PALOMA: Binary Separable Goppa-based KEM[1]

(KpqC Round 2 Proposal)

Dong-Chan Kim†, Chang-Yeol Jeon, Minji Kim
Dong Hyun Park, Dong Hyeon Kim, Yeonghyo Kim

Future cryptography Design Lab., Kookmin University

†dckim@kookmin.ac.kr

February 23, 2024

[1]This work is submitted to 'Korean Post-Quantum Cryptography Competition' (www.kpqc.or.kr).

Contents

List of Figures

List of Algorithms

List of Tables

About PALOMA for Round 2

In this proposal, we propose PALOMA, a new code-based key encapsulation mechanism, which is designed by combining an NP-hard SDP(Syndrome Decoding Problem)-based trapdoor with a binary separable Goppa code and FO(Fujisaki-Okamoto) transformation. Cryptographic schemes based on an SDP defined with a binary Goppa code have not been found to be vulnerable to critical attacks, and the FO transformation ensures IND-CCA2 security in the ROM(Random Oracle Model). The combination is highly regarded in cryptographic communities for its strong security guarantees. PALOMA has a public key size of approximately 300KB or more due to its SDP-based trapdoor nature. Furthermore, the key generation process, which involves generating the parity-check matrix of the scrambled Goppa code, is relatively slow compared to other post-quantum ciphers. However a primary role of post-quantum cryptography is to serve as an alternative to current cryptosystems that are vulnerable to quantum computing attacks. Therefore, in post-quantum cryptography, ensuring strong security guarantees is more important than efficiency. Consequently, we have designed PALOMA with a focus on conservative security guarantees, while ensuring that there is no significant degradation in application quality.

Description of Modifications from Round 1 to Round 2

Specification

The specifications of round 2 PALOMA differ as follows from round 1.

	1R	2R	Reason
GenPermMat	$\mathbf{P} \leftarrow \prod_{j=0}^{n-1} \mathbf{P}_{j,l_j}$	$\mathbf{P} \leftarrow [u_{l_0} \mid \cdots \mid u_{l_{n-1}}]$	Ensuring that $\mathbf{P}$ is sampling from a uniform distribution
Secret key sk	$(L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}})$	$(L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}}, r)$	A 256-bit string r, which is independent of $(L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}})$, is added for implicit rejection.

Security Proof

The security proof has been enhanced.

Performance

Data. The inclusion of r for implicit rejection in a secret key sk has resulted in an increase of 32-byte in the secret key size compared to the round 1.

Speed. Through optimization of the extended Patterson decoding in terms of algorithms, the decapsulation speed has improved compared to round 1.

Notations and Symbols

The notations used throughout this proposal are listed below.

$\{0,1\}^l$	set of all l-bit strings
$[i:j]$	integer set $\{i, i+1, \ldots, j-1\}$
$a_{[i:j]}$	substring $a_i \Vert a_{i+1} \Vert \cdots \Vert a_{j-1}$ of a bit string $a = a_0 \Vert a_1 \Vert \cdots$
$\mathbb{F}_q$	finite field with q elements
$\mathbb{F}_q^{m \times n}$	set of all $m \times n$ matrices over a field $\mathbb{F}_q$
$\mathbb{F}_q^l$	set of all $l \times 1$ matrices over a field $\mathbb{F}_q$, i.e., $\mathbb{F}_q^l := \mathbb{F}_q^{l \times 1}$ ($v \in \mathbb{F}_q^l$ is considered as a column vector)
0^l	zero vector with length l
v_I	subvector $(v_j)_{j \in I} \in \mathbb{F}_q^{\lvert I \rvert}$ of a vector $v = (v_0, v_1, \ldots, v_{l-1}) \in \mathbb{F}_q^l$
$\mathrm{supp}\,(v)$	index set of non-zero entries of a vector $v = (v_0, \ldots, v_{l-1}) \in \mathbb{F}_q^l$
$w_H(e)$	function that returns Hamming weight of a given vector e
$\mathbf{I}_l$	$l \times l$ identity matrix
$\mathbf{M}_I$	submatrix $[m_{r,c}]_{c \in I}$ of a matrix $\mathbf{M} = [m_{r,c}]$ where r and c are row index and column index, respectively
$\mathbf{M}_{I \times J}$	submatrix $[m_{r,c}]_{r \in I, c \in J}$ of a matrix $\mathbf{M} = [m_{r,c}]$ where r and c are row index and column index, respectively
$[\mathbf{A} \mid \mathbf{B}]$	concatenated matrix of two matrices $\mathbf{A}$ and $\mathbf{B}$
$\mathcal{P}_l$	set of all $l \times l$ permutation matrices
$x \xleftarrow{\$} X$	x uniformly chosen in a set X
$\gcd(f(X), g(X))$	function that returns the monic greatest common divisor polynomial of $f(X)$ and $g(X)$

Chapter 1

Introduction

We propose PALOMA, a new code-based KEM(Key Encapsulation Mechanism), which has the following features:

- Trapdoor based on SDP(Syndrome Decoding Problem) with a binary separable (not irreducible) Goppa code.
- IND-CCA2-secure KEM based on FO(Fujisaki-Okamoto) transformation.
- Parameter sets that ensure security strengths of 128, 192, and 256-bit.

1.1 Trapdoor

1.1.1 Syndrome Decoding Problem

SDP is a problem of finding the preimage error vector e with a specific Hamming weight for a given random binary parity-check matrix $\mathbf{H}$ and a syndrome $s(= \mathbf{H}e)$. In 1978, SDP was proven to be NP-hard because it is equivalent to the 3-dimensional matching problem [3, 15]. The McEliece and Niederreiter cryptosystems are designed with a trapdoor based on SDP [23, 25]. However, because the public key of an SDP-based trapdoor is a random-looking matrix, the public key is larger than that of other ciphers. Therefore, there have been attempts to reduce the size of a public key through cryptographic design using SDP-variant, such as rank metric-based SDP and quasi-cyclic code-based SDP. However, SDP-variants assume the problem's difficulty due to the lack of guaranteed NP-hardness for SDP and the insufficient maturity of security analysis.

A primary role of post-quantum cryptography is to serve as an alternative to current cryptosystems that are vulnerable to quantum computing attacks. Therefore, in post-quantum cryptography, ensuring strong security guarantees is more important than efficiency. We think the analysis method for SDP is sufficiently mature. Consequently, we have designed PALOMA based on SDP with a focus on conservative security guarantees, while ensuring that there is no significant degradation in application quality.

1.1.2 Niederreiter-type Code Scrambling (a.k.a. syndrome scrambling)

An SDP defined by a parity-check matrix $\mathbf{H}$ for a linear code $\mathcal{C}$ featuring an efficient decoding algorithm is an easy problem. Therefore, to conceal the information required for decoding, $\mathcal{C}$ is transformed into an equivalent code $\widehat{\mathcal{C}}$ that appears as a random code, and we define an SDP with the parity-check matrix $\widehat{\mathbf{H}}$ of $\widehat{\mathcal{C}}$. We call $\widehat{\mathcal{C}}$ a scrambled code of $\mathcal{C}$. Consequently, if one knows both

the information required for decoding and the code transformation information, it is possible to convert the SDP defined by $\widehat{\mathbf{H}}$ into the SDP defined by $\mathbf{H}$. In this manner, the preimage error vector can be determined.

In general, code-based cryptographic schemes use the information of a scrambled code $\widehat{\mathcal{C}}$, which is an equivalent code of the underlying code $\mathcal{C}$, as a public key pk, while the decoding information for $\mathcal{C}$ serves as a secret key sk. The McEliece scheme scrambles codewords, while the Niederreiter scheme scrambles syndromes. Syndrome scrambling has the advantage of being shorter than codeword scrambling and more intuitive for decoding, as syndromes serve as ciphertext. However, it has the drawback of requiring higher computational complexity in converting input plaintext into specific Hamming weight vectors. By the way, in the case of KEM, which does not involve encryption and thus no message input, this conversion process is unnecessary. Therefore, PALOMA adopts the syndrome scrambling approach.

Similar to the Niederreiter scheme, PALOMA uses the parity-check matrix $\widehat{\mathbf{H}}$ of a scrambled code $\widehat{\mathcal{C}}$ defined by $\mathbf{SHP}$. Here, $\mathbf{H}$ represents the parity-check matrix of $\mathcal{C}$, while $\mathbf{S}$ and $\mathbf{P}$ denote an invertible matrix and a permutation matrix, respectively. The $\mathbf{P}$ used in PALOMA is determined by a uniformly chosen 256-bit seed. However, to reduce the size of a public key, the invertible matrix $\mathbf{S}$ is derived from the reduced row echelon form procedure applied to $\mathbf{HP}$, resulting in $\widehat{\mathbf{H}}$ being in a systematic form, denoted as $\widehat{\mathbf{H}} = [\mathbf{I}_{n-k} \mid \mathbf{M}]$. PALOMA uses the submatrix $\mathbf{M}$ of $\widehat{\mathbf{H}}$ as a public key, similar to Classic McEliece [4]. Fig. 1.1 depicts the trapdoor framework of PALOMA.

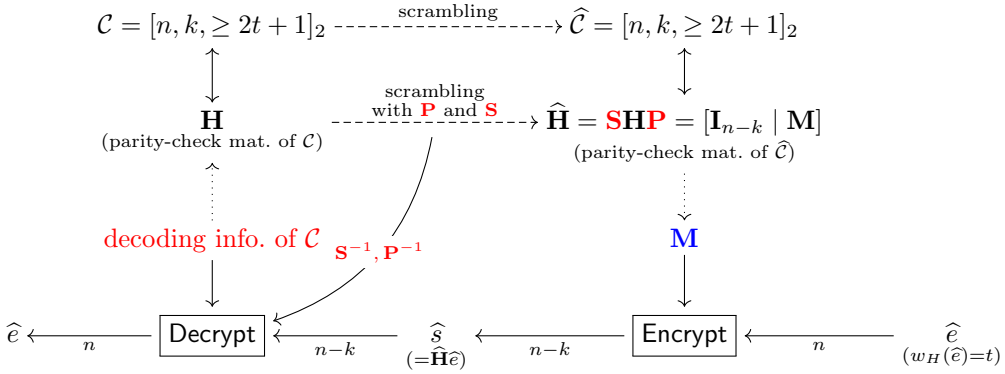


Figure 1.1: PALOMA: Trapdoor Framework

1.1.3 Binary Separable (not irreducible) Goppa Code

There are no critical attacks on cryptographic schemes based on an SDP defined with a binary separable Goppa code [13], for example, McEliece scheme, which is the first code-based cipher [23]. Many researchers have attempted to design code-based ciphers using various codes such as GRS(General Reed-Solomon) and RM(Reed-Muller) to enhance efficiency in terms of public key size and decryption speed. However, most of these schemes have been vulnerable to attacks due to their structural properties, and the remaining ones still require more rigorous security proofs [24, 28]. Therefore, PALOMA adopts a binary separable Goppa code that has no attack even though it has been studied for a long time with a conservative perspective.

A binary separable Goppa code $\mathcal{C} = [n, k, \geq 2t+1]_2$ is defined by a support set L consisting of n distinct elements in $\mathbb{F}_{2^m}$ and a separable Goppa polynomial $g(X) \in \mathbb{F}_{2^m}[X]$ with degree t, for some integer $m > 1$. Typically, an irreducible polynomial is chosen as the Goppa polynomial, as every irreducible polynomial is separable. However, since the algorithms generating irreducible

polynomials are probabilistic, i.e., not guaranteed to have constant-time complexity. For a generation in constant-time, PALOMA defines L and $g(X)$ with uniformly chosen $n+t$ elements in $\mathbb{F}_{2^m}$ as follows: For a random 256-bit string r,

$$[\underbrace{\alpha_0, \alpha_1, \ldots, \alpha_{n-1}}_{n \text{ elements for } L}, \underbrace{\alpha_n, \ldots, \alpha_{n+t-1}}_{t \text{ elements for } g(X)}, \alpha_{n+t}, \ldots, \alpha_{2^m-1}] \leftarrow \mathsf{Shuffle}(\mathbb{F}_{2^m}, r)$$

$$\Rightarrow \quad L \leftarrow [\alpha_0, \alpha_1, \ldots, \alpha_{n-1}], \quad g(X) \leftarrow \prod_{j=n}^{n+t-1} (X - \alpha_j).$$

After shuffling all $\mathbb{F}_{2^m}$ elements, the set of the first n elements is defined as a support set L and the next t elements are the zeros of a Goppa polynomial $g(X)$ with degree t. Note that $g(X)$ is separable but not irreducible in $\mathbb{F}_{2^m}[X]$, and we call the Goppa codes generated by the separable polynomial $g(X)$ totally decomposed Goppa codes [8]. The shuffling function, Shuffle, is a deterministic modification of the Fisher-Yates shuffling algorithm. It shuffles the set using a 256-bit string r. As a result, PALOMA efficiently generates a binary separable Goppa code in constant time.

Patterson and Berlekamp-Massey are decoding algorithms commonly used for binary separable Goppa codes [2, 20, 26]. Patterson shows better speed performance compared to Berlekamp-Massey. However, it only operates when the Goppa polynomial $g(X)$ is irreducible. Therefore, PALOMA adapts the extended Patterson to handle cases where the Goppa polynomial is not irreducible [7].

