An IND-CCA2 Attack Against the 1st- and 2nd-round Versions of NTS-KEM

Tung Chou

Academia Sinica, 128 Academia Road, Section 2, Nankang, Taipei 11529, Taiwan
blueprint@crypto.tw

Abstract. This paper presents an IND-CCA2 attack against the 1st- and 2nd-round versions of NTS-KEM, i.e., the versions before the update in December 2019. Our attack works against the 1st- and 2nd-round specifications, with a number of decapsulation queries upper-bounded by $n-k$ and an advantage lower-bounded by roughly $0.5(n-k)t/n^2$, where n, k, and t stand for the code length, code dimension, and the designed decoding capacity, for all the three parameter sets of NTS-KEM. We found that the non-reference implementations are also vulnerable to our attack, even though there are bugs. There are also bugs in the reference implementations, but in a way invulnerable to our attack.

Keywords: NIST PQC standardization · Post-quantum cryptogrphy · Code-based cryptography · IND-CCA2

1 Introduction

NTS-KEM [1] is a key encapsulation mechanism submitted to the NIST Post-Quantum Cryptography Standardization Process. NTS-KEM is one of the 25 submissions that entered the second round of the process. On 3 December, 2019, during the second round of the process, Paterson, one of the principal submitters of NTS-KEM, wrote the message in Appendix A (without the emphasis added) in an email to NIST's `pqc-forum` mailing list, to announce an update in their specification.

We call this new version the *updated* 2nd-round version of NTS-KEM, of which the specification can be found in [3], while the 2nd-round version of NTS-KEM refers to the version submitted in March 2019 [2]. NTS-KEM did not advance to the 3rd round as NTS-KEM and Classic McEliece [5] have merged. The merged submission is called Classic McEliece and equals the 2nd-round version of Classic McEliece.

Maram's recent paper [13] discusses more about this "subtle issue" caused by omission of re-encryption. In particular, [13, Section 3.1] argues that it might be possible for an IND-CCA2 adversary against the 1st- and 2nd-round NTS-KEM to modify the last $n-k$ bits of the challenge ciphertext (where n and k stand for

This work was supported by Taiwan Ministry of Science and Technology (MOST) Grant 109-2222-E-001-001-MY3. Permanent ID of this document: `eb8cb246e2f1ea188ad29d70680e73f8`. Date: 2020.12.8.

the code length and code dimension), such that the decapsulation oracle returns the encapsulated session key. If this happens with a sufficiently high probability, clearly IND-CCA2 security of the 1st- and 2nd-round version of NTS-KEM is broken. However, Maram did not present any concrete IND-CCA2 attack in the paper.

1.1 Our Contribution

In this paper, we present a simple IND-CCA2 attack against the 1st- and 2nd-round versions of NTS-KEM [1, 2]. The attack takes only a few decapsulation queries and bit flips, and it succeeds with a high probability. Our attack follows Maram's strategy: the adversary recovers the encapsulated session key by modifying the last $n - k$ bits of the challenge ciphertext. Our attack does exploit NTS-KEM's Berlekamp-Massey algorithm, but in a way without forcing the algorithm to "operate beyond its natural decoding capacity": what we did is to force the algorithm to operate *below* the designed decoding capacity t.

To be more precise, our attack works against the 1st- and 2nd-round specifications. The attack takes at most $n - k$ decapsulation queries and at most $n - k$ bit flips, and has an advantage lower-bounded by roughly $0.5(n - k)t/n^2$, for all the three parameter sets of NTS-KEM. We found that the non-reference implementations are also vulnerable to our attack, even though there are bugs. There are also bugs in the reference implementations, but in a way invulnerable to our attack.

One might argue that since the NTS-KEM team has updated their specification, it is not meaningful to study the security of the old specifications. However, we think it is meaningful to study the security of the 1st- and 2nd-round versions of NTS-KEM for the following reasons.

- In April 2018, Cheng from PQ Solutions Limited wrote the following message in an email to the `pqc-forum` mailing list.

  > "We are particularly excited that one entity is already going to perform a substantial test on the performance and resilience of NTS-KEM in the not too distant future."

  On the 22 May, 2019, Cho from ADVA gave a talk [7] in the 7th Code-Based Cryptography Workshop. Cho presented experimental results of using code-based KEMs, including NTS-KEM, for secure optical communication. This shows that the source code of the 1st- and 2nd- round versions of NTS-KEM has been used by some people, and clearly they need to be warned about the attack.

- Maram wrote the following in [13, Section 3.1].

  > "At the same time, we stress that the above described attack is just a possibility and is not a concrete attack. Because it is quite possible that, by analyzing the decoding algorithm used in NTS-KEM decapsulation, one might show such invalid ciphertexts are computationally hard to generate adversarially."

> The sentences and Paterson's message suggest that the 1st- and 2nd-round versions of NTS-KEM are not nessarily insecure. Indeed, a scheme can be IND-CCA2 secure even if people do not know how to prove that. However, we show that there is a concrete IND-CCA2 attack against the 1st- and 2nd-round versions of NTS-KEM.

To demonstrate that our attack works against the 1st- and 2nd-round specifications and non-reference implementations, we have modified the files `nts_kem.c`, `ntskem_test.c`, and `berlekamp_massey.c` in the submission packages. The contents of the modified `ntskem_test.c` and `berlekamp_massey.c` are available at Appendix E and F. More details about these modified files and how to use them to demonstrate our attack are shown in Appedix D.

1.2 Related Works

We note that re-encryption is not a new countermeasure against attacks. For example, re-encryption is required in the well-known Fujisaki-Okamoto transform [10, 11], which converts weakly secure public-key encryption schemes into CCA-secure ones. Dent [9] also makes use of re-encryption to construct CCA-secure key encapsulation mechanisms from weakly secure public-key encryption schemes. For comparison, a re-encryption step is included in the decapsulation algorithm of the 1st- and 2nd-round versions of Classic McEliece [5].

1.3 Organization

Section 2 gives some basic knowledge about key encapsulation mechanisms and code-based cryptography. Section 3 introduces the 1st- and 2nd-round NTS-KEM. Section 4 presents our attack and how it works against the 1st- and 2nd-round specifications and non-reference implementations. For completeness, Appendix C explains why our attack does not work against the reference implementations.

2 Preliminaries

This section presents some basic knowledge on key encapsulation mechanisms and code-based cryptography.

2.1 Key Encapsulation Mechanisms (KEMs)

The concept of KEM was first introduced by Shoup [18]. A KEM `KEM` consists of the following three algorithms.

