NTRU Cryptosystems Technical Report

Report # 019, Version 1
Title: Timing Attacks on NTRUEncrypt via Variation in the Number of Hash Calls
Author: Joseph H. Silverman, William Whyte
Release Date: February 2006

Abstract. This report studies timing attacks on NTRUEncrypt based on variable number of hash calls. The attacks apply to the parameter sets of [3]. We describe the attacks and describe simple countermeasures.

§Section 1. NTRUEncrypt overview

In this section we briefly review how NTRUEncrypt works in order to set notation. For further details, see [1,2,3]. Recall that NTRUEncrypt uses the ring of truncated polynomials (also sometime called the ring of convolution polynomials)

$$\mathbf{Z}[X]/(X^N - 1).$$

We denote multiplication in this ring by $*$. At various stages of encryption and decryption the coefficients of these polynomials are reduced modulo q and/or modulo p, where p and q are relatively prime integers. This reduction is always performed so that the reduced coefficients lie in the range from 0 to $p-1$ (respectively 0 to $q-1$). In particular, reduction modulo p and reduction modulo q do not commute with one another. For example,

$$(11 \bmod 7) \bmod 2 = 4 \bmod 2 = 0 \quad \text{and} \quad (11 \bmod 2) \bmod 7 = 1 \bmod 7 = 1.$$

For simplicity in this note, we restrict attention to the case $p = 2$, in which case various polynomials are chosen to be binary (i.e., all coefficients 0 or 1), and in some cases with a fixed number of zeros and ones. To ease notation, we let

$$\begin{aligned} \mathcal{B}_N &= \{\text{binary polynomials}\}, \\ \mathcal{B}_N(d) &= \{\text{binary polynomials with exactly } d \text{ ones}\}. \end{aligned}$$

An NTRUEncrypt private key consists of a pair of (binary) polynomials f and g. The associated public key is the polynomial

$$h = f_q^{-1} * g \bmod q,$$

where f_q^{-1} denotes the inverse of f modulo q. Similarly, we let f_p^{-1} denote the inverse of f modulo p. To speed decryption, the polynomial f is often taken in

the form $f = 1 + pF$ with $F \in \mathcal{B}_N(d_F)$, in which case $f_p^{-1} = 1$. See [2,3] for a discussion. The special form $1 + pF$ will play an important role in our attack.

Encryption and decryption use two hash functions. We denote them by Hash and Hash$'$. In practice, they are built using either SHA-1 or SHA-256 in various ways, depending on the desired security level, see [3]. The attack that we describe is based on the fact that for the number of SHA calls required by Hash depends on the input to Hash. Thus by measuring decryption time, an attacker may glean information about the input to Hash, which in turn reveals information about the private key f.

The enryption process works as follows.

NTRUEncrypt Encryption Algorithm

$M \in \mathcal{B}_N$	*Padded plaintext*
$r = \text{Hash}(M) \in \mathcal{B}_N(d_r)$	*Randomizer*
$m' = M \oplus \text{Hash}'(r * h \bmod q)$	*Message representative*
$e = (r * h + m') \bmod q$	*Ciphertext*

The decryption algorithm first recovers the (padded) message representative m' and plaintext M and then uses them to recreate the blinding value r and verify that (m', e) is a valid NTRUEncrypt pair.

NTRUEncrypt Decryption Algorithm

$$m' = \big((f * e \bmod q) \bmod 2\big) * f_2^{-1} \bmod 2$$
$$M = m' \oplus \text{Hash}'(e - m' \bmod q)$$
$$r = \text{Hash}(M)$$
Verify that e equals $r * h + m' \bmod q$

Note that on decryption, e and m' completely determine all the following operations.

The basis for our timing attack lies in the way in which Hash uses SHA to create r from M. The blinding value r is required to be a binary polynomial with exactly d_r ones, and the process described in [3] for creating r from M may take a different number of SHA calls for different values of M. Later we will describe exactly how this is done, but for now we simply observe that this leads to a time variation that an attacker may be able to measure and show how these timing observations may be converted into information about the private key f.

§Section 2. The time trail of a ciphertext

As we saw in Section 1, 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 (m', e). Each hash call requires a nontrivial amount of time, so an adversary might be able to determine how many hash calls Bob uses in attempting to decrypt a (possibly bogus) ciphertext e.