1.2 KEM Structure

In general, IND-CCA2-secure schemes are constructed with OW-CPA-secure trapdoors and hash functions that are treated as random oracles. The FO transformation is a method for designing IND-CCA2-secure schemes, and it has been proven to be IND-CCA2-secure in ROM [12,14,30]. To achieve IND-CCA2-secure KEM, PALOMA is designed based on the implicit rejection $\mathsf{KEM}^{\not\perp} = \mathsf{U}^{\not\perp}[\mathsf{PKE}_1 = \mathsf{T}[\mathsf{PKE}_0, G], H]$, among FO-like transformations proposed by Hofheinz et al. [14]. This is combined with two modules: (1) T, which converts an OW-CPA-secure PKE_0 into an OW-PCA(Plaintext Checking Attack)-secure PKE_1, and (2) $\mathsf{U}^{\not\perp}$, which converts it into an IND-CCA2-secure KEM.

1.3 Parameter Sets

The security of PALOMA is evaluated by the number of bit computations of generic attacks to SDP as there are currently no known attacks on binary separable Goppa codes. ISD(Information Set Decoding) is the most powerful generic attack of an SDP. The complexity of ISD has been improved by modifications to the specific conditions for the information set [1, 18, 19, 21, 22, 27, 29] and birthday-type search algorithms. PALOMA determines the level of security strength by evaluating the computational complexity of the most effective attack.

Criteria for Parameter Set Selection. PALOMA provides three parameter sets: PALOMA-128, PALOMA-192, and PALOMA-256, which correspond to security strength levels of 128-bit, 192-bit, and 256-bit, respectively. Each parameter set was carefully chosen to meet the following conditions, ensuring efficient implementation.

(1) Binary separable Goppa codes are defined in $\mathbb{F}_{2^{13}}$ which can be used for PALOMA-128, PALOMA-192, and PALOMA-256 simultaneously,

(2) $n + t \leq 2^{13}$ to define a support set and a Goppa polynomial,

(3) $n \equiv k \equiv t \equiv 0 \pmod{64}$ for 64-bit word-aligned implementation, and

(4) $k/n > 0.7$ to reduce the size of a public key, and

(5) Enough security margin.

Chapter 2

Binary Separable Goppa Codes

In this chapter, we provide the definition of Goppa codes and the necessary mathematical background to understand the operating principles of PALOMA.

2.1 Binary Linear Codes

A k-dimensional binary linear code $\mathcal{C}$ of length n defined in a binary finite field $\mathbb{F}_2$ is a k-dimensional subspace of the n-dimensional vector space $\mathbb{F}_2^n$. It means that $\mathcal{C}$ is the solution space of the following $n-k$ linear equations.

$$\begin{array}{rcccccccc} h_{0,0}X_0 & + & h_{0,1}X_1 & + & \cdots & + & h_{0,n-1}X_{n-1} & = & 0, \\ h_{1,0}X_0 & + & h_{1,1}X_1 & + & \cdots & + & h_{1,n-1}X_{n-1} & = & 0, \\ \vdots & & & & & & & & \\ h_{n-k-1,0}X_0 & + & h_{n-k-1,1}X_1 & + & \cdots & + & h_{n-k-1,n-1}X_{n-1} & = & 0. \end{array}$$

Therefore, a binary linear code $\mathcal{C}$ can be represented as follows.

$$\mathcal{C} = \{c \in \mathbb{F}_2^n : \mathbf{H}c = 0^{n-k}\},$$

where 0^{n-k} is a zero vector in $\mathbb{F}_2^{n-k}$ and

$$\mathbf{H} = [h_{i,j}] := \begin{pmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n-k-1,0} & h_{n-k-1,1} & \cdots & h_{n-k-1,n-1} \end{pmatrix} \in \mathbb{F}_2^{(n-k)\times n}.$$

Note that all vectors are considered as column vectors in this proposal. The vector $c \in \mathcal{C}$ and the matrix $\mathbf{H}$ are called a *codeword* and a *parity-check matrix* of $\mathcal{C}$, respectively. For an error vector $e \in \mathbb{F}_2^n$, $\mathbf{H}e \in \mathbb{F}_2^{n-k}$ is called a *syndrome* of e.

The *minimum distance* d of $\mathcal{C}$ is defined by

$$d := \min_{c \in \mathcal{C}\setminus\{0^n\}} w_H(c).$$

A k-dimensional linear code $\mathcal{C}$ of length n defined in a finite field $\mathbb{F}_q$ is denoted by $\mathcal{C} = [n,k]_q$. If the minimum distance d is given, $\mathcal{C}$ is denoted by $\mathcal{C} = [n,k,d]_q$

2.2 Syndrome Decoding Problem

SDP is the problem of finding the preimage error vector with a specific Hamming weight of a given syndrome. The formal definition of SDP is as follows.

Definition 2.1 (SDP)**.** Given a parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$ of a random binary linear code $\mathcal{C} = [n, k]_2$, a syndrome $s \in \mathbb{F}_2^{n-k}$ and an integer $t \in \{1, \ldots, n\}$, find the error vector $e \in \mathbb{F}_2^n$ that satisfies $\mathbf{H}e = s$ and $w_H(e) = t$.

SDP has been proven to be an NP-hard problem due to its equivalence to the 3-dimensional matching problem, as demonstrated in 1978 [3, 15].

Number of Roots of SDP. Suppose that there exist two distinct error vectors $e_0, e_1 \in \mathbb{F}_2^n$ satisfying $\mathbf{H}e_0 = \mathbf{H}e_1$ and $w_H(e_0), w_H(e_1) \leq \lfloor \frac{d-1}{2} \rfloor$, where d is the minimum distance of $\mathcal{C}$. Then we have the following contradiction since $e_0 - e_1 \in \mathcal{C} \setminus \{0^n\}$.

$$d \leq |\mathrm{supp}\,(e_0 - e_1)\,| \leq |\mathrm{supp}\,(e_0)\,| + |\mathrm{supp}\,(e_1)\,| \leq 2\left\lfloor \frac{d-1}{2} \right\rfloor \leq d - 1.$$

Therefore, the preimage error vector with Hamming weight less than or equal to $\lfloor \frac{d-1}{2} \rfloor$ is unique. Generally, in SDP-based schemes, the Hamming weight condition w of SDP is set to $\lfloor \frac{d-1}{2} \rfloor$ for the uniqueness of root.

2.3 Binary Separable Goppa Code

Binary separable Goppa codes are special cases of algebraic-geometric codes proposed by V. D. Goppa in 1970 [13]. The formal definition of a binary separable Goppa code over $\mathbb{F}_2$ is as follows.

Definition 2.2 (Binary Separable Goppa code)**.** For a set of distinct $n(\leq 2^m)$ elements $L = [\alpha_0, \alpha_1, \ldots, \alpha_{n-1}]$ of $\mathbb{F}_{2^m}$ and a separable polynomial[1] $g(X) = \sum_{j=0}^{t} g_j X^j \in \mathbb{F}_{2^m}[X]$ of degree t such that none of the elements of L are zeros of $g(X)$, i.e., $g(\alpha) \neq 0$ for all $\alpha \in L$, a binary separable Goppa code of length n over $\mathbb{F}_2$ is the subspace $\mathcal{C}_{L,g}$ of $\mathbb{F}_2^n$ defined by

$$\mathcal{C}_{L,g} := \{(c_0, \ldots, c_{n-1}) \in \mathbb{F}_2^n : \sum_{j=0}^{n-1} c_j (X - \alpha_j)^{-1} \equiv 0 \pmod{g(X)}\},$$

where $(X - \alpha)^{-1} \in \mathbb{F}_{2^m}[X]$ is the polynomial of degree $t - 1$ satisfying $(X - \alpha)^{-1}(X - \alpha) \equiv 1 \pmod{g(X)}$. L and $g(X)$ are referred to as a *support set* and a *Goppa polynomial*, respectively. If $g(X)$ is irreducible in $\mathbb{F}_{2^m}$, then $\mathcal{C}$ is called a *binary irreducible Goppa code*.

[1] A polynomial $g(X) \in \mathbb{F}_q[X]$ is separable if its roots are distinct in an algebraic closure $\overline{\mathbb{F}_q}$.

Parity-check Matrix. The parity-check matrix $\mathbf{H}$ of $\mathcal{C}_{L,g}$ is defined with each coefficient of the polynomial $(X-\alpha_j)^{-1}$, and $\mathbf{H}$ can be decomposed into $\mathbf{ABC}$, defined by

$$\mathbf{A} := \begin{pmatrix} g_1 & g_2 & \cdots & g_t \\ g_2 & g_3 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_t & 0 & \cdots & 0 \end{pmatrix} \in \mathbb{F}_{2^m}^{t\times t}, \quad \mathbf{B} := \begin{pmatrix} \alpha_0^0 & \alpha_1^0 & \cdots & \alpha_{n-1}^0 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{t-2} & \alpha_1^{t-2} & \cdots & \alpha_{n-1}^{t-2} \\ \alpha_0^{t-1} & \alpha_1^{t-1} & \cdots & \alpha_{n-1}^{t-1} \end{pmatrix} \in \mathbb{F}_{2^m}^{t\times n},$$
$$\text{and } \mathbf{C} := \begin{pmatrix} g(\alpha_0)^{-1} & 0 & \cdots & 0 \\ 0 & g(\alpha_1)^{-1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g(\alpha_{n-1})^{-1} \end{pmatrix} \in \mathbb{F}_{2^m}^{n\times n}. \tag{2.1}$$

Since the matrix $\mathbf{A}$ is invertible ($g_t \neq 0$), $\mathbf{BC}$ is another parity-check matrix of $\mathcal{C}_{L,g}$. Therefore, as Goppa codes are subfield-subcodes of generalized Reed-Solomon codes (i.e., Alternant codes), Berlekamp-Massey decoding can be applied. The Classic McEliece employs a binary Goppa code as its parity-check matrix $\mathbf{BC}$ and utilizes Berlekamp-Massey decoding. However, PALOMA employs a binary Goppa code as its parity-check matrix $\mathbf{ABC}$ and utilizes extended Patterson decoding.

Dimension and Minimum Hamming Distance. The dimension k and the minimum Hamming distance d of $\mathcal{C}_{L,g}$ satisfy $k \geq n-mt$ and $d \geq 2t+1$. PALOMA set the dimension k of $\mathcal{C}_{L,g}$ to $n-mt$ and the Hamming weight condition of the SDP to t to ensure the uniqueness of the root.

2.4 Extended Patterson Decoding

2.4.1 Patterson Decoding

Patterson decoding is the algorithm for a binary irreducible Goppa code $\mathcal{C} = [n, n-mt, \geq 2t+1]_2$, not a separable Goppa code. However, it can be extended for a binary separable Goppa code [7,26].

Given a syndrome vector $s \in \mathbb{F}_2^{mt}$, Patterson decoding procedure to find the preimage error vector $e \in \mathbb{F}_2^n$ of s with $w_H(e) = t$ is as follows.

(Step 1) Parse the syndrome vector $s \in \mathbb{F}_2^{mt}$ as the vector $s = (s_0, \ldots, s_{t-1}) \in \mathbb{F}_{2^m}^t$ and convert s into the syndrome polynomial $s(X) = \sum_{j=0}^{t-1} s_j X^j \in \mathbb{F}_{2^m}[X]$ of degree $t-1$ or less.

(Step 2) Derive the key equation for finding the error locator polynomial $\sigma(X) = \prod_{j\in \mathrm{supp}(e)}(X-\alpha_j) \in \mathbb{F}_{2^m}[X]$ of degree t.

(Step 3) Solve the key equation using the extended Euclidean algorithm.

(Step 4) Calculate $\sigma(X)$ using a root of the key equation.

(Step 5) Find all zeros of $\sigma(X)$ and compute the preimage error vector e.

In the above decoding procedure, the error locator polynomial $\sigma(X)$ satisfies the following identity.

$$\sigma(X)s(X) \equiv \sigma'(X) \pmod{g(X)}. \tag{2.2}$$

Note that $\sigma(X)$ satisfying Eq. (2.2) is unique since the number of errors is t. In $\mathbb{F}_{2^m}[X]$, all polynomials $f(X)$ has two polynomials $a(X)$ and $b(X)$ such that $f(X) = a(X)^2 + b(X)^2 X$,

$\deg(a) \leq \lfloor \frac{t}{2} \rfloor$, and $\deg(b) \leq \lfloor \frac{t-1}{2} \rfloor$. Thus, if $\sigma(X) = a(X)^2 + b(X)^2 X$, Eq. (2.2) can be rewritten as follows.

$$b(X)^2(1 + Xs(X)) \equiv a(X)^2 s(X) \pmod{g(X)}. \tag{2.3}$$

When $g(X)$ is irreducible, $s^{-1}(X)$ and $\sqrt{s^{-1}(X) + X}$ exist in modulo $g(X)$. Patterson decoding uses the extended Euclidean algorithm to find $a(X)$ and $b(X)$ of the following *key equation* to generate the error locator polynomial $\sigma(X)$.

$$b(X)\sqrt{(s^{-1}(X) + X)} \equiv a(X) \pmod{g(X)},\ \deg(a) \leq \left\lfloor \frac{t}{2} \right\rfloor,\ \deg(b) \leq \left\lfloor \frac{t-1}{2} \right\rfloor. \tag{2.4}$$

However, if $g(X)$ is separable, the existence of $s^{-1}(X)$ cannot be guaranteed because $g(X)$ and $s(X)$ are unlikely to be relatively prime.