– The key generation algorithm `KEM.KeyGen` is a probabilistic, polynomial-time algorithm that outputs a key pair $(\mathtt{PK}, \mathtt{SK})$, where `PK` is the *public key* and `SK` is the *secret key*.

– The encapsulation algorithm `KEM.Enc` is a probabilistic polynomial-time algorithm that on input a public key `PK`, outputs (K, ψ), where $K \in \{0,1\}^\ell$ is the *session key* and ψ is the *ciphertext* encapsulating K.

– The decapsulation algorithm `KEM.Dec` is a deterministic polynomial-time algorithm that on input a secret key `SK` and a ciphertext ψ, outputs either a session key K or the special symbol $\perp$.

A KEM is required to be *sound*. For the purpose of this paper, one may simply assume that this means that the decapsulation algorithm always outputs the encapsulated session key as long as the input ciphertext is valid.

2.2 IND-CCA2 Security of KEMs

In order to define IND-CCA2 security of a KEM `KEM`, we consider a game consisting of the following steps played by an *adversary* and a *challenger*.

1. The challenger generates a key pair $(\texttt{PK}, \texttt{SK})$ by running `KEM.KeyGen` and sends `PK` to the adversary.
2. The adversary runs until it is ready to move to the next step. During this step, the adversary may make a sequence of queries to a *decapsulation oracle*. In each of the queries, the adversary submits a ciphertext ψ, and the oracle responds with $\texttt{KEM.Dec}(\texttt{SK}, \psi)$.
3. The challenger prepares a pair (K^*, ψ^*) by carrying out the following steps and sends the pair to the adversary.

$$\begin{aligned}(K_0, \psi^*) &\leftarrow \texttt{KEM.Enc}(\texttt{PK});\\ K_1 &\xleftarrow{\$} \{0,1\}^\ell;\\ \tau &\xleftarrow{\$} \{0,1\};\\ K^* &\leftarrow K_\tau;\end{aligned}$$

4. The adversary runs until it is ready to move to the next step. During this step, the adversary may make a sequence of queries to the decapsulation oracle, under the condition that any ciphertext submitted by the adversary must be different from ψ^*.
5. The adversary outputs $\tau' \in \{0,1\}$.

The *advantage* of an adversary is defined as $|\Pr[\tau = \tau'] - 1/2|$. Traditionally, a KEM is said to be IND-CCA2 secure if for all probablistic, polynomial-time adversary, the advantage grows negligible in the security parameter λ. However, this definition requires that the KEM is defined as a family of systems. For KEMs with specific parameter sets, such as NTS-KEM, we evaluate the efficiency of an adversary by its actually running time and advantage.

2.3 Linear Codes

A *linear code* of *length* n and *dimension* k over a field $\mathbb{F}_q$ is a dimension-k linear subspace of $\mathbb{F}_q^n$. The elements in a code are called *codewords*. A linear code C can thus be represented by the row space of a matrix, in which case we call such a matrix a *generator matrix*. A linear code can also be represented by the right kernel space of a matrix, in which case we call such a matrix a *parity-check matrix*. Note that a generator matrix has at least k rows, and a parity-check matrix has at least $n-k$ rows.

Given a generator matrix $G \in \mathbb{F}_q^{k\times n}$ for a linear code, it is easy to compute a parity-check matrix H of the code using simple linear algebra techniques. In particular, if G has *systematic form*, which means $G = (I_k|Q)$ where Q is a $k \times (n-k)$ matrix, then $H = (-Q^T|I_{n-k})$ is a parity-check matrix for the same code, and vice versa. The *syndrome* of $v \in \mathbb{F}_q^n$ with respect to a parity-check matrix H is defined as vH^T.

The *Hamming weight* of a vector in $\mathbb{F}_q^n$ is the number of non-zero coordinates in it. We denote the Hamming weight of a vector v as $|v|$. The *minimum distance* of a nonzero linear code is the smallest Hamming weight of any nonzero codeword in C.

For a linear code C, a *decoding algorithm* takes $r \in \mathbb{F}_q^n$ and a positive integer w as inputs and outputs $e \in \mathbb{F}_q^n$ such that $|e| \le w$ and $r - e \in C$, if such e exists. When the minimum distance is at least $2w+1$, for any r, the vector e such that $|e| \le w$ and $r - e \in C$ must be unique if it exists; In this case, we say that C can correct w errors.

2.4 Binary Goppa Codes

Given a field $\mathbb{F}_{2^m}$, a sequence $\alpha_1, \ldots, \alpha_n$ (called the *support*) of n distinct elements from $\mathbb{F}_{2^m}$, and a degree-t polynomial $g \in \mathbb{F}_{2^m}[x]$ (called the *Goppa polynomial*) such that $g(\alpha_1)\cdots g(\alpha_n) \neq 0$, the Goppa code $\Gamma_2(\alpha_1, \ldots, \alpha_n, g)$ is defined as the set of vectors $c = (c_1, \ldots, c_n) \in \mathbb{F}_2^n$ such that

$$\sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} \equiv 0 \mod g(x).$$

The dimension k of the code is at least $n - mt$. When g is square-free, the minimum distance of $\Gamma_2(\alpha_1, \ldots, \alpha_n, g)$ is known to be at least $2t+1$, and

$$\Gamma_2(\alpha_1, \ldots, \alpha_n, g) = \Gamma_2(\alpha_1, \ldots, \alpha_n, g^2).$$

A specific parity-check matrix of $\Gamma_2(\alpha_1, \ldots, \alpha_n, g)$, which we denote as $\mathcal{H}(\alpha_1, \ldots, \alpha_n, g)$, is given as follows.

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{\mathrm{Deg}(g)-1}/g(\alpha_1) & \alpha_2^{\mathrm{Deg}(g)-1}/g(\alpha_2) & \cdots & \alpha_n^{\mathrm{Deg}(g)-1}/g(\alpha_n) \end{pmatrix},$$

where each of the tn entries is actually a column vector in $\mathbb{F}_2^m$, formed by the coordinates with respect to a chosen $\mathbb{F}_2$-basis of $\mathbb{F}_{2^m}$.

2.5 The McEliece Cryptosystem

The McEliece cryptosystem [15] is a public-key encryption system: it allows a sender to encrypt his/her messages as a ciphertext using the receiver's public key, such that the messages can only be decrypted from the ciphertexts using the receiver's secret key.

