NTS-KEM

Principal submitter:

This submission is from the following team, listed in alphabetical order:

- Martin Albrecht, martin.albrecht@rhul.ac.uk, Information Security Group & Institute for Cyber Security Innovation, Royal Holloway University of London, Egham, Surrey, TW20 0EX, United Kingdom, +44 (0)1784 434455
- Carlos Cid, carlos.cid@rhul.ac.uk, Information Security Group & Institute for Cyber Security Innovation, Royal Holloway University of London, Egham, Surrey, TW20 0EX, United Kingdom, +44 (0)1784 434455
- Kenneth G. Paterson, kenny.paterson@rhul.ac.uk, Information Security Group & Institute for Cyber Security Innovation, Royal Holloway University of London, Egham, Surrey, TW20 0EX, United Kingdom, +44 (0)1784 434455
- Cen Jung Tjhai, cjt@post-quantum.com, PQ Solutions Ltd, 50 Liverpool Street, 5th floor, London, EC2M 7PR, United Kingdom, +44 203 713 7388
- Martin Tomlinson, mt@post-quantum.com, PQ Solutions Ltd, 50 Liverpool Street, 5th floor, London, EC2M 7PR, United Kingdom, +44 203 713 7388

Auxiliary submitters: There are no auxiliary submitters.

Inventors/developers: The inventors/developers of this submission are the same as the principal submitter. Relevant prior work is credited below where appropriate.

Owner: PQ Solutions Ltd, 50 Liverpool Street, 5th floor, London, EC2M 7PR, United Kingdom.

Signature:

BLANK	BLANK	BLANK	BLANK	BLANK
Martin Albrecht	Carlos Cid	Kenneth G. Paterson	Cen Jung Tjhai	Martin Tomlinson

Contents

1 Introduction

This document contains the proposal of the NTS-KEM key encapsulation mechanism (KEM) to the NIST Post-Quantum Cryptography Standardization process. NTS-KEM can be seen as a variant of the McEliece and Niederreiter public-key encryption schemes [McE78, Nie86]. However, compared to those original schemes, NTS-KEM is no longer concerned with the communication of an encrypted message but rather with the secure communication of a random key.

Neither McEliece nor Niederreiter, in their pure form, achieve indistinguishability under either chosen plaintext (IND-CPA) or chosen ciphertext (IND-CCA) attacks. In contrast, NTS-KEM achieves IND-CCA security (as a KEM) in the Random Oracle Model by employing a transform akin to the Fujisaki-Okamoto [FO13] or Dent [Den03] transforms.[1]

The IND-CCA security of NTS-KEM reduces directly to the difficulty of breaking the one-wayness of the McEliece scheme, which is in turn related to the well-known problem of decoding random linear codes. We propose three sets of parameters in this submission, matching three of the security strength categories defined in NIST's Call for Proposals [NIS16]. This allows NTS-KEM to be used in applications with a range of pre- and post-quantum security requirements.

NTS-KEM features efficient key encapsulation and decapsulation operations, the latter due to efficient decoding algorithms for Goppa codes. Ciphertexts are relatively compact, making the scheme suitable for applications with limited communication bandwidth. In common with most code-based schemes, NTS-KEM requires large public keys. This arises from our conservative choice of codes, in which we avoid any cyclic or quasi-cyclic structure. There is a range of applications in which the use of large public keys would not be considered as a handicap and where fast operations and compact ciphertexts are considered more important. Examples of such applications are any that use long-term fixed public keys.

Our choice for a conservative KEM design based on error-correcting codes is motivated by our desire to provide long-term post-quantum security, which is based on a simple, flexible and efficient approach that has been extensively studied and trusted for almost four decades. The security guarantees come from a well-known mathematical problem and good estimates of the complexity of classical decoding algorithms, which can be leveraged to provide suitable levels of security against chosen ciphertext attacks.

Organisation of this document. In Section 2, we define the notation used in this document, and provide the definitions of the mathematical objects and security notions of relevance to this submission. In Section 3, we provide the full specification of NTS-KEM, with the recommended sets of parameters given in Section 4. Section 5 contains the design rationale for NTS-KEM. We discuss and analyse the scheme's implementation and performance

[1] In this submission, we do not distinguish between non-adaptive and adaptive IND-CCA security. The reason being that this distinction does not exist for KEMs as the adversary does not choose any messages in the IND-CCA security game, cf. Section 2.2.

in Section 6. In Section 7, we present and justify our security claims for the different parameter sets, and provide a security analysis of NTS-KEM. We list and discuss the main advantages, together with any known limitations, of our proposal in Section 8. After a list of cited references, we provide some introductory material on binary field arithmetic and how it can be implemented without look-up tables in Appendix A. An overview of the additive Fast Fourier Transform over a finite field, which we use in key-generation and decapsulation, is presented in Appendix B. Appendix C describes how to obtain the first derivative of a polynomial and the greatest common divisor (GCD) of two polynomials, which we need as part of NTS-KEM key-generation. In Appendix D, we discuss how we uniformly shuffle a sequence of finite length and how we generate random bits uniformly. We claim that NTS-KEM achieves IND-CCA security in the Random Oracle Model and this is proved in Appendix E. In Appendix F, we describe how to generate the KATs and outline the intermediate values that we produce.

2 Notation and Definitions

In this section we outline the notation adopted in this document, and provide the definition of the main mathematical objects and security notions of relevance to our submission.

We denote by $\mathbb{F}_2$ the field with two elements, and by $\mathbb{F}_{2^m}$ an extension field of $\mathbb{F}_2$ with 2^m elements. If $\mathbb{F}$ is a field, let $\mathbb{F}[x]$ be the ring of univariate polynomials with coefficients in $\mathbb{F}$. Then for an extension field of order 2^m, there exists an irreducible polynomial $f(x) \in \mathbb{F}_2[x]$ of degree m, such that $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/f(x)$. If we construct an extension field $\mathbb{F}_{2^m}$ via the polynomial $f(x)$, then any element of $\mathbb{F}_{2^m}$ can be represented as $b_0 + b_1\beta + b_2\beta^2 + \cdots + b_{m-1}\beta^{m-1}$, where $b_i \in \mathbb{F}_2$ and $\beta \in \mathbb{F}_{2^m}$ is a root of $f(x)$. A *primitive* element $\alpha \in \mathbb{F}_{2^m}$ is one that generates the cyclic group of $2^m - 1$ elements under multiplication of $\mathbb{F}_{2^m}$.

We denote by $\mathbb{F}_2^n$ the n-dimensional vector space with entries in $\mathbb{F}_2$, and by $\mathbb{F}_2^{k\times n}$ the kn-dimensional vector space of matrices with k rows and n columns with entries in $\mathbb{F}_2$. We denote vectors of $\mathbb{F}_2^n$ in bold lowercase, for example $\mathbf{e} = (e_0, e_1, \ldots, e_{n-1}) \in \mathbb{F}_2^n$; and matrices of $\mathbb{F}_2^{k\times n}$ in bold uppercase, for example $\mathbf{G} \in \mathbb{F}_2^{k\times n}$. If $\mathbf{G} \in \mathbb{F}_2^{k\times n}$ is a matrix, we denote the transpose of $\mathbf{G}$ by $\mathbf{G}^T \in \mathbb{F}_2^{n\times k}$. The i-th row of matrix $\mathbf{G}$ is denoted by $\mathbf{G}_i = (g_{i,0}, g_{i,1}, \ldots, g_{i,n-1})$. The *Hamming weight* of a vector $\mathbf{e}$ is the number of non-zero components in the vector and is denoted by $\mathrm{hw}(\mathbf{e})$. Given a vector $\mathbf{e}$ of length n over a field $\mathbb{F}$, and positive integers $\ell < k < n$, we adopt the following notation to denote the partition of $\mathbf{e}$ into three "sub-vectors": $\mathbf{e} = (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c)$, where $\mathbf{e}_a \in \mathbb{F}^{k-\ell}$, $\mathbf{e}_b \in \mathbb{F}^{\ell}$ and $\mathbf{e}_c \in \mathbb{F}^{n-k}$. More generally, if $\mathbf{v} \in \mathbb{F}^{n_1}$ and $\mathbf{w} \in \mathbb{F}^{n_2}$ are vectors over $\mathbb{F}$, we will denote by $(\mathbf{v} \mid \mathbf{w})$ the vector in $\mathbb{F}^{n_1+n_2}$ constructed as the "concatenation" of $\mathbf{v}$ and $\mathbf{w}$.

A permutation π on a ordered sequence of n elements may be represented by a permutation matrix $\mathbf{P} \in \mathbb{F}_2^{n\times n}$ where there is exactly one entry of 1 in each row and column and 0 elsewhere. It may also be represented by a *permutation vector* $\mathbf{p} = (p_0, p_1, \ldots, p_{n-1})$ where p_i denotes the row of $\mathbf{P}$ that has a 1 at column i. Then, given the sequence $\mathbf{b} = (b_0, b_1, \ldots, b_{n-1})$, we denote the permuted sequence $\mathbf{b}' = \mathbf{b} \cdot \mathbf{P} = \pi_{\mathbf{p}}(\mathbf{b})$ such that $b'_i = b_{p_i}$, and the inverse permutation is given by $\mathbf{b} = \mathbf{b}' \cdot \mathbf{P}^{-1} = \pi_{\mathbf{p}}^{-1}(\mathbf{b}')$ such that $b_{p_i} = b'_i$. Likewise for a matrix $\mathbf{M}$, $\pi_{\mathbf{p}}(\mathbf{M}) = \mathbf{M} \cdot \mathbf{P}$ reorders the columns of $\mathbf{M}$ according to the permutation vector $\mathbf{p}$.

If b is a binary string $b_0 b_1 \dots b_{m-1}$ of length m, then b_0 represents its least significant bit, and b_{m-1} the most significant bit. When representing vectors over $\mathbb{F}_2$ or elements of the field extension $\mathbb{F}_{2^m}$ as binary strings, we use the following convention: $(b_0, b_1, \dots, b_{m-1}) \leftrightarrow b_0 b_1 \dots b_{m-1}$ and $b_0 + b_1\beta + \cdots + b_{m-1}\beta^{m-1} \leftrightarrow b_0 b_1 \dots b_{m-1}$. Finally, given a set X, we will denote by $x \leftarrow_{\$} X$ the operation of sampling an element $x \in X$ uniformly at random.

2.1 Code-based Cryptography

Let $\mathcal{C} = [n, k, d]_2$ be a linear error-correcting code over $\mathbb{F}_2$ of length n and dimension k, with minimal distance d. The code $\mathcal{C}$ is capable of correcting at most $\tau = \lfloor \frac{d-1}{2} \rfloor$ errors, and can be described by a generator matrix $\mathbf{G} \in \mathbb{F}_2^{k \times n}$, or a parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k) \times n}$, such that $\mathbf{G} \cdot \mathbf{H}^T = \mathbf{0}$. Then, a vector $\mathbf{w} \in \mathbb{F}_2^k$ can be encoded as a codeword in $\mathcal{C}$ as $\mathbf{c} = \mathbf{w} \cdot \mathbf{G} \in \mathbb{F}_2^n$. Moreover, for any codeword $\mathbf{c}$ in $\mathcal{C}$, we have $\mathbf{c} \cdot \mathbf{H}^T = \mathbf{0}$. More generally, given any vector $\mathbf{v} \in \mathbb{F}_2^n$, the vector $\mathbf{s} = \mathbf{v} \cdot \mathbf{H}^T \in \mathbb{F}_2^{n-k}$ is called a *syndrome*. The problem of syndrome decoding is to find a vector of minimum weight $\mathbf{v} \in \mathbb{F}_2^n$ such that $\mathbf{s} = \mathbf{v} \cdot \mathbf{H}^T$ given a syndrome $\mathbf{s}$.

Definition 1. *A binary separable Goppa code $\mathcal{C}_\mathcal{G}$ is a class of $[n, k, d]_2$ linear error-correcting codes defined by a Goppa polynomial $G(z) = g_0 + g_1 z + \cdots + g_\tau z^\tau \in \mathbb{F}_{2^m}[z]$ and $d = 2\tau + 1$. The polynomial $G(z)$ has the following properties:*

- *$G(z)$ has no roots in $\mathbb{F}_{2^m}$, which implies $n = 2^m$;*
- *$G(z)$ has no repeated roots in any extension field, which guarantees that $\mathcal{C}_\mathcal{G}$ is capable of correcting up to τ errors.*

Let $(a_0, a_1, \dots, a_{n-1})$ be a sequence of all elements in $\mathbb{F}_{2^m}$. We may write the parity-check matrix $\mathbf{H}_m \in \mathbb{F}_{2^m}^{\tau \times n}$ of $\mathcal{C}_\mathcal{G}$ using $G^2(z)$ as follows [MS77]:

$$\mathbf{H}_m = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ a_0 & a_1 & a_2 & \cdots & a_{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_0^{\tau-1} & a_1^{\tau-1} & a_2^{\tau-1} & \cdots & a_{n-1}^{\tau-1} \end{bmatrix} \cdot \begin{bmatrix} G(a_0)^{-2} & 0 & 0 & \cdots & 0 \\ 0 & G(a_1)^{-2} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & G(a_{n-1})^{-2} \end{bmatrix}$$
$$= \begin{bmatrix} G(a_0)^{-2} & G(a_1)^{-2} & G(a_2)^{-2} & \cdots & G(a_{n-1})^{-2} \\ a_0 G(a_0)^{-2} & a_1 G(a_1)^{-2} & a_2 G(a_2)^{-2} & \cdots & a_{n-1} G(a_{n-1})^{-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_0^{\tau-1} G(a_0)^{-2} & a_1^{\tau-1} G(a_1)^{-2} & a_2^{\tau-1} G(a_2)^{-2} & \cdots & a_{n-1}^{\tau-1} G(a_{n-1})^{-2} \end{bmatrix}. \tag{1}$$

Let $B(a_i) = (b_{i0}, b_{i1}, \dots, b_{i(m-1)})$ be a representation of a_i over $\mathbb{F}_2$, i.e.

$$a_i = b_{i0} + b_{i1}\beta + b_{i2}\beta^2 + \cdots + b_{i(m-1)}\beta^{m-1},$$

where $b_{ij} \in \mathbb{F}_2$. Then by replacing each entry of $\mathbf{H}_m$ with $B(\cdot)^T$, we have the binary parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{m\tau \times n}$, which has rank $m\tau = n - k$. The generator matrix $\mathbf{G} \in \mathbb{F}_2^{k \times n}$ can then be easily obtained from $\mathbf{H}$.

With knowledge of its structure, binary Goppa codes can be efficiently decoded by using Patterson's method [Pat75] or the Berlekamp-Massey algorithm [Ber68, Mas69]. However, if the code structure is hidden, then decoding binary Goppa codes is expected to be as hard as decoding a random linear code. In this case, the best currently known algorithms are based on the technique known as *information-set decoding*, originally proposed by Prange [Pra62].

In 1978 [McE78], McEliece proposed a public-key encryption scheme whose security is based on two hardness assumptions. Firstly, the intractability of decoding a random linear code, a problem which is known to be NP-hard [BMvT78]. Secondly, the difficulty of distinguishing an unknown and permuted binary Goppa code from a random code. The McEliece scheme may be described as follows:

KGen: generate a Goppa polynomial $G(z)$ of degree τ, which defines a binary Goppa code $\mathcal{C}_\mathcal{G}$ with generator matrix $\mathbf{G}' \in \mathbb{F}_2^{k \times n}$. Let $\mathbf{S}$ be a non-singular matrix in $\mathbb{F}_2^{k \times k}$ and $\mathbf{P}$ be a permutation matrix in $\mathbb{F}_2^{n \times n}$, both generated at random. Then, define $\mathbf{G} = \mathbf{S} \cdot \mathbf{G}' \cdot \mathbf{P}$. The public key is given by $pk = (\mathbf{G}, \tau)$ and the private key is $sk = (G(z), \mathbf{S}^{-1}, \mathbf{P}^{-1})$.

Enc: to encrypt a message $\mathbf{m} \in \mathbb{F}_2^k$, sample $\mathbf{e} \in \mathbb{F}_2^n$ with Hamming weight τ and output the ciphertext $\mathbf{c} = \mathbf{m} \cdot \mathbf{G} + \mathbf{e} \in \mathbb{F}_2^n$.

Dec: to recover the message $\mathbf{m}$, compute $\mathbf{c}' = \mathbf{c} \cdot \mathbf{P}^{-1} = \mathbf{m} \cdot \mathbf{S} \cdot \mathbf{G}' + \mathbf{e} \cdot \mathbf{P}^{-1}$, and decode $\mathbf{c}'$ using an algebraic decoder for $\mathcal{C}_\mathcal{G}$ to recover the permuted $\mathbf{e}$, and hence $\mathbf{m}' = (\mathbf{m} \cdot \mathbf{S}) \in \mathbb{F}_2^k$. Finally, recover $\mathbf{m} = \mathbf{m}' \cdot \mathbf{S}^{-1}$ and output $\mathbf{m}$.

A variant of the McEliece scheme reducing the public key size was introduced by Neiderreiter in 1986 [Nie86]. In the Niederreiter scheme the public key is the (usually) smaller parity-check matrix instead of the generator matrix of a code. Moreover, in the Niederreiter scheme, the message is encoded as a low-weight vector $\mathbf{u} \in \mathbb{F}_2^n$, and the ciphertext is represented as the syndrome of $\mathbf{u}$ rather than as the information bits of a codeword corrupted with a random τ-weight error vector $\mathbf{e}$. In particular, the Niederreiter scheme may be described as follows:

KGen: generate a Goppa polynomial $G(z)$ of degree τ, which defines a binary Goppa code $\mathcal{C}_\mathcal{G}$ with parity-check matrix $\mathbf{H}' \in \mathbb{F}_2^{(n-k) \times n}$. Let $\mathbf{S}$ be a non-singular matrix in $\mathbb{F}_2^{(n-k) \times (n-k)}$, and $\mathbf{P}$ be a permutation matrix in $\mathbb{F}_2^{n \times n}$, both generated at random. Then define $\mathbf{H} = \mathbf{S} \cdot \mathbf{H}' \cdot \mathbf{P}$. The public key is given by $pk = (\mathbf{H}, \tau)$ and the private key is $sk = (G(z), \mathbf{S}^{-1}, \mathbf{P}^{-1})$.

Enc: to encrypt the message $\mathbf{m}$, encode it as a vector $\mathbf{u} \in \mathbb{F}_2^n$ with Hamming weight at most τ using a keyless, invertible encoding scheme ϕ. The ciphertext is $\mathbf{c} = \mathbf{H} \cdot \mathbf{u}^T \in \mathbb{F}_2^{n-k}$.

Dec: to recover the message $\mathbf{m}$, compute $\mathbf{S}^{-1} \cdot \mathbf{c} = \mathbf{H}' \cdot \mathbf{P} \cdot \mathbf{u}^T$, perform a syndrome decoding algorithm on $\mathbf{S}^{-1} \cdot \mathbf{c}$ to recover $\mathbf{P} \cdot \mathbf{u}^T$, then compute $\mathbf{P}^{-1} \cdot \mathbf{P} \cdot \mathbf{u}^T$ to recover $\mathbf{u}$. Finally, recover $\mathbf{m}$ from $\mathbf{u}$ as $\mathbf{m} = \phi^{-1}(\mathbf{u})$.

The Neiderreiter scheme was originally proposed to be used in conjunction with generalised Reed-Solomon codes. However, this construction was broken in [SS92]. On the other hand, Li et al. [LDW94] have shown that the security of the Niederreiter scheme and that of the McEliece scheme are equivalent. Thus, considering the Niederreiter scheme using Goppa codes comes with no loss of security compared to McEliece. Overbeck and Sendrier provide a comprehensive overview of code-based cryptography in [OS09].

Our NTS-KEM proposal can be considered as a mixture of McEliece and Niederreiter schemes combined with a transform akin to the Fujisaki-Okamoto [FO13] or Dent [Den03] transforms, resulting in a key encapsulation mechanism which is resistant against chosen ciphertext attacks.

2.2 Security Notions

A basic security notion for a public-key encryption (PKE) scheme is IND-CPA: *indistinguishability under chosen plaintext attacks*. This security notion states that it is hard for an adversary $\mathcal{A}$ to decide which of two messages of its choosing $\mathbf{m}_0, \mathbf{m}_1$ is encrypted in a given ciphertext $\mathbf{c}$.

In its basic form, the McEliece PKE scheme is not IND-CPA secure, and only achieves a weaker notion of security: *one-wayness* (OW). This notion states that an attacker cannot recover the underlying $\mathbf{m}$ for some ciphertext $\mathbf{c}$. The notion is formalised in the game below, where we write $\{0,1\}^{\mathsf{poly}(\lambda)}$ for the plaintext space, indicating that it consists of bit-strings of some length that depends on the security parameter via some polynomial function. Note that messages are random strings in the $\mathrm{OW}^{\mathcal{A}}_{\mathsf{Enc}}$ game and are not chosen by the adversary.

```
$\mathrm{OW}^{\mathcal{A}}_{\mathsf{Enc}}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$
$\mathbf{m} \leftarrow\!\!\$\ \{0,1\}^{\mathsf{poly}(\lambda)}$
$\mathbf{c} \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{pk}, \mathbf{m})$
$\mathbf{m}' \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, \mathsf{pk}, \mathbf{c})$
return $(\mathbf{m}' = \mathbf{m})$
```

We say that an adversary $\mathcal{A}$ is a (t, ε)-adversary against OW security of a PKE scheme if that adversary causes the $\mathrm{OW}^{\mathcal{A}}_{\mathsf{Enc}}$ game to output ’“1” with probability at least ε (where $0 < \varepsilon \leq 1$) and runs in time at most t.

A PKE scheme is said to be (t, ε)-secure with respect to a given security notion (for example, IND-CPA or OW security) if no (t, ε)-adversary exists for that notion. We sometimes simply write that a scheme is, for example, IND-CPA secure, if for some appropriate choice of parameters, we can prove (under some reasonable assumptions) that it is (t, ε)-secure where t can be made sufficiently high (e.g. $t = 2^{128}$) and ε sufficiently low (e.g. $\varepsilon = 2^{-128}$).

Our proposal is not for a PKE scheme, but rather for a *Key Encapsulation Mechanism* (KEM). A KEM is a public-key scheme to securely encapsulate bit-strings (keys) chosen uniformly at random. These keys can then be used in a symmetric encryption scheme commonly called a *Data Encapsulation Mechanism* (DEM), producing what is known as the KEM-DEM paradigm for public-key encryption. Formally, a KEM is defined as follows [Den03]:

Definition 2. *A Key Encapsulation Mechanism (KEM) consists of a triple of algorithms* $(\mathsf{KGen}, \mathsf{Encap}, \mathsf{Decap})$ *with the following syntax and operation:*

- *The Key Generation algorithm* KGen *takes as input a security parameter* 1^λ *and outputs a public/private key-pair* $(\mathsf{pk}, \mathsf{sk})$*.*
- *The Encapsulation algorithm* Encap *takes as input a public key* pk *and outputs an encapsulated key and ciphertext* (K, C)*.*
- *The Decapsulation algorithm* Decap *takes as input a ciphertext* C *and a private key* sk*, and outputs a key* K *encapsulated in* C *(or an error message indicating a decapsulation failure).*

Informally, we say that a KEM is (t, ε)-secure in the IND-CPA sense if no (t, ε)-adversary can decide whether a given pair (K, C) is such that C encapsulates K. We can also define a one-wayness security notion for KEMs, which like that for PKE schemes, informally states that it is difficult to recover an encapsulated key K from its ciphertext C. This is captured in the following game:

```
$\mathrm{OW}^{\mathcal{A}}_{\mathsf{KEM}}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^\lambda)$
$(K_0, C^*) \leftarrow_{\$} \mathsf{Encap}(\mathsf{pk})$
$K_1 \leftarrow_{\$} \mathcal{A}(1^\lambda, \mathsf{pk}, C^*)$
return $(K_1 = K_0)$
```

The desired security notion for a PKE scheme or a KEM is IND-CCA security, i.e. *indistinguishability under chosen ciphertext attacks.* In the KEM version of this security notion, in addition to a challenge ciphertext C^* and one of K_0 or K_1, the attacker is also given access to an oracle that returns the output of $\mathsf{Decap}(C', \mathsf{sk})$ for any $C' \neq C^*$ (with this restriction being needed to prevent trivial wins). We denote this capability as $\mathcal{A}^{\mathsf{Decap}(\cdot, \mathsf{sk})}(1^\lambda, \mathsf{pk}, K_b, C^*)$ below and require that such an adversary cannot decide whether K_0 or K_1 was encapsulated. Formally, a (t, ε)-adversary against the IND-CCA security of a KEM causes the following game to return “1” with probability at least $1/2 + \varepsilon$ (where $0 < \varepsilon \leq 1/2$) and runs in time at most t.

```
$\text{IND-CCA}^{\mathcal{A}}_{\mathsf{KEM}}$
$b \leftarrow_{\$} \{0,1\}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^{\lambda})$
$(K_0, C^*) \leftarrow_{\$} \mathsf{Encap}(\mathsf{pk})$
$K_1 \leftarrow_{\$} \{0,1\}^{|K_0|}$
$b' \leftarrow_{\$} \mathcal{A}^{\mathsf{Decap}(\cdot,\mathsf{sk})}(1^{\lambda}, \mathsf{pk}, K_b, C^*)$
return $(b' = b)$
```

We say that a KEM is (t, ε)-secure in the IND-CCA sense if no (t, ε)-adversary exists.

