly the $m'(e_i)$ terms[1]:

$$m'(e_i) = \sum_{j=0}^{N-1} \begin{pmatrix} 1 & \text{if } F_j = F_{j-i} = 1 \\ 0 & \text{otherwise} \end{pmatrix} X^j$$

which in terns yields the following partial information about F [1]

$$F(e_i) = \sum_{j=0}^{N-1} \begin{pmatrix} 1 & \text{if } m'(e_i)_j = 1 \\ & \text{or } m'(e_i)_{i+j} = 1 \\ 0 & \text{if } m'(e_i)_{j-i} = 1 \text{ and } m'(e_i)_j \neq 1 \\ & \text{or } m'(e_i)_{j+2i} = 1 \text{ and } m'(e_i)_{j+i} \neq 1 \\ ? & \text{otherwise} \end{pmatrix} X^j$$

Validating a choice

The initial set ε (see [1]) is relatively small to reduce pre-computation. Recognizing a *time trail* tells with high probability to an attacker that he had identified $m'(e_i)$ for the relevant e_i in his database. However if there is a nontrivial chance that the *time trail* is unique, B may want to check that he has in fact identified the correct e_i . To do that we note that if $e_i = \lambda + \lambda X^i$ decrypts to m' , then so do the alternate forms:

$$e_i^* = (\lambda + 2) + \lambda X^i, \text{ or } \lambda + \lambda X^i \text{ or } \dots$$

or indeed many polynomials $\lambda_1 + \lambda_2 X^i$ with λ_1 and λ_2 even integers in the vicinity of $q/4$ and satisfying $\lambda_1 + \lambda_2 \geq q/2$. B therefore selects one of the possible e_i^* , calculates the *time trail* $T(m', e_i^*)$ for the message representative m' that he thinks is produced by decrypting the original e_i , and then submits e_i^* to the pre-computed *time trails* to find its time. If the measured and the calculated *time trail* match, he has confirmed the guess for m' . Otherwise the original match on the *time trail* is simply a coincidence.

Practical Hash Timing Attack for $f = 1 + 2F$

Let us go further and see how does this attack functions practically for private key of the same form $f = 1 + 2F$. For the purpose we will use some particular parameter sets. As before this practicality depends, among other things, on the probability that different inputs require different number of hash calls. We begin by describing how the hash calls are used to compute r (section 1) and then we compute the probability that this process takes a different number of hash calls. The blinding value r (a binary polynomial with exactly d_r ones) is created from a hash function via repeated calls to some version of SHA (already given example in section 1.2). Here is the process[1]:

1. Take value c that satisfies $2^c \geq N$. Also let $b = \lceil c/8 \rceil$ and $n = \lfloor 2^c/N \rfloor$. b is the smallest integer such that b bytes contains at least c bits. Similarly n is the smallest multiple of N that is less than 2^c .
2. Call the specified version of SHA and separate the output into smaller pieces of length b bytes. Within each of the separated parts, keep the lower order c bits and discard the upper order $8b - c$ bits. Convert the lower order c bits into integers i_1, i_2, \dots, i_t . t is the integer such that the output of the specified version SHA consists of tb bytes.
3. create a list of indices j_1, j_2, \dots by looping through the list of i values from step 2. If $i \leq n$ and $i \bmod N$ is not already in the list, the adjoint $i \bmod N$ to the list, otherwise discard i . Continue until the list contains d_r values of j . If at any point there are no spare i values, then call SHA and create additional i values as specified in step 2.