To generate the public key and secret key, the receiver first generates a code C over $\mathbb{F}_q$ of length n and dimension k with a decoder which is able to correct t errors. The receiver then computes a generator matrix G of C and generates a uniform random permutation matrix $P \in \mathbb{F}_q^{n \times n}$ and a uniform random invertible matrix $S \in \mathbb{F}_q^{k \times k}$. The receiver then publishes $\hat{G} = SGP$ as its public key and keeps (G, P, S) as its secret key.

To perform encryption, the sender computes the ciphertext $y = m\hat{G} + e$ where $m \in \mathbb{F}_q^k$ is the message and $e \in \mathbb{F}_q^n$ is a uniform random vector of weight t. To perform decryption, the receiver computes $yP^{-1} = mSG + eP^{-1}$ and applies a decoding algorithm to find mSG. From mSG the receiver then computes mS and m using linear algebra.

3 The 1st- and 2nd-round versions of NTS-KEM

This section presents the specifications of the 1st- and 2nd-round versions of NTS-KEM. The difference between the two versions is small: the 1st-round NTS-KEM uses *explicit rejection* in the decapsulation algorithm, meaning that the decapsulation algorithm returns $\perp$ when the ciphertext is considered invalid. The 2nd-round NTS-KEM uses *implicit rejection* in the decapsulation algorithm, meaning that it returns an ℓ-bit string when the ciphertext is considered invalid.

The reader might find that the key generation algorithm and the decoding algorithm presented in this section look simpler than the ones shown in the 1st- and 2nd-round specifications. We emphasize that this is because we decided to omit irrelevant details in the specifications to simplify our discussions. It is easy to see that the algorithms presented in this section are in fact equivalent to the ones shown in the specifications.

3.1 Public Parameters and Parameter Sets

The public parameters of an instance of NTS-KEM are as follows.

- $n = 2^m$, the length of the binary Goppa code.
- t, the number of errors that the code is designed to correct.
- $f(x) \in \mathbb{F}_2[x]$, an irreducible polynomial of degree m, which is used to construct $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/f(x)$.

parameter set	m	n	t
`ntskem1264`	12	4096	64
`ntskem1380`	13	8192	80
`ntskem13136`	13	8192	136

Table 1. The three parameter sets of NTS-KEM.

– $\ell = 256$, which denotes the length of session keys.

NTS-KEM also makes use of a pseudorandom bit generator, denoted as $H_\ell(\cdot)$, which outputs ℓ-bit strings. The three parameter sets of NTS-KEM are listed in Table 1.

3.2 Key Generation

To generate a secret key, the user starts with generating a uniform random square-free Goppa polynomial $g(x) \in \mathbb{F}_{2^m}[x]$ of degree t, and a uniform random support $\alpha_1, \ldots, \alpha_n$. Note that the support contains all elements of $\mathbb{F}_{2^m}$. The support and the Goppa polynomial then define the binary Goppa code $\Gamma_2(\alpha_1, \ldots, \alpha_n, g)$.

To compute the public key, the user first computes a "parity-check matrix"

$$H = \begin{pmatrix} 1/g^2(\alpha_1) & 1/g^2(\alpha_2) & \cdots & 1/g^2(\alpha_n) \\ \alpha_1/g^2(\alpha_1) & \alpha_2/g^2(\alpha_2) & \cdots & \alpha_n/g^2(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g^2(\alpha_1) & \alpha_2^{t-1}/g^2(\alpha_2) & \cdots & \alpha_n^{t-1}/g^2(\alpha_n) \end{pmatrix}$$

where each of the tn entries is again a column vector in $\mathbb{F}_2^m$ and then computes its "reduced row echelon form". Let $k = n - mt$. Once the reduced row echelon form is obtained, the user "reorders its columns if necessary" to get a parity-check matrix of the form $(Q^T | I_{n-k})$ and the corresponding generator matrix $(I_k | Q)$. Note that elements in the support need to be reordered in the same way as the columns. The public key is then (Q, t, ℓ).

It is not clear why H is used instead of $\mathcal{H}(\alpha_1, \ldots, \alpha_n, g)$. In fact, we have not found any existing literature showing that H is guaranteed to be a parity-check matrix of the binary Goppa code. Any H that has full rank are guaranteed to be a parity-check matrix, though. Indeed, as $\Gamma_2(\alpha_1, \ldots, \alpha_n, g) = \Gamma_2(\alpha_1, \ldots, \alpha_n, g^2)$, $\mathcal{H}(\alpha_1, \ldots, \alpha_n, g^2)$ is a parity-check matrix of the code; Observe that H consists of the first mt rows of $\mathcal{H}(\alpha_1, \ldots, \alpha_n, g^2)$, and the dimension of the code is at least $n - mt$; Therefore, if H has rank mt, it must have the same row space and right kernel as $\mathcal{H}(\alpha_1, \ldots, \alpha_n, g^2)$.

A secret key of the 1st-round version of NTS-KEM consists of three pieces of data which can be easily derived from $(\alpha_1, \ldots, \alpha_n, g)$. A secret key of the 2nd-round version of NTS-KEM consists of four pieces of data where the first three are the same as those in the 1st-round NTS-KEM, and the last one is simply a

uniform random bit string $z \in \mathbb{F}_2^\ell$. To simplify our discussion, we consider that the secret key is simply $(\alpha_1, \ldots, \alpha_n, g)$ or $(\alpha_1, \ldots, \alpha_n, g, z)$.

The key generation algorithm above is *not* well-defined for the following reasons.

- It is possible that H is not full-rank, in which case it will be impossible to bring H to the form $(Q^T|I_{n-k})$, it is not written in the specifications how to deal with this case.
- It is not written in the specifications what "reduced row echelon form" means and how the columns should be re-ordered exactly. Even if one assumes the most common definition of reduced row echelon form, there are still multiple deterministic and non-deterministic ways to reorder the columns.

The implementations of NTS-KEM, however, show that H is reduced to a matrix that satisfies the following criteria.

- There is a sequence $c_{n-k-r+1} < c_{n-k-r+2} < \cdots < c_{n-k}$ such that for each $i \in \{n-k-r+1, \ldots, n-k\}$, row i ends with a 1 in column c_i, which is the only non-zero entry in column c_i.
- All rows before row $n-k-r+1$ are zero rows.

Existence of zero rows means H is not full-rank. The implementations simply fail to generate a key pair when H is not full-rank, which seems extremely unlikely to happen for NTS-KEM's parameter sets. After reducing H, the implementations then swap column c_{n-k} with column $n-k$, swap column c_{n-k-1} with column $n-k-1$, and so on to produce $(Q^T|I_{n-k})$.