3 NTS-KEM: algorithm specification

NTS-KEM is a key encapsulation mechanism targeting IND-CCA security derived as a combination of the McEliece and Niederreiter PKE schemes. In this section, we specify the scheme's three operations: *Key Generation*, *Encapsulation* and *Decapsulation*. For each of the three operations, we first specify all steps to fully carry out the operation, followed by a more detailed description of the algorithmic aspects of each step, with suggested subroutines to implement the full operations.

The public parameters for an instance of NTS-KEM are:

- $n = 2^m$: a power-of-two, positive integer, which denotes the length of codewords.
- τ: a positive integer denoting the number of errors corrected by the code.
- $f(x)$: an irreducible polynomial of degree m over $\mathbb{F}_2$, defining the extension field $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/f(x)$.
- $\ell = 256$: a positive integer, set to 256 in this submission, which denotes the length of the random key to be encapsulated.

We require that $\log_2 \binom{n}{\tau} \geq \ell$, and $\ell < k < n$, where $k = n - \tau m$. The proposed parameter sets in this submission are given in Section 4.

The NTS-KEM scheme makes use of a pseudorandom bit generator to produce ℓ-bit binary strings, which we denote by $H_\ell(\cdot)$. In this submission, we have set $\ell = 256$, and use the SHA3–256 hash function [NIS15] to implement $H_\ell(\cdot)$.

We note that, apart from the $H_\ell(\cdot)$ function, we chose not to prescribe in this specification any particular procedure for sampling pseudorandom values required for the different operations described below (e.g. for generating random Goppa codes and permutations in key generation; and random error vectors in encapsulation). The structure of NTS-KEM permits us to consider entropy generation as being out of scope of this submission. That said, a natural way in which implementations may sample entropy for NTS-KEM is by expanding a random seed using the SHAKE256 extendable output function [NIS15], which is also based on the SHA3–256 hash function.

3.1 Key Generation

The procedure to generate a NTS-KEM key-pair is as follows:

1. Randomly generate a monic Goppa polynomial of degree τ

$$G(z) = g_0 + g_1 z + \cdots + g_{\tau-1} z^{\tau-1} + z^{\tau},$$

where $g_i \in \mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/f(x)$, with $g_0 \neq 0$. The polynomial $G(z)$ defines a binary Goppa code $\overline{\mathcal{C}_{\mathcal{G}}}$ of length $n = 2^m$, dimension $k = n - \tau m$, capable of correcting up to τ errors.

2. Randomly generate a permutation vector $\mathbf{p}$ of length n, representing a permutation $\pi_{\mathbf{p}}$ on the set of n elements.

3. Construct a generator matrix in the reduced row echelon form $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{Q}]$ of a *permuted* code $\mathcal{C}_{\mathcal{G}}$ as follows:

   (a) let $\mathbf{a}'$ be a ordered sequence of all elements of $\mathbb{F}_{2^m}$, as given in equation (3) in Section 3.1.1, and $\mathbf{a} = \pi_{\mathbf{p}}(\mathbf{a}') = (a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}) \in \mathbb{F}_{2^m}^n$ be the sequence obtained by re-ordering the elements of $\mathbf{a}'$ according to $\pi_{\mathbf{p}}$.

   (b) construct the parity-check matrix $\mathbf{H}_m \in \mathbb{F}_{2^m}^{\tau \times n}$ using the sequence $\mathbf{a}$, as described in equation (1). Let $\mathbf{h} = (h_{p_0}, h_{p_1}, \ldots, h_{p_{n-1}}) \in \mathbb{F}_{2^m}^n$ be the first row of $\mathbf{H}_m$.

   (c) transform $\mathbf{H}_m$ to $\mathbf{H} \in \mathbb{F}_2^{m\tau \times n}$ using operator $B(\cdot)^T$, as described in Section 2.1.

   (d) transform $\mathbf{H}$ to reduced row echelon form, re-ordering its columns if necessary, such that the identity matrix $\mathbf{I}_{n-k}$ occupies the last $n-k$ columns of $\mathbf{H}$. If ρ is the permutation representing this re-ordering of columns, apply the same re-ordering to the vectors $\mathbf{a}$, $\mathbf{h}$ and $\mathbf{p}$, i.e. make $\mathbf{a} = \rho(\mathbf{a})$, $\mathbf{h} = \rho(\mathbf{h})$ and $\mathbf{p} = \rho(\mathbf{p})$.[2]

   (e) construct the generator matrix $G = [\mathbf{I}_k \mid \mathbf{Q}] \in \mathbb{F}_2^{k \times n}$ of the permuted code $\mathcal{C}_{\mathcal{G}}$ from the parity-check matrix[3] $\mathbf{H} = [\mathbf{Q}^T \mid \mathbf{I}_{n-k}]$.

4. Partition the vectors $\mathbf{a}$ and $\mathbf{h}$ as $\mathbf{a} = (\mathbf{a}_a \mid \mathbf{a}_b \mid \mathbf{a}_c)$ and $\mathbf{h} = (\mathbf{h}_a \mid \mathbf{h}_b \mid \mathbf{h}_c)$, where $\mathbf{a}_a, \mathbf{h}_a \in \mathbb{F}_{2^m}^{k-\ell}$, $\mathbf{a}_b, \mathbf{h}_b \in \mathbb{F}_{2^m}^{\ell}$ and $\mathbf{a}_c, \mathbf{h}_c \in \mathbb{F}_{2^m}^{n-k}$. Finally, define

$$\mathbf{a}^* = (\mathbf{a}_b \mid \mathbf{a}_c) \quad \text{and} \quad \mathbf{h}^* = (\mathbf{h}_b \mid \mathbf{h}_c)\,.$$

The NTS-KEM public and private keys are given as follows.

[2] Although not required for the implementation of the scheme itself, for the purpose of showing correctness of NTS-KEM (Section 3.4) we will assume that the permutation ρ is also applied to the parity-check matrix $\mathbf{H}_m \in \mathbb{F}_{2^m}^{\tau \times n}$, i.e. we make $\mathbf{H}_m = \rho(\mathbf{H}_m)$.

[3] Strictly speaking, $\mathbf{H} = [-\mathbf{Q}^T \mid \mathbf{I}_{n-k}]$; however since $-1 = 1 \mod 2$, we omit the negative sign.

- The public key is pk $= (\mathbf{Q}, \tau, \ell)$, where $\mathbf{Q} \in \mathbb{F}_2^{k\times(n-k)}$ and τ, ℓ are positive integers (determined in the parameter sets).
- The private key is sk $= (\mathbf{a}^*, \mathbf{h}^*, \mathbf{p})$, where $\mathbf{a}^*, \mathbf{h}^* \in \mathbb{F}_{2^m}^{n-k+\ell}$, and $\mathbf{p} \in \mathbb{F}_{2^m}^n$.

3.1.1 Detailed description

We provide more details concerning steps 1–3 above, with some suggested subroutines to perform required operations.

To randomly generate a monic Goppa polynomial $G(z)$ of degree τ (**Step 1**), the coefficients g_i for $i = \{0, 1, \ldots, \tau - 1\}$ are uniformly sampled from $\mathbb{F}_{2^m}$, with $g_0 \neq 0$. We uniformly sample m bits for each coefficient g_i, and as a result at least $m\tau$ bits of random data are required per polynomial generation trial. The generation procedure is outlined below:

(a) Sample uniformly at random $m\tau$ bits of random data and sequentially assign m bits for g_i in $\mathbf{g} = (g_0, g_1, \ldots, g_{\tau-1})$.

(b) Set $g_\tau = 1$ and let $G(z) = \sum_{i=0}^{\tau} g_i z^i$.

(c) Check the validity of $G(z)$. A Goppa polynomial is considered valid if:

   - $g_0 \neq 0$.
   - $G(z)$ has no roots in $\mathbb{F}_{2^m}$. This condition guarantees that $n = 2^m$. An additive FFT [GM10] (see Appendix B) is an efficient method to evaluate whether or not $G(z)$ has zeros in $\mathbb{F}_{2^m}$.
   - $G(z)$ has no repeated roots in any extension field. This condition guarantees that the binary Goppa code is capable of correcting up to τ errors. This condition is met if GCD $\left(G(z), \frac{d}{dz}G(z)\right) = 1$. Refer to Appendix C on how to realise the polynomial derivative of $G(z)$ and the GCD.

   If either one of the above conditions is not met, restart at (a).

To generate a length n permutation vector $\mathbf{p}$ (**Step 2**), we may proceed as follows:

(a) Initialise $\mathbf{p} = (p_0, p_1, \ldots, p_{n-1}) = (0, 1, \ldots, n-1)$.

(b) Shuffle $\mathbf{p}$ using the Fisher-Yates shuffling algorithm; see Appendix D. The algorithm generates unbiased permutations of n elements in linear time [Knu97].

We note that the permutation vector $\mathbf{p}$ may also be equivalently represented by a permutation matrix $\mathbf{P}$.

The generator matrix $\mathbf{G} \in \mathbb{F}_2^{k\times n}$ (**Step 3**) can be obtained as follows.

(a) Let $\beta \in \mathbb{F}_{2^m}$ be a root of $f(x)$, where $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/f(x)$ and B be a basis of $\mathbb{F}_{2^m}$, $B = \langle \beta^{(m-1)}, \dots, \beta, 1\rangle$. The i-th element of $\mathbb{F}_{2^m}$ in the basis of B is defined by

$$B[i] = \{b_0\beta^{(m-1)} + b_1\beta^{(m-2)} + \ldots + b_{m-2}\beta + b_{m-1} \ : \ b_j \in \{0,1\}\}, \tag{2}$$

where $i = \sum_{j=0}^{m-1} b_j 2^j$.

(b) Let $\mathbf{a}'$ be the sequence of all elements of $\mathbb{F}_{2^m}$ given by

$$\mathbf{a}' = (a_0, a_1, a_2, \dots, a_{n-2}, a_{n-1}) = (B[0], B[1], B[2], \dots, B[n-2], B[n-1]), \tag{3}$$

where $B[i]$ is defined above.

(c) Obtain the vector $\bar{\mathbf{h}} = \text{AdditiveFFT}(G(z)) = (G(B[0]), G(B[1]), \dots, G(B[n-1]))$; see Algorithm 4 in Appendix B. Here, $\bar{\mathbf{h}}$ contains the evaluation of $G(z)$ on all elements of $\mathbb{F}_{2^m}$ in the order defined by basis B, i.e. $(B[0], B[1], \dots, B[n-1])$.

(d) Apply inversion and squaring on $\bar{\mathbf{h}}$, resulting in

$$\mathbf{h}' = (\bar{h}_0^{-2}, \bar{h}_1^{-2}, \dots, \bar{h}_{n-1}^{-2}).$$

(e) Permute the sequences $\mathbf{a}'$ and $\mathbf{h}'$ with permutation $\mathbf{p}$:

$$\begin{aligned} \mathbf{a} &= \pi_{\mathbf{p}}(\mathbf{a}') = (a_{p_0}, a_{p_1}, \dots, a_{p_{n-1}}) \\ \mathbf{h} &= \pi_{\mathbf{p}}(\mathbf{h}') = (h_{p_0}, h_{p_1}, \dots, h_{p_{n-1}}). \end{aligned}$$

(f) Construct the permuted parity-check matrix $\mathbf{H}_m$ with $\mathbf{a}$ and $\mathbf{h}$ as per equation (1) as follows:

$$\mathbf{H}_m = \begin{bmatrix} h_{p_0} & h_{p_1} & \cdots & h_{p_{n-1}} \\ a_{p_0}h_{p_0} & a_{p_1}h_{p_1} & \cdots & a_{p_{n-1}}h_{p_{n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p_0}^{\tau-1}h_{p_0} & a_{p_1}^{\tau-1}h_{p_1} & \cdots & a_{p_{n-1}}^{\tau-1}h_{p_{n-1}} \end{bmatrix}. \tag{4}$$

(g) Transform $\mathbf{H}_m$ to $\mathbf{H} \in \mathbb{F}_2^{m\tau \times n}$ by applying operator $B(\cdot)^T$ on each component of $\mathbf{H}_m$.

(h) Transform $\mathbf{H}$ to reduced row echelon form by means of row operations such that $\mathbf{H}$ forms an $(n-k) \times (n-k)$ identity matrix $\mathbf{I}_{n-k}$. Let $\mathbf{r}$ be a permutation vector representing the permutation ρ, required to obtain $\mathbf{I}_{n-k}$ on the last $n-k$ columns of $\mathbf{H}$. Reorder the vectors $\mathbf{a}$, $\mathbf{h}$ and $\mathbf{p}$ following the permutation vector $\mathbf{r}$.

(i) The parity-check matrix of the permuted code $\mathcal{C}_{\mathcal{G}}$ is now in the form $\mathbf{H} = [\mathbf{Q}^T \mid \mathbf{I}_{n-k}]$. Obtain the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{Q}]$.

The matrix $\mathbf{G}$ and the vectors $\mathbf{a}, \mathbf{h}, \mathbf{p}$ are then used to produce a NTS-KEM public and private key pair $\text{pk} = (\mathbf{Q}, \tau, \ell)$ and $\text{sk} = (\mathbf{a}^*, \mathbf{h}^*, \mathbf{p})$.

3.2 Encapsulation

Given a NTS-KEM public key $\mathrm{pk} = (\mathbf{Q}, \tau, \ell)$, the encapsulation process produces two vectors over $\mathbb{F}_2$, one of which is a random vector $\mathbf{k}_r$, where $|\mathbf{k}_r| = \ell = 256$; the other is the ciphertext $\mathbf{c}^*$ encapsulating $\mathbf{k}_r$.

The NTS-KEM encapsulation algorithm is as follows.

1. Generate uniformly at random an error vector $\mathbf{e} \in \mathbb{F}_2^n$ with Hamming weight τ (for example, following Algorithm 1 below).
2. Partition $\mathbf{e}$ as $\mathbf{e} = (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c)$, where $\mathbf{e}_a \in \mathbb{F}_2^{k-\ell}$, $\mathbf{e}_b \in \mathbb{F}_2^{\ell}$ and $\mathbf{e}_c \in \mathbb{F}_2^{n-k}$.
3. Compute $\mathbf{k}_e = H_\ell(\mathbf{e}) \in \mathbb{F}_2^{\ell}$.
4. Construct the message vector $\mathbf{m} = (\mathbf{e}_a \mid \mathbf{k}_e) \in \mathbb{F}_2^{k}$.
5. Perform systematic encoding of $\mathbf{m}$ with $\mathbf{Q}$:

$$\begin{aligned}\mathbf{c} &= (\mathbf{m} \mid \mathbf{m} \cdot \mathbf{Q}) + \mathbf{e} \\ &= (\mathbf{e}_a \mid \mathbf{k}_e \mid (\mathbf{e}_a \mid \mathbf{k}_e) \cdot \mathbf{Q}) + (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c) \\ &= (\mathbf{0}_a \mid \mathbf{k}_e + \mathbf{e}_b \mid (\mathbf{e}_a \mid \mathbf{k}_e) \cdot \mathbf{Q} + \mathbf{e}_c) \\ &= (\mathbf{0}_a \mid \mathbf{c}_b \mid \mathbf{c}_c)\,,\end{aligned}$$

   where $\mathbf{c}_b = \mathbf{k}_e + \mathbf{e}_b$ and $\mathbf{c}_c = (\mathbf{e}_a \mid \mathbf{k}_e) \cdot \mathbf{Q} + \mathbf{e}_c$. Then remove the first $k - \ell$ coordinates (all zero) from $\mathbf{c}$ to obtain $\mathbf{c}^* = (\mathbf{c}_b \mid \mathbf{c}_c) \in \mathbb{F}_2^{n-k+\ell}$.
6. Output the pair $(\mathbf{k}_r, \mathbf{c}^*)$ where $\mathbf{k}_r = H_\ell(\mathbf{k}_e \mid \mathbf{e}) \in \mathbb{F}_2^{\ell}$.

Algorithm 1 Generate random vector $\mathbf{e}$ of length n and weight τ

```
function RandomVector($n, \tau$)
    $\mathbf{e} \leftarrow (0, 0, \ldots, 0, 1, 1, \ldots, 1)$ where $\mathrm{hw}(\mathbf{e}) = \tau$
  /* Shuffle the non-zero entries of $\mathbf{e}$ with Fisher-Yates algorithm */
    $i \leftarrow n - 1$
    while $i \geq n - \tau$ do
        $r \leftarrow_{\$} \{0, 1, \ldots, i\}$
        Swap $e_i$ with $e_r$
        $i \leftarrow i - 1$
    end while
    return $\mathbf{e}$
end function
```

Remark 1. *Algorithm 1 is not constant-time but it is possible to transform the algorithm to be constant-time using the approach suggested in [BCS13, BCLvV16]. We note that Algorithm 1 is only used for generating the ephemeral vector* $\mathbf{e}$.

3.3 Decapsulation

At a high level, the decapsulation of a NTS-KEM ciphertext $\mathbf{c}^* = (\mathbf{c}_b \mid \mathbf{c}_c)$ proceeds as follows:

1. Consider the vector $\mathbf{c} = (\mathbf{0}_a \mid \mathbf{c}_b \mid \mathbf{c}_c) \in \mathbb{F}_2^n$, and apply a decoding algorithm — using the secret key — to recover a permuted error pattern $\mathbf{e}'$.
2. Compute the error vector $\mathbf{e} = \pi_{\mathbf{p}}(\mathbf{e}')$.
3. Partition $\mathbf{e} = (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c)$, where $\mathbf{e}_a \in \mathbb{F}_2^{k-\ell}$, $\mathbf{e}_b \in \mathbb{F}_2^{\ell}$ and $\mathbf{e}_c \in \mathbb{F}_2^{n-k}$, and compute $\mathbf{k}_e = \mathbf{c}_b - \mathbf{e}_b$.
4. Verify that $H_\ell(\mathbf{e}) = \mathbf{k}_e$ and $\mathrm{hw}(\mathbf{e}) = \tau$. If yes, return $\mathbf{k}_r = H_\ell(\mathbf{k}_e \mid \mathbf{e}) \in \mathbb{F}_2^{\ell}$; otherwise return $\perp$.

3.3.1 Detailed description

In more detail, given a NTS-KEM ciphertext $\mathbf{c}^* \in \mathbb{F}_2^{n-k+\ell}$ and a private key $\mathrm{sk} = (\mathbf{a}^*, \mathbf{h}^*, \mathbf{p})$, where $\mathbf{a}^*, \mathbf{h}^* \in \mathbb{F}_{2^m}^{n-k+\ell}$ and $\mathbf{p} \in \mathbb{F}_{2^m}^{n}$, the process of decapsulation can be implemented as follows.

1. Decoding of $\mathbf{c} = (\mathbf{0}_a \mid \mathbf{c}_b \mid \mathbf{c}_c)$ to recover a permuted error pattern $\mathbf{e}'$.
   (a) Partition the private-key vectors $\mathbf{a}^*$ and $\mathbf{h}^*$ as

$$\begin{aligned}\mathbf{a}^* &= (\mathbf{a}_b \mid \mathbf{a}_c) = (a_{b,0}, a_{b,1}, \ldots, a_{b,\ell-1} \mid a_{c,0}, a_{c,1}, \ldots, a_{c,r-1})\\ \mathbf{h}^* &= (\mathbf{h}_b \mid \mathbf{h}_c) = (h_{b,0}, h_{b,1}, \ldots, h_{b,\ell-1} \mid h_{c,0}, h_{c,1}, \ldots, h_{c,r-1})\end{aligned}$$

   where $r = m \cdot \tau = n - k$.

   (b) Construct a truncated parity-check matrix $\mathbf{H}_m^* \in \mathbb{F}_{2^m}^{2\tau \times (\ell + m\cdot\tau)}$ from $\mathbf{a}^*$ and $\mathbf{h}^*$ as follows[4]:

$$\mathbf{H}_m^* = \left[\begin{array}{ccc|ccc} h_{b,0} & \cdots & h_{b,\ell-1} & h_{c,0} & \cdots & h_{c,r-1}\\ a_{b,0}h_{b,0} & \cdots & a_{b,\ell-1}h_{b,\ell-1} & a_{c,0}h_{c,0} & \cdots & a_{c,r-1}h_{c,r-1}\\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots\\ a_{b,0}^{\tau-1}h_{b,0} & \cdots & a_{b,\ell-1}^{\tau-1}h_{b,\ell-1} & a_{c,0}^{\tau-1}h_{c,0} & \cdots & a_{c,r-1}^{\tau-1}h_{c,r-1}\\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots\\ a_{b,0}^{2\tau-1}h_{b,0} & \cdots & a_{b,\ell-1}^{2\tau-1}h_{b,\ell-1} & a_{c,0}^{2\tau-1}h_{c,0} & \cdots & a_{c,r-1}^{2\tau-1}h_{c,r-1} \end{array}\right]. \qquad (5)$$

[4] Unlike the parity-check matrix $\mathbf{H}_m$ in equation (4), $\mathbf{H}_m^*$ is obtained by extending the number of rows to a total of 2τ rows and removing the large number of columns that correspond to section a of $\mathbf{c}$. This row extension allows us to use the Berlekamp-Massey algorithm [Ber68, Mas69] to obtain an error locator polynomial $\sigma(x)$ that can correct all τ errors [Ret75].

(c) Compute all 2τ syndromes of $\mathbf{c}^* = (\mathbf{c}_b \mid \mathbf{c}_c)$ as follows:

$$\begin{aligned}\mathbf{s} &= (\mathbf{c}_b \mid \mathbf{c}_c) \cdot (\mathbf{H}_m^*)^T \\ &= (s_0, s_1, \ldots, s_{2\tau-1})\end{aligned}$$

Due to the structure of $\mathbf{H}_m^*$, the syndromes $\mathbf{s}$ may be computed following Algorithm 2.