In practice, there will be a number K so that the number of hash calls required to create r from (m', e) is usually either K or $K+1$. For each pair (m', e), regardless of whether or not it is a valid NTRUEncrypt pair, we define $r(m', e)$ to be the output from the decryption algorithm,

$$r(m', e) = \mathrm{Hash}\big(\big(m' + \mathrm{Hash}'(e - m' \bmod q)\big) \bmod 2\big),$$

and we set $\beta(m', e) \in \{0, 1\}$ by the rule

$$\beta(m', e) = \begin{cases} 0 & \text{if it takes } \le K \text{ hashes to create } r(m', e), \\ 1 & \text{if it takes } > K \text{ hashes to create } r(m', e). \end{cases}$$

Note that the computation of $r(m', e)$, and thus of $\beta(m', e)$, requires no private knowledge.

For a given (m', e), we look also at the rotations $(X^i m', X^i e)$ for $i = 0, 1, \ldots$. We define the *Time Trail* of (m', e) to be the binary vector

$$\begin{aligned} T(m', e) &= \big(\beta(m', e), \beta(Xm', Xe), \beta(X^2 m', X^2 e), \ldots, \beta(X^{N-1} m', X^{N-1} e)\big) \\ &\in \{0, 1\}^N. \end{aligned}$$

The Time Trail tells us how many hashes are required for each of the rotations of the pair (m', e).

Let P be the probability that a randomly chosen (m', e) requires (at most) K hash calls and similarly $1 - P$ is the probability that a randomly chosen (m', e) requires (at least) $K + 1$ hash calls. If neither P nor $1 - P$ is too small, then the probability that two pairs (m'_1, e_1) and (m'_2, e_2) have the same time trails is quite small. More precisely, it is not hard to derive the formula

$$\mathrm{Prob}\big(T(m'_1, e_1) = T(m'_2, e_2)\big) = (1 - 2P + 2P^2)^N.$$

(We defer the derivation of this formula to Section A.1.)

§3. A timing attack based on variable number of hash calls

In this section we explain how an adversary Oscar might use time trails in order to derive information about Bob's private key.

Oscar first chooses a collection of (possibly bogus) ciphertexts $\mathcal{E}$ (i.e., $\mathcal{E}$ is a collection of polynomials modulo q). He also chooses a set of message representative values $\mathcal{M}$ (i.e., a collection of binary polynomials) with the property that $\mathcal{M}$ contains many of the polynomials in the set

$$\big\{((f * e \bmod q) \bmod 2) * (f^{-1} \bmod 2) : e \in \mathcal{E}\big\}.$$

Note that this is exactly the set of message representative that Bob would create during the process of decrypting the ciphertexts in $\mathcal{E}$. More precisely, we assume that the probability

$$p_{\mathcal{M},\mathcal{E}} := \operatorname*{Prob}_{e\in\mathcal{E}}\big(((f * e \bmod q) \bmod 2) * (f^{-1} \bmod 2) \in \mathcal{M}\big)$$

is not too small.

Before starting the active part of the attack, Oscar creates a table consisting of the time trails of every pair in $\mathcal{M} \times \mathcal{E}$. In other words, he creates a searchable list of binary vectors

$$\big(T(m', e) : m' \in \mathcal{M} \text{ and } e \in \mathcal{E}\big\}.$$

Thus the precomputation required for the attack has time and space requirements that are $O\big(\#\mathcal{M} \cdot \#\mathcal{E}\big)$.

To initiate the attack, Oscar chooses a random $e \in \mathcal{E}$, sends it to Bob, and records how long it takes Bob to decipher it. In this way, Oscar determines how many hash calls are required to create r from the ciphertext e and the message representative

$$m'(e) := ((f * e \bmod q) \bmod 2) * (f^{-1} \bmod 2),$$

so Oscar finds the value of $\beta(m'(e), e)$. Of course, Oscar does not know the value of $m'(e)$.