2.4.2 Extended Patterson Decoding

To address this situation, extended Patterson decoding redefines the key equation Eq. (2.4).

New Key Equation. We define $\widetilde{s}(X), g_1(X), g_2(X) \in \mathbb{F}_{2^m}[X]$ as follows.

$$\widetilde{s}(X) := 1 + Xs(X), \quad g_1(X) := \gcd(g(X), s(X)), \quad g_2(X) := \gcd(g(X), \widetilde{s}(X)).$$

Since $\gcd(s(X), \widetilde{s}(X)) = \gcd(s(X), \widetilde{s}(X) \bmod s(X)) = \gcd(s(X), 1) \in \mathbb{F}_{2^m} \setminus \{0\}$, we know

$$g \mid b^2\widetilde{s} + a^2 s \xrightarrow{g_1 \mid g} g_1 \mid b^2\widetilde{s} + a^2 s \xrightarrow{g_1 \mid s} g_1 \mid b^2\widetilde{s} \xrightarrow{g_1 \nmid \widetilde{s}} g_1 \mid b^2 \Rightarrow g_1 \mid b,$$
$$g \mid b^2\widetilde{s} + a^2 s \xrightarrow{g_2 \mid g} g_2 \mid b^2\widetilde{s} + a^2 s \xrightarrow{g_2 \mid \widetilde{s}} g_2 \mid a^2 s \xrightarrow{g_2 \nmid s} g_2 \mid a^2 \Rightarrow g_2 \mid a.$$

Therefore, the following five polynomials can be defined in $\mathbb{F}_{2^m}[X]$.

$$b_1(X) := \frac{b(X)}{g_1(X)}, \quad a_2(X) := \frac{a(X)}{g_2(X)}, \quad g_{12}(X) := \frac{g(X)}{g_1(X)g_2(X)},$$
$$\widetilde{s}_2(X) := \frac{\widetilde{s}(X)}{g_2(X)}, \quad s_1(X) := \frac{s(X)}{g_1(X)}.$$

Eq. (2.3) can be rewritten as follows.

$$\begin{aligned} & b(X)^2\widetilde{s}(X) \equiv a(X)^2 s(X) \pmod{g(X)} \\ & \quad \Rightarrow\ b_1(X)^2 g_1(X)\widetilde{s}_2(X) \equiv a_2(X)^2 g_2(X) s_1(X) \pmod{g_{12}(X)}. \end{aligned}$$

Because $\gcd(g_2(X), g_{12}(X)), \gcd(s_1(X), g_{12}(X))$ is an element of $\mathbb{F}_{2^m}$, we know $\gcd(g_2(X)s_1(X), g_{12}(X)) \in \mathbb{F}_{2^m}$. Therefore, the inverse of $g_2(X)s_1(X)$ modulo $g_{12}(X)$ exists, and we have the following equation.

$$b_1(X)^2 u(X) \equiv a_2(X)^2 \pmod{g_{12}(X)} \ \text{ where } u(X) := g_1(X)\widetilde{s}_2(X)(g_2(X)s_1(X))^{-1}.$$

Since $g_{12}(X)$ is separable, $u(X)$ has a square root modulo $g_{12}(X)$. Therefore, $a(X) = a_2(X)g_2(X)$ and $b(X) = b_1(X)g_1(X)$ are obtained by calculating $a_2(X)$ and $b_1(X)$ that satisfy the following

new key equation using the extended Euclidean algorithm.

$$b_1(X)\sqrt{u(X)} \equiv a_2(X) \pmod{g_{12}(X)}, \tag{2.5}$$
$$\deg(a_2) \le \left\lfloor \frac{t}{2} \right\rfloor - \deg(g_2),\ \deg(b_1) \le \left\lfloor \frac{t-1}{2} \right\rfloor - \deg(g_1).$$

Computation of $\sqrt{u(X)} \bmod g_{12}(X)$ in PALOMA. All zeros of the Goppa polynomial $g(X)$ used in PALOMA belong to $\mathbb{F}_{2^m}$. Since all elements of $\mathbb{F}_{2^m}$ are roots of the equation $X^{2^m} - X = 0$ and $g_{12}(X) \mid X^{2^m} - X$, we know $\sqrt{X} = X^{2^{m-1}} \bmod g_{12}(X)$. A polynomial $u(X) = \sum_{i=0}^{l} u_i X^i \in \mathbb{F}_{2^m}[X]$ of degree l can be written as $u(X) = \left(\sum_{i=0}^{\lfloor \frac{l}{2} \rfloor} \sqrt{u_{2i}} X^i\right)^2 + \left(\sum_{i=0}^{\lfloor \frac{l-1}{2} \rfloor} \sqrt{u_{2i+1}} X^i\right)^2 X$ where $\sqrt{u_j} = (u_j)^{2^{m-1}}$ for all j. Thus, the square root $\sqrt{u(X)}$ of $u(X)$ modulo $g_{12}(X)$ is

$$\sqrt{u(X)} = \left(\sum_{i=0}^{\lfloor \frac{l}{2} \rfloor} \sqrt{u_{2i}} X^i\right) + \left(\sum_{i=0}^{\lfloor \frac{l-1}{2} \rfloor} \sqrt{u_{2i+1}} X^i\right) \sqrt{X} \bmod g_{12}(X). \tag{2.6}$$

In summary, the operational process of extended Patterson decoding (Alg. 1) for PALOMA is as follows.

(Step 1) Parse the syndrome vector $s \in \mathbb{F}_2^{mt}$ as the vector $s = (s_0, \ldots, s_{t-1}) \in \mathbb{F}_{2^m}^t$ and convert s into the syndrome polynomial $s(X) = \sum_{j=0}^{t-1} s_j X^j \in \mathbb{F}_{2^m}[X]$ of degree $t-1$ or less. (Alg. 2)

(Step 2) Derive the new key equation Eq. (2.5) for finding the error locator polynomial $\sigma(X) = \prod_{j \in \mathrm{supp}(e)} (X - \alpha_j) \in \mathbb{F}_{2^m}[X]$ of degree t. (Alg. 3)

(Step 3) Solve the new key equation using the extended Euclidean algorithm. (Alg. 4)

(Step 4) Calculate $\sigma(X)$ using a root of the key equation.

(Step 5) Find all zeros of $\sigma(X)$ and compute the preimage error vector e. At this stage, to ensure resistance against timing attacks, we find the solution through an exhaustive search. (Alg. 5)

We give a SAGE code for the extended Patterson decoding in Appendix A.

Algorithm 1 Extended Patterson Decoding: RecoverErrVec

Input: A support set L, a Goppa polynomial $g(X)$ and a syndrome vector $s \in \mathbb{F}_2^{n-k}$
Output: An error vector $e \in \mathbb{F}_2^n$ with $w_H(e) = t$

1: **procedure** RecoverErrVec($L, g(X); s$)
2: $s(X) \leftarrow$ ToPoly(s) ▷ Alg. 2
3: $v(X), g_1(X), g_2(X), g_{12}(X) \leftarrow$ ConstructKeyEqn($s(X), g(X)$) ▷ Alg. 3
4: $(a_2(X), b_1(X)) \leftarrow$ SolveKeyEqn($v(X), g_{12}(X), \lfloor \frac{t}{2} \rfloor - \deg(g_2), \lfloor \frac{t-1}{2} \rfloor - \deg(g_1)$) ▷ Alg. 4
5: $a(X), b(X) \leftarrow a_2(X)g_2(X), b_1(X)g_1(X)$
6: $\sigma(X) \leftarrow a(X)^2 + b(X)^2 X$
7: $e \leftarrow$ FindErrVec($\sigma(X), L$) ▷ Alg. 5
8: **return** e
9: **end procedure**

Algorithm 2 Extended Patterson Decoding: ToPoly

Input: A syndrome vector $s = (s_0, s_1, \ldots, s_{mt-1}) \in \mathbb{F}_2^{mt}$
Output: A syndrom polynomial $s(X) \in \mathbb{F}_{2^m}[X]$

procedure ToPoly(s)
  $w_j \leftarrow \sum_{i=0}^{m-1} s_{mj+i} z^i \in \mathbb{F}_{2^m}$ for $j = 0, 1, \ldots, t-1$
  $s(X) \leftarrow \sum_{j=0}^{t-1} w_j X^j \in \mathbb{F}_{2^m}[X]$
  return $s(X)$
end procedure

Algorithm 3 Extended Patterson Decoding: ConstructKeyEqn

Input: A syndrome polynomial $s(X)$ and a Goppa polynomial $g(X)$
Output: $v(X), g_1(X), g_2(X), g_{12}(X) \in \mathbb{F}_{2^m}[X]$

procedure ConstructKeyEqn($s(X), g(X)$)
  $\widetilde{s}(X) \leftarrow 1 + Xs(X)$
  $g_1(X), g_2(X) \leftarrow \gcd(g(X), s(X)), \gcd(g(X), \widetilde{s}(X))$ ▷ $g_1(X), g_2(X)$ are monic.
  $g_{12}(X) \leftarrow \frac{g(X)}{g_1(X)g_2(X)}$
  $\widetilde{s}_2(X), s_1(X) \leftarrow \frac{\widetilde{s}(X)}{g_2(X)}, \frac{s(X)}{g_1(X)}$
  $u(X) \leftarrow g_1(X)\widetilde{s}_2(X)(g_2(X)s_1(X))^{-1} \bmod g_{12}(X)$
  $v(X) \leftarrow \sqrt{u(X)} \bmod g_{12}(X)$ ▷ Eq. (2.6)
  return $v(X), g_1(X), g_2(X), g_{12}(X)$
end procedure

Algorithm 4 Extended Patterson Decoding: SolveKeyEqn

Input: $v(X)$, $g_{12}(X)$, $dega$, $degb$
Output: $(a_2(X), b_1(X))$ s.t. $b_1(X)v(X) \equiv a_2(X) \pmod{g_{12}(X)}$ and $\deg(a_2) \leq dega, \deg(b_1) \leq degb$

procedure SolveKeyEqn($v(X), g_{12}(X), dega, degb$)
  $\eta_0(X), \eta_1(X) \leftarrow v(X), g_{12}(X)$
  $\rho_0(X), \rho_1(X) \leftarrow 1, 0$
  while $\eta_1(X) \neq 0$ **do**
    $q(X), r(X) \leftarrow div(\eta_0(X), \eta_1(X))$ ▷ $\eta_0(X) = \eta_1(X)q(X) + r(X),\ 0 \leq \deg(r) < \deg(\eta_1)$
    $\eta_0(X), \eta_1(X) \leftarrow \eta_1(X), r(X)$
    $\rho_2(X) \leftarrow \rho_0(X) - q(X)\rho_1(X)$
    $\rho_0(X), \rho_1(X) \leftarrow \rho_1(X), \rho_2(X)$
    if $\deg(\eta_0) \leq dega$ and $\deg(\rho_0) \leq degb$ **then**
      `break`
    end if
  end while
  return $(\eta_0(X), \rho_0(X))$
end procedure

Algorithm 5 Extended Patterson Decoding: FindErrVec

Input: An error locator polynomial $\sigma(X)$ and a support set L
Output: An error vector $e \in \mathbb{F}_2^n$

```
1: procedure FindErrVec(σ, L)
2:     e = (e_0, ..., e_{n-1}) ← (0, 0, ..., 0)
3:     for j = 0 to n − 1 do
4:         if σ(α_j) = 0 then
5:             e_j ← 1
6:         end if
7:     end for
8:     return e
9: end procedure
```

Chapter 3

Specification

3.1 Parameter Sets

PALOMA consists of three parameter sets: PALOMA-128, PALOMA-192, and PALOMA-256 offering 128/192/256-bit security strength, respectively. Tab. 3.1 shows each parameter set.

Table 3.1: Parameter Sets of PALOMA

Parameter set	n	t	$k(= n - mt)$	m
PALOMA-128	3904	64	3072	13
PALOMA-192	5568	128	3904	13
PALOMA-256	6592	128	4928	13

In Tab. 3.1, the parameters n, t, and k denote the length of a codeword, the number of correctable errors, and the dimension of a binary Goppa code, respectively. The parameter $m(= 13)$ represents the degree of a binary field extension. The binary extension field $\mathbb{F}_{2^{13}}(= \mathbb{F}_{2^m})$ used in PALOMA is defined by an irreducible polynomial $f(z) = z^{13} + z^7 + z^6 + z^5 + 1 \in \mathbb{F}_2[z]$, i.e., $\mathbb{F}_{2^{13}} = \mathbb{F}_2[z]/\langle f(z)\rangle$. Each parameter set satisfies $n + t \leq 2^m$ and $k = n - mt$.

Representation of the parameter set. Since all three parameters are set to $m = 13$ and k is determined by n and t, the parameter set is denoted as (n, t).