Algorithm 2 Syndrome Computation on ciphertext $\mathbf{c}^* = (\mathbf{c}_b \mid \mathbf{c}_c)$

```
function ComputeSyndrome($\mathbf{c}^*, \mathbf{a}^*, \mathbf{h}^*$)
Require: $\mathbf{c}^* \leftarrow (c_{b,0}, c_{b,1}, \ldots, c_{b,\ell-1} \mid c_{c,0}, c_{c,1}, \ldots, c_{c,r-1})$
Require: $\mathbf{a}^* \leftarrow (a_{b,0}, a_{b,1}, \ldots, a_{b,\ell-1} \mid a_{c,0}, a_{c,1}, \ldots, a_{c,r-1})$
Require: $\mathbf{h}^* \leftarrow (h_{b,0}, h_{b,1}, \ldots, h_{b,\ell-1} \mid h_{c,0}, h_{c,1}, \ldots, h_{c,r-1})$
    $s_0 \leftarrow \sum_{j=0}^{\ell-1}(c_{b,j} \cdot h_{b,j}) + \sum_{j=0}^{r-1}(c_{c,j} \cdot h_{c,j})$
    $i \leftarrow 1$
    while $i < 2\tau$ do
        $\mathbf{h}^* \leftarrow \mathbf{h}^* \cdot \mathbf{a}^*$        // Pointwise component multiplication modulo $\mathbb{F}_{2^m}$
        $s_i \leftarrow \sum_{j=0}^{\ell-1}(c_{b,j} \cdot h_{b,j}) + \sum_{j=0}^{r-1}(c_{c,j} \cdot h_{c,j})$
        $i \leftarrow i + 1$
    end while
    return $\mathbf{s} = (s_0, s_1, \ldots, s_{2\tau-1})_{\mathbb{F}_{2^m}}$
end function
```

(d) Compute the error locator polynomial $\sigma(x)$ and the first coordinate error indicator ξ from the syndrome vector $\mathbf{s}$ using the Berlekamp-Massey algorithm.[5] The original Berlekamp-Massey algorithm [Ber68, Mas69] requires the computation of the inverse of a field element at each iteration. As is evident from Appendix A.5, inversion is expensive but this can be avoided as shown by [Bur71, You91]. Our inversion-free implementation of Berlekamp-Massey algorithm is shown in Algorithm 3.

(e) Evaluate the polynomial $\sigma(x)$ on all elements of $\mathbb{F}_{2^m}$ defined by basis B, i.e. $\mathbf{\Lambda} = \{\sigma(B[0]), \sigma(B[1]), \ldots, \sigma(B[n-1])\}$. This multi-point evaluation may be efficiently computed by means of the additive Fast Fourier Transform over a finite field [GM10]; see Algorithm 4 in Appendix B.

(f) Given $\mathbf{\Lambda} = \{\sigma(B[0]), \sigma(B[1]), \ldots, \sigma(B[n-1])\}$, obtain the error vector $\mathbf{e}'$ where the non-zero positions of $\mathbf{e}'$ are determined as follows:

- Initialise $\mathbf{e}' = \mathbf{0} \in \mathbb{F}_2^n$.
- Set $e'_j = 1$ if $\sigma(B[j]) = 0$.

Furthermore, if ξ is 1, set $e'_0 = 1$.

2. Apply the permutation $\mathbf{p}$ to $\mathbf{e}'$ to obtain $\mathbf{e} = \pi_{\mathbf{p}}(\mathbf{e}')$. This is because the syndrome $\mathbf{s}$ is computed using $\mathbf{h}^*$ on $\mathbf{c}^*$, and thus we have $\mathbf{e}' = \pi_{\mathbf{p}^{-1}}(\mathbf{e})$.

[5] The roots of the polynomial $\sigma(x)$ tell us the locations of the τ bit errors and as Berlekamp-Massey cannot return $\sigma(x)$ with root zero, the error in the first coordinate is indicated by ξ in Algorithm 3. If the first coordinate is in error, $\xi = 1$; otherwise it is 0.

Algorithm 3 Berlekamp Massey Algorithm

```
function BerlekampMassey($\mathbf{s}$)
Require: $\mathbf{s} = (s_0, s_1, \ldots, s_{2\tau-1})$
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 $i \leftarrow 0$ to $2\tau - 1$ step 1 do
        $d \leftarrow \sum_{j=0}^{\min\{i,\tau\}} \sigma_j s_{i-j}$
        $\varphi(x) \leftarrow \delta\,\sigma(x) - d\,\beta(x)$
        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
        $\sigma(x) \leftarrow \varphi(x)$
    end for
    if Degree of $\sigma(x) < (\tau - \frac{R}{2})$ then
        $\xi \leftarrow 1$
    end if
    $\sigma^*(x) \leftarrow x^{\tau-\xi}\sigma(x^{-1})$          // $\sigma^*(x)$ is the reciprocal polynomial of $\sigma(x)$
    return $(\sigma^*(x), \xi)$
end function
```

3. Consider $\mathbf{e} = (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c)$, and compute $\mathbf{k}_e = \mathbf{c}_b - \mathbf{e}_b$.

4. Verify that $H_\ell(\mathbf{e}) = \mathbf{k}_e$ and $\mathrm{hw}(\mathbf{e}) = \tau$. If yes, return $\mathbf{k}_r = H_\ell(\mathbf{k}_e \mid \mathbf{e}) \in \mathbb{F}_2^\ell$; otherwise return $\perp$.

3.4 Correctness of NTS-KEM

It is straightforward to show correctness of the NTS-KEM scheme by drawing attention to the similarities with the McEliece scheme. If we denote by $\overline{\mathbf{H}}$ the parity-check matrix of the Goppa code $\overline{\mathcal{C}_\mathcal{G}}$ defined by $G(z)$ with elements in the order given in Section 3.1.1, then the parity-check matrix in row echelon form $\mathbf{H}$ for the permuted code $\mathcal{C}_\mathcal{G}$ used in the NTS-KEM key generation algorithm (Section 3.1) can be described as $\mathbf{H} = \mathbf{U} \cdot \overline{\mathbf{H}} \cdot \mathbf{P}$, where $\mathbf{P}$ is the random permutation matrix generated during key generation[6] and $\mathbf{U} \in \mathbb{F}_2^{n-k}$ is the

[6] As explained in Section 3.1, item 3d, we assume that the permutation ρ, which may have been required during the row echelon reduction of $\mathbf{H}$ for the identity matrix $\mathbf{I}_{n-k}$ to occupy the last $n-k$ columns of $\mathbf{H}$, has been applied to the matrices $\mathbf{P}$ and $\mathbf{H}_m$, and vectors $\mathbf{a}^*$ and $\mathbf{h}^*$.

transformation matrix representing the row echelon reduction.

The NTS-KEM public key $\mathbf{Q}$ can then be considered as a compact representation of a McEliece public key $\mathbf{G} = \overline{\mathbf{G}} \cdot \mathbf{P} = [\mathbf{I}_k \mid \mathbf{Q}]$ using the Goppa code $\overline{\mathcal{C}_\mathcal{G}}$, where $\overline{\mathbf{G}}$ is a generator matrix of $\overline{\mathcal{C}_\mathcal{G}}$ (and setting the invertible matrix $\mathbf{S}$ to the identity). Indeed we have

$$\begin{aligned}\overline{\mathbf{G}} \cdot \overline{\mathbf{H}}^T &= \overline{\mathbf{G}} \cdot (\mathbf{P} \cdot \mathbf{P}^T) \cdot \overline{\mathbf{H}}^T \cdot (\mathbf{U}^T \cdot (\mathbf{U}^T)^{-1}) \\ &= (\overline{\mathbf{G}} \cdot \mathbf{P}) \cdot (\mathbf{U} \cdot \overline{\mathbf{H}} \cdot \mathbf{P})^T \cdot (\mathbf{U}^T)^{-1} \\ &= \mathbf{G} \cdot \mathbf{H}^T \cdot (\mathbf{U}^T)^{-1} = 0,\end{aligned}$$

since $\mathbf{G} \cdot \mathbf{H}^T = 0$ by construction (in the NTS-KEM key generation algorithm).

In the NTS-KEM encapsulation algorithm described in Section 3.2, one first samples at random an error vector $\mathbf{e}$ of Hamming weight τ, which will carry the information for the generation of the encapsulated key $\mathbf{k}_r$, and constructs a message $\mathbf{m} = (\mathbf{e}_a \mid \mathbf{k}_e) \in \mathbb{F}_2^k$, where $\mathbf{k}_e = H_\ell(\mathbf{e})$. The NTS-KEM ciphertext is computed by performing a "McEliece encryption" of the message $\mathbf{m}$ with public key $\mathbf{G}$ and error $\mathbf{e}$, thus:

$$\begin{aligned}\mathbf{c} &= \mathbf{m} \cdot \mathbf{G} + \mathbf{e} \\ &= \mathbf{m} \cdot [\mathbf{I}_k \mid \mathbf{Q}] + \mathbf{e} \\ &= (\mathbf{m} \mid \mathbf{m} \cdot \mathbf{Q}) + \mathbf{e} \\ &= (\mathbf{e}_a \mid \mathbf{k}_e \mid (\mathbf{e}_a \mid \mathbf{k}_e) \cdot \mathbf{Q}) + (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c) \\ &= (\mathbf{0}_a \mid \mathbf{k}_e + \mathbf{e}_b \mid (\mathbf{e}_a \mid \mathbf{k}_e) \cdot \mathbf{Q} + \mathbf{e}_c) \\ &= (\mathbf{0}_a \mid \mathbf{c}_b \mid \mathbf{c}_c)\,.\end{aligned}$$

We note that setting the messages to be encrypted to be of the form $\mathbf{m} = (\mathbf{e}_a \mid \mathbf{k}_e)$ in the NTS-KEM scheme will always result on the the first $k - \ell$ bits of $\mathbf{c}$ being zero, and thus the transmitted ciphertext can be taken to have the compact form $\mathbf{c}^* = (\mathbf{c}_b \mid \mathbf{c}_c) \in \mathbb{F}_2^{n-k+\ell}$.

Finally, for the NTS-KEM decapsulation, we note that the private key $(\mathbf{a}^*, \mathbf{h}^*, \mathbf{p})$ contains the information for constructing the *permuted* and *truncated* parity-check matrix $\mathbf{H}_m^*$, which is used for syndrome decoding. Unlike the (permuted) parity-check matrix $\mathbf{H}_m = \overline{\mathbf{H}} \cdot \mathbf{P}$, the matrix $\mathbf{H}_m^*$ is obtained by extending the number of rows to a total of 2τ rows and removing a large number of columns that correspond to section a of $\mathbf{c}$. While the row extension allows us to directly use the Berlekamp-Massey algorithm [Ber68, Mas69] for syndrome decoding, the removal of the first $k - \ell$ columns can be done because the first $k - \ell$ bits of the ciphertext are always zero, and thus do not contribute to the computation of syndromes. As a result, for NTS-KEM ciphertexts, syndrome decoding using the matrix $\mathbf{H}_m^*$ as described in Section 3.3 is equivalent to syndrome decoding with the original parity-check matrix $\mathbf{H}_m$, and we have:

$$\begin{aligned}\mathbf{c} \cdot \mathbf{H}_m^T &= \mathbf{m} \cdot \mathbf{G} \cdot \mathbf{H}_m^T + \mathbf{e} \cdot \mathbf{H}_m^T \\ &= \mathbf{m} \cdot (\overline{\mathbf{G}} \cdot \mathbf{P}) \cdot (\overline{\mathbf{H}} \cdot \mathbf{P})^T + \mathbf{e} \cdot (\overline{\mathbf{H}} \cdot \mathbf{P})^T \\ &= \mathbf{m} \cdot \overline{\mathbf{G}} \cdot \mathbf{P} \cdot \mathbf{P}^T \cdot \overline{\mathbf{H}}^T + \mathbf{e} \cdot \mathbf{P}^T \cdot \overline{\mathbf{H}}^T \\ &= \mathbf{e}' \cdot \overline{\mathbf{H}}^T,\end{aligned}$$

algorithm version	security category	security target[8]	n	k	d	τ	$f(x)$	pk size[7] (in bytes)	sk size (in bytes)	ctext size (in bits)
NTS-KEM(12,64)	1	128-bit	4096	3328	129	64	$1+x^3+x^{12}$	319,488	9,216	1,024
NTS-KEM(13,80)	3	192-bit	8192	7152	161	80	$1+x+x^3+x^4+x^{13}$	929,760	17,524	1,296
NTS-KEM(13,136)	5	256-bit	8192	6424	273	136	$1+x+x^3+x^4+x^{13}$	1,419,704	19,890	2,024

Table 1: NTS-KEM recommended parameter sets

where $\mathbf{e}' = \mathbf{e} \cdot \mathbf{P}^T = \mathbf{e} \cdot \mathbf{P}^{-1} = \pi_{\mathbf{p}^{-1}}(\mathbf{e})$, since $\mathbf{P}^T = \mathbf{P}^{-1}$ for permutation matrices. That our syndrome decoding algorithm will recover the correct $\mathbf{e}'$ (and thus $\mathbf{e} = \pi_{\mathbf{p}}(\mathbf{e}')$) follows from the basic coding theory fact that there is a unique syndrome $\mathbf{s}$ for every vector $\mathbf{e}$ with $\text{hw}(\mathbf{e}) \leq \tau$ [MS77]. Thus the decoding procedure in the NTS-KEM decapsulation operation can be used to recover the error vector $\mathbf{e}$. This can in turn be verified against the hash $\mathbf{k}_e$ in the ciphertext, and if successful, both values can be used to recover the encapsulated key $\mathbf{k}_r = H_\ell(\mathbf{k}_e \mid \mathbf{e}) \in \mathbb{F}_2^\ell$.

This shows *correctness* of the NTS-KEM algorithm. We discuss *security* of NTS-KEM in Section 7.

4 Parameter sets

In this submission, we recommend three parameters sets for NTS-KEM, which are shown in Table 1. We specify the different parameters of the $[n, k, d]_2$ Goppa code used in the NTS-KEM scheme, including the minimum-weight irreducible polynomial $f(x)$ used to define the finite field $\mathbb{F}_{2^m}$ so as to facilitate fast multiplicative arithmetic in the field. We also provide the sizes of the public key, private key and ciphertext for each set of parameters. We refer to the different versions as NTS-KEM(m, τ), where $m = \log_2 n$ and $\tau = (d-1)/2$.

The recommended NTS-KEM parameters sets were selected to meet three of the security strength categories defined in NIST's call for proposals [NIS16]:

- *Category 1*: any attack that breaks the IND-CCA security of NTS-KEM must require computational resources comparable to or greater than those required for key search on a block cipher with a 128-bit key (e.g. AES-128).
- *Category 3*: any attack that breaks the IND-CCA security of NTS-KEM must require computational resources comparable to or greater than those required for key search on a block cipher with a 192-bit key (e.g. AES-192).
- *Category 5*: any attack that breaks the IND-CCA security of NTS-KEM must require computational resources comparable to or greater than those required for key search on a block cipher with a 256-bit key (e.g. AES-256).

[7] We assume that the public key values τ and ℓ are stored in two-byte strings each.

[8] Classical security target.

We claim that NTS-KEM(12,64), NTS-KEM(13,80) and NTS-KEM(13,136) provide at least 128-bit, 192-bit and 256-bit *classical* security, respectively, and at least 64-bit, 96-bit and 128-bit *post-quantum* security. These claims are based on the analysis of state-of-art algorithms for decoding random binary linear codes and the reasonable assumption that our scheme is not subject to dedicated quantum attacks, other than the generic speed-ups due to Grover's algorithm, quantum walks, etc. In the same spirit of our conservative code-based design, we have also decided to propose conservative security parameters, which are likely to offer a reasonable security margin within the security categories we aimed for. We discuss and justify our security claims in more detail in Section 7.

As in McEliece's original scheme, we have decided to use codes of length power of two, i.e. $n = 2^m$. The main reason is that for any given security level, choosing codes with a maximum length of $n = 2^m$ minimises the length of the ciphertext. This is because in our NTS-KEM implementation, it is the number of parity bits that defines the length of the ciphertext. Another point is that while some recent implementations suggest the potential use of non-power of two code lengths [BCS13], we believe that software and hardware implementations are cleaner and usually faster with $n = 2^m$. Moreover, setting parameters is also simpler: once the code length $n = 2^m$ has been chosen, different security levels may be achieved by selecting the only independent variable τ, the number of errors. In our choice of parameters, we also decided to select τ, m such that the number of parity bits $m \cdot \tau$ is an integral number of bytes, as it is already the case for the code length $n = 2^m$. This minimises the chances of software bugs and avoids messy bit/byte boundary formatting problems in software implementations.

Finally we note that while the three suggested parameter sets, and corresponding security levels and implementation profiles, should cater for most users requiring a secure key encapsulation mechanism, the NTS-KEM scheme is very versatile and offers a high degree of flexibility in the setting of parameters. The code length $n = 2^m$ and the number of errors τ may be chosen to meet users' needs, with the corresponding security level and implementation profile easily derived.

5 Design rationale

In this submission, we target long-term security, with a conservative design based on a simple and well-studied mathematical problem to provide a quantum-resistant, IND-CCA key encapsulation mechanism. Our choice is for a code-based cryptographic scheme, more specifically a combination of the McEliece and Niederreiter public-key encryption schemes, although they require large public keys. McEliece's construction is nearly 40 years old and has received considerable attention. Despite enormous cumulative efforts by the cryptographic community, it remains unbroken when instantiated with Goppa codes for suitable parameters. Our proposal NTS-KEM, a variant of McEliece and Niederreiter, is a fitting candidate to meet our design goals.

A major advantage of code-based schemes is that the security achieved is highly adjustable by appropriate choice of the code length n and the number of errors τ contained in the error vector. For many software and hardware implementations, different sets of parameters with their associated security levels may be accommodated straightforwardly without redesign. Moreover, by setting parameters conservatively, large security margins may also be built in, in case there are new improvements developed in the cryptanalysis of McEliece-like schemes. This is in fact the approach we have taken in this submission.

Another point we considered when designing NTS-KEM as a combination of the McEliece and Niederreiter schemes, is that the security level of our scheme may be increased by increasing the code length n, without however *substantially* increasing the ciphertext length. With standard McEliece if the code length is doubled then the ciphertext length is also doubled; on the other hand if the code length is doubled in standard Niederreiter, the ciphertext length only increases by the factor $\frac{\log_2(n)+1}{\log_2(n)}$. Similar to the latter, doubling the code length in NTS-KEM will result in an increase of ciphertext length from $n - k + \ell = m\tau + 256$ bits to $(m+1)\tau + 256$ bits. Of course doubling the code length substantially increases the size of the public keys of our schemes.

Although NTS-KEM public keys are large, the ciphertext length is short: 2024 bits at the highest, 256-bit security level. At the 192-bit security level the ciphertext length is 1296 bits, reducing to 1024 bits at the 128-bit security level.[9] To achieve these compact ciphertext sizes, we exploit the property of a binary Goppa code (or any binary linear code) that the syndrome of a truncated codeword is equal to the syndrome of the portion of the codeword that was truncated. It is this property that makes it possible for ciphertexts to be reduced in length from 8192 bits for McEliece to 2024 bits in NTS-KEM, at the 256-bit security level, with no consequent loss in security.

On the selection of parameters, we elected to make the code length $n = 2^m$ because this choice minimises the ciphertext length, for all security levels. Shortening the code by reducing n, for the same security level, necessitates increasing the number of errors τ in the error vector, requiring additional parity bits thereby increasing the ciphertext length. It should be noted that shortening the code can lead to a reduced public-key size at the expense of increased ciphertext length. We chose to make shortening the ciphertext length our top design priority. We also chose code parameters that provide clean, byte level boundaries between parity bits and information bits. This makes the implementation more straightforward and reduces the possibilities for software bugs, at the expense of a small increase in security margin that may not be required.

As for the random error vector $\mathbf{e}$, an n-bit binary vector containing τ bit errors used to carry information for the encapsulated key, we decided that τ should be a constant since there is negligible security advantage in making τ variable; and a variable τ would considerably complicate a constant-time decapsulation implementation. For similar reasons we rejected the notion of increasing τ beyond the error correcting capability of the underlying Goppa code. Bit flipping decoders can extend τ by 1 or 2 bits, thus increasing security, but at the cost of longer decapsulation times and a more difficult constant-time implementation. Similarly, we rejected the idea of using a list decoder to extend τ because this would mean that the probability of successful decryption by the private-key holder would no longer be 1,

[9] All classical security.

due to the possibility of the list decoder failing to decode.

We also elected to make the generation of the error vector to be entirely random with no internal structure or redundant formatting on the grounds that any such structure or formatting may invite an attack. This decision thereby precludes any Niederreiter-type, self-authenticating error vector schemes. It is obvious that using completely random error vectors, limited only by the random number generator, is the safest approach, and it is the one we have taken in our proposal.

To achieve IND-CCA security, we deploy a transform in the spirit of Dent's transform [Den03], which converts an OW secure public-key encryption into an IND-CCA secure KEM. Making careful use of the properties of the underlying OW secure scheme, we achieve a construction which comes equipped with a tight proof of security in the Random Oracle Model.

In our approach to implementation, we have aimed to provide fast execution times whilst minimising any leakage due to processing time variations in both encapsulation and decapsulation. We believe that with further development the implementation may be made constant time. Where there is a free choice we also elected to choose the simplest, most transparent algorithms in order to minimise the possibility of undetected software bugs. For example, in decapsulation we use a bit-sliced version of the well-understood Berlekamp-Massey algorithm combined with a BCH decoding approach commonly used for decoding standard Reed-Solomon codes. This is instead of using the somewhat complicated Patterson's method for decoding Goppa codes. In some cases alternative representations can result in faster execution. For example, we generate the parity-check matrix directly from the square of the Goppa polynomial $G^2(z)$ rather than the traditional approach of generating the parity check matrix from $G(z)$. This has the advantage of producing all of the component syndromes necessary for decoding without requiring the additional step of interpolation that is needed to produce the missing component syndromes when the traditional parity-check matrix is used.

Look-up tables are avoided for $\mathbb{F}_{2^m}$ arithmetic and the lowest Hamming weight irreducible polynomial is used to define $\mathbb{F}_{2^m}$. This ensures that polynomial-based $\mathbb{F}_{2^m}$ multiplication is fast and runs in constant-time. Short ciphertexts are used to advantage in decapsulation. At the 256-bit security level only 2024 columns of the parity check matrix need be used to calculate the decoding syndrome instead of the 8192 columns of the standard McEliece system. This means faster calculation of the syndromes required to determine the error vector, as well as reducing the size of the private key since only 2024 columns of the parity-check matrix need to be stored.

Code-based cryptography makes use of long codes, much longer than the codes traditionally used, with increased number of errors, to achieve the security targets. Consequently with conventional implementation approaches there are noticeable bottlenecks particularly in the determination of the roots of the error locator polynomial during decapsulation. The use of an additive FFT can however be very efficient, and this has also been adopted in other implementations of McEliece [BCS13, Cho17]. In this submission, we have used a bit-sliced additive FFT implementation to speed up the root finding stage during decapsulation and the validity check of a Goppa polynomial during key-generation.

In key generation, the major bottleneck, particularly with the long codes used, is in determining the $m\tau \times n$ binary reduced-row echelon parity-check matrix for the binary Goppa code. This is a (1768×8192)-bit matrix for NTS-KEM(13,136), meeting the 256-bit security level. A particularly efficient algorithm for Gauss-Jordan elimination [ABP11] has been used in this part of the implementation to produce satisfactory performance, as described in Section 6.

Finally, we declare that we have not knowingly introduced any back-door in the NTS-KEM with our proposed design and suggested parameters.

6 Performance analysis

The performance profile of NTS-KEM key-generation, encapsulation and decapsulation is obtained from the following machines:

- A server with 16 cores Intel® Xeon® E5-2667 v2 3.3GHz processors, 256GB of RAM, running Debian Linux 8.8. It has an Ivy Bridge-EP processor microarchitecture supporting SSE2/SSE4.1 instruction set extensions.
- A server with 28 cores Intel® Xeon® E5-2690 v4 2.6GHz processors, 256GB of RAM, running Debian Linux 8.9. It has a Broadwell processor microarchitecture supporting AVX2.0 instruction set extensions.
- A Macbook with Intel® Core™ m3-6Y30 1.1GHz processor, 8GB of RAM (early 2016 model), running OS X 10.11. It has a Skylake processor microarchitecture supporting AVX2.0 instruction set extensions.