In a similar manner, Oscar sends each of the polynomials

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

to Bob and obtains the values $\beta(m'(X^ie), X^ie)$ for $i = 0, 1, \ldots, N-1$. We now observe that

$$\begin{aligned} m'(X^ie) &= ((f * X^ie \bmod q) \bmod 2) * (f^{-1} \bmod 2) \\ &= X^i * ((f * e \bmod q) \bmod 2) * (f^{-1} \bmod 2) \\ &= X^i m'(e) \end{aligned}$$

Thus Oscar has determined $\beta(X^i m'(e), X^i e)$ for $i = 0, 1, \ldots, N-1$, so he knows the time trail $T(m'(e), e)$ of the pair $(m'(e), e)$.

Oscar now searches his precomputed list and, with reasonable probability, finds a small number of possibilities for $(m'(e), e)$. In other words, Oscar now has a known polynomial e and a known polynomial m' so that when Bob decrypted e, Bob got m' as the message representative. Hence Oscar knows that there is an equation of the form

$$m' * f \equiv (f * e \bmod q) \pmod 2. \tag{1}$$

(More precisely, Oscar knows e and he has a small list of possible m', one of which satisfies (1). In Section 4 we discuss how Oscar can disambiguate between the possible m' in a plausible attack scenario.) Equation (1) certainly contains a significant amount of information concerning Bob's private key f, although exploiting this information will depend on the specific form of e. For example, if the elements of $\mathcal{E}$ consist of polynomials with very few nonzero coefficients, then equation (1) may give information concerning the spacing between the nonzero coefficients of f. In Section 4 we describe a specific collection $\mathcal{E}$ that leads to a practical hash timing attack when the key f has the form $f = 1 + pF$. (This form is sometimes used to decrease decryption time.)

§4. A practical hash timing attack for $f = 1 + 2F$ — Theory

For this section we consider the case where $p = 2$, so q is necessarily odd, and where private keys have the form

$$f = 1 + 2F \qquad \text{for some binary polynomial } F.$$

The parameters recommended by NTRU Cryptosystems currently take this form [2,3]. Note that $f_2^{-1} = (f \bmod 2)^{-1}$ is equal to 1, so the formula that Bob uses to recover the message representative m' from a ciphertext e simplifies to

$$m'(e) = (f * e \bmod q) \bmod 2. \tag{2}$$

For later computations, we write $F = \sum F_i X^i$ with $F_i \in \{0, 1\}$, and for any $i \in \mathbf{Z}$, we let F_i denote the coefficient $F_{(i \bmod N)}$.

Let $\lambda = 2\lceil q/8 \rceil$ be the smallest even integer that is larger than $q/4$. To mount the attack, Oscar uses the set of (bogus) ciphertexts defined by

$$\mathcal{E} = \big\{ \lambda + \lambda X^i : 1 \le i < N \big\}.$$

In other words, the $e \in \mathcal{E}$ are polynomials with two coefficients equal to λ and all other coefficients equal to 0.

We now need to figure out the possible values of $m'(e)$ that arise in (2) when Bob decrypts the ciphertexts in $\mathcal{E}$. During decryption, Bob first computes

$$\begin{aligned} a &= f * e \bmod q \\ &\equiv (1+2F) * (\lambda + \lambda X^i) \pmod q \\ &\equiv \lambda + \lambda X^i + \sum_{j=0}^{N-1} 2\lambda(F_j + F_{j-i})X^j. \end{aligned}$$

Thus the j^{th} coefficient of a is given by

$$a_j = \begin{cases} \lambda(1 + 2F_0 + 2F_{-i}) \bmod q & \text{if } j = 0, \\ \lambda(1 + 2F_i + 2F_0) \bmod q & \text{if } j = i, \\ \lambda(2F_j + 2F_{j-i}) \bmod q & \text{if } j \neq 0, i. \end{cases} \tag{3}$$

The key observation is that since $\lambda = 2\lceil q/8 \rceil$ is just slightly larger than $q/4$, the quantities on the righthand side of (3) are between 0 and $q-1$ unless $F_j = F_{j-i} = 1$, in which case they are greater than q. Thus there is nontrivial reduction modulo q if and only if $F_j = F_{j-i} = 1$, which implies that