3.2 Utility Functions

3.2.1 Array Shuffling: Shuffle

Shuffle parses a 256-bit seed $r = r_0\|r_1\|\cdots\|r_{255}$ as sixteen 16-bit non-negative integers $\widehat{r}_0, \ldots, \widehat{r}_{15}$ where $\widehat{r}_w = \sum_{j=0}^{15} r_{16w+15-j}2^j < 2^{16}$ and uses each as a random integer required in the Fisher-Yates shuffle [17]. Alg. 6 shows the process of Shuffle in detail. Section 5.1.1.1 gives the security analysis of Shuffle.

Algorithm 6 Shuffle: Array Shuffling with a 256-bit Seed

Input: An array $A = [A_0, A_1, \ldots, A_{l-1}]$ and a 256-bit seed $r \in \{0,1\}^{256}$
Output: A shuffled array A

procedure $\mathsf{Shuffle}(A, r)$
  for $j = 0$ to 15 **do**
    $\widehat{r}_j \leftarrow \mathsf{BitToInt}(r_{[16j:16(j+1)]})$ ▷ $\mathsf{BitToInt}(r_0 \| \cdots \| r_{15}) := \sum_{j=0}^{15} r_{15-j} 2^j < 2^{16}$
  end for
  $w \leftarrow 0$
  for $i \leftarrow |A| - 1$ downto 1 **do** ▷ $|A| = l$, the number of elements of A
    $j \leftarrow \widehat{r}_w \bmod i + 1$
    $A_i, A_j \leftarrow A_j, A_i$ ▷ Swapping of A_i and A_j
    $w \leftarrow w + 1 \bmod 16$
  end for
  return A
end procedure

3.2.2 Generation of Permutation Matrix: GenPermMat

GenPermMat takes a 256-bit string and performs Shuffle (Alg. 6) with it. It then returns the $n \times n$ permutation matrix $\mathbf{P}$ and its inverse $\mathbf{P}^{-1}$ corresponding to the shuffled array. Alg. 7 illustrates the operation process of GenPermMat.

Algorithm 7 GenPermMat: Generating a n-bit Permutation with a 256-bit Seed

Input: A permutation size n and a 256-bit seed $r \in \{0,1\}^{256}$
Output: An $n \times n$ permutation matrix $\mathbf{P}, \mathbf{P}^{-1} \in \mathbb{F}_2^{n \times n}$

procedure $\mathsf{GenPermMat}(n, r)$
  $[l_0, l_1, \ldots, l_{n-1}] \leftarrow \mathsf{Shuffle}([0, 1, \ldots, n-1], r)$ ▷ Alg. 6
  $\mathbf{P} \leftarrow [u_{l_0} \mid u_{l_1} \mid \cdots \mid u_{l_{n-1}}] \in \mathbb{F}_2^{n \times n}$ ▷ u_j is the standard unit column vector such that $\mathrm{supp}(u_j) = j$
  return $\mathbf{P}, \mathbf{P}^{-1}$
end procedure

3.2.3 Vector Permutation: Perm and PermInv

Perm and PermInv substitute a vector $v \in \mathbb{F}_2^n$ with a 256-bit string r and permutation matrices $\mathbf{P}$ and $\mathbf{P}^{-1}$ generated by GenPermMat. The substitution is carried out by replacing the v with $\mathbf{P}v$ and $\mathbf{P}^{-1}v$, respectively. Alg. 8 illustrates these processes.

Algorithm 8 Perm and PermInv: Vector Permutation

Input: A vector $v \in \mathbb{F}_2^n$ and $r \in \{0,1\}^{256}$
Output: A vector $v \in \mathbb{F}_2^n$

procedure $\mathsf{Perm}(v, r)$
  $\mathbf{P}, \mathbf{P}^{-1} \leftarrow \mathsf{GenPermMat}(n, r)$
  $v \leftarrow \mathbf{P}v$
  return v
end procedure

Input: A vector $v \in \mathbb{F}_2^n$ and $r \in \{0,1\}^{256}$
Output: A vector $v \in \mathbb{F}_2^n$

procedure $\mathsf{PermInv}(v, r)$
  $\mathbf{P}, \mathbf{P}^{-1} \leftarrow \mathsf{GenPermMat}(n, r)$
  $v \leftarrow \mathbf{P}^{-1}v$
  return v
end procedure

3.2.4 Generation of Error Vector: GenErrVec

GenErrVec generates a t-Hamming weight error vector $e \in \mathbb{F}_2^n$ by shuffling an array $[0, 1, \ldots, n-1]$ using a 256-bit seed r and selecting the upper t elements to form the support set. Alg. 9 illustrates the process.

Algorithm 9 GenErrVec: Generating a t-Hamming weight Error Vector with a 256-bit Seed

Input: A vector length n, a Hamming weight t, and a 256-bit seed $r \in \{0,1\}^{256}$
Output: An error vector $e = (e_0, e_1, \ldots, e_{n-1}) \in \mathbb{F}_2^n$ with $w_H(e) = t$

procedure GenErrVec(n, t, r)
  $[l_0, l_1, \ldots, l_{n-1}] \leftarrow$ Shuffle$([0, 1, \ldots, n-1], r)$
  $e = (e_0, e_1, \ldots, e_{n-1}) \leftarrow (0, 0, \ldots, 0)$
  for $j = 0$ to $t - 1$ **do**
    $e_{l_j} \leftarrow 1$
  end for
  return e ▷ supp $(e) = \{l_0, l_1, \ldots, l_{t-1}\}$
end procedure

3.2.5 Random Oracles: RO_G, RO_H

PALOMA is a KEM designed in the random oracle model. PALOMA uses two random oracles, namely RO_G and RO_H, which are defined using the Korean KS standard hash function LSH-512 [16]. Alg. 10 presents the definition.

Algorithm 10 RO_G, RO_H: Random Oracles

Input: A bit string $x \in \{0,1\}^*$
Output: A 256-bit string $r \in \{0,1\}^{256}$

procedure $\mathsf{RO}_G(x)$
  return LSH("PALOMAGG"$\|x)_{[0:256]}$
end procedure

procedure $\mathsf{RO}_H(x)$
  return LSH("PALOMAHH"$\|x)_{[0:256]}$
end procedure

3.3 Key Generation

The trapdoor of PALOMA is designed with SDP based on a scrambled code $\widehat{\mathcal{C}}$ of a binary separable Goppa code $\mathcal{C}$. The public key is the submatrix of the systematic parity-check matrix of $\widehat{\mathcal{C}}$, and the secret key is the necessary information for decoding and scrambling of $\mathcal{C}$. Alg. 11 presents the pseudo code for the key generation GenKeyPair of PALOMA, and the detailed processes of each subroutine are outlined in Section 3.3.1, Section 3.3.2, and Section 3.3.3.

3.3.1 Generation of a random binary separable Goppa code $\mathcal{C}$

GenRandGoppaCode generates a support set L in $\mathbb{F}_{2^{13}}$ and a Goppa polynomial $g(X) \in \mathbb{F}_{2^{13}}[X]$ for a Goppa code $\mathcal{C}$ using a uniformly chosen 256-bit string $r_{\mathcal{C}}$, and computes the parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{13t \times n}$ of $\mathcal{C}$. The operation process is outlined below, and Alg. 12 presents the pseudo code of it.

(Step 1) A 256-bit string $r_{\mathcal{C}}$ is uniformly chosen.

Algorithm 11 PALOMA: Generation of Key Pair

Input: Parameter set (n, t)
Output: A public key pk and a secret key sk

procedure GenKeyPair(n, t)
  $r_{\mathcal{C}}, L, g(X), \mathbf{H} \leftarrow$ GenRandGoppaCode(n, t) ▷ Alg. 12
  $r_{\widehat{\mathcal{C}}}, \mathbf{S}^{-1}, \widehat{\mathbf{H}} \leftarrow$ GenScrambledCode$(\mathbf{H})$ ▷ Alg. 13
  $pk \leftarrow \widehat{\mathbf{H}}_{[n-k:n]}$
  $r \xleftarrow{\$} \{0,1\}^{256}$ ▷ r is used in the implicit rejection case
  $sk \leftarrow (L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}}, r)$ ▷ or $sk \leftarrow (r_{\mathcal{C}}, r_{\widehat{\mathcal{C}}}, r) \in \{0,1\}^{768}$
  return pk and sk
end procedure

(Step 2) Reorder elements of $\mathbb{F}_{2^{13}}$ with the $r_{\mathcal{C}}$ using Shuffle. (Alg. 6)

$$[\alpha_0, \ldots, \alpha_{2^{13}-1}] \leftarrow \mathsf{Shuffle}(\mathbb{F}_{2^{13}} = \underbrace{[0, 1, z, z+1, z^2, \ldots, z^{12} + \cdots + 1]}_{\text{lexicographic order}}, r_{\mathcal{C}}).$$

Note that we consider a field element $\alpha = \sum_{j=0}^{12} a_j z^j \in \mathbb{F}_{2^{13}}$ as an integer $\sum_{j=0}^{12} a_j 2^j \in \mathbb{Z}$ for using Shuffle.

(Step 3) Set the support set $L = [\alpha_0, \ldots, \alpha_{n-1}]$, and set the separable Goppa polynomial $g(X) = \sum_{j=0}^{t} g_j X^j = \prod_{j=n}^{n+t-1}(X - \alpha_j) \in \mathbb{F}_{2^{13}}[X]$ of degree t.

$$L \leftarrow [\alpha_0, \ldots, \alpha_{n-1}], \quad g(X) \leftarrow \prod_{j=n}^{n+t-1} (X - \alpha_j).$$

(Step 4) Compute the parity-check matrix $\mathbf{H} = \mathbf{ABC} \in \mathbb{F}_{2^{13}}^{t \times n}$ where $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are defined in Eq. (2.1).

(Step 5) Parse $\mathbf{H}$ as a matrix in $\mathbb{F}_2^{13t \times n}$ as follows.

$$\mathbf{H} = [h_{r,c}] \in \mathbb{F}_{2^{13}}^{t \times n} \quad \Rightarrow \quad \mathbf{H} = [h_0 \mid h_1 \mid \cdots \mid h_{n-1}] \in \mathbb{F}_2^{13t \times n},$$

where $h_c := (h_{0,c}^{(0)}, \ldots, h_{0,c}^{(12)}, h_{1,c}^{(0)}, \ldots, h_{1,c}^{(12)}, \ldots, h_{t-1,c}^{(12)}) \in \mathbb{F}_2^{13t}$ and $h_{r,c}^{(j)} \in \mathbb{F}_2$ such that $h_{r,c} = \sum_{j=0}^{12} h_{r,c}^{(j)} z^j \in \mathbb{F}_{2^{13}}$ for $r \in [0:t]$ and $c \in [0:n]$.

3.3.2 Generation of a scrambled code $\widehat{\mathcal{C}}$ of $\mathcal{C}$

GenScrambledCode scrambles the parity-check matrix $\mathbf{H}$ below, and Alg. 13 presents the pseudo code of it.

(Step 1) A 256-bit string $r_{\widehat{\mathcal{C}}}$ is uniformly chosen.

(Step 2) Generate a $n \times n$ random permutation matrix $\mathbf{P}$ and its inverse $\mathbf{P}^{-1}$ using GenPermMat and $r_{\widehat{\mathcal{C}}}$. (Alg. 7)

(Step 3) Compute $\mathbf{HP}$.

(Step 4) Compute the reduced row echelon form $\widehat{\mathbf{H}}$ of $\mathbf{HP}$. If $\widehat{\mathbf{H}}_{[0:n-k]} \neq \mathbf{I}_{n-k}$, back to (Step 1). Note that $\Pr[\widehat{\mathbf{H}}_{[0:n-k]} = \mathbf{I}_{n-k}] > 0.288788$.

Algorithm 12 GenRandGoppaCode: Generating a Random Binary Separable Goppa Code

Input: Parameter set (n, t)

Output: A 256-bit string $r_{\mathcal{C}} \in \{0,1\}^{256}$, a support set L, a Goppa poly. $g(X)$ and a parity-check matrix $\mathbf{H}$ of $\mathcal{C}_{L,g}$

1: **procedure** GenRandGoppaCode(n, t)
2: $r_{\mathcal{C}} \xleftarrow{\$} \{0,1\}^{256}$
3: $[\alpha_0, \ldots, \alpha_{2^{13}-1}] \leftarrow \mathsf{Shuffle}(\mathbb{F}_{2^{13}}, r_{\mathcal{C}})$ ▷ Alg. 6
4: $L \leftarrow [\alpha_0, \ldots, \alpha_{n-1}]$ ▷ Support set
5: $g(X) \leftarrow \prod_{j=n}^{n+t-1} (X - \alpha_j)$ ▷ Goppa polynomial
6: $\mathbf{H} = [h_{r,c}] \leftarrow \mathbf{ABC} \in \mathbb{F}_{2^{13}}^{t \times n}$ ▷ $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are defined in Eq. (2.1)
7: Parse $\mathbf{H}$ as a matrix in $\mathbb{F}_2^{13t \times n}$ ▷ Parity-check matrix
8: **return** $r_{\mathcal{C}}$, $L, g(X)$, $\mathbf{H}$
9: **end procedure**

Algorithm 13 GenScrambledCode: Scrambling a Goppa code

Input: A parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{mt \times n}$ of $\mathcal{C}$

Output: A 256-bit string $r_{\widehat{\mathcal{C}}} \in \{0,1\}^{256}$, an invertible matrix $\mathbf{S}^{-1} \in \mathbb{F}_2^{mt \times mt}$, and a parity-check matrix $\widehat{\mathbf{H}} \in \mathbb{F}_2^{mt \times n}$ of $\widehat{\mathcal{C}}$

1: **procedure** GenScrambledCode$(\mathbf{H})$
2: $r_{\widehat{\mathcal{C}}} \xleftarrow{\$} \{0,1\}^{256}$
3: $\mathbf{P}, \mathbf{P}^{-1} \leftarrow \mathsf{GenPermMat}(n, r_{\widehat{\mathcal{C}}})$ ▷ Alg. 7
4: $\widehat{\mathbf{H}} \leftarrow \mathsf{RREF}(\mathbf{HP})$
5: **if** $\widehat{\mathbf{H}}_{[0:n-k]} \neq \mathbf{I}_{n-k}$ **then**
6: Go back to line 2.
7: **end if**
8: $\mathbf{S}^{-1} \leftarrow (\mathbf{HP})_{[0:n-k]}$
9: **return** $r_{\widehat{\mathcal{C}}}, \mathbf{S}^{-1}$, $\widehat{\mathbf{H}}$
10: **end procedure**

(Step 5) Define the invertible matrix $\mathbf{S}^{-1} := (\mathbf{HP})_{[0:n-k]} \in \mathbb{F}_2^{(n-k)\times(n-k)}$. Note that $\widehat{\mathbf{H}} = \mathbf{SHP}$.