We tested our code on `gcc` version 4.9.2, 5.4.0 and 6.3.0 on Linux, and `clang-800` on OS X platforms. Furthermore, the code is compiled with the following compiler flags:

```
-O3 -ansi -std=c99 -fomit-frame-pointer -fwrapv
```

and additionally `-msse2 -msse4.1` or `-mavx2` depending on the processor architecture in order to make use of the wide single instruction, multiple data (SIMD) registers. We collected the number of CPU cycles at various stages of key-generation, encapsulation and decapsulation from $5,000$ runs. These are reported in Tables 2, 3 and 4, respectively. Note that on the Macbook, the computations were carried out in a single-user mode in order to obtain a more accurate figure.

The key-generation process in NTS-KEM consumes the most CPU time. It can be broken down into three main steps, namely sampling of a random Goppa polynomial $G(z)$, generation of a random permutation vector $\mathbf{p}$ and construction of the matrix $\mathbf{Q}$. Table 2 shows the

CPU	NTS-KEM parameter	Random $G(z)$ (Step 1)	Random $\mathbf{p}$ (Step 2)	Matrix $\mathbf{Q}$ (Step 3)
E5-2667 v2 3.3GHz Ivy Bridge-EP	NTS-KEM(12, 64)	$261,717$	$1,649,365$	$39,835,291$
	NTS-KEM(13, 80)	$678,262$	$3,631,370$	$131,504,205$
	NTS-KEM(13, 136)	$1,027,569$	$3,661,217$	$245,250,759$
E5-2690 v4 2.6GHz Broadwell	NTS-KEM(12, 64)	$175,297$	$1,295,656$	$38,008,363$
	NTS-KEM(13, 80)	$486,009$	$2,955,066$	$120,320,437$
	NTS-KEM(13, 136)	$763,752$	$2,853,602$	$217,488,808$
Core m3-6Y30 1.1GHz Skylake	NTS-KEM(12, 64)	$81,862$	$917,274$	$17,691,871$
	NTS-KEM(13, 80)	$210,537$	$1,976,192$	$49,088,739$
	NTS-KEM(13, 136)	$347,217$	$1,977,219$	$106,176,811$

Table 2: The average number of CPU cycles consumed during key-generation for different parameter sets on various processor architectures. A reference to the corresponding step in the specification is also given.

	Encapsulation		
NTS-KEM parameter	E5-2667 v2 3.3GHz Ivy Bridge-EP	E5-2690 v4 2.6GHz Broadwell	Core m3-6Y30 1.1GHz Skylake
NTS-KEM(12, 64)	$172,463$	$130,675$	$51,563$
NTS-KEM(13, 80)	$429,301$	$368,946$	$178,334$
NTS-KEM(13, 136)	$574,406$	$478,323$	$265,676$

Table 3: The average number of CPU cycles consumed during encapsulation for different parameter sets on various processor architectures.

average number of CPU cycles on these three steps for the three parameter sets proposed in Section 4.

Empirical results have shown that on average, approximately 3 sets of $m\tau$ bits are required in order to obtain a valid Goppa polynomial $G(z)$. The cost of sampling a random $G(z)$ is dominated by GCD computation between $G(z)$ and its derivative, which is used to check whether or not $G(z)$ has repeated roots in any extension field. As shown in Table 2, the highest cost is on the generation of matrix $\mathbf{Q}$. This cost is largely attributed to transforming a matrix into reduced-row echelon form, which has average-case complexity of $\mathcal{O}(n^3/\log_2 n)$ using M4RI [AP10]. We optimise this step following the approach in [ABP11] that exploits data locality, i.e. the cost of reading data from RAM is often higher than the combined cost of doing arithmetic operations and reading data from CPU cache. Note that unlike McBits [BCS13, Cho17] where on average 3 Gaussian elimination attempts are required per key-generation, only one attempt is required in NTS-KEM because we can update $\mathbf{p}$, see Step 3d of the key generation procedure.

The encapsulation in NTS-KEM is simple and relatively cheap compared to the key-generation process. Table 3 shows the average CPU cycles for encapsulation, which includes the generation of a random vector of weight τ, vector multiplication with $\mathbf{Q}$ and running SHAKE256 twice. The vector multiplication step is carried out with $\tau \left\lceil \frac{n-k}{W} \right\rceil$ XOR operations where W is the width of an SIMD register.

CPU	NTS-KEM parameter	Syndrome computation (Steps 1a–1c)	Berlekamp-Massey algorithm (Step 1d)	Root finding (Step 1e)	Obtain $\mathbf{e}'$ Permute $\mathbf{e}'$ & recover $\mathbf{k}_e$ (Step 1f, 2–3)	Compute $\mathbf{k}_r$ & hash check (Step 4)
E5-2667 v2	NTS-KEM(12, 64)	511,562	106,321	25,056	27,979	18,169
3.3GHz	NTS-KEM(13, 80)	996,303	157,559	60,328	55,483	30,429
Ivy Bridge-EP	NTS-KEM(13, 136)	2,334,224	412,076	78,428	55,552	30,840
E5-2690 v4	NTS-KEM(12, 64)	227,330	61,743	14,396	23,317	13,984
2.6GHz	NTS-KEM(13, 80)	395,687	94,573	30,181	58,998	25,020
Broadwell	NTS-KEM(13, 136)	754,569	246,202	40,377	57,412	25,319
Core m3-6Y30	NTS-KEM(12, 64)	110,535	35,860	6,548	14,891	9,054
1.1GHz	NTS-KEM(13, 80)	213,418	56,569	17,610	28,539	16,043
Skylake	NTS-KEM(13, 136)	435,830	141,889	21,572	28,376	16,058

Table 4: The average number of CPU cycles consumed during decapsulation for different parameter sets on various processor architectures. A reference to the corresponding step in the specification is also given.

Table 4 shows the break-down of the average CPU cycles consumed at different stages in NTS-KEM decapsulation. For the syndrome computation, NTS-KEM operates on vectors with $(\ell + m\tau)$ elements only, instead of n as in a typical McEliece implementation. This gives the syndrome computation complexity of $m\tau^2$. To compute an error-locator polynomial, we use the Berlekamp-Massey algorithm, which has complexity τ^2. Our implementation of this algorithm aims to minimise any potential timing side-channel information by removing all if-else branches and ensuring that it always runs for 2τ iterations for a given set of syndrome components. We use the additive-FFT, which requires $2n \log_2 n$ finite-field multiplications and $\frac{n}{2}(\log_2 n)^2$ finite-field additions [GM10], to perform root finding. The step to obtain $\mathbf{e}'$ is low cost compared to permuting $\mathbf{e}'$ to produce $\mathbf{e}$. The cost in the last decapsulation step is attributed to SHAKE256 computations.

A significant gain in performance has been obtained by bit-slicing the finite-field arithmetic operations. The field operations described in Appendix A are rather inefficient. Bit-slicing allows the same operation to be carried out simultaneously on a block of W items with relatively few bitwise logical operations on W-bit SIMD registers. For example, for $m = 12$ in NTS-KEM(12, 64), a naive field multiplication over $\mathbb{F}_{2^m}$ requires 30 bitwise operations.[10] Following the work on circuit minimization [BDP+], it is possible to build a circuit to multiply two polynomials of degree less than m with 126 XOR and 81 AND gates respectively. A modulo reduction circuit requires 22 XOR gates only. Therefore, on a processor with 256-bit

[10] 12 AND and 11 XOR operations for schoolbook multiplication, and 3 AND and 4 XOR operations for modulo reduction.

SIMD registers (AVX2.0 support), bit-slicing allows us to perform field multiplication on 256 elements in parallel with just 229 elementary operations. In contrast using the equivalent naive approach would require $7,680$ operations.

Our implementations of syndrome computation, the Berlekamp-Massey algorithm, and additive FFT all make use of bit-slicing. In particular, we follow the work of [Cho17] on bit-sliced additive FFT implementation. Root finding is usually the most time-consuming step in decapsulation [BS08], however as is evident from Table 4, using the bit-sliced additive FFT for root finding turns this step into one of the most efficient steps. The trade-off of bit-sliced FFT is the cost of storage where we need to store the *scaling factors* of the Taylor expansion of a polynomial ($\delta_{i,j}$ in Step 16 of Algorithm 4) and the *twiddle factors* of the FFT butterflies ($\Gamma_j[i]$ in Step 27 of Algorithm 4). The storage requirement of the *scaling factors* is small, under 500 bytes for NTS-KEM(12, 64), 1.2 kilobytes for NTS-KEM(13, 80) and just under 3 kilobytes for NTS-KEM(13, 136). On the other hand, the *twiddle factors* require a slightly more modest storage space, around 6 kilobytes for NTS-KEM(12, 64) and around 14 kilobytes for the other two NTS-KEM versions. Table 5 gives the high-level overview of the memory requirements of the current implementation of NTS-KEM key-generation, encapsulation and decapsulation.

We believe that the performance of NTS-KEM can be further improved. As shown in Table 4, the most CPU consuming step in decapsulation is syndrome computation. While bit-slicing has been used in our syndrome computation, we believe that a considerable amount of performance gain will be obtained by using a bit-sliced implementation of a transposed additive FFT as demonstrated in [BCS13, Cho17]. Moreover, following the results in error-correcting codes, it may be possible to simplify the Berlekamp-Massey algorithm to run in τ iterations, instead of 2τ [Bla83, LC04].

Survey of hardware implementation. A number of works have addressed the subject of hardware implementation. One of the first published implementations of the McEliece PKE scheme on an 8-bit microcontroller is MicroMcEliece [EGHP09]. It uses a binary Goppa code with parameters $m = 11$ and $t = 27$, with a claimed security of 80-bits [EGHP09]. Implemented on AVR ATxMega192 microcontroller clocked at 32MHz, it achieves encryption and decryption throughput of $3,889$ and $2,835$ bits per second respectively for the McEliece scheme; whereas for the Niederreiter scheme, it achieves encryption and decryption throughput of $119,890$ and $1,062$ bits per second respectively. The same Goppa code is also implemented in a Xilinx Virtex-6 FPGA to achieve 1.5 million encryptions per second and $17,000$ decryptions per second [HG12, Hey13]. Strenzke [Str10a, Str13a] demonstrated a prototype of the McEliece PKE scheme on a 16-bit smartcard. The author uses two binary Goppa codes, the first one has parameters $m = 10$, $t = 40$ and it takes around 1.2 seconds to encrypt and just under 1 second to decrypt; the second one has parameters $m = 11$, $t = 50$ and both encryption and decryption take less than 2 seconds each.

As stated earlier, the key-generation is the most time consuming step in McEliece-like schemes. Due to this constraint, this step may be prohibitive to implement on a low-end hardware platform. Furthermore most hardware implementations consider the lower end of the

		NTS-KEM(12, 64)	NTS-KEM(13, 80)	NTS-KEM(13, 136)
Key-Gen	$G(z)$	96	208	312
	FFT	13,152	28,288	30,368
	$\mathbf{p}$	8,192	16,384	16,384
	$\mathbf{a}$	6,144	13,312	13,312
	$\mathbf{H}$	393,216	1,064,960	1,810,432
	Public-key	319,488	929,760	1,419,704
	Private-key	9,216	17,524	19,890
Encap	Public-key	319,488	929,760	1,419,704
	$\mathbf{e}$	512	1,024	1,024
	$\mathbf{k}_e$	32	32	32
	Key $\mathbf{k}_r$	32	32	32
	Ciphertext	128	162	253
Decap	Private-key	9,216	17,524	19,890
	Ciphertext	128	162	253
	$\mathbf{s}$	192	416	624
	$\sigma(x)$	192	208	416
	FFT	13,152	28,288	30,368
	$\mathbf{e}$	512	1,024	1,024
	$\mathbf{k}_e$	32	32	32
	Key $\mathbf{k}_r$	32	32	32

Table 5: High-level overview of the memory requirements (in bytes) in NTS-KEM key-generation, encapsulation and decapsulation operations.

security target. For post-quantum security at the 128-bit level, Wang et al [WSN17] present a tunable FPGA implementation of key-generation which takes under 900,000 clock cycles on a Stratix V FPGA. The authors use a binary Goppa code recommended by the PQCRYPTO project [PQC] and claim 128-bit post-quantum security, with parameters $m = 13$, $t = 119$ and $n = 6960$. It is interesting to note that the additive FFT [GM10] is also implemented on the FPGA in [WSN17] and it requires 1,000 clock cycles to evaluate a Goppa polynomial of degree 119. Furthermore, they also use the Fisher-Yates shuffle in generating random permutations.

7 NTS-KEM security

In this section, we justify our parameter selection. In Section 7.1, we present Theorem 1 which states that an attack defeating the IND-CCA security of NTS-KEM can be used to break the one-wayness of a McEliece PKE scheme with the same security parameters and similar adversarial advantage. The proof of the theorem is given in Appendix E. In Section 7.2, we provide quantitative estimates for breaking the one-wayness of[11] the McEliece PKE scheme, based on the complexity of the best-known decoding algorithms. We provide justification in Section 7.3 for our assumption that NTS-KEM is not subject of dedicated quantum attacks, other than the speed-ups due to Grover's algorithm and other generic quantum algorithms. Section 7.4 summarises the security of NTS-KEM against other forms of attack, including side-channel attacks and reaction attacks. Finally, Section 7.5 summarises our security claims for the three different parameter sets that we propose for NTS-KEM.

7.1 IND-CCA security of NTS-KEM

Theorem 1 shows that the NTS-KEM scheme is secure against chosen-ciphertext attacks, under the reasonable assumption that inverting the McEliece PKE scheme is hard. In more detail, it proves that an adversary $\mathcal{A}$ capable of winning a IND-CCA game for NTS-KEM can be used to construct an adversary $\mathcal{B}$ that defeats the OW security of the McEliece PKE scheme with similar advantage.

Theorem 1. *If there exists a (t, ε)-adversary $\mathcal{A}$ winning the* IND-CCA *game for NTS-KEM, then there exists a $\left(2\,t, \varepsilon - \frac{q_D}{2^\ell}\right)$-adversary $\mathcal{B}$ against the* OW *security of the McEliece PKE scheme with same code parameters:*

- *in the Random Oracle Model; and,*
- *when the decapsulation algorithm succeeds with probability 1 for all public keys $(\mathbf{Q}, \tau, \ell)$ and all well-formed ciphertexts;*

with q_D being the number of queries made by $\mathcal{A}$ to its decapsulation oracle.

The full proof of the theorem can be found in Appendix E. We note that while our proof is tight, we use it mainly as *design validation*. In particular, when selecting parameters we do not take the additive term of $q_D/2^\ell$ into account.[12] That is, we use the cost of inverting the McEliece PKE scheme directly to select parameters.

7.2 McEliece OW security: decoding complexity

The best, known attack against the one-wayness of the McEliece PKE scheme is to attempt to determine the error vector from the ciphertext and the public key. The most efficient class of this type of attack is *information-set decoding* (ISD), which was anticipated and described by McEliece in his original paper [McE78]. The ISD technique was originally proposed by

[11] In what follows, we write "inverting" as short-hand for "breaking the one-wayness of".

[12] This omission does not substantially alter the promised security for two out of three of our proposed parameter sets as we have $\ell = 256$ and, per NIST's call [NIS16], $q_D \leq 2^{64}$.

Prange [Pra62] in the 1960s and was aimed at error correction of random binary codes when used in digital communication systems. It has been shown that decoding a random binary code is NP-hard [BMvT78] and, indeed, Prange's original decoder and all subsequent ISD algorithms have exponential running times. They remain, however, the most efficient attack currently known against the McEliece scheme, and more generally against Goppa code based cryptographic schemes, under the assumption that the structure of the original code used is hidden.

7.2.1 Information-set decoding

An OW adversary for the McEliece PKE scheme can treat the code as a random binary code and attempt to use Prange's decoder or subsequently improved ISD algorithms to obtain the plaintext message. In its basic form, an information-set decoding algorithm on any binary code with parameters $[n, k, 2\tau + 1]_2$ will randomly select k columns of the generator matrix, and carry out Gauss-Jordan elimination of the rows. There is a non-zero probability that a reduced echelon generator matrix will be obtained which may be used to generate an error-free codeword from k coordinates of the McEliece ciphertext.

First of all, the $k \times k$ sub-matrix resulting from the k selected columns needs to be full rank. The probability of this depends on the particular code, but for the long Goppa codes used in the McEliece scheme, this probability turns out to be the same as the probability of a randomly chosen $k \times k$ binary matrix being full rank, which has an asymptotic value of 0.2887.

Given a McEliece ciphertext containing τ errors, an attacker then selects k bits randomly and constructs the corresponding permuted, reduced echelon generator matrix with the chance of success of 0.2887. The attacker uses this matrix to generate a codeword from the corresponding k bits of the target ciphertext and finds the Hamming distance between this codeword and the ciphertext. If the Hamming distance is exactly τ, then it can recover the plaintext message. If not, it can start again with a different selection of k bits.

For an information-set attack to work, the selected k bits of the ciphertext need to be free from error. The probability of this event is

$$\prod_{i=0}^{k-1} \frac{n-\tau-i}{n-i} = \frac{(n-\tau)!(n-k)!}{(n-\tau-k)!n!}$$

Thus, in its basic form each iteration of the ISD algorithm has a probability of

$$0.2887 \cdot \left(\frac{(n-\tau)!(n-k)!}{(n-\tau-k)!n!} \right)$$

to successfully invert the McEliece PKE scheme.

We note however that if the $k \times k$ sub-matrix is not full rank it may not be necessary to start again. By selecting $k + \epsilon$ columns rather than k, a simple back tracking algorithm can be used to find a set of k full rank columns.

Then, assuming ϵ additional columns have been selected so that the selected matrix has rank k, in the basic ISD algorithm the expected number of selections of k bits from the ciphertext required for recovering the plaintext message is roughly given by

$$\frac{(n-\tau-k)!n!}{(n-\tau)!(n-k)!}.$$

One can already use this initial crude analysis of complexity of basic ISD to derive *minimum* parameter values for the McEliece PKE scheme for comparison between the computational resources required for inverting the scheme and for exhaustive key search on a λ-bit block cipher. Assuming a somewhat simplistic computational equivalence, the parameters need to satisfy

$$N_{(m,\tau)} = \frac{(m\tau-\tau)!2^m!}{(2^m-\tau)!(m\tau)!} = \frac{(n-\tau-k)!n!}{(n-\tau)!(n-k)!} \geq 2^{\lambda}, \tag{6}$$

where the binary Goppa code used is of full length $n = 2^m$, with the number of parity bits $n-k = m\tau$. This basic ISD analysis allows one to derive minimum values for the pair (m, τ) to satisfy this inequality. In particular we have:

- $m \geq 12$ and $\tau \geq 42 \Longrightarrow N_{(m,\tau)} \geq 2^{128}$.
- $m \geq 13$ and $\tau \geq 53 \Longrightarrow N_{(m,\tau)} \geq 2^{192}$.
- $m \geq 13$ and $\tau \geq 90 \Longrightarrow N_{(m,\tau)} \geq 2^{256}$.

While an analysis of basic ISD can indicate minimum parameter values for code-based cryptography, the literature documents a succession of developments and refinements to information-set decoding, particularly when applied to the task of inverting the McEliece PKE scheme. Canteaut et al. [CS98] formulated an attack based on extending the code by augmenting the generator matrix with the target ciphertext. For this new code, there will be only one codeword with Hamming weight τ, which is the error vector $\mathbf{e}$. With this approach any algorithm that finds low-weight codewords of a code may be used to attack the McEliece PKE scheme.

Most modern ISD attacks are based on collisions between the calculated syndrome of the target ciphertext and syndromes calculated from selected columns of the parity-check matrix. Equivalently stated, the target ciphertext has a syndrome $\mathbf{s}$ and there are τ columns of the parity-check matrix whose modulo 2 sum is equal to $\mathbf{s}$. There are many different variants of this type of attack with different strategies of partitioning sets of columns of the parity-check matrix and looking for collisions of sub-sum syndrome evaluations.

Starting with Stern [Ste88] there have been a large number of improvements in the work factors of this and similar approaches [LB88, BLP08, MMT11]. Good surveys of the different improvements of the syndrome based approach are provided in [FS09, BLP11].

(m,τ)	time complexity
(12,64)	$2^{158.4}$
(13,80)	$2^{239.9}$
(13,136)	$2^{305.1}$

Table 6: Proposed code parameters and ISD attack time complexities based on [BLP11]

7.2.2 Quantitative complexity estimates for inverting the McEliece PKE scheme

Our selection of parameters for NTS-KEM and corresponding security claims follow complexity estimates for inverting the McEliece PKE scheme using recent improvements in information-set decoding algorithms. In particular, we make use of scripts and results from [BLP08, FS09, MMT11] to assess the security levels of the proposed code parameters used in the NTS-KEM(12,64), NTS-KEM(13,80) and NTS-KEM(13,136) versions of NTS-KEM. Specifically, we used the scripts by Peters [BLP11] to produce the estimates on ISD attack time complexity for the three proposed sets of parameters for NTS-KEM shown in Table 6.

We also considered more recent results by Both and May [BM17a], which use the BJMM algorithm [BJMM12] with nearest neighbour search to derive time complexities for full and bounded distance decoding. Their work indicates work factors of 2^{128}, 2^{198} resp. 2^{256} operations [BM17b]. As noted in [BM17a], the analysis there neglects all polynomial factors, so these work factor estimates do not necessarily give upper bounds on the bit security level of the corresponding McEliece PKE scheme instantiations. We also note that the algorithm in [BM17a] has exponential space complexity. Thus, the area-time complexity of known algorithms for inverting McEliece is larger than indicated above, even when ignoring polynomial factors.

Yet, in the spirit of our conservative design, we chose to err on the side of caution and selected parameters which lead to an estimated cost of at least 2^{λ} for the three targeted classical security levels $\lambda \in \{128, 192, 256\}$ for an analysis based on [BM17a]. We stress that this is optimistic for the attacker.

7.3 Quantum attacks

There are currently no known *dedicated* quantum algorithms for attacking McEliece, and thus the NTS-KEM scheme. To the best of our knowledge, the best approach exploiting quantum computers to attack code-based schemes are the applications of generic quantum techniques such as Grover's algorithm and random walks to speed up information-set decoding algorithms.

Bernstein in [Ber10] showed that Grover's algorithm could be used to speed up the search step in Prange's original ISD algorithm. He showed that the basic search for an information set could be done on a quantum computer using Grover's algorithm in about $\sqrt{\binom{n}{k}/0.2887\binom{n-\tau}{k}} \approx c^{(1/2)n/\log n}$ iterations, where $c = (1 - \frac{k}{n})^{-(1-\frac{k}{n})}$, with each iteration requiring $O(n^3)$ qubit operations. Bernstein's algorithm thus represents a quadratic speed-up on Prange's *basic* ISD algorithm.

More recently, Kachigar and Tillich [KT17] considered how to speed up some of the more advanced information-set decoding algorithms on quantum computers. This was done by proposing a method to solve the generalised subset-sum problem with quantum walks and Grover search. They were then able to obtain a (below quadratic) speed-up, however on more sophisticated ISD algorithms, e.g. for the algorithm of May et al. [MMT11]. Their quantum information-set decoding algorithm represents an improvement (numerically) compared to Bernstein's, yet does not go beyond the square-root improvement offered by generic quantum algorithms.

Thus we can conservatively assume that quantum attacks will *at best* offer a square-root reduction in the classical security level offered by NTS-KEM.[13] We have not estimated the exact circuit depth and size required, but the research on quantum information-set decoding algorithms outlined above indicates that the quantum resources required to carry out such a quantum attack against NTS-KEM(12,64), NTS-KEM(13,80) and NTS-KEM(13,136) would be comparable to or greater than those required for a (quantum) key search on the AES block cipher with 128-bit, 192-bit and 256-bit keys, respectively.

We stress that the above analysis assumes not only the existence of an algorithmic square-root speed-up but also the existence of perfect quantum computers that can run indefinitely. In contrast, in [NIS16] NIST encourages designers to assume a much smaller depth achievable on a quantum computer, further limiting potential Grover-style speed-ups.

7.4 Security against other known attacks

In addition to ISD-based message recovery attacks discussed in Section 7.2.1, our security analysis may also consider attacks to recover the private key of code-based schemes. These so-called *structural attacks* attempt to recover the original code (i.e. to recognise the code structure) from the public key. This in turn would allow the attacker to construct an algebraic decoder, and break any ciphertext for the particular instantiation of the scheme.