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

The next step is to reduce a modulo 2, which yields the message representative $m'(e_i)$ for the (bogus) ciphertext $e_i = \lambda + \lambda X^i$. Recalling that λ is even and q is odd, we see that

$$a_j \bmod 2 = \begin{cases} 0 & \text{if } F_j = 0 \text{ or } F_{j-i} = 0, \\ 1 & \text{if } F_j = F_{j-i} = 1. \end{cases}$$

This gives the following explicit description of $m'(e_i)$:

$$m'(e_i) = \sum_{j=0}^{N-1} \left(\begin{array}{l} 1 \text{ if } F_j = F_{j-i} = 1 \\ 0 \text{ otherwise} \end{array} \right) X^j \ ,$$

which in turn yields the following partial information about F:

$$F(e_i) = \sum_{j=0}^{N-1} \left(\begin{array}{l} 1 \text{ if } m'(e_i)_j = 1 \text{ or } m'(e_i)_{j+i} = 1 \\ 0 \text{ otherwise} \end{array} \right) X^j \ ,$$

Therefore, every m' with $d_{m'}$ ones that Oscar can recover will directly yield $2d_{m'}$ non-zero coefficients of F, allowing him to reduce the search space for F and possibly to recover F completely.

Now we consider the amount of precomputation that Oscar must carry out in order to mount the attack. We claim that $m'(e_i)$ tends to have a comparatively small number of nonzero coefficients. More precisely, suppose that we take F to lie in the set $\mathcal{B}_N(d)$ of binary polynomials having d ones and $N-d$ zeros. Then the average number of ones in $m'(e_i)$ as F ranges over $\mathcal{B}_N(d)$ is

$$\frac{d(d-1)}{N}.$$

(We give the derivation of this formula in Section A.2.)

So if Oscar takes $\mathcal{M} = \mathcal{B}_N(\delta)$ with $\delta \approx (d^2-d)/N$, then there is an approximately 50% chance that $m'(e_i)$ appears in $\mathcal{M}$. And if Oscar takes a somewhat larger value for δ, he can significantly increase the probability of success. Of course, the cost is that Oscar's time and space requirements for the precomputation are on the order of

$$\#\mathcal{M} \cdot \#\mathcal{E} = N \cdot \#\mathcal{B}_N(\delta) = N\binom{N}{\delta}.$$

However, as we will see in the next section, there are NTRUEncrypt parameter sets for which this number is considerably smaller than the associated security level.

With these preliminaries, we can now describe Oscar's attack.

(1) Choose a value δ somewhat larger than $(d^2-d)/N$.
(2) Let $\mathcal{E} = \{e_i = \lambda + \lambda X^i : 0 \le i < N\}$ and $\mathcal{M} = \mathcal{B}_N(\delta)$.
(3) Precompute and store in a suitably searchable database the time trails $T(m', e)$ for every $m' \in \mathcal{M}$ and every $e \in \mathcal{E}$.
(4) For each i, send e_i, $Xe_i, \ldots, X^{N-1}e_i$ to Bob and use the decryption times to determine the time trail $T(m'(e_i), e_i)$ as described in Section 3.
(5) Search the database to determine $m'(e_i)$, either exactly or up to a small number of choices. (This will generally succeed for 50% or more of the e_i, depending on the size of δ.)
(6) Use the resulting values of $m'(e_i)$ to reconstruct F, either by an exact computation or by cutting down on the search space for F and performing a direct search of that subset.

Remark. For convenience, we have restricted attention to a small set of (bogus) ciphertexts $\mathcal{E}$. If several of the $e_i = \lambda + \lambda X^i$ in $\mathcal{E}$ lead to message representatives $m'(e_i)$ that are not in Oscar's database, he may not obtain enough information to recover the private key faster than is obtained by other attacks. In this case, Oscar can mount a similar attack using more general two term binary polynomials

$$\lambda_1 + \lambda_2 X^i.$$

He simply takes λ_1 and λ_2 to be even integers in the vicinity of $q/4$ and satisfying $\lambda_1 + \lambda_2 > q/2$. This leads to a plentiful supply of (bogus) ciphertexts whose message representatives have a good chance of lying in $\mathcal{M} = \mathcal{B}_N(\delta)$.