3.3.3 Define a public key pk and a secret key sk

Public key. The public operation of PALOMA involves computing the syndrome(image) of a given error vector in the scrambled code $\widehat{\mathcal{C}}$. Therefore PALOMA defines the submatrix $\widehat{\mathbf{H}}_{[n-k:n]}$ of the systematic form parity-check matrix $\widehat{\mathbf{H}}$ as a public key pk.

$$pk \leftarrow \widehat{\mathbf{H}}_{[n-k:n]}.$$

Secret key. The secret operation of PALOMA involves computing the error vector(preimage) of a given syndrome. Therefore PALOMA defines the decoding information L and $g(X)$ for the Goppa code $\mathcal{C}$ and scrambling information $r_{\widehat{\mathcal{C}}}$ and $\mathbf{S}^{-1}$ for the scrambled code $\widehat{\mathcal{C}}$ as a secret key sk. Additionally, PALOMA includes 256-bit string r for implicit rejection, generated independently with $\mathcal{C}$ and $\widehat{\mathcal{C}}$.

$$sk \leftarrow (L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}}, r).$$

Remark. The $\mathbf{S}^{-1}$ of a secret key can be derived from the $(L, g(X), r_{\widehat{\mathcal{C}}})$. Both L and $g(X)$ are generated by the $r_{\mathcal{C}}$, used in GenRandGoppaCode. Therefore the secret key sk can be defined as a 768-bit string $(r_{\mathcal{C}}, r_{\widehat{\mathcal{C}}}, r) \in \{0,1\}^{768}$.

$$sk \leftarrow (r_{\mathcal{C}}, r_{\widehat{\mathcal{C}}}, r).$$

3.4 Encryption and Decryption

Encryption. The encryption Encrypt of PALOMA involves computing the syndrome $\widehat{s}$(ciphertext) of a given t-Hamming weight error vector $\widehat{e}$(plaintext) in the scrambled code $\widehat{\mathcal{C}}$ using a public key $pk = \widehat{\mathbf{H}}_{[n-k:n]}$. The process is outlined below.

(Step 1) Retrieve the parity-check matrix $\widehat{\mathbf{H}} = [\mathbf{I}_{n-k} \mid \widehat{\mathbf{H}}_{[n-k:n]}] \in \mathbb{F}_2^{(n-k)\times n}$ of the scrambled code $\widehat{\mathcal{C}}$ from the public key $pk = \widehat{\mathbf{H}}_{[n-k:n]} \in \mathbb{F}_2^{(n-k)\times k}$.

(Step 2) Compute the $(n-k)$-bit syndrome $\widehat{s}(= \widehat{\mathbf{H}}\widehat{e})$ of an n-bit error vector input $\widehat{e}$ with $w_H(\widehat{e}) = t$, and return $\widehat{s}$ as the ciphertext of $\widehat{e}$.

$$\widehat{s}(= \widehat{\mathbf{H}}\widehat{e}) \leftarrow \mathsf{Encrypt}(pk; \widehat{e}).$$

Decryption. The decryption Decrypt of PALOMA involves computing the t-Hamming weight error vector $\widehat{e}$(plaintext) of a given syndrome $\widehat{s}$(ciphertext) in the scrambled code $\widehat{\mathcal{C}}$ using a secret key $sk = (L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}}, r)$. The process is outlined below.

(Step 1) Convert the syndrome $\widehat{s}$ in $\widehat{\mathcal{C}}$ into the syndrome $s(= \mathbf{S}^{-1}\widehat{s})$ in $\mathcal{C}$ by multiplying the secret key $\mathbf{S}^{-1}$.

(Step 2) Recover the error vector e corresponding to s with the secret key $L, g(X)$, which are decoding information of $\mathcal{C}$. At that stage, we use the extended Patterson decoding introduced by Section 2.4.2. (Alg. 1)

(Step 3) Return the error vector $\widehat{e}(= \mathbf{P}^{-1}e)$ of $\widehat{\mathcal{C}}$ obtained from e and the permutation matrix $\mathbf{P}^{-1}$ generated by the secret key $r_{\widehat{\mathcal{C}}}$. (Alg. 8)

Alg. 14 shows the pseudo codes of the encryption and the decryption, and Fig. 3.1 depicts these operations.

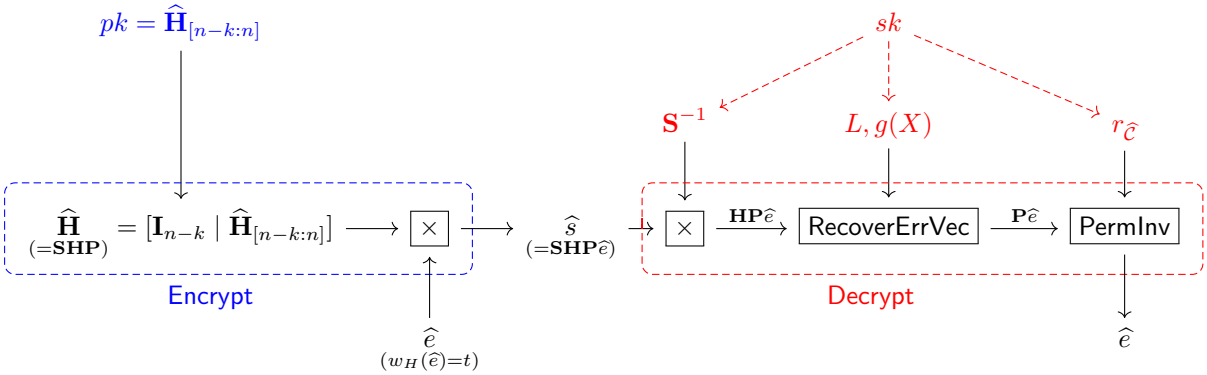


Figure 3.1: PALOMA: Encryption and Decryption

Algorithm 14 PALOMA: Encryption and Decryption

Input: A public key $pk = \widehat{\mathbf{H}}_{[n-k:n]} \in \mathbb{F}_2^{(n-k)\times k}$ and an error vector $\widehat{e} \in \mathbb{F}_2^n$ with $w_H(\widehat{e}) = t$
Output: A syndrome vector $\widehat{s} \in \mathbb{F}_2^{n-k}$

1: **procedure** Encrypt($pk = \widehat{\mathbf{H}}_{[n-k:n]}; \widehat{e}$)
2: $\quad \widehat{\mathbf{H}} \leftarrow [\mathbf{I}_{n-k} \mid \widehat{\mathbf{H}}_{[n-k:n]}] \in \mathbb{F}_2^{(n-k)\times n}$
3: $\quad \widehat{s} \leftarrow \widehat{\mathbf{H}}\widehat{e} \in \mathbb{F}_2^{n-k}$
4: $\quad$ **return** $\widehat{s}$
5: **end procedure**

Input: A secret key $sk = (L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}}, r)$ and a syndrome vector $\widehat{s} \in \mathbb{F}_2^{n-k}$
Output: An error vector $\widehat{e} \in \mathbb{F}_2^n$ with $w_H(\widehat{e}) = t$

1: **procedure** Decrypt($sk = (L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}}, r); \widehat{s}$)
2: $\quad s \leftarrow \mathbf{S}^{-1}\widehat{s}$
3: $\quad e \leftarrow$ RecoverErrVec($L, g(X); s$) ▷ Alg. 1
4: $\quad \widehat{e} \leftarrow$ PermInv($e, r_{\widehat{\mathcal{C}}}$) ▷ $\widehat{e} = \mathbf{P}^{-1}e$, Alg. 8
5: $\quad$ **return** $\widehat{e}$
6: **end procedure**

3.5 Encapsulation and Decapsulation

Encapsulation. Encap takes a public key pk as an input and returns a key κ and the ciphertext $c = (\widehat{r}, \widehat{s})$ of κ. The procedure is as follows. (Alg. 15)

(Step 1) A 256-bit string r^* is uniformly chosen.

(Step 2) Generate a random n-bit error vector e^* with $w_H(e^*) = t$ using GenErrVec and r^*. (Alg. 9)

(Step 3) Query e^* to RO_G and obtain a 256-bit string $\widehat{r}$. (Alg. 10)

(Step 4) Compute $\widehat{e}(= \mathbf{P}e^*) = \mathsf{Perm}(e^*, \widehat{r})$. (Alg. 8)

(Step 5) Obtain the $(n-k)$-bit syndrome $\widehat{s}$ of $\widehat{e}$ using Encrypt with pk. (Alg. 14)

(Step 6) Query $(e^*\|\widehat{r}\|\widehat{s})$ to RO_H and obtain a 256-bit key κ. (Alg. 10)

(Step 7) Return a ciphertext $c = (\widehat{r}, \widehat{s})$ and a key κ.

Decapsulation. Decap returns the key κ when given the secret key sk and the ciphertext $c = (\widehat{r}, \widehat{s})$ as inputs. The process is as follows. (Alg. 15)

(Step 1) Obtain the error vector $\widehat{e}$ by entering $\widehat{s}$ and sk into the Decrypt. (Alg. 14)

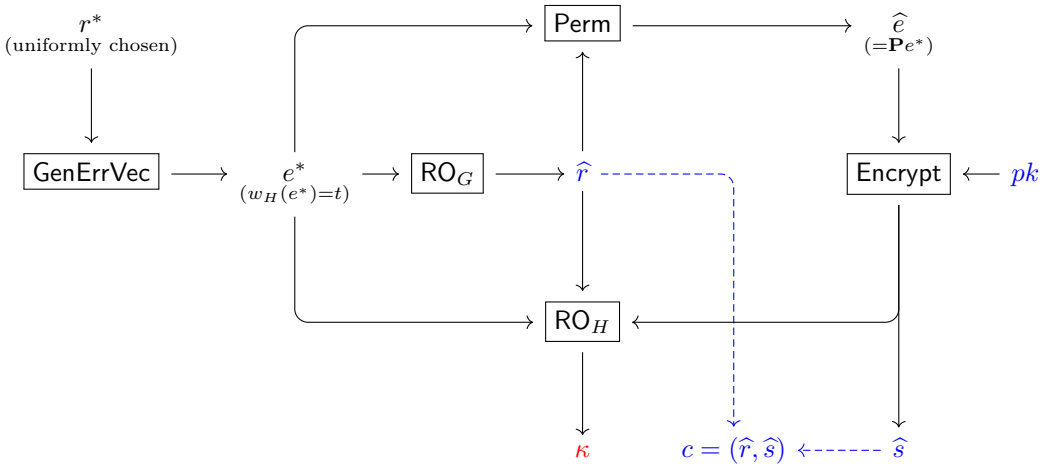
(Step 2) Compute $e^*(= \mathbf{P}^{-1}\widehat{e}) = \mathsf{PermInv}(\widehat{e}, \widehat{r})$. (Alg. 8)

(Step 3) Query e^* to the RO_G and obtain a 256-bit string $\widehat{r}'$. (Alg. 10)

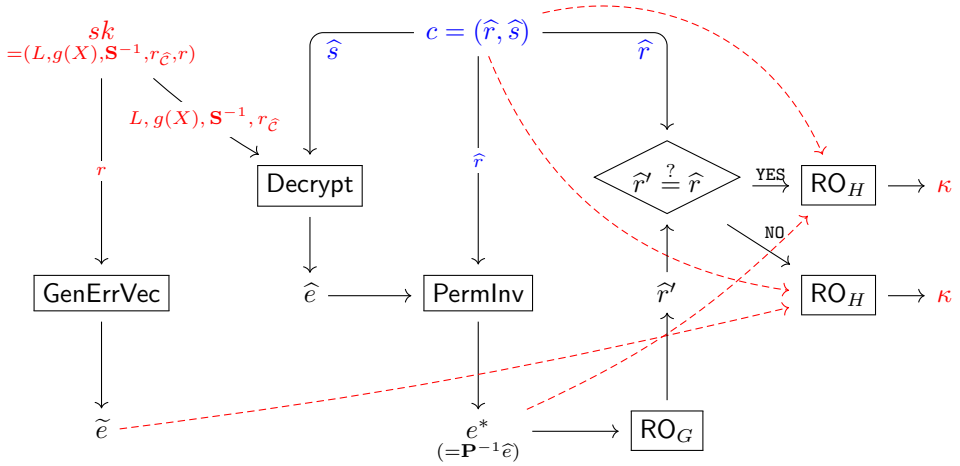
(Step 4) Generate the error vector $\widetilde{e}$ using GenErrVec with the secret key r. (Alg. 9)

(Step 5) If $\widehat{r}' = \widehat{r}$, then query $(e^*\|\widehat{r}\|\widehat{s})$ to the random oracle RO_H, and if not, query $(\widetilde{e}\|\widehat{r}\|\widehat{s})$ to RO_H. Return the received bit string from RO_H as a key κ. (Alg. 10)

Fig. 3.2 outlines Encap and Decap.