Overbeck and Sendrier discuss in [OS09, Chapter 4.3] how one may attempt to recognise the code structure for schemes based on different classes of algebraic codes. For Goppa code based schemes, a structural attack means recovering either the Goppa polynomial $G(z)$ or the permutation $\mathbf{P}$; knowledge of either would allow recovery of the full private key from the public key in polynomial time. Faugère et al. discuss structural attacks on *high rate* Goppa codes, with $\frac{k}{n} > 0.95$, in [FGUPT11], and Loidreau and Sendrier showed that binary Goppa codes derived from *binary* Goppa polynomials [LS01] are vulnerable to structural attacks.

[13] We also note that best quantum speed-ups in [KT17] appear to occur for codes with rate around $0.4 - 0.5$, while the codes used in NTS-KEM have rate $0.78 - 0.87$.

However, for standard Goppa code based schemes, having a systematic generator matrix of the permuted Goppa code is considered sufficient to hide the original code structure, in which case the best known structural attacks have exponential time complexity. This is the case for NTS-KEM, where the public key can be seen as the systematic generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{Q}]$ over $\mathbb{F}_2$ of the permuted Goppa code $\mathcal{C}_\mathcal{G}$. Thus, for the NTS-KEM scheme and the suggested parameters in this proposal, such structural attacks are universally accepted as being infeasible, in fact of higher complexity than the decoding attacks used to derive our suggested parameter sets.

Ciphertexts in textbook McEliece are highly malleable. This allows so-called *reaction attacks*, in which an attacker manipulates a target ciphertext (e.g. performing bit flips on it), then observes the receiver's reaction (whether decoding fails or not), and uses this information to recover the secret message. Reaction attacks may also be applied against code-based schemes that require the use of a probabilistic decoding procedure for decryption, in some cases allowing for recovery of the private key [GJS16]. NTS-KEM employs a deterministic decoding algorithm during decapsulation, and this decoding algorithm always produces some output (with that output being correct whenever a valid ciphertext is presented for decryption). Moreover, NTS-KEM is designed to be IND-CCA secure, such that any ciphertext manipulations are detected after the decoding step is complete. This allows us to assert that NTS-KEM is not vulnerable to reaction attacks.

An overview of *side-channel attacks* against McEliece is given in [RZ14, Section 7]. Of particular applicability to code-based schemes are *timing attacks*, in which an attacker may observe the time required for key generation, encapsulation or decapsulation to recover secret information; see for example [STM+08, SSMS09, Str10b, AHPT12, Str13b]. As a variant of the McEliece scheme, NTS-KEM would also be vulnerable to timing attacks if implemented without care. While not all algorithms defined and recommended in Section 3 are constant-time – at times our choice for sub-routines were done for the sake of simplicity of presentation – implementations of NTS-KEM can be made constant-time by applying, for example, the approach suggested in [BCS13, BCLvV16]. *Power attacks* may also be attempted against software and hardware implementations of NTS-KEM [HMP10]. The implementation may however be protected against simple power analysis using standard techniques, see for example [EGHP09, CEvMS16]. Differential power analysis has also been applied against a number of implementations of McEliece, for example see [CEvMS15] for QC-MDPC-based scheme and [PRD+16] for Goppa-based scheme. The differential power cryptanalysis of Chen et al. [CEvMS15] shows that it is possible to recover the complete private key after a few observed decryptions. On the other hand, Petrvalsky et al. [PRD+16] attack the software implementation of a secure bit permutation proposed by Strenzke et al. [STM+08] and demonstrate that part of the private-key and permutation matrix can be recovered. Clearly care would be needed in the implementation of NTS-KEM in environments where DPA is a concern.

Code-based schemes may also be at risk from potential *misuse attacks* by one of the communicating parties. For NTS-KEM, the cases we foresee are the following:

1. in encapsulation, the sending party may generate error vectors either with weight lower

than τ, or not uniformly at random among the space of weight-τ vectors of length n;

2. in key generation, the receiving party may deliberately generate *weak* keys, for example reducing the coefficient space of the Goppa polynomial by restricting the coefficients to be binary values (a case for which there is an attack [LS01]), or by restrictions in the construction of the public key via binary mappings of field elements to an echelon reduced-row parity check matrix by means of Gauss-Jordan elimination.

In case 1, if $\text{hw}(\mathbf{e}) \ll \tau$ an attacker may efficiently decode the ciphertext with knowledge of the public key only, by using a bit flipping decoder or a general purpose decoder for random binary codes. However our algorithm checks the weight of the error vector on decapsulation, returning $\perp$ if $\text{hw}(\mathbf{e}) \neq \tau$. On the other hand, if error vectors are not chosen uniformly at random, for example if the error positions are restricted, this information may be used for faster recovery of the error vector and thus of the encapsulated key. The NTS-KEM decapsulation algorithm will not recognise such cases; if this is considered a threat, analysis may be carried out to verify that error vectors are generated uniformly at random. In case 2, an attacker may efficiently decode the ciphertext with knowledge of the public key if an instantiation of NTS-KEM generates keys in this manner. Again, if this form of attack is considered a threat, analysis of key generation may be carried out. We note however that all misuse cases discussed above require non-compliance with the specification of NTS-KEM given in Section 3 and/or use of algorithms against our recommendations given there.

7.5 Security claims

Our proof relating the IND-CCA security of NTS-KEM to the hardness of inverting the McEliece PKE scheme, combined with complexity estimates for state-of-art information-set decoding algorithms, and the absence of dedicated quantum attacks against our scheme, underpin our quantitative security claims for the three versions of NTS-KEM.

1. **NTS-KEM(12,64)**: scheme version based on $[2^{12}, 3328, 129]_2$ Goppa codes capable of correcting up to $\tau = 64$ errors. We claim that any attack that can break the IND-CCA security of NTS-KEM(12,64) will require computational resources of the order of at least 2^{128} operations on a classical computer, and at least 2^{64} operations on a quantum computer. Those computational resources are comparable to or greater than those required for key search on the AES-128 block cipher. Our claim places NTS-KEM(12,64) in the **Security Strength Category 1**, as defined in NIST's call for proposals [NIS16].

2. **NTS-KEM(13,80)**: scheme version based on $[2^{13}, 7152, 161]_2$ Goppa codes capable of correcting up to $\tau = 80$ errors. We claim that any attack that can break the IND-CCA security of NTS-KEM(13,80) will require computational resources of the order of at least 2^{192} operations on a classical computer, and at least 2^{96} operations on a quantum computer. Those computational resources are comparable to or greater than those required for key search on the AES-192 block cipher. Our claim places NTS-KEM(13,80) in the **Security Strength Category 3**.

3. **NTS-KEM(13,136)**: scheme version based on $[2^{13}, 6424, 273]_2$ Goppa codes capable of correcting up to $\tau = 136$ errors. We claim that any attack that can break the IND-CCA security of NTS-KEM(13,136) will require computational resources of the order of at least 2^{256} operations on a classical computer, and at least 2^{128} operations on a quantum computer. Those computational resources are comparable to or greater than those required for key search on the AES-256 block cipher. Our claim places NTS-KEM(13,136) in the **Security Strength Category 5**.

8 Advantages and limitations

NTS-KEM is a code-based key encapsulation mechanism, with a conservative design aiming for long-term security against classical and quantum attacks. We discuss in more detail below the main features of our proposal.

- A main advantage of our proposal is its **strong security guarantees**. NTS-KEM is a conservative proposal, a variant of the McEliece and Niederreiter schemes. These are schemes that have received considerable attention from the cryptographic community for nearly four decades. The NTS-KEM construction itself is a key encapsulation mechanism which offers resistance against chosen ciphertext attacks. This is shown via a proof of security which demonstrates a tight relationship between the IND-CCA security of NTS-KEM and the problem of inverting the McEliece PKE scheme.

- The security of NTS-KEM is therefore based on a **simple and well-understood mathematical problem**. The main approach to tackle this problem, namely information-set decoding algorithms, has been extensively studied and has good complexity estimates.

- These estimates were used to set **conservative parameter sets**, which are likely to offer a reasonable security margin within the security categories we aimed for.

- Moreover, the **absence of dedicated quantum attacks** indicates the best-case post-quantum scenario, with *at best* a quadratic speed-up on the classical ISD algorithms for cryptanalysing NTS-KEM on quantum computers. These features lead us to conclude that NTS-KEM offers **long-term post-quantum security**.

- Although we proposed conservative parameters in this submission, the scheme offers a **high degree of flexibility** in the setting of parameters. The two NTS-KEM parameters (the code length and the weight of the error vector) can be adjusted in case of future progress in ISD algorithms, or, conversely, if some of the current estimates are proven to be too optimistic. More generally, it is straightforward to derive from the parameter choices the security level estimates and associated costs in performance and size of keys and ciphertexts. This makes **easy to consider potential trade-offs** between performance and security. Finally, parameters may be set deliberately low in reduced versions of the algorithm, to test any new proposed cryptanalytic technique in practice.

- One further advantage of NTS-KEM is that it provides good **long-term keys**. The most efficient attacks target the keys encapsulated in individual ciphertexts instead. Other features of the scheme, for example the use of a deterministic decoding algorithm during decapsulation, mean that brute force is essentially the only known practical means of attack to determine the private key from the public key, and thus private-public key pairs may be deployed for long periods of time.

- NTS-KEM has **compact ciphertexts**, around 2,000 bits at the highest security level. This makes the scheme particularly suitable for low bandwidth applications with long-term keys.

- NTS-KEM has also **efficient operations**, particularly encapsulation, leading to reasonably fast software implementations. Moreover, the simplicity of the operations and subroutines allow for the straightforward deployment of protection measures against side-channel attacks by, for example implementing constant-time versions of the scheme's operations.

The notable disadvantage of our design is the size of the public key. At the highest security level proposed, the NTS-KEM public key is approximately 1.39MB in size (312KB for the 128-bit security version). Large public keys are a common feature of all Goppa code based schemes. There are however applications to which large public keys are not of major concern. In these applications NTS-KEM long-term security, compact ciphertexts and efficient public and private key operations may be considered as more relevant and attractive features.

Finally, we note that NTS-KEM does not currently come equipped with a proof of security in the Quantum Random Oracle Model (QROM), but only in the classical Random Oracle Model. However, in recent weeks, several new proof techniques have become available for proving security in the QROM. Thus, we are optimistic that we will be able to prove NTS-KEM (or a mild tweak of it) secure also in the QROM in future work.

References

[ABP11] Martin R. Albrecht, Gregory V. Bard, and Clément Pernet. Efficient Dense Gaussian Elimination over the Finite Field with Two Elements. *Computing Research Repository*, abs/1111.6549, 2011.

[AHPT12] Roberto Avanzi, Simon Hoerder, Dan Page, and Michael Tunstall. Erratum to: Side-channel attacks on the McEliece and Niederreiter public-key cryptosystems. *J. Cryptographic Engineering*, 2(1):75, 2012.

[AP10] Martin R. Albrecht and Clément Pernet. Efficient Decomposition of Dense Matrices over GF(2). *Computing Research Repository*, abs/1006.1744, 2010.

[Arn11] Jörg Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. Springer-Verlag, 2011.

[BBHT17] Axel Bacher, Olivier Bodini, Hsien-Kuei Hwang, and Tsung-Hsi Tsai. Generating Random Permutations by Coin Tossing: Classical Algorithms, New Analysis, and Modern Implementation. *ACM Trans. Algorithms*, 13(2):24:1–24:43, February 2017.

[BCLvV16] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime: reducing attack surface at low cost. Cryptology ePrint Archive, Report 2016/461, 2016. https://eprint.iacr.org/2016/461.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 250–272, Santa Barbara, CA, USA, August 20–23, 2013. Springer, Heidelberg, Germany.

[BDP$^+$] Joan Boyar, Morris Dworkin, Rene Peralta, Meltem Turan, Cagdas Calik, and Luis Brandao. Circuit minimization work. http://www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html. Past collaborators include: Michael Bartock, Ramon Collazo, Magnus Find, Michael Fischer, Christopher Wood, Andrea Visconti, Chiara Schiavo, Holman Gao, Bruce Strackbein and Larry Bassham. A web page including explicit formulas for multiplication over the binary field by the Circuit Minimization Team at the Yale University (last accessed 1 Nov 2017).

[Ber68] Elwyn R. Berlekamp. *Algebraic coding theory*. McGraw-Hill series in systems science. McGraw-Hill, 1968.

[Ber71] Elwyn R. Berlekamp. Factoring Polynomials over Large Finite Fields. In *Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation*, SYMSAC '71, page 223, New York, NY, USA, 1971. ACM.

[Ber10] Daniel J. Bernstein. Grover vs. McEliece. In *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings*, pages 73–80, 2010.

[BJMM12] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 520–536, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany.

[BK15] E. B. Barker and J. M. Kelsey. Recommendation for random number generation using deterministic random bit generators. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, 2015.

[Bla83] Richard E. Blahut. *Theory and Practice of Error Control Codes.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[BLP08] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the McEliece cryptosystem. Cryptology ePrint Archive, Report 2008/318, 2008. http://eprint.iacr.org/2008/318.

[BLP11] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Smaller decoding exponents: Ball-collision decoding. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 743–760, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany.

[BM17a] Leif Both and Alexander May. Optimizing BJMM with Nearest Neighbors: Full Decoding in $2^{2/21n}$ and McEliece Security. The Tenth International Workshop on Coding and Cryptography 2017, 2017.

[BM17b] Leif Both and Alexander May. private communication, November 2017.

[BMvT78] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (Corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, May 1978.

[BS08] Bhaskar Biswas and Nicolas Sendrier. McEliece Cryptosystem Implementation: Theory and Practice. In *Proceedings of the 2nd International Workshop on Post-Quantum Cryptography*, PQCrypto '08, pages 47–62, Berlin, Heidelberg, 2008. Springer-Verlag.

[Bur71] Herbert O. Burton. Inversionless decoding of binary BCH codes. *IEEE Trans. Information Theory*, 17(4):464–466, Jul 1971.

[CEvMS15] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt. Differential power analysis of a McEliece cryptosystem. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15: 13th International Conference on Applied Cryptography and Network Security*, volume 9092 of *Lecture Notes in Computer Science*, pages 538–556, New York, NY, USA, June 2–5, 2015. Springer, Heidelberg, Germany.

[CEvMS16] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt. Masking large keys in hardware: A masked implementation of McEliece. In Orr Dunkelman and Liam Keliher, editors, *SAC 2015: 22nd Annual International Workshop on Selected Areas in Cryptography*, volume 9566 of *Lecture Notes in Computer Science*, pages 293–309, Sackville, NB, Canada, August 12–14, 2016. Springer, Heidelberg, Germany.

[Chi64] Robert Chien. Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes. *IEEE Transactions on Information Theory*, 10(4):357–363, Oct 1964.

[Cho17] Tung Chou. McBits Revisited. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2017.

[CS98] Anne Canteaut and Nicolas Sendrier. Cryptanalysis of the original McEliece cryptosystem. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology – ASIACRYPT'98*, volume 1514 of *Lecture Notes in Computer Science*, pages 187–199, Beijing, China, October 18–22, 1998. Springer, Heidelberg, Germany.

[Den03] Alexander W. Dent. A designer's guide to KEMs. In Kenneth G. Paterson, editor, *9th IMA International Conference on Cryptography and Coding*, volume 2898 of *Lecture Notes in Computer Science*, pages 133–151, Cirencester, UK, December 16–18, 2003. Springer, Heidelberg, Germany.

[Dur64] Richard Durstenfeld. Algorithm 235: Random Permutation. *Communications of the ACM*, 7(7):420–, July 1964.

[EGHP09] Thomas Eisenbarth, Tim Güneysu, Stefan Heyse, and Christof Paar. MicroEliece: McEliece for Embedded Devices. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009: 11th International Workshop Lausanne, Switzerland, September 6-9, 2009 Proceedings*, pages 49–64. Springer Berlin Heidelberg, 2009.

[FGUPT11] Jean-Charles Faugère, Ayoub Gauthier-Umaña, Valérie Otmani, Ludovic Perret, and Jean-Pierre Tillich. A Distinguisher for High Rate McEliece Cryptosystems. In *Information Theory Workshop (ITW), 2011 IEEE*, pages 282–286, October 2011.

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, January 2013.

[FS09] Matthieu Finiasz and Nicolas Sendrier. Security Bounds for the Design of Code-based Cryptosystems. In M. Matsui, editor, *Asiacrypt 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2009.

[FY48] Ronald A. Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research.* Oliver and Boyd, London, 3rd edition, 1948.

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. A key recovery attack on MDPC with CCA security using decoding errors. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 789–815, Hanoi, Vietnam, December 4–8, 2016. Springer, Heidelberg, Germany.

[GM10] Shuhong Gao and Todd Mateer. Additive Fast Fourier Transforms Over Finite Fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272, Dec 2010.

[Hey13] Stefan Heyse. *Post Quantum Cryptography: Implementing Alternative Public Key Scheme on Embedded Devices.* PhD thesis, Faculty of Electrical Engineering and Information Technology, Faculty of Electrical Engineering and Information Technology, Oct 2013.

[HG12] Stefan Heyse and Tim Güneysu. Towards one cycle per bit asymmetric encryption: Code-based cryptography on reconfigurable hardware. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 340–355, Leuven, Belgium, September 9–12, 2012. Springer, Heidelberg, Germany.

[HMP10] Stefan Heyse, Amir Moradi, and Christof Paar. Practical Power Analysis Attacks on Software Implementations of McEliece. In *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings*, pages 108–125, 2010.

[HMV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, 2004.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[KT17] Ghazal Kachigar and Jean-Pierre Tillich. Quantum Information Set Decoding Algorithms. In *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, pages 69–89, 2017.

[KY76] Donald E. Knuth and Andrew C. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.

[LB88] Pil Joong Lee and Ernest F. Brickell. An observation on the security of McEliece's public-key cryptosystem. In C. G. Günther, editor, *Advances in Cryptology – EUROCRYPT'88*, volume 330 of *Lecture Notes in Computer Science*, pages 275–280, Davos, Switzerland, May 25–27, 1988. Springer, Heidelberg, Germany.

[LC04] Shu Lin and Daniel J. Costello. *Error Control Coding.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 2004.

[LDW94] Yuan Xing Li, Robert H. Deng, and Xin Mei Wang. On the equivalence of McEliece's and Niederreiter's public-key cryptosystems. *IEEE Transactions on Information Theory*, 40:271–273, January 1994.

[LS01] P. Loidreau and N. Sendrier. Weak keys in the McEliece public-key crtptosystem. *IEEE Transactions on Information Theory*, 47(3):1207–1211, 2001.

[Lum13] Jérémie Lumbroso. Optimal Discrete Uniform Generation from Coin Flips, and Applications. *Computing Research Repository*, abs/1304.1916, 2013.

[Mas69] James L. Massey. Shift-register synthesis and BCH decoding. *IEEE Trans. Information Theory*, 15(1):122–127, Jan 1969.

[McE78] Robert J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.

[MMT11] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding random linear codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 107–124, Seoul, South Korea, December 4–8, 2011. Springer, Heidelberg, Germany.

[MS77] Florence J. MacWilliams and Neil J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland Publishing Company, 1977.

[Nie86] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. In *Problems of Control and Information Theory 15*, pages 159–166, 1986.

[NIS15] NIST. FIPS PUB 202 Federal Information Processing Standards Publication: SHA-3 Standard: Permutation-Baed Hash and Extendable-Output Functions, August 2015.

[NIS16] NIST. Post-Quantum Cryptography Standardization: Call for Proposals, December 2016.

[OS09] Raphael Overbeck and Nicolas Sendrier. Code-based cryptography. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, *Post-Quantum Cryptography*, pages 95–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Pat75] Nicholas J. Patterson. The Algebraic Decoding of Goppa Codes. *IEEE Transactions on Information Theory*, 21(2):203–207, September 1975.

[PQC] Post-quantum cryptography for long-term security PQCRYPTO ICT-645622. https://pqcrypto.eu.org. Last accessed 1 Nov 2017.

[Pra62] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Trans. Information Theory*, 8(5):5–9, 1962.

[PRD+16] Martin Petrvalsky, Tania Richmond, Milos Drutarovsky, Pierre-Louis Cayrel, and Viktor Fischer. Differential power analysis attack on the secure bit permutation in the McEliece cryptosystem. In *2016 26th International Conference Radioelektronika (RADIOELEKTRONIKA)*, pages 132–137, April 2016.

[Ret75] Charles Retter. Decoding Goppa codes with a BCH decoder. *IEEE Transactions on Information Theory*, 21(1):112–112, Jan 1975.

[RZ14] Marek Repka and Pavol Zajac. Overview of the McEliece Cryptosystem and its Security. *ATatra Mountains Mathematical Publications*, 60(1):57–83, Sep 2014.

[SS92] Vladimir M. Sidel'nikov and Sergey O. Shestakov. On insecurity of cryptosystems based on generalized Reed-Solomon codes. *Discrete Mathematics and Applications*, 2, January 1992.

[SSMS09] Abdulhadi Shoufan, Falko Strenzke, H. Gregor Molter, and Marc Stöttinger. A Timing Attack against Patterson Algorithm in the McEliece PKC. In *Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers*, pages 161–175, 2009.

[Ste88] Jacques Stern. A method for finding codewords of small weight. In *Coding Theory and Applications, 3rd International Colloquium, Toulon, France, November 2-4, 1988, Proceedings*, pages 106–113, 1988.

[STM+08] Falko Strenzke, Erik Tews, H. Gregor Molter, Raphael Overbeck, and Abdulhadi Shoufan. Side Channels in the McEliece PKC. In *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, pages 216–229, 2008.

[Str10a] Falko Strenzke. A Smart Card Implementation of the McEliece PKC. In *Proceedings of the 4th IFIP WG 11.2 International Conference on Information Security Theory and Practices: Security and Privacy of Pervasive Systems and Smart Devices*, WISTP'10, pages 47–59, Berlin, Heidelberg, 2010. Springer-Verlag.

[Str10b] Falko Strenzke. A Timing Attack against the Secret Permutation in the McEliece PKC. In *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings*, pages 95–107, 2010.

[Str13a] Falko Strenzke. *Efficiency and Implementation Security of Code-based Cryptosystems.* PhD thesis, Technischen Universität Darmstadt, Nov 2013.

[Str13b] Falko Strenzke. Timing attacks against the syndrome inversion in code-based cryptosystems. In *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, pages 217–230, 2013.

[von51] John von Neumann. Various techniques used in connection with random digits. In A.S. Householder, G.E. Forsythe, and H.H. Germond, editors, *Monte Carlo*

Method, pages 36–38. National Bureau of Standards Applied Mathematics Series, 12, Washington, D.C.: U.S. Government Printing Office, 1951.

[WSN17] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based Key Generator for the Niederreiter Cryptosystem Using Binary Goppa Codes. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 253–274, Taipei, Taiwan, September 25–28 2017. Springer, Heidelberg, Germany.

[WZ88] Y. Wang and X. Zhu. A fast algorithm for the Fourier transform over finite fields and its VLSI implementation. *IEEE Journal on Selected Areas in Communications*, 6(3):572–577, Apr 1988.

[You91] Xu Youzhi. Implementation of Berlekamp-Massey algorithm without inversion. *IEE Proceedings I - Communications, Speech and Vision*, 138(3):138–140, June 1991.

A Binary Field Arithmethic

A finite field $\mathbb{F}_{2^m}$ is constructed from an irreducible polynomial $f(x) \in \mathbb{F}_2[x]$ of degree m. The polynomial $f(x)$ is irreducible in $\mathbb{F}_2$ and it has a root β in $\mathbb{F}_{2^m}$. There exists a primitive element $\alpha \in \mathbb{F}_{2^m}$ that generates a cyclic multiplicative group of order $2^m - 1$, i.e. $\{\alpha, \alpha^2, ..., \alpha^{2^m-2}, 1\}$. We can write $\beta = \alpha^i$ for some integer i and depending on the choice of $f(x)$, β may be equal to α. Each element of $\mathbb{F}_{2^m}$ may be represented as a polynomial of degree at most $m-1$ with coefficients in $\mathbb{F}_2$, i.e.

$$\mathbb{F}_{2^m} = \{b_0 + b_1\beta + b_2\beta^2 + \ldots + b_{m-1}\beta^{m-1} \; : \; b_i \in \{0,1\}\}.$$

This representation is commonly known as *polynomial basis representation*. The set of binary coefficients $(b_{m-1}, \ldots, b_1, b_0)$ is represented as a machine integer in software implementations.