Now consider the case where where $T(m_1', e) = T(m_2', e)$ and $e = \lambda + \lambda X^\iota$ for some ι. Oscar has determined the time trail $T(m'(e), e)$ and wishes to determine whether $m'(e) = m_1'$ or m_2'. He picks a new ciphertext of the form $e' = \lambda_1 + \lambda_2 X^\iota$ and calculates $T(m_1', e')$, $T(m_2', e')$. Since

$$(e * f) \bmod q \bmod 2 = (e' * f) \bmod q \bmod 2 \ ,$$

Oscar knows that $m'(e') = m'(e) = m_1'$ or m_2'. With high probability $T(m_1', e') \neq T(m_2', e')$, so determining $T(m'(e'), e')$ will allow Oscar to work out which of m_1' and m_2' is equal to $m'(e')$.

§5. A practical hash timing attack for $f = 1 + 2F$ — Practice

In this section we evaluate the practicality of the attack described in Section 4 for some specific NTRUEncrypt parameter sets that appear in [3]. This depends, among other things, on the probability that different inputs require a greater or lesser number of SHA calls. We begin by describing how [3] uses SHA to compute r and then we compute the probability that this process takes a varying number of SHA calls.

The blinding value r, which is a binary polynomial with exactly d_r ones, is created from a hash function via repeated calls to some version of SHA. Here is the process as described in [3]:

(1) Fix a value of c satisfying $2^c > N$. This value of c is specified in [3] for each of the sample NTRUEncrypt parameter sets. Also let

$$b = \lceil c/8 \rceil \qquad \text{and} \qquad n = \lfloor 2^c/N \rfloor$$

Thus b is the smallest integer such that b bytes contains at least c bits. (In practice, b will be 1 or 2.) Similarly, nN is the smallest multiple of N that is less than 2^c.

(2) Call the specified version of SHA and break the output into chunks of b bytes each. Within each b byte chunk, keep the lower order c bits and discard the upper order $8b - c$ bits. Convert the lower order c bits into (little endian) integers $i_1, i_2, \ldots, i_t$. (Here t is the integer such that the output of the specified version of SHA consist of tb bytes.) This process of splitting the output from SHA is illustrated in Figure 1.

(3) Create a list of indices $j_1, j_2, \ldots$ by looping through the list of i values from (2). If $i < n$ and $i \bmod N$ is not already in the list, the adjoin $i \bmod N$ to the list, otherwise discard i. Continue until the list contains d_r values of j. If at any point you run out of i values, then call SHA and create additional i values as specified in (2). The complete r generation algorithm is illustrated with pseudocode in Figure 2.

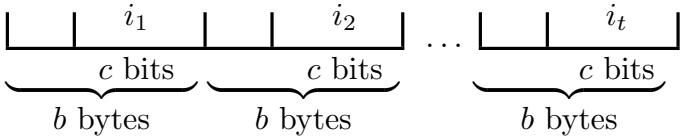

Figure 1. Converting SHA output into c bit integers

(1) `jList` $= \{\,\}$
(2) `Call SHA to get` $i_1, i_2, \ldots, i_t$
(3) `Loop` $\alpha = 1, 2, \ldots, t$
(4) `If` $i_\alpha < n$ `and` $(i_\alpha \bmod N) \notin$ `jList`
`then adjoin` $i_\alpha \bmod N$ `to jList`
(5) `If jList contains` d_r `elements, then exit`
(6) `End` α `loop`
(7) `Go to Step (2) to get more` i `values`

Figure 2. Generating r from SHA output

This description makes it clear why the number of calls to SHA may vary for different input values. If we treat the list of numbers $i_1, i_2, \ldots$ as a random sequence of integers in the range $0 \le i < 2^c$, the fundamental probabilities that we need to compute are

$$P_{C,N,n}(L,d) = \text{Prob}\left(\begin{array}{l}\text{A set of } L \text{ randomly chosen integers } i \in [0,C)\\ \text{includes exactly } d \text{ numbers satisfying both}\\ i \in [0,nN) \text{ and the values are distinct modulo } N\end{array}\right)$$

It is not hard to find a recursive formula that allows one to compute $P_{C,N,n}(L,d)$ reasonably quickly. See Section A.3 for details.