(a) $\kappa, c = (\widehat{r}, \widehat{s}) \leftarrow \mathsf{Encap}(pk)$



(b) $\kappa \leftarrow \mathsf{Decap}(sk; c = (\widehat{r}, \widehat{s}))$

Figure 3.2: PALOMA: Encapsulation and Decapsulation

Algorithm 15 PALOMA: Encapsulation and Decapsulation

Input: A public key $pk \in \{0,1\}^{(n-k)\times k}$
Output: A ciphertext $c = (\widehat{r}, \widehat{s}) \in \{0,1\}^{256} \times \{0,1\}^{n-k}$ and a key $\kappa \in \{0,1\}^{256}$

procedure $\mathsf{Encap}(pk)$
  $r^* \xleftarrow{\$} \{0,1\}^{256}$
  $e^* \leftarrow \mathsf{GenErrVec}(n, t, r^*)$ ▷ $w_H(e^*) = t$
  $\widehat{r} \leftarrow \mathsf{RO}_G(e^*)$ ▷ $\widehat{r} \in \{0,1\}^{256}$
  $\widehat{e} \leftarrow \mathsf{Perm}(e^*, \widehat{r})$ ▷ $\widehat{e} = \mathbf{P}e^*$
  $\widehat{s} \leftarrow \mathsf{Encrypt}(pk; \widehat{e})$ ▷ $\widehat{s} \in \{0,1\}^{n-k}$
  $\kappa \leftarrow \mathsf{RO}_H(e^* \| \widehat{r} \| \widehat{s})$ ▷ $\kappa \in \{0,1\}^{256}$
  return $c = (\widehat{r}, \widehat{s})$ and κ
end procedure

Input: A secret key $sk = (L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}}, r)$ and a ciphertext $c = (\widehat{r}, \widehat{s})$
Output: A key $\kappa \in \{0,1\}^{256}$

procedure $\mathsf{Decap}(sk = (L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}}, r); c = (\widehat{r}, \widehat{s}))$
  $\widehat{e} \leftarrow \mathsf{Decrypt}(sk; \widehat{s})$ ▷ $w_H(\widehat{e}) = t$
  $e^* \leftarrow \mathsf{PermInv}(\widehat{e}, \widehat{r})$ ▷ $e^* = \mathbf{P}^{-1}\widehat{e}$
  $\widehat{r}' \leftarrow \mathsf{RO}_G(e^*)$
  $\widetilde{e} \leftarrow \mathsf{GenErrVec}(n, t, r)$
  if $\widehat{r}' \neq \widehat{r}$ **then**
    $\kappa \leftarrow \mathsf{RO}_H(\widetilde{e} \| \widehat{r} \| \widehat{s})$ ▷ implicit rejection
  else
    $\kappa \leftarrow \mathsf{RO}_H(e^* \| \widehat{r} \| \widehat{s})$
  end if
  return κ
end procedure

Chapter 4

Performance

In this chapter, we provide the performance analysis result of PALOMA.

4.1 Description of C Implementation

4.1.1 Data Structure for $\mathbb{F}_{2^{13}}[X]$

The elements of $\mathbb{F}_{2^{13}} = \mathbb{F}_2[z]/\langle f(z)\rangle$ are stored in the 2-byte data type, where $f(z) = z^{13} + z^7 + z^6 + z^5 + 1$. The data structure for a field element $a(z) = \sum_{i=0}^{12} a_i z^i$ is defined as $0^3\|a_{12}\|\cdots\|a_0 \in \{0,1\}^{16}$. A polynomial $a(X) = \sum_{i=0}^{l} a_i X^i \in \mathbb{F}_{2^{13}}[X]$ of degree l is stored in $2(l+1)$-byte as $a_0\|\cdots\|a_l \in (\{0,1\}^{16})^{l+1}$.

4.1.2 Arithmetics in $\mathbb{F}_{2^{13}}$

PALOMA uses the pre-computed tables for multiplication, squaring, square rooting, and inversion in $\mathbb{F}_{2^{13}}$.

Multiplication. To store the multiplication of all pairs in $\mathbb{F}_{2^{13}}$, a table of 128MB ($=2\times 2^{26}$-byte) is required. In order to reduce the size of the table, PALOMA employs the multiplication of three smaller tables. Every field element $a(z) \in \mathbb{F}_{2^{13}}$ can be expressed as $a_1(z)z^7 + a_0(z)$ where $\deg(a_0) \leq 6$ and $\deg(a_1) \leq 5$. So, the multiplication of $a(z) = a_1(z)z^7 + a_0(z)$ and $b(z) = b_1(z)z^7 + b_0(z)$ in $\mathbb{F}_{2^{13}}$ can be computed as follows.

$$\begin{aligned} a(z)b(z) \bmod f(z) = &\big(a_1(z)b_1(z)z^{14} \bmod f(z)\big) + \big(a_1(z)b_0(z)z^7 \bmod f(z)\big) \\ &+ \big(a_0(z)b_1(z)z^7 \bmod f(z)\big) + \big(a_0(z)b_0(z) \bmod f(z)\big)\,. \end{aligned}$$

Thus, the multiplication can be calculated using the following three tables $\mathsf{MUL}_{00} : \{0,1\}^7 \times \{0,1\}^7 \to \{0,1\}^{16}$, $\mathsf{MUL}_{10} : \{0,1\}^6 \times \{0,1\}^7 \to \{0,1\}^{16}$, and $\mathsf{MUL}_{11} : \{0,1\}^6 \times \{0,1\}^6 \to \{0,1\}^{16}$ for all possible pairs.

$$\begin{aligned} \mathsf{MUL}_{00}[a_0, b_0] &:= a_0(z)b_0(z) \bmod f(z), \\ \mathsf{MUL}_{10}[a_1, b_0] &:= a_1(z)b_0(z)z^7 \bmod f(z), \\ \mathsf{MUL}_{11}[a_1, b_1] &:= a_1(z)b_1(z)z^{14} \bmod f(z). \end{aligned}$$

Note that $a_0(z)b_1(z)z^7 \bmod f(z)$ is computed using the table MUL_{10}.

Squaring, Square root, and Inversion. Tables SQU, SQRT, and INV store the results of the squares, the square roots, and the inverses, respectively, for all elements in $\mathbb{F}_{2^{13}}$. Note that we define the inverse of 0 as 0. Tab. 4.1 presents the size of pre-computed arithmetic tables.

Table 4.1: Pre-computed Tables for Arithmetics in $\mathbb{F}_{2^{13}}$ used in PALOMA

Table	Size (in bytes)		Description
MUL_{00}	32,768	$(= 2^{14} \times 2)$	$a_0(z)b_0(z) \bmod f(z)$, $\deg(a_0), \deg(b_0) < 7$
MUL_{10}	16,384	$(= 2^{13} \times 2)$	$a_1(z)b_0(z)z^7 \bmod f(z)$, $\deg(a_1) < 6$, $\deg(b_0) < 7$
MUL_{11}	8,192	$(= 2^{12} \times 2)$	$a_1(z)b_1(z)z^{14} \bmod f(z)$, $\deg(a_1), \deg(b_1) < 6$
SQU	16,384	$(= 2^{13} \times 2)$	$a(z)^2 \bmod f(z)$, $\deg(a) < 13$
SQRT	16,384	$(= 2^{13} \times 2)$	$\sqrt{a(z)}$ where $a(z) = (\sqrt{a(z)})^2 \bmod f(z)$, $\deg(a) < 13$
INV	16,384	$(= 2^{13} \times 2)$	$a(z)^{-1}$ where $1 = a(z)^{-1}a(z) \bmod f(z)$, $\deg(a) < 13$
Total	106,496	(= 104 KB)	

4.2 Data Size

We determine the size of a public key $pk = \widehat{\mathbf{H}}_{[n-k:n]}$, a secret key $sk = (L, g(X), \mathbf{S}^{-1}, r_{\widehat{\mathcal{C}}}, r)$, and a ciphertext $c = (\widehat{r}, \widehat{s})$ in terms of byte strings. Each size in bytes is computed by the following formula.

$$\begin{aligned}
\mathsf{bytelen}(pk) &= \mathsf{bytelen}(\widehat{\mathbf{H}}_{[n-k:n]}) = \left\lceil \frac{(n-k)k}{8} \right\rceil, \\
\mathsf{bytelen}(sk) &= \mathsf{bytelen}(L) + \mathsf{bytelen}(g(X)) + \mathsf{bytelen}(\mathbf{S}^{-1}) + \mathsf{bytelen}(r_{\widehat{\mathcal{C}}}) + \mathsf{bytelen}(r) \\
&= 2n + 2t + \left\lceil \frac{(n-k)^2}{8} \right\rceil + 32 + 32, \\
\mathsf{bytelen}(c) &= \mathsf{bytelen}(\widehat{r}) + \mathsf{bytelen}(\widehat{s}) = 32 + \left\lceil \frac{n-k}{8} \right\rceil.
\end{aligned}$$

Note that a monic Goppa polynomial $g(X) \in \mathbb{F}_{2^{13}}[X]$ of degree t is stored in $2t$-byte. Tab. 4.2 shows the sizes of a public key, a secret key, and a ciphertext of PALOMA. As stated in Section 3.3.3, the size of a secret key can be 768-bit. However, using such a key size may adversely affect the decryption speed performance.

Tab. 4.3 shows the data size comparison among the NIST competition round 4 code-based ciphers and PALOMA. The data size of PALOMA is similar to Classic McEliece because of the usage of SDP-based trapdoor. Compared to HQC and BIKE, the size of a public key and a secret key is relatively large. However, the size of the ciphertext which is the actual transmitted value is smaller than HQC and BIKE.

4.3 Speed

PALOMA is implemented in ANSI C. A speed benchmark was performed on the following two platforms using the Apple clang with the `-O3` optimization option:

(Platform 1) macOS, Sonoma 14.1.2, Apple M1 Max, 32GB RAM, Apple clang version 15.0.0

Table 4.2: Data Size Performance of PALOMA (in bytes)

		PALOMA-128	PALOMA-192	PALOMA-256
Public key pk	$\widehat{\mathbf{H}}_{[n-k:n]} \in \mathbb{F}_2^{(n-k)\times k}$	319,488	812,032	1,025,024
Secret key sk	$L \in \mathbb{F}_{2^{13}}^n$	7,808	11,136	13,184
	$g(X) \in \mathbb{F}_{2^{13}}[X]$	128	256	256
	$\mathbf{S}^{-1} \in \mathbb{F}_2^{(n-k)\times(n-k)}$	86,528	346,112	346,112
	$r_{\widehat{\mathcal{C}}} \in \{0,1\}^{256}$	32	32	32
	$r \in \{0,1\}^{256}$	32	32	32
	Total	94,528	357,568	359,616
Ciphertext c	$\widehat{r} \in \{0,1\}^{256}$	32	32	32
	$\widehat{s} \in \mathbb{F}_2^{n-k}$	104	208	208
	Total	136	240	240
Key κ	$\kappa \in \{0,1\}^{256}$	32	32	32

(Platform 2) macOS, Sonoma 14.2.1, Apple M3, 8GB RAM, Apple clang version 14.0.3

The results are shown in Tab. 4.4 and Tab. 4.5. We conducted 100 iterations each and measured the average value of a single operation, and measured the round 4 code presented by the Classic McEliece developers. We also apply the AES-256-CTR-based DRBG provided in OpenSSL. Comparing PALOMA with Classic McEliece reveals that PALOMA exhibits faster Encap and Decap speeds. However, GenKeyPair of Classic McEliece is faster than that of PALOMA.