For the purpose of this document, we are only interested in $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{13}}$. Each element of these fields are represented as a 16-bit integer. Throughout this document, it is assumed that the following irreducible polynomials are used because these are the lowest weight irreducible polynomials in their respective fields, apart from their reciprocal polynomial counterparts:

Field	$f(x)$	Relationship of α and β
$\mathbb{F}_{2^{12}}$	$1 + x^3 + x^{12}$	$\alpha = 1 + \beta + \beta^2$
$\mathbb{F}_{2^{13}}$	$1 + x + x^3 + x^4 + x^{13}$	$\alpha = \beta$

In the following sub-sections, we describe how the operations such as addition, substraction, multiplication, squaring, modulo reduction and inversion are implemented. The emphasis is on an implementation that does not require the use of look-up tables as this could lead to timing side-channel vulnerabilities. The material in this section is drawn from [HMV04] and [Arn11].

A.1 Addition and Subtraction

In a binary field, both addition and substraction are identical and they can be easily computed as the XOR output of the two input operands in polynomial basis representation.

A.2 Multiplication

Given two field elements α^i, α^j, their product is simply $\alpha^i \cdot \alpha^j = \alpha^k$ where $k = i + j \mod 2^m - 1$. This can be easily implemented by means of a look-up table that maps a polynomial representation of an element to its logarithmic value and vice-versa. However, this may introduce a timing side-channel and we give algorithms avoiding look-up tables below.

Let $a_i(x)$ be the polynomial representation of $\alpha^i \in \mathbb{F}_{2^m}$ for some integer i, the product of α^i and α^j may also be computed as the polynomial product of $a_i(x)$ and $a_j(x)$ reduced modulo $f(x)$, the irreducible polynomial that defines the field. The product of $a_i(x)$ and $a_j(x)$ may be implemented as a sequence of shift-and-add operations. Refer to Section A.3 on how to perform reduction modulo $f(x)$. The pseudocode below shows how to perform multiplication over $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{13}}$. Note that the operator $\times$ on the pseudocode below denotes integer multiplication.

```
function FFMultiply12($\alpha$, $\beta$)
/* $\alpha, \beta \in \mathbb{F}_{2^{12}}$ are 16-bit integers          */
/* $\sigma$ is a 32-bit integer                         */
    $\sigma = \alpha \times (\beta$ AND 1)
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 2))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 4))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 8))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 16))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 32))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 64))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 128))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 256))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 512))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 1024))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 2048))
    return FFReduce12($\sigma$)
end function
```

```
function FFMultiply13($\alpha$, $\beta$)
/* $\alpha, \beta \in \mathbb{F}_{2^{13}}$ are 16-bit integers          */
/* $\sigma$ is a 32-bit integer                         */
    $\sigma = \alpha \times (\beta$ AND 1)
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 2))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 4))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 8))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 16))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 32))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 64))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 128))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 256))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 512))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 1024))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 2048))
    $\sigma = \sigma$ XOR ($\alpha \times (\beta$ AND 4096))
    return FFReduce13($\sigma$)
end function
```

A.3 Modulo Reduction

The modulo operation $a(x) \mod f(x)$ may be computed using long division but this is not efficient for software implementation. If $f(x)$ is chosen such that it has the lowest possible weight, for example a trinomial or pentanomial, the modulo operation can be efficiently performed with a few shift and add operations.

Consider the case for $\mathbb{F}_{2^{12}}$ where we can use the trinomial $f(x) = 1+x^3+x^{12}$. The intermediate value σ of the product of two elements of $\mathbb{F}_{2^{12}}$ is a polynomial of degree at most 22. Following $f(x)$, we can write the following two sets of congruences:

$$\begin{array}{cc} x^{22} = x^{13} + x^{10} & x^{15} = x^6 + x^3 \\ \vdots & \vdots \\ x^{16} = x^7 + x^4 & x^{12} = x^3 + 1 \end{array}$$

In the first set of congruences above, we isolate the bits at indices 16 to 22, shift them to the right by $(22 - 13) = 9$ positions and add these to σ; we also shift right them to the right by $(22 - 10) = 12$ positions and add these to σ. Once the operations on the first set of congruence are completed, the same operations are applied to the second set of congruences.

Likewise, for the case of $\mathbb{F}_{2^{13}}$, we use the pentanomial $f(x) = 1 + x + x^3 + x^4 + x^{13}$. The intermediate product value σ is a polynomial of degree at most 24 and as before, we have the following two sets of congruences:

$$\begin{array}{cc} x^{24} = x^{15} + x^{14} + x^{12} + x^{11} & x^{15} = x^6 + x^5 + x^3 + x^2 \\ \vdots & x^{14} = x^5 + x^4 + x^2 + x \\ x^{16} = x^7 + x^6 + x^4 + x^3 & x^{13} = x^4 + x^3 + x + 1 \end{array}$$

The pseudocode below shows how modulo reduction is computed for $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{13}}$. Note that the symbol $\gg$ denotes a right shift operation.

```
function FFReduce12($\sigma$)
/* $\sigma, \mu$ are 32-bit integers                 */
    $\mu = \sigma$ AND 0x007F0000
    $\sigma = \sigma$ XOR ($\mu \gg 9$)
    $\sigma = \sigma$ XOR ($\mu \gg 12$)
/* The first set of congruence is done     */
    $\mu = \sigma$ AND 0x0000F000
    $\sigma = \sigma$ XOR ($\mu \gg 9$)
    $\sigma = \sigma$ XOR ($\mu \gg 12$)
/* The second set of congruence is done */
    return ($\sigma$ AND 0x0FFF)
end function
```

```
function FFReduce13($\sigma$)
/* $\sigma, \mu$ are 32-bit integers                 */
    $\mu = \sigma$ AND 0x01FF0000
    $\sigma = \sigma$ XOR ($\mu \gg 9$)
    $\sigma = \sigma$ XOR ($\mu \gg 10$)
    $\sigma = \sigma$ XOR ($\mu \gg 12$)
    $\sigma = \sigma$ XOR ($\mu \gg 13$)
/* The first set of congruence is done     */
    $\mu = \sigma$ AND 0x0000E000
    $\sigma = \sigma$ XOR ($\mu \gg 9$)
    $\sigma = \sigma$ XOR ($\mu \gg 10$)
    $\sigma = \sigma$ XOR ($\mu \gg 12$)
    $\sigma = \sigma$ XOR ($\mu \gg 13$)
/* The second set of congruence is done */
    return ($\sigma$ AND 0x1FFF)
end function
```

A.4 Squaring

Let $\alpha^i \in \mathbb{F}_{2^m}$ for some integer i and $a_i(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{m-1}x^{m-1}$ be its polynomial representation. Squaring α^i produces $a_i(x)^2 = b_0 + b_1x^2 + b_2x^4 + \cdots + b_{m-1}x^{2(m-1)}$. If we represent α^i in its binary form, we have $(b_{m-1}, b_{m-2}, \ldots, b_1, b_0)_2$ and the effect of squaring is the insertion of 0 between consecutive bits of α^i, i.e. $(b_{m-1}, 0, b_{m-2}, 0, \ldots, 0, b_1, 0, b_0)_2$. This operation can be computed efficiently as shown in the following pseudocode. Note that the symbol $\ll$ denotes a left shift operation.

```
function FFSquare12($\alpha$)
/* $\alpha \in \mathbb{F}_{2^{12}}$ is a 16-bit integer */
/* $\sigma$ is a 32-bit integer */
    $\sigma = \alpha$
    $\sigma = (\sigma$ OR $(\sigma \ll 8))$ AND 0x00FF00FF
    $\sigma = (\sigma$ OR $(\sigma \ll 4))$ AND 0x0F0F0F0F
    $\sigma = (\sigma$ OR $(\sigma \ll 2))$ AND 0x33333333
    $\sigma = (\sigma$ OR $(\sigma \ll 1))$ AND 0x55555555
    return FFReduce12($\sigma$)
end function
```

```
function FFSquare13($\alpha$)
/* $\alpha \in \mathbb{F}_{2^{13}}$ is a 16-bit integer */
/* $\sigma$ is a 32-bit integer */
    $\sigma = \alpha$
    $\sigma = (\sigma$ OR $(\sigma \ll 8))$ AND 0x00FF00FF
    $\sigma = (\sigma$ OR $(\sigma \ll 4))$ AND 0x0F0F0F0F
    $\sigma = (\sigma$ OR $(\sigma \ll 2))$ AND 0x33333333
    $\sigma = (\sigma$ OR $(\sigma \ll 1))$ AND 0x55555555
    return FFReduce13($\sigma$)
end function
```

A.5 Inversion

Following Fermat's Little Theorem, if $\alpha \in \mathbb{F}_{2^m}$ and $\alpha \neq 0$ then its inverse $\alpha^{-1} = \alpha^{2^m-2}$. The value α^{2^m-2} may be obtained by repeated squaring and a few multiplication operations. Consider the case of $\mathbb{F}_{2^{12}}$, the integer 4094 may be written as follows

$$4094 = \underbrace{\left(\underbrace{\left(\underbrace{\left(\underbrace{\left(\underbrace{\left(\underbrace{\left(\underbrace{((3 \times 2^2) + 3)}_{15} \times 2^4\right)}_{240} +15\right)}_{255} \times 2^2\right)}_{1020} +3\right)}_{1023} \times 2\right)}_{2046} +1\right)}_{2047} \times 2.$$

In the case of $\mathbb{F}_{2^{13}}$, we can write integer 8190 as follows

$$8190 = \underbrace{\left(\underbrace{\left(\underbrace{((((((3 \times 2^2) + 3) \times 2^4) + 15) \times 2^2) + 3)}_{1023} \times 2^2\right)}_{4092} +3\right)}_{4095} \times 2.$$

Based on the above representation of integers 4094 and 8190, the following pseudocode shows how inversion is computed. As evident from the pseudocode, inversion is an expensive operation.

```
function FFInverse12($\alpha$)
/* $\alpha, \alpha_3, \alpha_{15}, \sigma \in \mathbb{F}_{2^{12}}$                */
    $\alpha_3 = $ FFSquare12($\alpha$)                // $\alpha^2$
    $\alpha_3 = $ FFMultiply12($\alpha_3, \alpha$)        // $\alpha^3$

    $\alpha_{15} = $ FFSquare12($\alpha_3$)             // $\alpha^6$
    $\alpha_{15} = $ FFSquare12($\alpha_{15}$)           // $\alpha^{12}$
    $\alpha_{15} = $ FFMultiply12($\alpha_{15}, \alpha_3$)  // $\alpha^{15}$

    $\sigma = $ FFSquare12($\alpha_{15}$)              // $\alpha^{30}$
    $\sigma = $ FFSquare12($\sigma$)                // $\alpha^{60}$
    $\sigma = $ FFSquare12($\sigma$)                // $\alpha^{120}$
    $\sigma = $ FFSquare12($\sigma$)                // $\alpha^{240}$
    $\sigma = $ FFMultiply12($\sigma, \alpha_{15}$)       // $\alpha^{255}$

    $\sigma = $ FFSquare12($\sigma$)                // $\alpha^{510}$
    $\sigma = $ FFSquare12($\sigma$)                // $\alpha^{1020}$
    $\sigma = $ FFMultiply12($\sigma, \alpha_3$)        // $\alpha^{1023}$

    $\sigma = $ FFSquare12($\sigma$)                // $\alpha^{2046}$
    $\sigma = $ FFMultiply12($\sigma, \alpha$)         // $\alpha^{2047}$
    return FFSquare12($\sigma$)               // $\alpha^{4094}$
end function
```

```
function FFInverse13($\alpha$)
/* $\alpha, \alpha_3, \alpha_{15}, \sigma \in \mathbb{F}_{2^{13}}$                */
    $\alpha_3 = $ FFSquare13($\alpha$)                // $\alpha^2$
    $\alpha_3 = $ FFMultiply13($\alpha_3, \alpha$)        // $\alpha^3$

    $\alpha_{15} = $ FFSquare13($\alpha_3$)             // $\alpha^6$
    $\alpha_{15} = $ FFSquare13($\alpha_{15}$)           // $\alpha^{12}$
    $\alpha_{15} = $ FFMultiply13($\alpha_{15}, \alpha_3$)  // $\alpha^{15}$

    $\sigma = $ FFSquare13($\alpha_{15}$)              // $\alpha^{30}$
    $\sigma = $ FFSquare13($\sigma$)                // $\alpha^{60}$
    $\sigma = $ FFSquare13($\sigma$)                // $\alpha^{120}$
    $\sigma = $ FFSquare13($\sigma$)                // $\alpha^{240}$
    $\sigma = $ FFMultiply13($\sigma, \alpha_{15}$)       // $\alpha^{255}$

    $\sigma = $ FFSquare13($\sigma$)                // $\alpha^{510}$
    $\sigma = $ FFSquare13($\sigma$)                // $\alpha^{1020}$
    $\sigma = $ FFMultiply13($\sigma, \alpha_3$)        // $\alpha^{1023}$

    $\sigma = $ FFSquare13($\sigma$)                // $\alpha^{2046}$
    $\sigma = $ FFSquare13($\sigma$)                // $\alpha^{4092}$
    $\sigma = $ FFMultiply13($\sigma, \alpha_3$)        // $\alpha^{4095}$
    return FFSquare13($\sigma$)               // $\alpha^{8190}$
end function
```

B Additive Fast Fourier Transform

The most time-consuming stage in decoding a Goppa code is finding the roots of an error locator polynomial. The simplest method is do this was discovered by Robert Chien [Chi64]. The Chien search basically evaluates the error locator polynomial with a specific order of the elements of the finite field that defines the code. Let $e(x) = e_0 + e_1 x + \ldots + e_\tau x^\tau$ be the error locator polynomial where $e_i \in \mathbb{F}_{2^m}$, Chien's method finds the roots of $e(x)$ by exploiting the following relationship

$$\begin{aligned} e(\alpha^i) &= e_0 + e_1\alpha^i + \ldots + e_\tau \left(\alpha^i\right)^\tau \\ e(\alpha^{i+1}) &= e_0 + e_1\alpha^i \cdot \alpha + \ldots + e_\tau \left(\alpha^i\right)^\tau \cdot \alpha^\tau, \end{aligned}$$

where α is a primitive element of $\mathbb{F}_{2^m}$. Whilst this method is ideal for hardware implementation, the theoretical complexity is $\mathcal{O}(\tau 2^m)$ assuming that the Goppa code is of length 2^m, correcting at most τ errors, and is not the lowest complexity that can be achieved.

Another method of root finding is due to Berlekamp [Ber71] and is commonly known as the Berlekamp Trace Algorithm (BTA). This method has a theoretical complexity of $\mathcal{O}(m\tau^2)$.

Both Chien search and BTA are efficient enough for practical purposes when $m \leq 11$. For larger m values, we need to use an approach like the Fast Fourier Transform (FFT) that offers sub-quadratic theoretical complexity. Evaluating a polynomial $e(x)$ for all $x \in \mathbb{F}_{2^m}$

is equivalent to taking the FFT of the polynomial. For binary finite fields however, it is not possible to perform an FFT operation in a traditional sense, i.e. execute a multiplicative FFT, because there do not exist primitive n-th roots of unity in $\mathbb{F}_{2^m}$. On the other hand, this is not the case for additive FFT as first shown by Wang and Zhu [WZ88] and more recently by Gao and Mateer [GM10].

Following [GM10], we describe the additive FFT and how to implement it in this section. Let $A = \langle \alpha_0, \alpha_1, \ldots, \alpha_{m-1} \rangle$ be a basis of $\mathbb{F}_{2^m}$ and

$$A[i] = \{b_0\alpha_0 + b_1\alpha_1 + \ldots + b_{m-1}\alpha_{m-1} \ : \ b_i \in \{0, 1\}\}$$

the i-th element of $\mathbb{F}_{2^m}$ under the basis A where $(b_0, b_1, \ldots, b_{m-1})_2$ is the binary representation of i. If $f(x) \in \mathbb{F}_{2^m}[x]$ has degree less than $n = 2^m$, then the additive FFT of $f(x)$ over basis A is denoted as

$$\mathrm{FFT}(f(x), m, A) = (f(A[0]), f(A[1]), \ldots, f(A[n-1])). \tag{7}$$

As shown in equation (7), we can use the additive FFT to evaluate $f(x)$ over all elements of $\mathbb{F}_{2^m}$. The roots of $f(x)$ are the set $\{A[i] \in \mathbb{F}_{2^m} \mid f(A[i]) = 0 \text{ for } 0 \le i < n\}$. As in the classical FFT algorithm, the efficiency of the FFT comes from reducing a problem of size n into two problems of size $n/2$, and recursively applying this reduction until we hit a problem of small enough size suitable for direct evaluation.

In order to perform the aforementioned reduction, we derive a new basis $\Gamma = \langle \gamma_0, \gamma_1, \ldots, \gamma_{m-2} \rangle$ from A of size $m - 1$ where $\gamma_i = \alpha_i \alpha_{m-1}^{-1}$. By denoting $g(x) = f(\alpha_{m-1}x)$, then we have $\mathrm{FFT}(f(x), m, A) = \mathrm{FFT}(g(x), m, A \cdot \alpha_{m-1}^{-1})$ where $A \cdot \alpha_{m-1}^{-1} = \Gamma \cup (1 + \Gamma)$. Consequently, we can reduce $\mathrm{FFT}(f(x), m, A)$ into two FFTs of half the size, i.e.

$$\mathrm{FFT}(f(x), m, A) = (\mathrm{FFT}(g(x), m-1, \Gamma), \mathrm{FFT}(g(x), m-1, 1+\Gamma))\,.$$

We can write the polynomial $g(x)$ as

$$g(x) = \sum_{i=0}^{n/2-1} (\bar{g}_i + \hat{g}_i x) \cdot (x^2 - x)^i \tag{8}$$

where $\bar{g}_i, \hat{g}_i \in \mathbb{F}_{2^m}$ and this is a *Taylor expansion* of $g(x)$ at $x^2 - x$. Let $\sigma \in \Gamma$ and $b \in \mathbb{F}_2$, evaluating the polynomial $g(x)$ at $(\sigma + b)$, we have

$$\begin{aligned} g(\sigma + b) &= \sum_{i=0}^{n/2-1} (\bar{g}_i + \hat{g}_i(\sigma + b)) \cdot ((\sigma + b)^2 - (\sigma + b))^i \\ &= \sum_{i=0}^{n/2-1} \bar{g}_i(\sigma^2 - \sigma)^i + \sigma \sum_{i=0}^{n/2-1} \hat{g}_i(\sigma^2 - \sigma)^i + b \sum_{i=0}^{n/2-1} \hat{g}_i(\sigma^2 - \sigma)^i \\ &= \bar{g}(\sigma^2 - \sigma) + \sigma\hat{g}(\sigma^2 - \sigma) + b\hat{g}(\sigma^2 - \sigma) \end{aligned} \tag{9}$$

where we define

$$\bar{g}(x) = \sum_{i=0}^{n/2-1} \bar{g}_i x^i \qquad \text{and} \qquad \hat{g}(x) = \sum_{i=0}^{n/2-1} \hat{g}_i x^i. \tag{10}$$

By deriving another basis $\Delta = \langle \delta_0, \delta_1, \ldots, \delta_{m-2} \rangle$ of the same size from Γ where $\delta_i = \gamma^2 - \gamma$, it follows that

$$\text{FFT}(g(x), m-1, \Gamma) = (w_0, w_1, \ldots, w_{n/2-1})$$

can be obtained from

$$\text{FFT}(\bar{g}(x), m-1, \Delta) = (u_0, u_1, \ldots, u_{n/2-1})$$

and

$$\text{FFT}(\hat{g}(x), m-1, \Delta) = (v_0, v_1, \ldots, v_{n/2-1})$$

whereby $w_i = u_i + \Gamma[i] \cdot v_i$ and $\Gamma[i]$ is the i-th element of the subfield of $\mathbb{F}_{2^m}$ defined by basis Γ. Now that we have $\text{FFT}(g(x), m-1, \Gamma)$, following equation (9) and by letting $b = 1$, we then have

$$\text{FFT}(g(x), m-1, 1+\Gamma) = \text{FFT}(g(x), m-1, \Gamma) + \text{FFT}(\hat{g}(x), m-1, \Delta).$$

Algorithm 4 summarises the additive FFT execution above in pseudocode format. How to compute the Taylor expansion of a polynomial at $(x^2 - x)$, i.e. as given by equation (8), is described in Algorithm 5.

In addition to being used to compute the roots of an error locator polynomial in decoding a Goppa code, the additive FFT is also used as part of syndrome computation to evaluate the Goppa polynomial $G(z)$ over all elements of $\mathbb{F}_{2^m}$. This kind of evaluation is also invoked during the key generation process to determine whether or not the randomly generated Goppa polynomial $G(z)$ is valid. The use of additive FFT for cryptography was first shown in McBits [BCS13].