In order to generate r, the algorithm described in Figure 2 needs to create a list of d_r distinct numbers satisfying $0 \le i < N$. Each time the algorithm calls SHA, it gets t numbers satisfying $0 \le i < 2^c$. Hence the probability that it suffices to call to 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

$$\begin{aligned}\text{Prob}(s \text{ calls to SHA suffices}) &= \text{Prob}\left(\begin{array}{l}st \text{ randomly chosen integers in } [0,2^c)\\ \text{includes } \underline{\text{at least}}\ d_r \text{ values in } [0,n)\\ \text{that are distinct modulo } N\end{array}\right)\\ &= \sum_{d_r \le d \le st} P_{2^c,N,n}(st,d).\end{aligned}$$

In Table 1 we have assembled the NTRUEncrypt parameters from [3] and computed the values of s such that it is most likely to take either s or $s+1$ calls

to SHA in order to generate r. The probabilities are listed in the last column of the table. The closer that the first probability is to 50%, the better that attack described in this note works. We see from the table that the attack will probably work best for the $N = 787$ parameter set, which [3] indicates gives a 256 bit security level.

Bit Security	N	d_r	SHA bits	c	b	n	t	s : Prob(s SHA calls suffices)	
80	251	48	160	8	1	1	20	3 : 98.14%	4 : 100.0%
112	347	66	160	14	2	47	10	7 : 15.65%	8 : 98.48%
128	397	74	160	11	2	5	10	8 : 12.77%	9 : 95.10%
160	491	91	160	9	2	1	10	10 : 13.87%	11 : 91.32%
192	587	108	256	11	2	3	16	8 : 4.52%	9 : 82.38%
256	787	140	256	12	2	5	16	10 : 53.04%	11 : 99.85%

Table 1. The probability that s calls to SHA generates r

Finally, we use the probabilities in Table 1 and the other material described in this note to calculate the chances that a time trail uniquely determines a message representative m' and give an approximate size for Oscar's database in order to apply the attack. We present the resulting information in Table 2. The last column of Table 2 gives the approximate space and time requirements for the attack. Although the listed numbers are fairly large, they are considerably smaller than the security levels suggested in [3]. Thus if an attacker is in a position to measure the number of SHA calls, then the bit security is reduced by anywhere from 30 bits for $N = 251$ to 100 bits for $N = 787$.

§6. Conclusions and recommendations

We have described a timing attack on the implementation of NTRUEncrypt described in [3]. The attack relies on the fact that decryption of different (possibly bogus) ciphertexts may require a different number of calls to a hash function such as SHA-1 or SHA-256. We draw some conclusions and make some recommendations.

(1) Although we have only described an attack that relies on keys of the special form $f = 1 + pF$, it is reasonable to assume that similar attacks are possible for more general keys. Thus the use of general keys is not a recommended method to thwart hash timing attacks on NTRUEncrypt.

(2) In order to prevent hash timing attacks, it suffices to make sure that almost all decryptions require the same number of SHA calls. This can be accomplished by fixing a parameter K_{SHA} so that almost all inputs (m', e) require

Bit Security	N	d_r d_F	P_{extra}	$P_{\text{nonunique}}$	$\bar{\nu}_i(d_F)$	$\binom{N}{\lceil \bar{\nu}_i \rceil}$
80	251	48	1.86%	$2^{-13.5}$	8.988	$2^{53.1}$
112	347	66	84.35%	2^{-154}	12.36	$2^{76.8}$
128	397	74	87.23%	2^{-144}	13.61	$2^{84.8}$
160	491	91	86.13%	2^{-193}	16.68	$2^{103.2}$
192	587	108	95.48%	2^{-76}	19.69	$2^{122.4}$
256	787	140	46.96%	2^{-783}	24.73	$2^{156.3}$

Table 2. Data for various for NTRUEncrypt parameter sets
P_{extra} = Probability of needing an extra hash call (see Table 1)
$P_{\text{nonunique}}$ = Probability that a time trail is not unique
$\bar{\nu}_i(d_F)$ = Average number of j with $F_j = F_{j-i} = 1$
$\binom{N}{\lceil \bar{\nu}_i \rceil}$ = Number of m' in attacker's database

at most K_{SHA} SHA calls and then performing extra SHA call(s) if necessary so that almost all inputs require exactly K_{SHA} SHA calls. Here, we can put a more concrete meaning on "almost all" by requiring that at the k-bit security level, there is a chance of 2^{-k} that a given (m', e) has $\beta(m', e) = 1$. This yields the values given in Table 3 for K_{SHA}.