As the key generation algorithm is not well-defined in the specifications, we simply assume that the implemented key generation algorithm is what the NTS-KEM team intended to specify, and we consider the implemented key generation algorithm for our discussion.

3.3 Encapsulation

Given an NTS-KEM public key (Q, t, ℓ), the encapsulation algorithm computes a session key and a ciphertext encapsulating it by carrying out the following steps.

- Generate a uniform random error vector $e \in \mathbb{F}_2^n$ with $|e| = t$.
- Partition e into $(e_a \mid e_b \mid e_c)$, where $e_a \in \mathbb{F}_2^{k-\ell}$, $e_b \in \mathbb{F}_2^\ell$, and $e_c \in \mathbb{F}_2^{n-k}$.
- Compute $k_e = H_\ell(e) \in \mathbb{F}_2^\ell$. Let $m = (e_a \mid k_e) \in \mathbb{F}_2^k$.
- Following McEliece encryption, compute $c \in \mathbb{F}_2^n$ as follows.

$$\begin{aligned} c &= m \cdot (I \mid Q) + e \\ &= (m \mid m \cdot Q) + e \\ &= (e_a \mid k_e \mid (e_a \mid k_e) \cdot Q) + (e_a \mid e_b \mid e_c) \\ &= (0_a \mid k_e + e_b \mid (e_a \mid k_e) \cdot Q + e_c) \\ &= (0_a \mid c_b \mid c_c), \end{aligned}$$

where 0_a is the zero vector of length $k - \ell$. Let the ciphertext be $(c_b \mid c_c)$.

- Compute the session key $k_r = H_\ell(k_e \mid e) \in \mathbb{F}_2^\ell$.
- Return $(k_r, (c_b \mid c_c))$.

3.4 Decapsulation

Given a ciphertext $(c_b \mid c_c)$, the decapsulation works as follows.

- Taking the vector $(0_a \mid c_b \mid c_c) \in \mathbb{F}_2^n$ and the secret key as inputs, compute $e \in \mathbb{F}_2^n$ using the decoding algorithm (see Section 3.5).
- Partition e into $(e_a \mid e_b \mid e_c)$, where $e_a \in \mathbb{F}_2^{k-\ell}$, $e_b \in \mathbb{F}_2^\ell$, and $e_c \in \mathbb{F}_2^{n-k}$, and compute $k_e = c_b - e_b$.
- Check if $H_\ell(e) = k_e$ and $|e| = t$. If both are true, return $k_r = H_\ell(k_e \mid e) \in \mathbb{F}_2^\ell$; Otherwise return $\perp$ for the 1st-round NTS-KEM or $H_\ell(z \mid 1_a \mid c_b \mid c_c)$ for the 2nd-round NTS-KEM.

3.5 The Decoding Algorithm

A strategy for decoding is to consider $\Gamma_2(\alpha_1, \ldots, \alpha_n, g)$ as an alternant code and use an alternant decoder to decode. A well-known alternant decoder is the Berlekamp decoder. To find the error positions, the Berlekamp decoder makes use of the Berlekamp-Massey algorithm ([4], [14]) to compute an *error locator*, of which the roots are $\{\alpha_i^{-1} \mid e_i = 1, \alpha_i \neq 0\}$. In other words, by finding the roots of the error locator, we can find the error positions $\{i \mid e_i = 1, \alpha_i \neq 0\}$. Some more operations are required to figure out whether $e_i = 0$ or not.

NTS-KEM's decoding algorithm uses the strategy above but introduces some modifications. NTS-KEM's decoding algorithm makes use of `NTSKEM_BM`, which is a modified Berlekamp-Massey algorithm. In `NTSKEM_BM`, the error locator is computed and converted into a polynomial σ^*, of which the roots are simply $\{\alpha_i \mid e_i = 1, \alpha_i \neq 0\}$. `NTSKEM_BM` also computes a value $\xi \in \{0, 1\}$, to indicate whether $e_{\mathrm{pos}(0)} = 1$. Given an input vector $r \in \mathbb{F}_2^n$, NTS-KEM's decoding algorithm works as follows.

- Compute the syndrome $s = r \cdot \mathcal{H}(\alpha_1, \ldots, \alpha_n, g^2)^T \in \mathbb{F}_{2^m}^{2t}$.
- Compute $(\sigma^*(x), \xi) \leftarrow \texttt{NTSKEM\_BM}(s)$, where $\sigma^*(x) \in \mathbb{F}_{2^m}[x]$ and $\xi \in \{0, 1\}$.
- Set $e = 0 \in \mathbb{F}_2^n$.
- Set $e_i = 1$ for all i such that $\sigma^*(\alpha_i) = 0$.
- Set $e_{\mathrm{pos}(0)} = 1$ if $\xi = 1$.
- Return e.

3.6 NTS-KEM's Berlekamp-Massey Algorithm

The Berlekamp-Massey algorithm in NTS-KEM's specifications is shown in Algorithm 1 (we put it in Appendix B due to the page limit). The algorithm is the same as Algorithm 3 in the 1st- and 2nd-round supporting documentations, except that we use t to indicate the designed decoding capacity. Without the lines involving R, ξ, or σ^*, the algorithm is the same as Xu's inversion-free Berlekamp-Massey algorithm [20]. The following section presents an attack against NTS-KEM. The attack makes use of the lines involving R, ξ, or σ^*, in particular line 16 to line 20, and the following theorem.

Theorem 1. *If the input to Algorithm 1 is in*

$$(c+e)\cdot\mathcal{H}(\alpha_1,\ldots,\alpha_n,g^2)^T,$$

where $g(x)\in\mathbb{F}_{2^m}[x]$ *is square-free and of degree-*t, $c\in\Gamma_2(\alpha_1,\ldots,\alpha_n,g)$, $e\in\mathbb{F}_2^n, |e|\leq t$, *then at the end of the algorithm we have*

$$\sigma(x)=\sigma_0\prod_{e_i=1}(1-\alpha_i x),$$

where $\sigma_0\in\mathbb{F}_{2^m}\setminus\{0\}$.