Algorithm 4 Additive FFT of $f(x) \in \mathbb{F}_{2^m}[x]$

```
function AdditiveFFT($(f(x), m, A)$)
Require: $A = \langle \alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{m-1} \rangle$ where $\alpha_i \in \mathbb{F}_{2^m}$
Require: Stack is initialised
Require: $m > 0$ and $\mathbf{w} = \mathbf{0}^{2^m}$
Require: $k \leftarrow 0$
    /* As we reduce the FFT into two FFTs of half the original size at stage $i$,          */
    /* two new bases $\Gamma_i = \langle \gamma_{i,0}, \gamma_{i,1}, \ldots, \gamma_{i,i} \rangle$ and $\Delta_i = \langle \delta_{i,0}, \delta_{i,1}, \ldots, \delta_{i,i} \rangle$ are          */
    /* constructed. We need to pre-compute $\Gamma_i$ and $\Delta_i$ for $i = \{1, 2, \ldots, m-1\}$          */
    $\Gamma_{m-1} = \Delta_{m-1} = A$
    for $i \leftarrow (m-2)$ downto 1 step $-1$ do
        $\Gamma_i = \Gamma_{i+1} \cdot \gamma_{i+1,i+1}^{-1}$
        $\Delta_i = \langle \delta_{i,0}, \delta_{i,1}, \ldots, \delta_{i,i} \rangle$ where $\delta_{i,j} = (\gamma_{i,j}^2 - \gamma_{i,j})$
    end for
    Stack $\xleftarrow{\text{Push}}$ $(f(x), m)$
    while Stack is not empty do
        $(f'(x), m') \xleftarrow{\text{Pop}}$ Stack
        if $m' = 1$ OR $\deg f'(x) \leq 0$ then
            $w_k \leftarrow f'_0$
            $w_{k+1} \leftarrow f'_0 + \delta_{0,0} \cdot f'_1$
            $k \leftarrow k + 2^{m'}$
            Continue
        end if
        $g(x) \leftarrow f'(\delta_{m'-2,m'-1}x)$
        $\ell \leftarrow \lceil (\deg g(x) + 1)/2 \rceil$
        $((\bar{g}_0 + \hat{g}_0 x), \ldots, (\bar{g}_{\ell-1} + \hat{g}_{\ell-1} x)) \leftarrow$ TaylorExpansion($g(x)$)
        $\bar{g}(x) \leftarrow \sum_{i=0}^{n/2-1} \bar{g}_i x^i$
        $\hat{g}(x) \leftarrow \sum_{i=0}^{n/2-1} \hat{g}_i x^i$
        Stack $\xleftarrow{\text{Push}}$ $(\hat{g}(x), m' - 1)$
        Stack $\xleftarrow{\text{Push}}$ $(\bar{g}(x), m' - 1)$
    end while
    /* Update $\mathbf{w}$          */
    for $s \leftarrow 1$ to $m$ step 1 do
        for $i \leftarrow 0$ to $2^m$ step $2^{s+1}$ do
            for $j \leftarrow i$ to $(2^s + i)$ step 1 do
                $w_j = w_j + (\Gamma_{s-1}[j-i] \cdot w_{2^s+j})$
                $w_{2^s+j} = w_{2^s+j} + w_j$
            end for
        end for
    end for
    return $\mathbf{w}$
end function
```

Algorithm 5 Taylor Expansion $f(x) \in \mathbb{F}_{2^m}[x]$ at $(x^2 - x)$

```
function TaylorExpansion($f(x)$)
Ensure: $\mathbf{L} \leftarrow ((\bar{f}_0 + \hat{f}_0 x), \ldots, (\bar{f}_{j-1} + \hat{f}_{j-1} x))$ where $j = \lceil (\deg f(x) + 1)/2 \rceil$
Require: $\mathbf{L} \leftarrow \emptyset$
Require: Stack is initialised
    Stack $\xleftarrow{\text{Push}}$ $(f(x), \deg f(x) + 2)$
    while Stack is not empty do
        $(f'(x), \ell) \xleftarrow{\text{Pop}}$ Stack
        if $\ell \leq 2$ then
            $\mathbf{L} \xleftarrow{\text{Append}} f'(x)$
            Continue
        end if
        Find $k$ that satisfies $2^{k+1} < \ell \leq 2^{k+2}$
        Partition $f'(x)$ into three blocks as $f'(x) = f'_0(x) + x^{2^{k+1}}(f'_1(x) + x^{2^k} f'_2(x))$ where
            • $\deg f'_0(x) < 2^{k+1}$,
            • $\deg f'_1(x) < \min\{\ell - 2^{k+1}, 2^k\}$, and
            • $f'_2(x) = 0$ if $(\ell - 2^{k+1}) < 2^k$ otherwise $\deg f'_2(x) < 2^k$
        Compute $g_0(x) \leftarrow f'_0(x) + x^{2^k}(f'_1(x) + f'_2(x))$
        Compute $g_1(x) \leftarrow (f'_1(x) + f'_2(x)) + x^{2^k} f'_2(x)$
        Stack $\xleftarrow{\text{Push}}$ $(g_1(x), \ell - 2^{k+1})$
        Stack $\xleftarrow{\text{Push}}$ $(g_0(x), 2^{k+1})$
    end while
    return $\mathbf{L}$
end function
```

C Polynomial Derivative and GCD

One of the conditions on the validity of a Goppa polynomial $G(z)$ is that it does not have repeated zeros. This condition is met if $G(z)$ and its derivative are relatively prime, i.e. $\text{GCD}\left(G(z), \frac{d}{dz}G(z)\right) = 1$. Let $G(z) = \sum_{i=0}^{\tau} g_i z^i$, its derivative is defined by

$$\begin{aligned} \frac{d}{dz}G(z) &= \sum_{i=1}^{\tau} i g_i z^{i-1} \\ &= \sum_{j=0}^{\lceil \tau/2 \rceil - 1} g_{2j+1} z^{2j}. \end{aligned}$$

Note that because we are working in the field of characteristic 2, $\frac{d}{dz}G(z)$ contains even powers only and it is a perfect square.

The GCD of two polynomials in $\mathbb{F}_{2^m}[z]$ may be computed by repeated modular reduction as shown in the following algorithm.

Algorithm 6 Greatest common divisor of $a(z)$ and $b(z)$

```
function GCD($(a(z), b(z))$)
Require: $\deg a(z) \geq \deg b(z)$
    while $\deg b(z) \geq 0$ do
        $t(z) \leftarrow b(z)$
        $b(z) \leftarrow a(z) \mod b(z)$
        $a(z) \leftarrow t(z)$
    end while
    return $a(z)$
end function
```

D Random Permutation

The Fisher-Yates shuffle is an algorithm to generate a random permutation on a sequence of finite length. The original Fisher-Yates shuffle [FY48] was not suitable for computer use. The version suitable for computer implementation, which is shown in Algorithm 7, was introduced by Durstenfeld [Dur64] and it also appears in Knuth's book [Knu97], hence it is commonly referred to as the Knuth shuffle.

As shown in Step 4 of Algorithm 7, the Fisher-Yates shuffle requires sampling of uniform random integers between 0 and i for some integer $i < n$ where n is the length of the sequence. However, most random number generators produce numbers in some fixed range M that is usually a power of 2. If we need to generate a uniform random number r between 0 and i where $i \leq M$ and is not a power of 2, we cannot simply force the generated random numbers to be in the range by means of a modular reduction operation as this will introduce a bias.

Algorithm 7 Fisher-Yates shuffle on sequence $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$

```
function RandomShuffle($\mathbf{a}$)
    $i \leftarrow n - 1$
    while $i > 0$ do
        $r \leftarrow_{\$} \{0, 1, \ldots, i\}$
        Swap $a_i$ with $a_r$
        $i \leftarrow i - 1$
    end while
    return the shuffled sequence $\mathbf{a}$
end function
```

One way to address this is to perform rejection sampling; we start by divide the range M into blocks of size i, sample a number r, if $r > i \lfloor M/i \rfloor$ reject it and repeat sampling, otherwise output $r = r/\lfloor M/i \rfloor$.

A more elegant method is to simulate successively the discrete uniform distribution by flipping unbiased coins, i.e. generating random bits [BBHT17]. To simulate the sampling of random number r, generate $\lceil \log_2 r \rceil$ random bits and if the integer value of these bits when read as a binary representation is less than r, return the sample r; otherwise reject these bits and restart the sampling until a value $< r$ is found. This rejection method is attributed to von Neumann [von51] and it is called *Simple Discard Method* in NIST's recommendation for random number generation [BK15]. However, the *Simple Discard Method* is not efficient; instead of rejecting the bits and starting again, we can use the difference between this value and r to seed the next sampling round. This latter method relies on Knuth-Yao's discrete distribution generating-tree (DDG-tree) [KY76] algorithm [Lum13, BBHT17] and it is shown in Algorithm 8.

Algorithm 8 Knuth-Yao algorithm to generate a uniform random number in $\{0, \ldots, i - 1\}$

```
function KnuthYaoUniformRNG($i$)
    $u \leftarrow 1$
    $x \leftarrow 0$
    while TRUE do
        while $u < i$ do
            $u \leftarrow 2u$
            $x \leftarrow 2x + \text{RandomBit}$
        end while
        $d \leftarrow u - i$
        if $x \geq d$ then
            return $x - d$
        else
            $u \leftarrow d$
        end if
    end while
end function
```

E IND-CCA Security Reduction for NTS-KEM

NTS-KEM achieves IND-CCA security in the Random Oracle Model by employing a transform akin to the Fujisaki-Okamoto [FO13] or Dent [Den03] transforms.

As a stepping stone towards our IND-CCA proof, we define a variant of NTS-KEM, which we denote NTS$^-$. Recall that NTS-KEM creates encapsulations which are essentially encryptions (in the McEliece PKE scheme with public key in systematic form) of message vectors of the form $\mathbf{m} = (\mathbf{e}_a \mid \mathbf{k}_e) \in \mathbb{F}_2^k$, where $\mathbf{k}_e = H_\ell(\mathbf{e}) \in \mathbb{F}_2^\ell$; the encapsulated key is then defined to be $\mathbf{k}_r = H_\ell(\mathbf{k}_e|\mathbf{e}) \in \mathbb{F}_2^\ell$. In contrast, the scheme NTS$^-$ creates encapsulations which are encryptions of message vectors of the form $(\mathbf{e}_a \mid \mathbf{r}_b)$, where $\mathbf{r}_b$ is a uniformly random string in $\mathbb{F}_2^\ell$; we take $\mathbf{r}_b$ as the encapsulated key for NTS$^-$. That is, NTS$^-$ outputs as encapsulations vectors $\mathbf{c} = (\mathbf{c}_b \mid \mathbf{c}_c)$ such that

$$\begin{aligned}(\mathbf{0}_a \mid \mathbf{c}_b \mid \mathbf{c}_c) &= (\mathbf{m} \mid \mathbf{m} \cdot \mathbf{Q}) + \mathbf{e} \\ &= (\mathbf{e}_a \mid \mathbf{r}_b \mid (\mathbf{e}_a \mid \mathbf{r}_b) \cdot \mathbf{Q}) + (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c) \in \mathbb{F}_2^n.\end{aligned}$$

Clearly, any adversary capable of recovering $\mathbf{e}$ from $\mathbf{c}$ as defined above can also recover $\mathbf{r}_b$, since $\mathbf{c}_b = \mathbf{r}_b + \mathbf{e}_b$. On the other hand, knowledge of $\mathbf{r}_b$ reduces the decoding problem with parameters (n, k, τ) into a smaller, albeit possibly still non-trivial, decoding problem with parameters roughly $(n-\ell, k-\ell, \tau-(\tau/n)\cdot\ell)$. Here, “roughly” means that the exact Hamming weight of the error depends on the randomness of the challenge ciphertext; $\tau - (\tau/n) \cdot \ell$ is the expectation. Thus, in contrast to the standard McEliece scheme, it is not obvious that recovering $\mathbf{r}_b$ implies the ability to recover $\mathbf{e}$ in the NTS$^-$ scheme.

To circumvent this issue, we will prove that NTS$^-$ satisfies a non-generic security notion specific to schemes like NTS-KEM, i.e. McEliece-type PKE resp. KEM schemes that encrypt resp. encapsulate messages of the form $\mathbf{m} = (\mathbf{e}_a \mid \mathbf{r}_b)$ with error vector $\mathbf{e} = (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c)$. This notion informally states that it is hard to recover the error vector $\mathbf{e}$ used to generate a challenge ciphertext $\mathbf{c}$. For this reason, we refer to “error one-wayness” and EOW security. We stress that this non-standard security notion merely serves as an intermediate step between the OW security of McEliece and the IND-CCA security of NTS-KEM in our proofs. Formally, EOW security for public-key encryption is defined via the following game:

```
$\mathrm{EOW}_{\mathsf{Enc}}^{\mathcal{A}}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow_\$ \mathsf{KGen}(1^\lambda)$
$\mathbf{r}_b \leftarrow_\$ \{0,1\}^{\mathsf{poly}(\lambda)}$
$C^* \leftarrow_\$ \mathsf{Enc}(\mathsf{pk}, \mathbf{r}_b)$
$\mathbf{e} \leftarrow$ error vector used to produce $C^*$
$\mathbf{e}' \leftarrow_\$ \mathcal{A}(1^\lambda, \mathsf{pk}, C^*)$
return $(\mathbf{e}' = \mathbf{e})$
```

Here, $\mathsf{Enc}(\mathsf{pk}, \mathbf{r}_b)$ denotes the NTS-KEM-like encryption of a message of the form $\mathbf{m} = (\mathbf{e}_a \mid \mathbf{r}_b)$.

We can also define an equivalent notion of EOW security for NTS-KEM-like KEMs via a security game. In this game, the adversary receives the encapsulation of a random key and is required to produce the error vector that led to that encapsulation:

```
$\mathrm{EOW}^{\mathcal{A}}_{\mathsf{KEM}}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^{\lambda})$
$(K, C^*) \leftarrow_{\$} \mathsf{Encap}(\mathsf{pk})$
$\mathbf{e} \leftarrow$ error vector used to produce $C^*$
$\mathbf{e}' \leftarrow_{\$} \mathcal{A}(1^{\lambda}, \mathsf{pk}, C^*)$
return $(\mathbf{e}' = \mathbf{e})$
```

The observation above that recovery of $\mathbf{e}$ implies recovery of $\mathbf{r}_b$ for NTS$^-$ can now be restated as that any adversary against EOW security of NTS$^-$ (as a KEM, encapsulating the key $\mathbf{r}_b$) can be turned into an OW adversary against NTS$^-$. The reverse implication does not necessarily hold.

We now give a proof that NTS$^-$ is EOW secure as a KEM if McEliece is OW secure as a PKE scheme. To this end, we first show that any (t, ε)-adversary against the OW security of McEliece with public key in systematic form $\mathbf{G}_{sys} = [\mathbf{I}_k \mid \mathbf{Q}]$ can be turned into an (t, ε)-adversary against the OW security of standard McEliece. We note that this reduction is well-known, but reproduce it here for completeness.

In what follows, to ease notation we will assume that the parameters ℓ and τ are public constants. Thus, NTS-KEM public keys are reduced to simply $\mathbf{Q}$. Similarly, for ease of exposure, we are going to assume that running times of all algorithms are expressed in the number of NTS-KEM encapsulations/decapsulations. That is, saying that algorithm $\mathcal{A}$ runs in time t is saying that $\mathcal{A}$ runs in the time required to perform t NTS-KEM encapsulations/decapsulations.

Lemma 1. *If there is an (t, ε)-adversary $\mathcal{A}$ against the* OW *security of the McEliece PKE scheme with public keys of the form $\mathbf{G}_{sys} = [\mathbf{I}_k \mid \mathbf{Q}]$, then there is a (t, ε)-adversary $\mathcal{B}$ against the* OW *security of the McEliece PKE scheme with any public key $\mathbf{G}$.*

Proof. By assumption, with probability ε and in time t, $\mathcal{A}$ returns $\mathbf{r}$ on input $(\mathbf{G}_{sys}, \mathbf{c}')$ with $\mathbf{c}' = \mathbf{r} \cdot \mathbf{G}_{sys} + \mathbf{e}$, $\mathrm{hw}(\mathbf{e}) = \tau$ and $\mathbf{r} \leftarrow_{\$} \mathbb{F}_2^k$. We show the existence of $\mathcal{B}$ — operating on challenge $(\mathbf{G}, \mathbf{c})$ — by constructing it explicitly from $\mathcal{A}$.

Adversary $\mathcal{B}$ receives as input $(\mathbf{G}, \mathbf{c})$, where $\mathbf{G}$ is the public key for the standard McEliece PKE scheme and $\mathbf{c} = \mathbf{m} \cdot \mathbf{G} + \mathbf{e}$ is a ciphertext for this scheme. It then proceeds as follows:

1. compute $\mathbf{G}_{sys} = \mathbf{U} \cdot \mathbf{G} \cdot \mathbf{P}$, where $\mathbf{P} \in \mathbb{F}_2^{n \times n}$ is a permutation matrix and $\mathbf{U}$ is the transformation matrix for turning $\mathbf{G}$ into reduced row-echelon form.

2. submit $(\mathbf{G}_{sys}, \mathbf{c} \cdot \mathbf{P})$ to $\mathcal{A}$ to recover $\mathbf{r}$.

3. return $\mathbf{m} = \mathbf{r} \cdot \mathbf{U}$.

The pair $(\mathbf{G}_{sys}, \mathbf{c} \cdot \mathbf{P})$ is a valid OW challenge for $\mathcal{A}$ because

$$\begin{aligned} \mathbf{c} \cdot \mathbf{P} &= \mathbf{m} \cdot \mathbf{G} \cdot \mathbf{P} + \mathbf{e} \cdot \mathbf{P} \\ &= \mathbf{r} \cdot \mathbf{U} \cdot \mathbf{G} \cdot \mathbf{P} + \mathbf{e} \cdot \mathbf{P}, \text{ where } \mathbf{r} = \mathbf{m} \cdot \mathbf{U}^{-1} \\ &= \mathbf{r} \cdot \mathbf{G}_{sys} + \mathbf{e} \cdot \mathbf{P}, \end{aligned}$$

with $\mathrm{hw}(\mathbf{e} \cdot \mathbf{P}) = \mathrm{hw}(\mathbf{e}) = \tau$.

The linear algebra in steps 1 and 3 is no more expensive than McEliece encryption. Hence, $\mathcal{B}$ essentially runs in time the same as that of $\mathcal{A}$, namely t. Furthermore, $\mathcal{B}$ succeeds if $\mathcal{A}$ succeeds. □

Next, we show that an EOW adversary against NTS$^-$ as a KEM can be turned into an OW adversary against the McEliece PKE scheme with public key in systematic form.

Lemma 2. *If there is an (t, ε)-adversary $\mathcal{A}$ against the* EOW *security of NTS$^-$ as a KEM, then there is a (t, ε)-adversary $\mathcal{B}$ against the* OW *security of the McEliece PKE scheme with public key in systematic form.*

Proof. Let $\mathbf{c} = \mathbf{r} \cdot \mathbf{G}_{sys} + \mathbf{e}$ with $\mathrm{hw}(\mathbf{e}) = \tau$ be a ciphertext for the McEliece PKE scheme in systematic form for some random message $\mathbf{r} \leftarrow_\$ \mathbb{F}_2^k$. Adversary $\mathcal{B}$ receives as input $(\mathbf{G}_{sys}, \mathbf{c})$ and wishes to recover $\mathbf{r}$.

Denote $\mathbf{G}_{sys} = [\mathbf{I}_k \mid \mathbf{Q}]$, $\mathbf{c} = (\mathbf{c}_a \mid \mathbf{c}_b \mid \mathbf{c}_c)$, $\mathbf{r} = (\mathbf{r}_a \mid \mathbf{r}_b) \in \mathbb{F}_2^k$ and $\mathbf{e} = (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c)$. It holds that

$$\mathbf{c} = \mathbf{r} \cdot \mathbf{G}_{sys} + \mathbf{e} = (\mathbf{r}_a + \mathbf{e}_a \mid \mathbf{r}_b + \mathbf{e}_b \mid (\mathbf{r}_a \mid \mathbf{r}_b) \cdot \mathbf{Q} + \mathbf{e}_c)\,.$$

$\mathcal{B}$ then constructs

$$\mathbf{c}^* = (\mathbf{c}_a \mid \mathbf{0}_b) \cdot \mathbf{G}_{sys} = (\mathbf{r}_a + \mathbf{e}_a \mid \mathbf{0}_b \mid (\mathbf{r}_a + \mathbf{e}_a \mid \mathbf{0}_b) \cdot \mathbf{Q})$$

and

$$\begin{aligned} \mathbf{c}' &= \mathbf{c} + \mathbf{c}^* \\ &= (\mathbf{r}_a + \mathbf{e}_a \mid \mathbf{r}_b + \mathbf{e}_b \mid (\mathbf{r}_a \mid \mathbf{r}_b) \cdot \mathbf{Q} + \mathbf{e}_c) + (\mathbf{r}_a + \mathbf{e}_a \mid \mathbf{0}_b \mid (\mathbf{r}_a + \mathbf{e}_a \mid \mathbf{0}_b) \cdot \mathbf{Q}) \\ &= (\mathbf{0}_a \mid \mathbf{r}_b + \mathbf{e}_b \mid (\mathbf{r}_a \mid \mathbf{r}_b) \cdot \mathbf{Q} + (\mathbf{r}_a + \mathbf{e}_a \mid \mathbf{0}_b) \cdot \mathbf{Q} + \mathbf{e}_c) \\ &= (\mathbf{0}_a \mid \mathbf{r}_b + \mathbf{e}_b \mid (\mathbf{e}_a \mid \mathbf{r}_b) \cdot \mathbf{Q} + \mathbf{e}_c) \\ &= (\mathbf{0}_a \mid \mathbf{c}'_b \mid \mathbf{c}'_c). \end{aligned}$$

The vector $(\mathbf{c}'_b \mid \mathbf{c}'_c)$ corresponds to an NTS$^-$ encapsulation of random key $\mathbf{r}_b$. Indeed, we have

$$\begin{aligned}(\mathbf{e}_a \mid \mathbf{r}_b) \cdot \mathbf{G}_{sys} + \mathbf{e} &= (\mathbf{e}_a \mid \mathbf{r}_b) \cdot [\mathbf{I}_k \mid \mathbf{Q}] + \mathbf{e} \\ &= (\mathbf{e}_a \mid \mathbf{r}_b \mid (\mathbf{e}_a \mid \mathbf{r}_b) \cdot \mathbf{Q}) + (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c) \\ &= (\mathbf{0}_a \mid \mathbf{r}_b + \mathbf{e}_b \mid (\mathbf{e}_a \mid \mathbf{r}_b) \cdot \mathbf{Q} + \mathbf{e}_c) \\ &= (\mathbf{0}_a \mid \mathbf{c}'_b \mid \mathbf{c}'_c)\end{aligned}$$

Adversary $\mathcal{B}$ then proceeds as follows:

1. Run $\mathcal{A}$ on $(\mathbf{Q}, (\mathbf{c}'_b \mid \mathbf{c}'_c))$ to recover $\mathbf{e}$.
2. Compute $\mathbf{r}' = (\mathbf{c}_a + \mathbf{e}_a \mid \mathbf{c}_b + \mathbf{e}_b)$ and return $\mathbf{r}'$.

The adversary $\mathcal{B}$ makes one call to $\mathcal{A}$ and adds essentially no extra running time to that of $\mathcal{A}$. It succeeds when $\mathcal{A}$ succeeds. □

Combining the preceding lemmas, we arrive at the following theorem:

Theorem 2. *If there is an (t, ε)-adversary $\mathcal{A}$ against the* EOW *security of NTS$^-$, then there is a (t, ε)-adversary $\mathcal{B}$ against the* OW *security of the McEliece PKE scheme.*

Next, in order to make our IND-CCA reduction for NTS-KEM "tight", i.e. so that the running times and success probabilities of the two adversaries are closely related, we will make use of the following proposition stating that for a given valid ciphertext there is a unique $\mathbf{e}$.

Proposition 1. *Let $C^* = (\mathbf{c}_b \mid \mathbf{c}_c)$ be a correctly formed ciphertext for NTS$^-$ or for NTS-KEM with public key in systematic form. Then there exists a unique pair of vectors $((\mathbf{e}_a \mid \mathbf{r}_b), \mathbf{e})$ such that* $\mathrm{hw}(\mathbf{e}) = \tau$ *and* $C^* = (\mathbf{e}_a \mid \mathbf{r}_b) \cdot [\mathbf{I}_k \mid \mathbf{Q}] + \mathbf{e}$.

Proof. First, we note that from the minimum distance $d = 2\tau + 1$ of the underlying code it follows that there is no $\mathbf{e}'$ with $\mathrm{hw}(\mathbf{e}') \leq \tau$ such that $C^* = \mathbf{r}' \cdot [\mathbf{I}_k \mid \mathbf{Q}] + \mathbf{e}'$ with $\mathbf{r}' \neq (\mathbf{e}_a \mid \mathbf{r}_b)$.

Now, from $(\mathbf{0}_a \mid \mathbf{c}_b \mid \mathbf{c}_c) + (\mathbf{e}_a \mid \mathbf{r}_b) \cdot [\mathbf{I}_k \mid \mathbf{Q}] = \mathbf{e}$, the vector $\mathbf{e}$ is completely determined. □

We are now ready to prove the main result of this appendix. At a high level, our proof for establishing the IND-CCA security of NTS-KEM proceeds as follows:

1. We show that, in the Random Oracle Model, it is possible to simulate the decapsulation oracle using only publicly available information, with the simulation being correct with high probability.

2. Then, using this simulated decapsulation oracle, we show that it is possible to convert any adversary against the IND-CCA security of NTS-KEM into an adversary against the EOW security of NTS^{-}.
3. The last step of the proof argues that the adversary must have made a query to $H(\cdot)$ involving the correct $\mathbf{e}$ to succeed or to detect that it is running in a simulation.

Lemma 3. *If there exists a (t, ε)-adversary $\mathcal{A}$ winning the* IND-CCA *game for NTS-KEM, then there exists a $\left(2\,t, \varepsilon - \frac{q_D}{2^\ell}\right)$-adversary $\mathcal{B}$ against the* EOW *security of NTS^{-}:*

- *in the Random Oracle Model; and,*
- *when the decapsulation algorithm succeeds with probability 1 for all public keys* $\mathbf{Q}$ *and all well-formed ciphertexts;*

with q_D being the number of queries made by $\mathcal{A}$ to the decapsulation oracle.