Table 4.3: Data Size Comparison of Code-based KEMs (in bytes)

Security	Algorithm	Public key	Secret key	Ciphertext	Key
128	hqc-128	2,249	40	4,481	64
	BIKE	1,541	281	1,573	32
	mceliece348864	261,120	6,492	96	32
	PALOMA-128	319,488	94,528	136	32
192	hqc-192	4,522	40	9,026	64
	BIKE	3,083	419	3,115	32
	mceliece460896	524,160	13,608	156	32
	PALOMA-192	812,032	357,568	240	32
256	hqc-256	7,245	40	14,469	64
	BIKE	5,122	580	5,154	32
	mceliece6688128	1,044,992	13,932	208	32
	PALOMA-256	1,025,024	359,616	240	32

Table 4.4: Speed Performance of PALOMA (in milliseconds(cycles))

	PALOMA-128		PALOMA-192		PALOMA-256	
	Plat. 1	Plat. 2	Plat. 1	Plat. 2	Plat. 1	Plat. 2
GenKeyPair	39.58 (947,517)	27.81 (1,021,606)	167.70 (4,025,479)	119.77 (2,939,786)	207.22 (5,001,633)	150.03 (3,665,877)
Encrypt	0.003 (56)	0.003 (42)	0.005 (89)	0.004 (74)	0.006 (109)	0.005 (88)
Decrypt	2.57 (61,778)	1.71 (42,278)	13.03 (310,759)	8.92 (214,352)	13.70 (328,884)	9.42 (225,706)
Encap	0.052 (1,257)	0.042 (1,014)	0.061 (1,444)	0.051 (1,279)	0.067 (1,611)	0.060 (1,368)
Decap	2.56 (62,322)	1.75 (40,848)	13.00 (312,372)	8.97 (212,087)	13.67 (329,616)	9.43 (225,231)

Table 4.5: Comparison between PALOMA and Classic McEliece in Plat. 2 (in milliseconds(cycles))

		GenKeyPair	Encap	Decap
128-bit	PALOMA-128	27.81 (1,021,606)	0.042 (1,014)	1.75 (40,848)
	mceliece348864f	28.24 (651,804)	0.038 (940)	15.03 (353,312)
192-bit	PALOMA-192	119.77 (2,939,786)	0.051 (1,279)	8.97 (212,087)
	mceliece460896f	79.67 (1,922,618)	0.073 (1,817)	37.16 (898,408)
256-bit	PALOMA-256	150.03 (3,665,877)	0.060 (1,368)	9.43 (225,231)
	mceliece6688128f	139.61 (3,331,050)	0.124 (2,920)	71.94 (1,726,978)

Chapter 5

Security

5.1 OW-CPA-secure PKE = (GenKeyPair, Encrypt, Decrypt)

When evaluating the security of PALOMA, it is important to consider that no critical attacks on binary separable Goppa codes have been reported thus far. However, for the purpose of security analysis, we assume that the scrambled code of a Goppa code is indistinguishable from a random code. It is considered difficult to generate an effective distinguisher for Goppa codes used in PALOMA, as their rates are all less than 0.8 [10]. Therefore, the most powerful attack considered is the ISD, which is a generic attack of SDP. Consequently, the security strength of PALOMA is assessed based on the number of bit operations required for the ISD process.

SDP based on $\mathcal{C} = [n, k, \geq 2t+1]_2$ is defined with a parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$, a syndrome $s \in \mathbb{F}_2^{n-k}$, and a Hamming weight t. We denote $\mathsf{SDP}(\mathbf{H}, s, t)$ as the root set of the SDP. We also denote the set of all n-bit vectors with a Hamming weight of t as $\mathcal{E}_t^n$ and represent the zero matrix as $\mathbf{0}$. It is worth noting that the parameters n and t of PALOMA are selected to ensure that the underlying SDP possesses a unique root, and that both n and t are even.

5.1.1 Assumptions for Analysis

5.1.1.1 Deterministic Fisher-Yates Shuffle based on a 256-bit string

PALOMA utilizes the Shuffle (Alg. 6) to generate Goppa codes, permutation matrices, and error vectors. The deterministic Shuffle based on a 256-bit input is a modified version of the probabilistic Fisher-Yates shuffle. The Shuffle satisfies the following property.

Proposition 5.1. *Let* $w \in \{3904, 5568, 6592, 8192\}$ *and* $A = [0, 1, \ldots, w-1]$. *If* $\mathsf{Shuffle}(A, r) = \mathsf{Shuffle}(A, \widehat{r})$ *for some* $r, \widehat{r} \in \{0,1\}^{256}$, *then* $r = \widehat{r}$.

Proof. Let $r = (r_0, r_1, \ldots, r_{15})$ and $\widehat{r} = (\widehat{r}_0, \widehat{r}_1, \ldots, \widehat{r}_{15})$ for $0 \leq r_i, \widehat{r}_i < 2^{16}$. According to the nature of the Fisher-Yates shuffle, in order for the two resulting arrays to be identical, the following equation must be satisfied.

$$r_{j \bmod 16} \equiv \widehat{r}_{j \bmod 16} \pmod{w-j} \quad \text{for } j = 0, 1, \ldots, w-2.$$

When $w = 3904$, we obtain $r_0 \equiv \widehat{r}_0 \pmod{3904}$ and $r_0 \equiv \widehat{r}_0 \pmod{3888}$, resulting in $\mathrm{lcm}(3904, 3888) = 948672 \mid r_0 - \widehat{r}_0$. However, since $0 \leq r_0, \widehat{r}_0 < 2^{16} = 65536$, we have $r_0 = \widehat{r}_0$. By employing a similar method, we obtain $r_i = \widehat{r}_i$ for $i = 1, \ldots, 15$. The same approach applies when $w \in \{5568, 6592, 8192\}$ and yields the same result. □

According to Proposition 5.1, Shuffle returns distinct 2^{256} arrays. PALOMA assumes that an arbitrarily selected array from the total of $w!$ possible arrays and an array obtained through Shuffle with an arbitrarily selected 256-bit input are indistinguishable.

5.1.1.2 Number of Equivalent Codes

PALOMA defines the support set L as the top n elements obtained by shuffling $\mathbb{F}_{2^{13}}$ using the Shuffle with a 256-bit input. The next t elements are defined as the zeros to the Goppa polynomial $g(X)$. Therefore, the expected number of equivalent codes among the 2^{256} Goppa codes generated using this method is as follows.

$$2^{256} \times \frac{\binom{2^{13}}{n} n! \binom{2^{13}-n}{t} t!}{2^{13}!} = \frac{2^{256}}{(2^{13}-n-t)!} \approx \begin{cases} 2^{-44532}, & \text{PALOMA-128,} \\ 2^{-24318}, & \text{PALOMA-192,} \\ 2^{-13117}, & \text{PALOMA-256.} \end{cases}$$

Due to the expectation values being extremely small for all three parameters of PALOMA, it is assumed that PALOMA defines the SDP using distinct 2^{256} Goppa codes.

5.1.1.3 Number of t-Hamming weight Error Vectors

PALOMA uses GenErrVec to generate an error vector $e^* \in \mathbb{F}_2^n$ with a Hamming weight of t. (Alg. 9) In other words, based on a 256-bit string r^*, it shuffles the array $[0, 1, \ldots, n-1]$ and defines $\mathrm{supp}\,(e^*)$ as the top t elements. The expected value for the number of identical vectors among the 2^{256} error vectors generated using this method is as follows.

$$2^{256} \times \frac{\binom{n}{t} t!}{n!} = \frac{2^{256}}{(n-t)!} \approx \begin{cases} 2^{-39933}, & \text{PALOMA-128,} \\ 2^{-59410}, & \text{PALOMA-192,} \\ 2^{-72248}, & \text{PALOMA-256.} \end{cases}$$

Since the expected value for all three parameter sets of PALOMA is significantly smaller than 2^{-256}, it is assumed that GenErrVec returns distinct 2^{256} error vectors.

5.1.1.4 Number of Plaintexts

In PALOMA, the plaintext $\widehat{e}$ of the SDP is generated from a 256-bit string r^* through the operations GenErrVec, RO_G, and Perm. (Fig. 3.2 (a)) PALOMA assumes that the probability of having different 256-bit strings that produce the same plaintext $\widehat{e}$ through this process is extremely low and can be disregarded. In other words, PALOMA considers that there are 2^{256} possible plaintext candidates.

5.1.2 Exhaustive Search

The naive algorithm for finding roots of an SDP in PALOMA is the exhaustive search, shown in Alg. 16. The computational search complexity is $O\left(\binom{n}{t}(n-k)\right)$ in terms of bit operations. The complexity for each PALOMA parameter set is as follows.

$$O\left(\binom{n}{t}(n-k)\right) \approx \begin{cases} O\left(2^{476.52}\right), & \text{PALOMA-128,} \\ O\left(2^{885.11}\right), & \text{PALOMA-192,} \\ O\left(2^{916.62}\right), & \text{PALOMA-256.} \end{cases}$$

Algorithm 16 Exhaustive Search of SDP

Input: $\mathbf{H} = [h_0 \mid h_1 \mid \cdots \mid h_{n-1}] \in \mathbb{F}_2^{(n-k)\times n}$, $s \in \mathbb{F}_2^{n-k}$, and t
Output: $e \in \mathcal{E}_t^n$ such that $\mathbf{H}e = s$

```
for j_1 = 0 to n − 1 − (t − 1) do
    v_1 ← s + h_{j_1}
    for j_2 = j_1 + 1 to n − 1 − (t − 2) do
        v_2 ← v_1 + h_{j_2}
        ...
        for j_t = j_{t−1} + 1 to n − 1 do
            v_t ← v_{t−1} + h_{j_t}
            if v_t = 0^{n−k} then
                set e with supp(e) = {j_1, ..., j_t}
                return e
            end if
        end for
    end for
end for
```

PALOMA assumes that its underlying SDP has 2^{256} candidate roots. (Section 5.1.1.4) Each candidate generation requires the operations of GenErrVec, RO_G, and Perm. The process of verifying if a candidate is a root requires $t-1$ $(n-k)$-bit additions. The computational cost of the Shuffle, which is the main operation in GenErrVec and Perm, is negligible compared to the hash function operation, RO_G. Similarly, the computational cost of $t-1$ $(n-k)$-bit additions is also negligible compared to the RO_G operation. Therefore, the total computational cost of exhaustively searching the root candidates is $O(2^{256}T_G)$ where T_G is the computational cost of the RO_G operation. Assuming $T_G < 2^{40}$, generating and verifying the root candidates in PALOMA is more efficient in terms of computational cost compared to investigating all vectors with a Hamming weight of t. The set of 2^{256} root candidates can be precomputed before the start of the SDP challenge, independent of the public/secret keys. However, this requires memory of $2^{256}t\lceil\log_2 n\rceil$ bits.

5.1.3 Birthday-type Decoding

For a random permutation matrix $\mathbf{P} \in \mathcal{P}_n$, $\mathsf{SDP}(\mathbf{H}, s, t)$ and $\mathsf{SDP}(\mathbf{HP}, s, t)$ have the necessary and sufficient conditions: $e \in \mathsf{SDP}(\mathbf{H}, s, t)$ if and only if $\mathbf{P}^{-1}e \in \mathsf{SDP}(\mathbf{HP}, s, t)$. Let $I := [0 : \frac{n}{2}]$, $J := [\frac{n}{2} : n]$, and $\widehat{\mathbf{H}} := \mathbf{HP}$. Birthday-type decoding transforms SDP until finding the root $\widehat{e} = (\widehat{e}_I \| \widehat{e}_J) := \mathbf{P}^{-1}e \in \mathsf{SDP}(\widehat{\mathbf{H}}, s, t)$ that satisfies $w_H(\widehat{e}_I) = w_H(\widehat{e}_J) = \frac{t}{2}$ for a random permutation matrix $\mathbf{P}$. To find $\widehat{e}_I$ and $\widehat{e}_J$, we check the intersection of $T_I := \{s + \widehat{\mathbf{H}}_I\widehat{e}_I \in \mathbb{F}_2^{n-k} : \widehat{e}_I \in \mathcal{E}_{t/2}^{n/2}\}$ and $T_J := \{\widehat{\mathbf{H}}_J\widehat{e}_J \in \mathbb{F}_2^{n-k} : \widehat{e}_J \in \mathcal{E}_{t/2}^{n/2}\}$. The probability that the two sets, T_I and T_J, have an intersection for a randomly chosen permutation matrix $\mathbf{P}$ is $p = \binom{n/2}{t/2}^2 / \binom{n}{t}$. Therefore, the process of transforming SDP with a new $\mathbf{P}$ must be repeated at least $1/p$ times. Alg. 17 shows this attack in detail.