Bit Security	N	Expected SHA calls	K_{SHA}
80	251	3	6
112	347	8	15
128	397	9	17
160	491	11	22
192	587	9	20
256	787	10	21

Table 3. Recommended number of SHA calls for different security levels.

Note that this recommendation will require an attacker to expend more than 2^k machine cycles to mount the attack, first because a SHA call takes more than one operation, and second because each attack involves $K_{\text{SHA}} > 1$ SHA calls.

(3) The method used to generate r from (m', e) in [3] is easy to implement, but it is somewhat wasteful of the pseudorandom bits produced by SHA. It might be worthwhile to look for more efficient ways to generate r which might also use a fixed number of calls to SHA, thereby eliminating the possibility of a hash timing attack. However, we note that the use of a new r-generation method would require changes to the exisiting standards, while equalization of the number of SHA calls as in (2) is a simple implementation change that maintains current standards.

APPENDICES

§A.1. Probability that two message representatives have the same time trail

A time trail is a binary vector of dimension N. We let P denote the probabilty that a randomly chosen coordinate is equal to 0, so $1-P$ is the corresponding probablity that a randomly chosen coordinate is equal to 1. Then the probability that (say) the first coordinates of two random time trails agree is

$$\text{Prob(both 0)} + \text{Prob(both 1)} = P^2 + (1-P)^2 = 1 - 2P + 2P^2.$$

In order for two entire time trails to be identical, they must agree on all N of their coordinates. Hence

$$\text{Probability that two Time Trails coincide} = (1 - 2P + 2P^2)^N.$$

Therefore for any given $e \in \mathcal{E}$ and $m' \in \mathcal{M}$, the probability that there exists some other message representative $m'' \in \mathcal{M}$ with $T(e, m'') = T(e, m')$ is approximately

$$\#\mathcal{M} \cdot (1 - 2P + 2P^2)^N.$$

§A.2. The average number of ones with a given separation distance

Let $\mathcal{B}_N(d)$ be the set of binary polynomials of degree less than N with exactly d ones and $N - d$ zeros. Fix i. We are interested in the average number of j such that F_j and F_{j-i} are both equal to 1, as F ranges over $\mathcal{B}_N(d)$. For a given F and i, we denote the number of such j by

$$\nu_i(F) = \#\{0 \le j < N : F_j = F_{j-i} = 1\}.$$

Clearly $\nu_0(F) = d$ for every $F \in \mathcal{B}_N(d)$. We now fix some $1 \le i < N$ and compute the average value $\bar{\nu}_i(d)$ of $\nu_i(F)$ as F ranges over $\mathcal{B}_N(d)$.

$$\begin{aligned}
\bar{\nu}_i(d) = \operatorname*{Average}_{F\in\mathcal{B}_N(d)} \quad \nu_i(F) &= \binom{N}{d}^{-1} \sum_{F\in\mathcal{B}_N(d)} \nu_i(F) \\
&= \binom{N}{d}^{-1} \sum_{F\in\mathcal{B}_N(d)} \sum_{j=0}^{N-1} F_j F_{j-i} \\
&= \binom{N}{d}^{-1} \sum_{j=0}^{N-1} \sum_{F\in\mathcal{B}_N(d)} F_j F_{j-i} \\
&= \binom{N}{d}^{-1} \sum_{j=0}^{N-1} \#\big\{F \in \mathcal{B}_N(d) : F_j = F_{j-i} = 1\big\} \\
&= \binom{N}{d}^{-1} \sum_{j=0}^{N-1} \binom{N-2}{d-2} \\
&= \binom{N}{d}^{-1} N\binom{N-2}{d-2} \\
&= \frac{d(d-1)}{N}.
\end{aligned}$$

This proves the formula cited in Section 4.