Proof. We construct the adversary $\mathcal{B}$ against the EOW security of NTS^{-} from adversary $\mathcal{A}$. Adversary $\mathcal{B}$ receives as input one NTS^{-} encapsulation C^* of some unknown $\mathbf{r}_b$ and one randomly generated public key $\mathbf{Q}$. Note that $\mathbf{Q}$ is also a randomly generated public key for NTS-KEM. We can write

$$\mathbf{c} = (\mathbf{e}_a \mid \mathbf{r}_b) \cdot [\mathbf{I}_k \mid \mathbf{Q}] + \mathbf{e}$$

where $\mathbf{e} = (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c)$ is an unknown error vector, and where $\mathbf{c} = (\mathbf{0}_a \mid C^*)$. $\mathcal{B}$ passes to $\mathcal{A}$ the pair $(\mathbf{y}, C^*)$, where $\mathbf{y} \leftarrow_{\$} \{0,1\}^\ell$, along with the public key $\mathbf{Q}$. Note that C^* is highly unlikely to be a correct NTS-KEM encapsulation of *any* key, let alone $\mathbf{y}$, a fact that we will need to account for in our analysis that follows. $\mathcal{B}$ also answers all queries of $\mathcal{A}$ to the random oracle and the decapsulation oracle.

Separating random oracles. In order to simplify the presentation of our analysis, we define three new hash functions $H_\ell^n(\cdot)$, $H_\ell^{\ell+n}(\cdot)$ and $H_\ell^{+}(\cdot)$. These are all implemented directly using $H_\ell(\cdot)$, but we give them different names to differentiate the operation of the different components of the scheme more clearly. The three hash functions are distinguished by the length of their *inputs*: $H_\ell^n(\cdot)$ accepts inputs of length n bits, $H_\ell^{\ell+n}(\cdot)$ accepts inputs of length $\ell + n$ bits, and $H_\ell^{+}(\cdot)$ accepts inputs of all lengths except n and $\ell + n$ bits. This separation by length of inputs means that we can treat $H_\ell^n(\cdot)$, $H_\ell^{\ell+n}(\cdot)$ and $H_\ell^{+}(\cdot)$ as separate random oracles in our security analysis, even though each is in the end implemented using a single random oracle $H_\ell(\cdot)$. Furthermore, when we write that $\mathcal{A}$ "queries the random oracle $H_\ell(\cdot)$" we mean that $\mathcal{A}$ picks a uniformly random string of length ℓ, maintaining tables to ensure consistency as usual.

Queries. $\mathcal{A}$ continues with $\mathcal{B}$ handling all queries made by $\mathcal{A}$ to the random oracles as follows:

1. When $\mathcal{A}$ queries the random oracle $H_\ell^{\ell+n}(\cdot)$ on input $(\mathbf{k}'_e \mid \mathbf{e}')$, $\mathcal{B}$ passes the query on to the random oracle $H_\ell(\cdot)$ and returns the answer $\mathbf{x}'$ to $\mathcal{A}$. If $\mathrm{hw}(\mathbf{e}') = \tau$ and $\mathbf{k}'_e = H_\ell(\mathbf{e}')$ then $\mathcal{A}$ computes C', the encapsulation of $\mathbf{x}'$ under public key $\mathbf{Q}$ and error vector $\mathbf{e}'$. Note that at this point $\mathcal{B}$ knows all inputs to the encapsulation algorithm needed to produce C' exactly. $\mathcal{A}$ then stores
$$\big(\mathbf{k}'_e, \mathbf{e}', \mathbf{x}', C', \ell + n\big) = \big(H_\ell(\mathbf{e}'), \mathbf{e}', H_\ell(H_\ell(\mathbf{e}') \mid \mathbf{e}'), C', \ell + n\big)$$
in a table T, unless that table already contains a row $(H_\ell(\mathbf{e}'), \mathbf{e}', H_\ell(H_\ell(\mathbf{e}') \mid \mathbf{e}'), C', n)$. The table T is organised in such a way as to allow constant-time lookup by C'.[14] The last entry "$\ell + n$" tags the row in the table as coming from a query to $H_\ell^{\ell+n}(\cdot)$.

2. When $\mathcal{A}$ queries the random oracle $H_\ell^n(\cdot)$ on input $\mathbf{e}'$, $\mathcal{B}$ passes the query on to the random oracle $H_\ell(\cdot)$ and returns the answer $\mathbf{k}'_e$ to $\mathcal{A}$. If $\mathrm{hw}(\mathbf{e}') = \tau$ then $\mathcal{A}$ also stores
$$\big(\mathbf{k}'_e, \mathbf{e}', \mathbf{x}', C', n\big) = \big(H_\ell(\mathbf{e}'), \mathbf{e}', H_\ell(H_\ell(\mathbf{e}') \mid \mathbf{e}'), C', n\big)$$
in T. If T already contains an entry $(H_\ell(\mathbf{e}'), \mathbf{e}', H_\ell(H_\ell(\mathbf{e}') \mid \mathbf{e}'), C', \ell + n)$, it is removed first. Here, again, C' is the encapsulation of $\mathbf{x}'$ under public key $\mathbf{Q}$ and error vector $\mathbf{e}'$. Note that, again, at this point $\mathcal{B}$ knows all inputs to the encapsulation algorithm needed to produce C' exactly.

3. Queries to $H_\ell^+(\cdot)$ are simply passed through to the random oracle without any record keeping.

Note that by Proposition 1 there is a one-to-one map from C' to rows of T, thus indexing by C' is well-defined. Now, whenever $\mathcal{A}$ requests a decapsulation of some ciphertext C', $\mathcal{B}$ will respond as follows:[15]

- If $C' = C^*$, return $\perp$.
- If C' is found as the penultimate component in an entry $(\mathbf{k}'_e, \mathbf{e}', \mathbf{x}', C', n) \in T$, then return $\mathbf{x}'$.
- Otherwise, return $\perp$.

Analysing the simulation of decapsulation. We argue that $\mathcal{B}$ simulates the decapsulation oracle for $\mathcal{A}$ perfectly, except with probability at most $q_D/2^\ell$, where q_D is the number of decapsulation queries made by the adversary. To see this, note that:

- Invalid encapsulations result in $\perp$ being returned by both the real decapsulation oracle and by the simulation described above. In particular, any encapsulation with $\mathrm{hw}(\mathbf{e}') \neq \tau$ is rejected.

[14] We write "constant-time" to express that an appropriate data structure is used to ensure that the running time of looking up values in T does not grow as more entries are added, i.e. a hash table.

[15] This logic is conservative in that it checks that the corresponding row in T is tagged with n. This restriction permits a more modular proof.

- Valid encapsulations produced by first making the right query to $H_\ell^n(\cdot)$ result in the correct encapsulated keys being returned by both the real decapsulation oracle and by the simulation described above. Here, we use that decapsulation succeeds with probability 1 for correctly formed ciphertexts.

Thus, the simulation fails only when $\mathcal{A}$ queries the decapsulation oracle on some correct encapsulation C' without having queried $H_\ell^n(\cdot)$ on the required inputs first. In this case, the real decapsulation will return the encapsulated key if $H_\ell^n(\mathbf{e}') = \mathbf{k}'_e$ but the simulation will return $\perp$.

First, note that, given C', there is a unique $\mathbf{e}'$ such that $H_\ell^n(\mathbf{e}') = \mathbf{k}'_e$ holds, and no other input to $H_\ell^n(\cdot)$ will lead to successful decapsulation by the real decapsulation algorithm. This is due to the uniqueness of the pair $(\mathbf{k}'_e, \mathbf{e}')$ given C' (Proposition 1).

Now, when the adversary has not queried $H_\ell^n(\mathbf{e}')$, the output of $H_\ell^n(\cdot)$ on input $\mathbf{e}'$ is still uniformly random from the adversary's point of view. This in turn implies that the probability that the equation $H_\ell^n(\mathbf{e}') = \mathbf{k}'_e$ holds is exactly $2^{-\ell}$ (because $H_\ell^n(\cdot)$ has ℓ-bit outputs).

Thus, the probability that the simulation fails when considering all q_D decryption queries can be bounded by $q_D/2^\ell$ by applying the union bound. As a consequence, we have that with probability at least $\left(1 - \frac{q_D}{2^\ell}\right)$, adversary $\mathcal{A}$ runs in a simulated environment in which all its decapsulation queries are correctly handled.

We let F denote the event that $\mathcal{B}$'s simulation of the decapsulation oracle is incorrect. The above analysis establishes that

$$\Pr[F] \leq \frac{q_D}{2^\ell}.$$

For ease of presentation, in what follows we define $\delta := \frac{q_D}{2^\ell}$.

Handling undefined behaviour. Note that $\mathcal{A}$'s behaviour is undefined when the simulation is incorrect, that is, when event F occurs and we cannot estimate $\mathcal{A}$'s success probability in such cases. However, when the event F does *not* occur then the adversary succeeds with the same probability as with the real decapsulation oracle. Denote by $\Pr[\mathcal{A}_{\text{real}}] = 1/2 + \varepsilon$ the probability of the IND-CCA adversary winning the original IND-CCA game and let $\Pr[\mathcal{A}_{\text{sim}}]$ denote the probability of the IND-CCA adversary winning with our simulated decapsulation oracle. Then we have:

$$\begin{aligned}
1/2 + \varepsilon &= \Pr[\mathcal{A}_{\text{real}}] \\
&= \Pr[\mathcal{A}_{\text{real}} \mid F]\ \Pr[F] + \Pr[\mathcal{A}_{\text{real}} \mid \overline{F}]\ \Pr[\overline{F}] \\
&\leq \Pr[F] + \Pr[\mathcal{A}_{\text{real}} \mid \overline{F}] \\
&\leq \Pr[F] + \Pr[\mathcal{A}_{\text{sim}} \mid \overline{F}] \\
&\leq \delta + \Pr[b = b' \mid \overline{F}].
\end{aligned}$$

Hence:

$$1/2 + \varepsilon - \delta \leq \Pr[b = b' \mid \overline{F}].$$

The critical queries. Now let G denote the event that $\mathcal{A}$ during its execution makes a query either to $H_\ell^n(\cdot)$ on $\mathbf{e}$ or to $H_\ell^{\ell+n}(\cdot)$ on $(\mathbf{t} \mid \mathbf{e})$ with $\mathbf{t} = H_\ell^n(\mathbf{e})$ for the $\mathbf{e}$ used to construct C^*. We refer to these as the *critical queries*. By further conditioning our preceding probability $\Pr[b = b' \mid \overline{F}]$ also on event G, we obtain:

$$\begin{aligned} 1/2 + \varepsilon - \delta &\leq \Pr[b = b' \mid \overline{F} \wedge G] \Pr[\overline{F} \wedge G] + \Pr[b = b' \mid \overline{F} \wedge \overline{G}] \Pr[\overline{F} \wedge \overline{G}] \\ &\leq \Pr[\overline{F} \wedge G] + \Pr[b = b' \mid \overline{F} \wedge \overline{G}]. \end{aligned}$$

Next, consider the term $\Pr[b = b' \mid \overline{F} \wedge \overline{G}]$. We argue that this probability is equal to $1/2$. First note that the probability is conditioned in part on the event $\overline{F}$, meaning that $\mathcal{B}$ correctly simulates decapsulation queries for $\mathcal{A}$. Then note that, if G does not occur, then the values of $H_\ell^n(\mathbf{e})$ and of $H_\ell^{\ell+n}(H_\ell^n(\mathbf{e}) \mid \mathbf{e})$ are uniformly random strings (in the Random Oracle Model) from the adversary's point of view. However, it is the value $H_\ell^n(\mathbf{e})$ that determines if C^* is a valid encapsulation of *any* key and it is the value $H_\ell^{\ell+n}(H_\ell^n(\mathbf{e}) \mid \mathbf{e})$ that determines whether C^* is a valid encapsulation of $\mathbf{y}$ (via testing the equation $H_\ell^{\ell+n}(H_\ell^n(\mathbf{e}) \mid \mathbf{e}) = \mathbf{y}$). Thus, if $\mathcal{A}$ does not make the critical queries then it cannot detect that $(\mathbf{Q}, \mathbf{y}, C^*)$ may not be a valid CCA challenge for NTS-KEM, nor can it learn anything about the hidden bit b (indicating whether C^* does encapsulate $\mathbf{y}$ or not).

Then, we consider the term $\Pr[\overline{F} \wedge G]$. Here $\mathcal{B}$ still correctly simulates decapsulation queries but one of the critical queries indicated by event G does occur, in which case $\mathcal{A}$ may be able to detect that C^* is an invalid encapsulation for *any* $\mathbf{y}$, in violation of the requirements concerning the construction of the challenge encapsulation. At the point when event G occurs, then the behaviour of $\mathcal{A}$ becomes undefined and we cannot make any arguments about event probabilities beyond this point. However, up until the point when G occurs, $\mathcal{B}$ provides a correct simulation to $\mathcal{A}$, meaning that the probabilities of all events remain the same as they would in such a simulation. This includes the probability of the event G itself.

Combining the above analyses, we end at:

$$1/2 + \varepsilon - \delta \leq 1/2 + \Pr[\overline{F} \wedge G].$$

Rearranging and using $\Pr[\overline{F} \wedge G] \leq \Pr[G]$ we finally obtain:

$$\varepsilon - \delta \leq \Pr[G].$$

Finalising the construction of $\mathcal{B}$. Now, $\mathcal{B}$ proceeds as follows:

- It runs $\mathcal{A}$, handling all queries as described above.
- After t steps, $\mathcal{B}$ terminates $\mathcal{A}$.[16]
- Finally, for all $(H_\ell^n(\mathbf{e}'), \mathbf{e}', \mathbf{x}', C', \cdot) \in T$, $\mathcal{B}$ tests if $\mathbf{e}'$ decodes C^*, that is, if $C^* + \mathbf{e}'$ is a codeword in the code defined by the generator matrix $[\mathbf{I}_k \mid \mathbf{Q}]$. If such an $\mathbf{e}'$ is found, $\mathcal{B}$ returns it, otherwise it returns $\perp$.

First, we note that if event G occurs, that is, if $\mathcal{A}$ queried $H_\ell^n(\mathbf{e})$ or $H_\ell^{\ell+n}(H_\ell^n(\mathbf{e}) \mid \mathbf{e})$, then the above procedure executed by $\mathcal{B}$ will return the correct $\mathbf{e}$. This follows from Proposition 1, i.e. that C^* cannot be also a valid ciphertext for some other error vector $\mathbf{e}'$. Hence $\mathcal{B}$ succeeds in returning the correct value $\mathbf{e}$ with probability $\Pr[G] \geq \varepsilon - \delta$.

$\mathcal{A}$ runs in time t that is at least q_H, where q_H is the total number of queries made by $\mathcal{A}$ to the random oracle and we have $|T| \leq q_H$ by construction. Thus, since $\mathcal{B}$ runs $\mathcal{A}$ and terminates it after t steps, and then performs a computation for each entry in T, with $|T| \leq q_H \leq t$ we see that $\mathcal{B}$ runs in time at most $2t$. Substituting δ with the value $\frac{q_D}{2^\ell}$ concludes the analysis. □

Combining Lemma 3 with Theorem 2, we arrive at Theorem 1.

F KATs and Intermediate Values

This section describes how to obtain the known answer tests (KATs) and the intermediate values from the reference code. Information on the types of intermediate values and their relation to NTS-KEM specifications (Section 3) is also provided.

The KATs are produced with the code provided by NIST[17], in particular `PQCgenKAT_kem.c` and the associated AES-CTR-DRBG random number generator. The KATs and intermediate values are produced as follows. Note that we specify 100 sets of KATs and intermediate values per run.

1. On the reference implementation code, execute `Makefile`. This produces executables `ntskem-`m`-`τ`-kat` and `ntskem-`m`-`τ`-intval` in `bin` directory, where the pair m and τ denotes a specific NTS-KEM parameter set.
2. Execute the combined KAT and intermediate value generator binary, for example for $m = 12$ and $\tau = 64$:

```
./bin/ntskem-12-64-intval > ntskem-12-64-kat.intvalues
```

[16] The behaviour of $\mathcal{A}$ is undefined if the events F or G occur. For example, it might enter an infinite loop in such cases. Thus, we must take care to terminate it after t steps and cannot rely on its guarantees to terminate itself, which only holds in the real IND-CCA security game. Note also that, up until the event $F \vee G$ occurs, $\mathcal{A}$'s view is identical to that in a correct execution of the IND-CCA game.

[17] They are available at Source Code Files for KATs.

This produces two KATs files, namely `PQCkemKAT_YYYYY.req` and `PQCkemKAT_YYYYY.rsp` where `YYYYY` is some integer denoting the private-key size in bytes; and the intermediate values are stored in `ntskem-12-64-kat.intvalues`. Note that the generation of KATs and the intermediate values will take a few minutes to complete.

We output the following intermediate values. Note that a line prefix by `#` denotes a comment.

Key-Generation

1. An array of $(\tau + 1)$ elements containing the coefficients of the random Goppa polynomial $G(z)$, i.e. $(g_0, g_1, \ldots, g_\tau)$ and $g_i \in \mathbb{F}_{2^m}$ (Step 1).

   ```
   Gz = 1EE 677 162 5EC 23B AA7 076 A65 A3B 519 ...
   ```

2. An array of n elements of the random permutation vector $\mathbf{p}$ (Step 2).

   ```
   p = 48E 6DB 364 16A 10E 826 785 505 9B4 B7F ...
   ```

3. An array of n elements of $\mathbf{a} = \pi_{\mathbf{p}}(\mathbf{a}') = (a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}})$, $a_{p_i} \in \mathbb{F}_{2^m}$ (Step 3a).

   ```
   a = 712 DB6 26C 568 708 641 A1E A0A 2D9 FED ...
   ```

4. An array of n elements of $\mathbf{h} = (h_{p_0}, h_{p_1}, \ldots, h_{p_{n-1}})$, the first row of $\mathbf{H}_m$ and $h_{p_i} \in \mathbb{F}_{2^m}$ (Step 3b).

   ```
   h = 771 2C7 7E0 257 D87 F02 E96 2AF FD5 5B2 ...
   ```

5. A string of $(n - k)\frac{n}{8}$ bytes of matrix $\mathbf{H}$ in row-major representation (Step 3c). Note that $\mathbf{H}$ is not in reduced-row echelon form yet.

   ```
   H = 70E5E1604CFAAF0CE0C0CB28FA2308A466ABE46 ...
   ```

6. A string of $(n - k)\frac{n}{8}$ bytes of reduced-row echelon matrix $\mathbf{H}$ in row-major representation (Step 3d).

   ```
   H = D82B0277361D04BD6752F6F7F2C1B3BFA3CB59D ...
   ```

7. An array of n elements of the random permutation vector $\mathbf{p}$ after permutation ρ (Step 3d).

   ```
   p = 48E 6DB 364 16A 10E 826 785 505 9B4 B7F ...
   ```

8. An array of n elements of $\mathbf{a}$ after permutation ρ and each element is a member of $\mathbb{F}_{2^m}$ (Step 3d).

   ```
   a = 712 DB6 26C 568 708 641 A1E A0A 2D9 FED ...
   ```

9. An array of n elements of $\mathbf{h}$ after permutation ρ and each element is a member of $\mathbb{F}_{2^m}$ (Step 3d).

   ```
   h = 771 2C7 7E0 257 D87 F02 E96 2AF FD5 5B2 ...
   ```

10. A string of $k\frac{n-k}{8}$ bytes of matrix $\mathbf{Q}$ in row-major representation (Step 3e).

    ```
    Q = 5E04CE2F46579905A765571B698A750EC3544E6 ...
    ```

11. An array of $s + (n - k)$ elements of $\mathbf{a}^*$ and each element is a member of $\mathbb{F}_{2^m}$ (Step 4).

    ```
    a_ast = 8F9 117 A20 5B7 5E2 BE8 2A8 16D 0EB E56 ...
    ```

12. An array of $s + (n - k)$ elements of $\mathbf{h}^*$ and each element is a member of $\mathbb{F}_{2^m}$ (Step 4).

    ```
    h_ast = 5DD 46F 39C 39D B7A 909 9F9 558 BE0 8CE ...
    ```

Encapsulation

1. A string of $\frac{n}{8}$ bytes of a random error vector $\mathbf{e}$ (Step 1).

   ```
   e = 0000000000000000000000000000000000000000800000 ...
   ```

2. A string of $\frac{s}{8}$ bytes of $\mathbf{k}_e = H_s(\mathbf{e})$ (Step 3).

   ```
   k_e = FA6E661808C004E296A24EE68ADA8669808DC24CE62AC6C4A5F4C3B320BBD2C7
   ```

3. A string of $\frac{k}{8}$ bytes of $\mathbf{m}$ (Step 4).

   ```
   m = 000000000000000000000000000000000000000080000 ... 62AC6C4A5F4C3B320BBD2C7
   ```

4. A string of $\frac{s}{8}$ bytes of $\mathbf{c}_b = \mathbf{k}_e + \mathbf{e}_b$ (Step 5).

   ```
   c_b = FA6C661808C004E296A24EC6CADA8669808DC34CE62AC6C421F4C3B320BBD2C7
   ```

5. A string of $\frac{n-k}{8}$ bytes of $\mathbf{c}_c = (\mathbf{e}_a \mid \mathbf{k}_e) \cdot \mathbf{Q} + \mathbf{e}_c$ (Step 5).

   ```
   c_c = 5574C21080112877A4B662587FC1F1BF952A9F7664AE4275A ...
   ```

6. A string of $\frac{\ell}{8}$ bytes of $\mathbf{k}_r$ (Step 6).

   ```
   k_r = 6FA29D451A4FE3B49833F7BE6C8F6AA6B74FA22570FD7D19FC25E4E1CE7E9DA6
   ```

Decapsulation

1. An array of $(n - k)$ elements of syndrome $\mathbf{s}$ and each element is a member of $\mathbb{F}_{2^m}$ (Step 1c).

   ```
   s = 9A3 FCF 869 721 F03 0E5 95C EA0 DA2 D6E ...
   ```

2. An array of $(\tau + 1)$ elements containing the coefficients of the error locator polynomial $\sigma(x)$, i.e. $(\sigma_0, \sigma_1, \dots, \sigma_\tau)$ and $\sigma_i \in \mathbb{F}_{2^m}$ (Step 1d).

   ```
   sigma = D32 2B8 144 8B0 717 ED5 2C3 FC2 892 CBC ...
   ```

3. An indicator ξ of whether there is an error in the last coordinate (Step 1d).

   ```
   xi = 0
   ```

4. An array of n elements of the evaluations of $\sigma(x)$ over all elements of $\mathbb{F}_{2^m}$ defined by basis A (Step 1e).

   ```
   evaluations = D32 38B A1A EEA C6A 50C 9CB FAC C1B 020 ...
   ```

5. A string of $\frac{n}{8}$ bytes of the inversely permuted error pattern $\mathbf{e}'$ (Step 1f).

```
e_prime = 00000400000000000000000000000800000000000000000000000 ...
```

6. A string of $\frac{n}{8}$ bytes of the error pattern $\mathbf{e} = \mathbf{e}' \cdot \mathbf{P}$ (Step 2).

```
e = 0000000000000000000000000000000000000000800000 ...
```

7. A string of $\frac{s}{8}$ bytes of $\mathbf{k}_e$ recovered, $\mathbf{k}_e = \mathbf{c}_b - \mathbf{e}_b$ (Step 3).

```
k_e = FA6E661808C004E296A24EE68ADA8669808DC24CE62AC6C4A5F4C3B320BBD2C7
```

8. A string of $\frac{s}{8}$ bytes of $H_s(\mathbf{e})$ (Step 4).

```
SHAKE256_e = FA6E661808C004E296A24EE68ADA8669808DC24CE62AC6C4A5F4C3B320BBD2C7
```

9. A string of $\frac{\ell}{8}$ bytes of $\mathbf{k}_r$, $\mathbf{k}_r = H_\ell\left(\mathbf{k}_e \mid \mathbf{e}\right)$ (Step 4).

```
k_r = 6FA29D451A4FE3B49833F7BE6C8F6AA6B74FA22570FD7D19FC25E4E1CE7E9DA6
```