Proof. As discussed in Section 3.2, $\mathcal{H}(\alpha_1,\ldots,\alpha_n,g^2)^T$ is a parity-check matrix of $\Gamma_2(\alpha_1,\ldots,\alpha_n,g)$. Following the discussion in [12], the original Berlekamp-Massey algorithm ([4], [14]) then computes $A(x)=\prod_{e_i=1}(1-\alpha_i x)$ on input $(c+e)\cdot\mathcal{H}(\alpha_1,\ldots,\alpha_n,g^2)^T$. It is shown in [20] that the inversion-free version computes a similar polynomial $\sigma(x)=\sigma_0\cdot A(x)$ with $\sigma_0\neq 0$.

4 Our Attack

This section presents our IND-CCA2 attack and explains why it can be used to attack the specifications and non-reference implementations. The reason why it does not work for the reference implementations is shown in Appendix C.

4.1 The Adversary

We consider an adversary who does nothing before receiving (K^*,ψ^*) from the challenger. After receiving (K^*,ψ^*) from the challenger, the adversary partitions ψ^* into (c_b,c_c) as in regular decapsulation and performs the following two simple steps for each of the $n-k$ bits of c_c:

- Flip the bit of c_c to obtain (c_b,c_c').
- Send (c_b,c_c') to the decapsulation oracle.
- If the decapsulation oracle returns K^*, return $\tau'=0$.

If the decapsulation oracle does not return K^* in any of the $n-k$ iterations, the adversary returns $\tau' = 1$.

Clearly, the adversary takes at most $n-k$ queries, and it takes only one bit flip for each query. The following discussion shows that $\Pr[\tau' = \tau]$ is lower-bounded by roughly $0.5(n-k)t/n^2 + 0.5$, so the advantage of the adversary is lower-bounded by roughly $0.5(n-k)t/n^2$.

4.2 Attacking the Specifications

To understand the success probability of our attack against the 1st- and 2nd-round specifications, assume that in one of the $n-k$ iterations, the adversary sends (c_b, c'_c) to the decapsulation oracle, such that

$$(0_a, c_b, c'_c) \in e' + \Gamma_2(\alpha_1, \ldots, \alpha_n, g),$$

where $|e'| = t-1$ and $e'_{\mathrm{pos}(0)} = 0$. Let $e \in \mathbb{F}_2^n$ be the unique vector such that $|e + e'| = 1$ and $e_{\mathrm{pos}(0)} = 1$. According to Theorem 1, on input $(0_a, c_b, c'_c)$, Algorithm 1 computes

$$\sigma(x) = \sigma_0 \prod_{e'_i=1} (1 - \alpha_i x),$$

which is of degree $t-1$. There are two cases for the return value of Algorithm 1, one for $\xi = 0$ and one for $\xi = 1$. If $\xi = 0$, Algorithm 1 returns

$$(x^t \sigma(x^{-1}), 0) = (\sigma_0 x \prod_{e'_i=1} (x - \alpha_i), 0),$$

and therefore the decoding algorithm will return e instead of e'. If $\xi = 1$, Algorithm 1 returns

$$(x^{t-1} \sigma(x^{-1}), 1) = (\sigma_0 \prod_{e'_i=1} (x - \alpha_i), 1),$$

and therefore the decoding algorithm will again return e instead of e'. In other words, no matter what the value of ξ is, the decoding algorithm will return e.

Now consider the case when $\mathrm{pos}(0) > k$ and $e_{\mathrm{pos}(0)} = 1$, where e is the error vector used to generate ψ^*. In this case, in one of the $n-k$ iterations carried out by the adversary, the adversary will send to the decapsulation oracle (c_b, c'_c) of the form discussed in the previous paragraph. Therefore, in the iteration, the error vector e will be returned by the decoding algorithm. As $H_\ell(e) = k_e = c_b - e_b$ and $|e| = t$, the session key encapsulated by (c_b, c_c) will be returned by the decapsulation oracle.

To compute $\Pr[\tau = \tau']$, it suffices to compute $\Pr[\tau = 0 \text{ and } \tau = \tau']$ and $\Pr[\tau = 1 \text{ and } \tau = \tau']$. Assuming $\tau = 0$, there is a probability at least

$$\Pr[e_{\mathrm{pos}(0)} = 1] \cdot \Pr[\mathrm{pos}(0) > k]$$

	n	k	#queries	advantage
`ntskem1264`	4096	3328	≤ 768	≥ 0.00146
`ntskem1380`	8192	7152	≤ 1040	≥ 0.00061
`ntskem13136`	8192	6424	≤ 1768	≥ 0.00179

Table 2. The number of decapsulation queries and advantage of our IND-CCA2 attack.

that our attack will recover the session key and thus return $\tau' = 0$. In other words, we have

$$\begin{aligned}\Pr[\tau = 0 \text{ and } \tau = \tau'] &\geq 0.5 \cdot \Pr[e_{\text{pos}(0)} = 1] \cdot \Pr[\text{pos}(0) > k] \\ &= 0.5 \cdot t/n \cdot \Pr[\text{pos}(0) > k].\end{aligned}$$

Assuming $\tau = 1$, the probability that one of $n-k$ decapsulation queries returns K^* is upper bounded by $(n-k)/2^\ell$ as K^* is a random string. Therefore,

$$\Pr[\tau = 1 \text{ and } \tau = \tau'] \geq 0.5 \cdot (2^\ell - (n-k))/2^\ell.$$

What is the actual value of $\Pr[\text{pos}(0) > k]$? Intuitively, $\Pr[\text{pos}(0) > k]$ should be $(n-k)/n$, and this seems to be true according to our experiments. Therefore, under the assumption that $\Pr[\text{pos}(0) > k] = (n-k)/n$, we may conclude that

$$\Pr[\tau = \tau'] \geq 0.5 \cdot (n-k)t/n^2 + 0.5 \cdot (2^\ell - (n-k))/2^\ell.$$

For real parameters, the term $0.5 \cdot (2^\ell - (n-k))/2^\ell$ is usually extremely close to 0.5, so one may also simply consider that the advantage is lower bounded by roughly $0.5(n-k)t/n^2$. Based on the discussion above, it is easy to compute the number of queries and advantage of our attack against the 3 parameter sets of NTS-KEM; The numbers are shown in Table 2.

We note that if the challenger generates a key pair with $\text{pos}(0) \leq k$, the advantage of our attack will be close to 0. If the challenger generates a key pair with $\text{pos}(0) > k$, the advantage of our attack will be lower-bounded by roughly $0.5 \cdot t/n$.

4.3 Attacking the Non-reference Implementations

In addition to the reference implementations, some other implementations are also included in the 1st- and 2nd-round submission packages. These are the implementations under the directories `Additional_Implementation` and `Optimized_Implementation`. We found the following bugs in the code for Algorithm 1 in these implementations.