We also observe that $\nu_i(F)$ appears as a coefficient of the product $F * F^{\text{rev}}$, where the reversal F^{rev} of F is the polynomial $F^{\text{rev}} = \sum F_{-i} X^i$. Thus

$$F * F^{\text{rev}} = \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} F_j F_{-k} X^{j+k} = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} F_j F_{j-i} X^i = \sum_{i=0}^{N-1} \nu_i(F) X^i.$$

Thus knowledge of $\nu_i(F)$ for $0 \le i < N$ is equivalent to knowledge of the product $F * F^{\text{rev}}$. Using this value and the public key $h = f^{-1} * g \bmod q$, there are practical methods for recovering F. In any case, it is certainly true that each valid (r, m') pair that Oscar finds contains significant information about the private key f, and there are numerous ways to exploit such information in order to recover f directly (if one has enough (r, m') pairs) or by cutting down the search space for f.

§A.3. The probability of choosing distinct values in a given range

In this section we describe a recursion that can be used to compute the probability

$$P_{C,N,n}(L,d) = \text{Prob}\left(\begin{array}{l}\text{A set of } L \text{ randomly chosen integers } i \in [0,C)\\ \text{includes exactly } d \text{ numbers satisfying}\\ i \in [0,nN) \text{ and whose values are distinct modulo } N\end{array}\right)$$

We obtain a recursion from the observation that $P_{C,N,n}(L,d)$ equals the sum of the following two quantities:

- The probability after $L-1$ picks of having $d-1$ values in $[0,nN)$ that are distinct modulo N multiplied by the probability of picking an integer in $[0,nN)$ multiplied by the probability that it does not a repeat a previous values modulo N.
- The probability after $L-1$ picks of having d values in $[0,nN)$ that are distinct modulo N multiplied by the probability of picking an integer that either is not in $[0,nN)$ or whose value modulo N repeats a previous value.

We observe that for the first case, the probability of picking an integer in $[0,nN)$ multiplied by the probability that it does not a repeat a previous values modulo N is

$$\frac{nN}{C} \cdot \frac{N-(d-1)}{N} = \frac{n(N-d+1)}{C}.$$

For the second case, there are $C-nN$ integers in $[0,C)$ that are not in $[0,nN)$, and there are nd integers in $[0,nN)$ that are in one of the d congruence classes modulo n that have already been selected, so the probability of picking an integer that either is not in $[0,nN)$ or whose value modulo N repeats a previous value is

$$\frac{C-nN+nd}{C} = 1 - \frac{n(N-d)}{C}.$$

This yields the recursion formula

$$P_{C,N,n}(L,d) = P_{C,N,n}(L-1,d-1) \cdot \left(\frac{n(N-d+1)}{C}\right) + P_{C,N,n}(L-1,d) \cdot \left(1 - \frac{n(N-d)}{C}\right)$$

Combining this recursion with the obvious initial values

$$P_{C,N,n}(L,d) = 0 \quad \text{if } L < d \qquad \text{and} \qquad P_{C,N,n}(L,0) = \left(1 - \frac{nN}{C}\right)^L,$$

it is an easy matter to compute $P_{C,N,n}(L,d)$ if the parameters are not too large.

References

[1] J. Hoffstein, J. Pipher, J.H. Silverman, NTRU: A new high speed public key cryptosystem, Algorithmic Number Theory (ANTS III), Portland, OR, June 1998, Lecture Notes in Computer Science 1423, J.P. Buhler (ed.), Springer-Verlag, Berlin, 1998, 267–288

[2] J. Hoffstein, J.H. Silverman, Optimizations for NTRU, Public Key Cryptography and Computational Number Theory (Warsaw, Sept. 11–15, 2000), Walter de Gruyter, Berlin–New York, 2001, 77–88.

[3] N. Howgrave-Graham, J. H. Silverman, W. Whyte Choosing Parameter Sets for NTRUEncrypt with NAEP and SVES-3, Topics in cryptology—CT-RSA 2005, 118–135, Lecture Notes in Comput. Sci., 3376, Springer, Berlin, 2005. www.ntru.com/cryptolab/articles.htm#2005_1

Comments and questions concerning this technical report should be addressed to

info@ntru.com

Additional information on NTRU Cryptosystems, Inc., the NTRU Public Key Cryptosystem (NTRUEncrypt) and the NTRU Signature Scheme (NTRUSign) are available at

www.ntru.com