In order to compute r , the algorithm described needs to create a list of d_r distinct numbers satisfying $0 \leq i \leq N$. Each time the algorithm uses hash calls, it gets t numbers satisfying $0 \leq i \leq 2^c$. Hence the probability that suffices to call to a SHA s times is equal to the probability that st random numbers in the range $[0, 2^c)$ contain at least d_r values in $[0, n)$ whose values modulo N are distinct. Hence [1]:

$$Prob(\text{calls to SHA suffices}) = Prob \left(\begin{array}{l} st \text{ randomly chosen integer in } [0, 2^c) \\ \text{includes at least } d_r \text{ values in } [0, n) \\ \text{that are distinct modulo } N \end{array} \right)$$

The closer the first probability to 0,5, the greater the chance that a *time trail* is unique. In most cases except perhaps for $k = 80$ and $k = 192$, it will not be necessary to validate a *time trail*.

Generalization

We have seen how the timing side channel attack could reveal the private key f from the NTRUEncrypt. We have also seen that this attack is only possible because of the fact, that for decrypting different cipher texts, we may need a different number of hash calls. Though that we have spent most of the time explaining how to attack a private key of the form $f = 1 + 2f$, it is reasonable to assume that there are similar attacks for more general keys. We have concentrated our focus on this particular kind of private key, mostly because of one reason - as we have seen in the first section, while we have been explaining the choice of parameters, we have said that for the parameter p we set this parameter to be equal to 2, simply because this value is most appropriate in the computer language.

The timing side channel can be beaten of course. We have seen in the second section how to do that. Again the answer is based on the same fact - for different cipher text, different number of hash calls may be used. The smartest thing to do is to fix the number of hash calls for each cipher text. What is meant is to calculate the number of hash calls for each possible cipher text. Then take the largest number and call it K_{max} . In case a random cipher text needs less hash calls than K_{max} , one should simply perform extra hash calls. More precisely if a cipher text requires K number of hash calls, do $K_{max} - K$ extra hash calls. In this way the number of SHA calls is the same for all cipher texts, which leads to avoiding the attack. This method of making the number of hash calls per message one and the same is called *padding*. Padding is not as simple as the example above - the *padding* has to be intermixed with the number of hash calls so that every bit of the *padding* affects every bit of the number of hash calls. While *padding* we have to be careful and to consider the security level. For example there are 80 bits of *padding* for 80-bit security, and with increasing the security level we also have to increase the *padding* bits.

Conclusion

In the chapters above we have seen that NTRUEncrypt is an asymmetric key encryption algorithm for public key cryptography. We have seen how the encryption and the decryption algorithms work, and that both of these algorithms use only simple polynomial multiplication. This gives NTRU an advantage in comparison to other asymmetric encryption schemes (RSA, El Gamal). Because of this fast simple polynomial multiplication the NTRU algorithm requires a low memory. This makes NTRU as a whole a lot faster than other known algorithms. We have seen the weaknesses too.

We have introduced the side channel attacks. We divided them into four main groups. We have explained briefly how they work. We have seen what software and hardware we need in case we want to use these attacks. In the final chapter we focused our attention on the timing attack. We have seen how it is applied on the already defined NTRUEncrypt, and of course a counter measure was defined too. We have seen that the timing side channel attack uses the different time, which is needed while hash calls are used. The base of the attack is simply the fact, that for different cipher text, different number of hash calls are used. We have used as a counter measure the *padded* scheme, which saves us from the timing side channel. There can be (and are) more side channel attacks, that can be applied on NTRUEncrypt, but NTRUEncrypt has not yet undergone enough amount of cryptographic analysis.

NTRU is a well-accomplished, widely usable, promising cryptosystem.

References

- [1] Joseph H. Silverman, William Whyte: NTRU Cryptosystems Technical Report Report 021, Version 1
Timing Attacks on NTRUEncryption via Variation in the Number of Hash Calls
- [2] S. Vanstone, P. Van Oorschot, A. Menezes: *Handbook of Cryptography* CRC Press, Boca Raton, (1996).
- [3] William Whyte, NTRU Cryptosystems: Choosing Parameter Sets for NTRUEncrypt with NAEP and SVES-3 February 17-th (2005)
- [4] Bidgoli, Hossein: Handbook of Information security (2006)