- In the last of the $2t$ iterations, R is not updated.
- In the first $2t-1$ of the $2t$ iterations, R is updated as follows.

```
if d == 0 OR i < 2L then
    if d == 0 then
        R ← R + 1
    end if
else
    R ← 0
end if
```

These bugs can make R smaller and thus can change the value of ξ. However, as discussed in Section 4.2, our attack is independent of the actual value of ξ, so the numbers in Table 2 still apply.

5 Acknowledgements

The author would like thank Daniel J. Bernstein and Tanja Lange for all their suggestions.

References

[1] Martin Albrecht, Carlos Cid, Kenneth G. Paterson, Cen Jung Tjhai, Martin Tomlinson, *NTS-KEM*, first round submission (2017). URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/NTS_KEM.zip. Citations in this document: §1.

[2] Martin Albrecht, Carlos Cid, Kenneth G. Paterson, Cen Jung Tjhai, Martin Tomlinson, *NTS-KEM*, second round submission (2019). URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/NTS-KEM-Round2.zip. Citations in this document: §1.

[3] Martin Albrecht, Carlos Cid, Kenneth G. Paterson, Cen Jung Tjhai, Martin Tomlinson, *NTS-KEM*, updated second round submission (2019). URL: https://nts-kem.io/. Citations in this document: §1.

[4] Elwyn R. Berlekamp, *Algebraic coding theory*, McGraw-Hill, 1968. MR 38 #6873. Citations in this document: §3.5, §3.6.

[5] Martin Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, Wen Wang, *Classic McEliece* (2020). URL: https://classic.mceliece.org/. Citations in this document: §1, §1.2.

[6] Guido Bertoni, Jean-Sébastien Coron (editors), *Cryptographic hardware and embedded systems—CHES 2013—15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings*, Lecture Notes in Computer Science, 8086, Springer, 2013. ISBN 978-3-642-40348-4.

[7] Joo Yeon Cho, *Implementation of code-based KEMs submitted to NIST on optical communication systems* (2019). URL: https://cbc2019.dii.univpm.it/program. Citations in this document: §1.1.

[8] Ronal Cramer, Victor Shoup, *Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack* (2001). URL: https://eprint.iacr.org/2001/108.

[9] Alexander W. Dent, *A designer'A designer's guide to KEMss guide to KEMs*, in Cryptography and Coding [**16**] (2003), 133–151. URL: https://eprint.iacr.org/2002/174.pdf. Citations in this document: §1.2.

[10] Eiichiro Fujisaki, Tatsuaki Okamoto, *Secure integration of asymmetric and symmetric encryption schemes*, in CRYPTO'99 [**19**] (1999), 537–554.

[11] Eiichiro Fujisaki, Tatsuaki Okamoto, *Secure integration of asymmetric and symmetric encryption schemes*, Journal of Cryptology **26** (2013), 80–101.

[12] Hermann J. Helgert, *Decoding of alternant codes*, IEEE Transactions on Information Theory **23** (1977), 513–514. Citations in this document: §3.6.

[13] Varun Maram, *On the security of NTS-KEM in the quantum random oracle model*, in PQCRYPTO 2020 (to appear) (2017). URL: https://eprint.iacr.org/2020/150.pdf. Citations in this document: §1, §1, §1.1.

[14] James L. Massey, *Shift-register synthesis and BCH decoding*, IEEE Transactions on Information Theory **15** (1969), 122–127. Citations in this document: §3.5, §3.6, §C.

[15] Robert J. McEliece, *A public-key cryptosystem based on algebraic coding theory*, JPL DSN Progress Report (1978), 114–116. URL: http://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF. Citations in this document: §2.5.

[16] Kenneth G. Paterson (editor), *Cryptography and coding, 9th IMA international conference, Cirencester, UK, December 16-18, 2003. proceedings*, Lecture Notes in Computer Science, 2898, Springer, 2003. ISBN 978-3-540-20663-7. See [9].

[17] Bart Preneel (editor), *Advances in cryptology – EUROCRYPT 2000, international conference on the theory and application of cryptographic techniques, Bruges, Belgium, May 14-18, 2000, proceedings*, Lecture Notes in Computer Science, 1807, Springer, 2000. ISBN 978-3-540-67517-4. See [18].

[18] Victor Shoup, *A composition theorem for universal one-way hash functions*, in EUROCRYPT 2000 [**17**] (2000), 445–452. URL: https://www.iacr.org/archive/eurocrypt2000/1807/18070451-new.pdf. Citations in this document: §2.1.

[19] Michael Wiener (editor), *Advances in cryptology — CRYPTO'99, 19th annual international cryptology conference, Santa Barbara, California, USA, August 15–19, 1999, proceedings*, Lecture Notes in Computer Science, 1666, Springer, 1999. ISBN 978-3-540-66347-8. See [10].

[20] Youzhi Xu, *Implementation of Berlekamp-Massey algorithm without inversion*, IEE Proceedings I - Communications, Speech and Vision **138** (1991), 138–140. Citations in this document: §3.6, §3.6.

A Paterson's Message

"We have added a re-encapsulation step during decapsulation, in order to fix a subtle issue in the ROM security proof for NTS-KEM. This issue was identified by Varun Maram from ETH Zurich. This change necessitates the inclusion of the public key as part of the private key and increases the running time of decapsulation. Fortuitously, this change facilitates a QROM proof for NTS-KEM which we plan to make public soon.

[In more detail: our proof did not fully address the possibility that certain adversarially generated ciphertexts not output by encapsulation might decapsulate correctly. This is due to possible behaviour of the decoder, including the Berlekamp-Massey algorithm, when operating beyond its natural decoding capacity. Adding the re-encapsulation step ensures that only correctly generated ciphertexts lead to valid decapsulations; other ciphertexts are implicitly rejected. Our new security proof still tightly relates breaking IND-CCA security of (the new version of) NTS-KEM to breaking one-wayness of the McEliece scheme with the same parameters. We also stress that we are not aware of any concrete attack arising from the issue identified in our proof. Since re-encapsulation makes use of the public key, we now include the public key as part of the private key; an alternative whose cost can be amortised over many invocations of decapsulation is to regenerate the public key from the private key when needed.]"

B NTS-KEM's Berlekamp-Massey Algorithm

Algorithm 1 NTS-KEM's Berlekamp-Massey Algorithm

function BerlekampMassey($\vec{s}$)
Require: $\vec{s} = (s_0, s_1, \ldots, s_{2t-1})$
Require: $\sigma(x) = \sum \sigma_i x^i = 1$
Require: $\beta(x) = \sum \beta_i x^i = x$
Require: $\delta = 1$
Require: $L = R = \xi = 0$
  for $i = 0$ to $2t - 1$ step 1 **do**
    $d \leftarrow \sum_{j=0}^{\min\{i,t\}} \sigma_j s_{i-j}$
    $\psi(x) \leftarrow \delta\sigma(x) - d\beta(x)$
    if $d == 0$ OR $i < 2L$ **then**
      $R \leftarrow R + 1$
      $\beta(x) \leftarrow x\beta(x)$
    else
      $R \leftarrow 0$
      $\beta(x) \leftarrow x\sigma(x)$
      $L \leftarrow i - L + 1$
      $\delta \leftarrow d$
    end if
    $\sigma(x) \leftarrow \psi(x)$
  end for
  if Degree of $\sigma(x) < t - \frac{R}{2}$ **then**
    $\xi \leftarrow 1$
  end if
  $\sigma^*(x) \leftarrow x^{t-\xi}\sigma(x^{-1})$
  return $(\sigma^*(x), \xi)$
end function

C Attacking the Reference Implementations

We found the following bugs in the code for Algorithm 1 in the reference implementations.

- In each of the $2t$ iterations, R is updated in the same way as the pseudocode in Section 4.3.
- $\sigma^*(x)$ is computed as $x^{\mathrm{Deg}(\sigma(x))}\sigma(x^{-1})$.

We also found that after obtaining $(\sigma^*(x), \xi)$, the reference implementations compute the error vector e as follows.

- Set $e = 0 \in \mathbb{F}_2^n$.
- Set $e_i = 1$ for all i such that $\sigma^*(\alpha_i) = 0$ and $\alpha_i \neq 0$.
- Set $e_{\mathrm{pos}(0)} = 1$ if $\xi = 1$.

Note that this is different from what is described in Section 3.5.

Our attack relies on forcing the decoding algorithm to take an input vector which is the sum of a codeword and an error vector e' with $|e'| = t - 1$ and $e'_{\mathrm{pos}(0)} = 0$. In this case, according to Theorem 1, Algorithm 1 computes

$$\sigma(x) = \sigma_0 \prod_{e'_i=1} (1 - \alpha_i x),$$

so the reference implementations compute

$$\sigma^*(x) = \sigma_0 \prod_{e'_i=1} (x - \alpha_i).$$

How about the value of ξ? It turns out that Algorithm 1 computes $\sigma(x)$ in the first $2t-2$ iteration; See [14] for discussions on the number of iterations required to compute the linear feedback shift register. This forces d to be 0 in the last 2 iterations, so we have $R \geq 2$. As $\mathrm{Deg}(\sigma) = t - 1 \geq t - R/2$, the reference implementations computes $\xi = 0$. Therefore, the decoding algorithm returns the weight-$(t-1)$ vector e', and the decapsulation oracle returns $\perp$ or $H_\ell(z, 1_a, c_b, c'_c)$ instead of the session key.

D Implementations

To demonstrate that our attack works against the non-reference implementations, We modified `ntskem_test.c` in the non-reference implementations included in the 1st-round and 2nd-round submission packages. The content of the modified file is available in Appendix F. The modified `testkem_nts` function keeps generating ciphertext-session-key pairs. For each ciphertext, it is checked whether flipping any of the last $n - k$ bits will result in a ciphertext that decapsulates to the same session key. If this happens, a message

```
Original session key returned!
```

will be printed. One can replace the original `ntskem_test.c` by the modified one and compile each non-reference implementation using `make`. Then by running the executables `ntskem-*-test`, the user can see that the message usually shows after trying a several hundreds of ciphertext-session-key pairs.

To demonstrate that our attack works against the specifications, we fixed the bugs in `berlekamp_massey.c` and `nts_kem.c` in the reference implementations included in the 1st-round and 2nd-round submission packages. The content of the modified `berlekamp_massey.c` is available in Appendix E. The modified `berlekamp_massey` function updates R and computes σ^* in the correct way. For `nts_kem.c`, we only change the code segment

```
memset(e_prime, 0, sizeof(e_prime));
for (i=1; i<NTS_KEM_PARAM_N; i++) {
    e_prime[i>>3] |= (CT_is_equal_zero(evals[i]) << (i & 7));
}
e_prime[0] |= ((uint8_t)extended_error);
```

in the `nts_kem_decapsulate` function into the following.

```
memset(e_prime, 0, sizeof(e_prime));
for (i=0; i<NTS_KEM_PARAM_N; i++) {
    e_prime[i>>3] |= (CT_is_equal_zero(evals[i]) << (i & 7));
}
e_prime[0] |= ((uint8_t)extended_error);
```

The modification changes the way the error vector e is computed from σ^* and ξ to the way specified in Section 3.5 (which is equivalent to what is specified in NTS-KEM's specifications). The user can replace `ntskem_test.c` by the modified one replace `berlekamp_massey.c` by the modified one, apply the same change to `nts_kem.c`, do `make` and execute `ntskem-*-test`. Then the user will again see that the message usually shows after trying a several hundreds of ciphertext-session-key pairs.

E The modified `berlekamp_massey.c`

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include "berlekamp_massey.h"
#include "bits.h"

poly *berlekamp_massey(const FF2m *ff2m,
                       const ff_unit *S,
                       int slen,
                       int *extended_error)
{
    poly *ex = NULL;
    ff_unit *sigma = NULL, *beta = NULL, *varphi = NULL;
    ff_unit *src0_ptr = NULL, *src1_ptr = NULL, *dst_ptr = NULL;
    ff_unit d, delta = 1;
    ff_unit inv = 0;
    uint32_t control, d_eq_0;
```

```
int32_t i, j, v, t, L = 0, R = 0;

t = slen >> 1;
sigma  = (ff_unit *)calloc(t+1, sizeof(ff_unit));
beta   = (ff_unit *)calloc(t+1, sizeof(ff_unit));
varphi = (ff_unit *)calloc(t+1, sizeof(ff_unit));
if (!sigma || !beta || !varphi) {
    goto BMA_fail;
}
sigma[0] = 1;   /* sigma(x) = 1 */
beta[1] = 1;    /* beta(x) = x */
*extended_error = 0;

/* Loop until we process all 2t syndromes */
for (i=0; i<slen; i++) {
    /**
     * d = \sum_{i}^{\min{i, t}} sigma_j * S_{i-j}
     **/
    v = CT_min(i, t);
    for (d=0,j=0; j<=v; j++) {
        d = ff2m->ff_add(ff2m, d,
                         ff2m->ff_mul(ff2m, sigma[j], S[i-j]));
    }
    /**
     * varphi(x) = delta.sigma(x) - d.beta(x)
     **/
    for (j=0; j<=t; j++) {
        varphi[j] = ff2m->ff_add(ff2m,
                                 ff2m->ff_mul(ff2m, delta, sigma[j]),
                                 ff2m->ff_mul(ff2m, d, beta[j]));
    }

    d_eq_0 = CT_is_equal_zero((uint32_t)d);   /* d == 0? */
    control = d_eq_0 || CT_is_less_than(i, (L << 1)); /* (d == 0) OR (i < 2L) */

    /**
     * if control is 1 -> beta(x) = x.beta(x)
     * otherwise       -> beta(x) = x.sigma(x)
     **/
    v = t;
    src0_ptr = (ff_unit *)&sigma[t-1];
    src1_ptr = (ff_unit *)&beta[t-1];
    dst_ptr = (ff_unit *)&beta[t];
    while (v-- > 0) {
        *dst_ptr = CT_mux(control, *src1_ptr, *src0_ptr);
        --dst_ptr;
        --src1_ptr;
        --src0_ptr;
    }
    beta[0] = 0x00;

    /**
     * if control is 1 ->
     *     R = R + 1 if d == 0
     * otherwise       ->
     *     R = 0
     *     L = i - L + 1
     *     delta = d
     **/
    L = (int32_t)CT_mux(control, L, i-L+1);
    R = (int32_t)CT_mux(control, R + 1, 0);
    delta = (ff_unit)CT_mux(control, delta, d);

    memcpy(sigma, varphi, (t+1)*sizeof(ff_unit));
}

ex = init_poly(t+1);
if (!ex) {
```

```
        goto BMA_fail;
    }
    ex->degree = t;
    while (ex->degree > 0 && !sigma[ex->degree]) --ex->degree;
    inv = ff2m->ff_inv(ff2m, sigma[0]);

    *extended_error = CT_is_less_than(ex->degree, t - (R>>1));

    for (i=0; i <= t-*extended_error ; i++) {
        if (t-*extended_error-i <= ex->degree)
            ex->coeff[i] = ff2m->ff_mul(ff2m, sigma[t-*extended_error-i], inv);
    }

    ex->degree = t - *extended_error;

BMA_fail:
    if (varphi) {
        memset(varphi, 0, (t+1)*sizeof(ff_unit));
        free(varphi);
    }
    if (beta) {
        memset(beta,  0, (t+1)*sizeof(ff_unit));
        free(beta);
    }
    if (sigma) {
        memset(sigma, 0, (t+1)*sizeof(ff_unit));
        free(sigma);
    }

    return ex;
}
```

F The modified `ntskem_test.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "api.h"
#include "ntskem_test.h"
#include "nts_kem_params.h"
#include "random.h"

int testkem_nts(int iterations)
{
    int i, it = 0;
    uint8_t *pk, *sk;
    uint8_t *encap_key, *decap_key, *ciphertext, *flipped;
    FILE *fp = NULL;
    unsigned char entropy_input[] = {
        0xaa, 0xe7, 0xd7, 0x4e, 0x3c, 0x3a, 0x52, 0xdd,
        0x87, 0xc7, 0x2a, 0xa4, 0x38, 0x54, 0x7e, 0x37,
        0x1e, 0x97, 0x29, 0x78, 0x22, 0xa2, 0xcd, 0x83,
        0x43, 0x64, 0x84, 0xcf, 0x77, 0x6b, 0x9e, 0xa5,
        0x53, 0xf3, 0x50, 0xc5, 0xc7, 0x8d, 0x46, 0xb3,
        0xa5, 0xf2, 0xe3, 0x99, 0x63, 0x10, 0x1d, 0x10
    };
    unsigned char nonce[48];

    fprintf(stdout, "NTS-KEM(%d, %d) Test\n", NTSKEM_M, NTSKEM_T);

    do {
        if ((fp = fopen("/dev/urandom", "r"))) {
            if ((sizeof(entropy_input) !=
                fread(entropy_input, 1, sizeof(entropy_input), fp)) ||
                (sizeof(nonce) != fread(nonce, 1, sizeof(nonce), fp))) {
                break;
```

```
            }
        }
        fclose(fp);

        memcpy(&entropy_input[48-sizeof(it)], &it, sizeof(it));

        fprintf(stdout, "Iteration: %d, Seed: ", it);
        for (i=0; i<sizeof(entropy_input); i++)
          fprintf(stdout, "%02x", entropy_input[i]);
        fprintf(stdout, "\n"); fflush(stdout);

        randombytes_init(entropy_input, nonce, 256);

        pk = (uint8_t *)calloc(CRYPTO_PUBLICKEYBYTES, sizeof(uint8_t));
        sk = (uint8_t *)calloc(CRYPTO_SECRETKEYBYTES, sizeof(uint8_t));
        crypto_kem_keypair(pk, sk);

        ciphertext = (uint8_t *)calloc(CRYPTO_CIPHERTEXTBYTES, sizeof(uint8_t));
        flipped = (uint8_t *)calloc(CRYPTO_CIPHERTEXTBYTES, sizeof(uint8_t));
        encap_key = (uint8_t *)calloc(CRYPTO_BYTES, sizeof(uint8_t));
        decap_key = (uint8_t *)calloc(CRYPTO_BYTES, sizeof(uint8_t));

        crypto_kem_enc(ciphertext, encap_key, pk);

        for (i = 0; i < NTS_KEM_PARAM_M*NTS_KEM_PARAM_T; i++)
        {
            memcpy(flipped, ciphertext, CRYPTO_CIPHERTEXTBYTES);

            flipped[CRYPTO_CIPHERTEXTBYTES-1 - i/8] ^= 1 << (i%8);

            crypto_kem_dec(decap_key, flipped, sk);

            if (0 == memcmp(encap_key, decap_key, CRYPTO_BYTES))
            {
                fprintf(stderr, "Original session key returned!\n");
                getchar();
            }
        }

        free(decap_key);
        free(encap_key);
        free(ciphertext);
        free(flipped);
        free(sk);
        free(pk);
    }
    while (++it < iterations || 1);

    return 0;
}
```