Since the number of bit computations for calculating $\widehat{\mathbf{H}}_I\widehat{e}_I$ and $\widehat{\mathbf{H}}_J\widehat{e}_J$ are $O(\binom{n/2}{t/2}(n-k))$, the total amount of computations for the PALOMA parameters is as follows.

$$O\left(2\binom{n}{t}(n-k)\Big/\binom{n/2}{t/2}\right) = \begin{cases} O(2^{245.77}), & \text{PALOMA-128,} \\ O(2^{450.81}), & \text{PALOMA-192,} \\ O(2^{466.57}), & \text{PALOMA-256.} \end{cases}$$

Algorithm 17 Finding a root of SDP: Birthday-type Decoding

Input: $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$ and $s \in \mathbb{F}_2^{n-k}$, w and $I = [0 : \frac{n}{2}]$, $J = [\frac{n}{2} : n]$
Output: $e \in \mathbb{F}_2^n$ such that $\mathbf{H}e = s$ and $w_H(e) = t$

while TRUE **do**
  $\mathbf{P} \xleftarrow{\$} \mathcal{P}_n$
  $\widehat{\mathbf{H}} \leftarrow \mathbf{HP}$
  $T[j] \leftarrow$ NULL for all $j \in \{0,1\}^{n-k}$
  for $\widehat{e}_I$ in $\mathcal{E}_{t/2}^{n/2}$ **do** ▷ exhaustive search
    $u \leftarrow s + \widehat{\mathbf{H}}_I \widehat{e}_I$ // num. of bit operations $= n-k$
    $T[u] \leftarrow \widehat{e}_I$
  end for
  for $\widehat{e}_J$ in $\mathcal{E}_{t/2}^{n/2}$ **do** ▷ exhaustive search
    $u \leftarrow \widehat{\mathbf{H}}_J \widehat{e}_J$ // num. of bit operations $= n-k$
    if $T[u] \neq$ NULL **then**
      $\widehat{e} \leftarrow (T[u] \| \widehat{e}_J)$ ▷ $T[u] = \widehat{e}_I$
      return $\mathbf{P}\widehat{e}$
    end if
  end for
end while

To increase the probability p to a value close to 1 in birthday-type decoding, define the two subsets $I = \left[0 : \frac{n}{2} + \varepsilon\right]$ and $J = \left[\frac{n}{2} - \varepsilon : n\right]$ for some $\varepsilon > 0$. When we find $e_1, e_2 \in \mathcal{E}_{t/2}^{\frac{n}{2}+\varepsilon}$ that satisfy $s + \widehat{\mathbf{H}}_I e_1 = \widehat{\mathbf{H}}_J e_2$, it cannot be assumed that $(e_1 \| 0^{\frac{n}{2}-\varepsilon}) + (0^{\frac{n}{2}-\varepsilon} \| e_2)$ is a root. If $w_H((e_1 \| 0^{\frac{n}{2}-\varepsilon}) + (0^{\frac{n}{2}-\varepsilon} \| e_2)) = t$, then $(e_1 \| 0^{\frac{n}{2}-\varepsilon}) + (0^{\frac{n}{2}-\varepsilon} \| e_2)$ is the root. Therefore it is necessary to include this discriminant. In this attack, ε is set to a value that makes the probability $p = \binom{n/2+\varepsilon}{t/2}^2 / \binom{n}{t}$ close to 1. The calculated amount of birthday-type decoding for the PALOMA parameters is counted as follows.

$$O\left(2(n-k)\binom{n/2+\varepsilon}{t/2}\right) \approx O\left(2(n-k)\sqrt{\binom{n}{t}}\right) = \begin{cases} O(2^{244.11}), & \text{PALOMA-128,} \\ O(2^{448.91}), & \text{PALOMA-192,} \\ O(2^{464.66}), & \text{PALOMA-256.} \end{cases}$$

We consider the computation cost of this approach as a birthday-type decoding calculation, even though the overall computational complexity decreases by only about 2 or 3 bits compared to the increase in memory complexity.

5.1.4 Improved Birthday-type Decoding

By defining two smaller SDPs from the SDP, and obtaining the roots of each SDP through birthday-type decoding, it is possible to find the root of the SDP while checking if the root candidate satisfies certain conditions. This is referred to as improved birthday-type decoding.

Consider $\mathbf{H} = \binom{\mathbf{H}_1}{\mathbf{H}_2} \in \mathbb{F}_2^{(n-k)\times n}$ as a concatenation of two submatrices, $\mathbf{H}_1 \in \mathbb{F}_2^{r\times n}$ and $\mathbf{H}_2 \in \mathbb{F}_2^{(n-k-r)\times n}$, where $r \leq n-k$. For the n-bit roots $x \in \mathsf{SDP}\left(\mathbf{H}_1, s_{[0:r]}, t/2+\varepsilon\right)$ and $y \in \mathsf{SDP}\left(\mathbf{H}_1, 0^r, t/2+\varepsilon\right)$ for $\mathbf{H}_1$, if x and y satisfy $\mathbf{H}_2(x+y) = s_{[r:n-k]}$ and $w_H(x+y) = t$, then $x+y \in \mathsf{SDP}(\mathbf{H}, s, t)$. Note that $|\mathsf{SDP}\left(\mathbf{H}_1, s_{[0:r]}, t/2+\varepsilon\right)| \approx |\mathsf{SDP}\left(\mathbf{H}_1, 0^r, t/2+\varepsilon\right)| \approx \frac{\binom{n}{t/2+\varepsilon}}{2^r}$. Alg. 18 shows this method in detail.

Algorithm 18 Finding a root of SDP: Improved Birthday-type Decoding

Input: $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$, $s \in \mathbb{F}_2^{n-k}$, t and r
Output: $e \in \mathbb{F}_2^n$ such that $\mathbf{H}e = s$ and $w_H(e) = t$

$T[j] \leftarrow \emptyset$ for all $j \in \{0,1\}^{n-k-r}$
for x in SDP $(\mathbf{H}_1, s_{[0:r]}, t/2+\varepsilon)$ **do** ▷ birthday-type decoding
  $u \leftarrow s_{[r:n-k]} + \mathbf{H}_2 x$ // num. of bit operations $= (t/2+\varepsilon)(n-k-r)$
  $T[u] \leftarrow T[u] \cup \{x\}$
end for
for y in SDP $(\mathbf{H}_1, 0^r, t/2+\varepsilon)$ **do** ▷ birthday-type decoding
  $u \leftarrow \mathbf{H}_2 y$ // num. of bit operations $= (t/2+\varepsilon)(n-k-r)$
  for x in $T[u]$ **do** // $|T[u]| \approx \frac{\binom{n}{t/2+\varepsilon}}{2^r} \times \frac{1}{2^{n-k-r}}$
    $e \leftarrow x + y$ // num. of bit operations $= \frac{\binom{n}{t/2+\varepsilon}}{2^r} \times \frac{n\binom{n}{t/2+\varepsilon}}{2^{n-k}}$
    if $w_H(e) = t$ **then**
      return e
    end if
  end for
end for

The number of bit operations in this algorithm is as follows.

$$4r\sqrt{\binom{n}{t/2+\varepsilon}} + \frac{\binom{n}{t/2+\varepsilon}}{2^r}\left((t+2\varepsilon)(n-k-r) + \frac{n\binom{n}{t/2+\varepsilon}}{2^{n-k}}\right).$$

Choice of ε. When two subsets A and B with the number of elements $t/2+\varepsilon$ are randomly selected from the set $[0:n]$, the expected value $E[|A \cap B|]$ is $\frac{(t/2+\varepsilon)^2}{n}$. Therefore, for the roots x and y of each small SDP, $E[w_H(x+y)]$ is as follows.

$$\begin{aligned} E[w_H(x+y)] &= E[2(|\mathrm{supp}\,(x)| - |\mathrm{supp}\,(x) \cap \mathrm{supp}\,(y)|)] \\ &= 2E[|\mathrm{supp}\,(x)|] - 2E[|\mathrm{supp}\,(x) \cap \mathrm{supp}\,(y)|)] \\ &= 2\,(t/2+\varepsilon) - \frac{2(t/2+\varepsilon)^2}{n}. \end{aligned}$$

Set ε to satisfy $\varepsilon = \frac{(t/2+\varepsilon)^2}{n}$, i.e., $\varepsilon = \frac{\sqrt{n^2-2nt}+(n-t)}{2}$. Then $E[w_H(x+y)] = t$.

Choice of r. For $e \in \mathrm{SDP}(\mathbf{H}, s, t)$, the number of (x,y) pairs satisfying $e = x+y$ is $|\{(x,y) \in (\mathcal{E}^n_{t/2+\varepsilon})^2 : e = x+y\}| = \binom{t}{t/2}\binom{n-t}{\varepsilon}$. Therefore, set r to satisfy $2^r \approx \binom{t}{t/2}\binom{n-t}{\varepsilon}$ to count the number of roots of small SDP accurately. The required amount of bit operations of improved birthday-type decoding for PALOMA parameters is as follows.

$$\begin{cases} O(2^{225.45})\ (\varepsilon = 3840,\ r = 61), & \text{PALOMA-128}, \\ O(2^{398.84})\ (\varepsilon = 5440,\ r = 125), & \text{PALOMA-192}, \\ O(2^{414.76})\ (\varepsilon = 6464,\ r = 125), & \text{PALOMA-256}. \end{cases}$$

5.1.5 Information Set Decoding

ISD is a generic decoding algorithm for random linear codes. The first phase of ISD involves transforming the parity-check matrix $\mathbf{H}$ into a systematic form to facilitate the identification of an error-free information set. In the second phase, error vectors satisfying specific conditions are

identified, utilizing a combination of birthday attack-type searches and partial brute force attacks. Initially proposed by E. Prange in 1962, ISD has demonstrated improved computational complexity by modifying the error vector conditions and incorporating search techniques inspired by birthday attacks [1, 5, 6, 9, 11, 18, 19, 21, 22, 27, 29].

5.1.5.1 Procedure

ISD utilizes Proposition 5.2, which describes the relationship between the code $\mathcal{C}$ and the scrambled code $\widehat{\mathcal{C}}$ of $\mathcal{C}$ in terms of the root of SDP.

Proposition 5.2. *Let $e \in SDP(\boldsymbol{H}, s, t)$. For an invertible matrix $\boldsymbol{S} \in \mathbb{F}_2^{(n-k)\times(n-k)}$ and a permutation matrix $\boldsymbol{P} \in \mathcal{P}_n$, we know $\boldsymbol{P}^{-1}e \in SDP(\boldsymbol{SHP}, \boldsymbol{S}s, t)$.*

ISD is a probabilistic algorithm that modifies SDP until it finds a root that satisfies certain conditions. ISD proceeds to the following two phases.

(Phase 1) Redefining a problem: Find $e \in \mathsf{SDP}(\mathbf{H}, s, t) \Rightarrow$ Find $\widehat{e} = \mathbf{P}^{-1}e \in \mathsf{SDP}(\widehat{\mathbf{H}} = \mathbf{SHP}, \widehat{s} = \mathbf{S}s, t)$ where $\mathbf{SHP} = \left(\begin{smallmatrix} \mathbf{I}_l & \mathbf{M}_1 \\ \mathbf{0} & \mathbf{M}_2 \end{smallmatrix}\right)$ is a partially systematic matrix obtained by applying elementary row operations.

(Phase 2) Find $\widehat{e}(= \mathbf{P}^{-1}e) \in \mathsf{SDP}(\widehat{\mathbf{H}}, \widehat{s}, t)$ that satisfies the specific Hamming weight condition and return $e(= \mathbf{P}\widehat{e})$. If no root satisfies the condition, go back to (Phase 1).

5.1.5.2 Computational Complexity

Let p be the probability that the root $\widehat{e}$ satisfies a specific Hamming weight condition in the modified problem. The computational complexity of the ISD is $\frac{1}{p} \times$((Phase 1)'s computational amount+ (Phase 2)'s computational amount). (Phase 1) involves modifying the problem using the Gaussian elimination. Most ISD algorithms require $O((n-k)^2 n)$ bit operations in this phase. ISD has been developed by improving the computational efficiency of (Phase 2) and the probability p.

We considered the BJMM-ISD to be the most effective ISD because the subsequent ISDs proposed after the BJMM-ISD in 2012 only provided minor improvements in specific situations [1]. Consequently, the parameters of PALOMA were chosen based on the precise calculation of the number of bit operations involved in the BJMM-ISD. The BJMM-ISD transforms the SDP into a small SDP and identifies the root of the SDP using birthday-type attacks.

5.1.5.3 Becker-Joux-May-Meurer

BJMM-ISD is an ISD that applies improved birthday-type decoding to the partial row-reduced echelon form [1]. Transform $\mathbf{H}$ into the form $\widehat{\mathbf{H}} = \left(\begin{array}{cc} \mathbf{I}_{n-k-l} & \mathbf{H}_1 \\ \hline \mathbf{0} & \mathbf{H}_2 \end{array}\right)$ where $\mathbf{H}_1 \in \mathbb{F}_2^{(n-k-l)\times(k+l)}$ and $\mathbf{H}_2 \in \mathbb{F}_2^{l\times(k+l)}$ by applying a partial RREF(row-reduced echelon form) operation for some $l(\leq n-k)$. For $I = [0 : n-k-l]$, $J = [n-k-l : n]$, and $L = [n-k-l : n-k]$, BJMM-ISD finds the root $\widehat{e} = (\widehat{e}_I \| \widehat{e}_J)$ of $\mathsf{SDP}(\widehat{\mathbf{H}} = \mathbf{SHP}, \widehat{s} = \mathbf{S}s, t)$ that satisfies the following conditions.

$$w_H(\widehat{e}_I) = t - p, \quad w_H(\widehat{e}_J) = p, \quad \widehat{e}_J \in \mathsf{SDP}(\mathbf{H}_2, \widehat{s}_L, p), \quad \widehat{e}_I + \widehat{e}_J\mathbf{H}_1 = \widehat{s}_I.$$

The process of BJMM-ISD is as follows.

(Phase 1) Randomly select a permutation matrix $\mathbf{P} \in \mathcal{P}_n$. Apply partial RREF to $\mathbf{HP}$ to obtain a partial canonical matrix $\widehat{\mathbf{H}} = \left(\begin{array}{cc} \mathbf{I}_{n-k-l} & \mathbf{H}_1 \\ \hline \mathbf{0} & \mathbf{H}_2 \end{array}\right)$. In this process, the invertible matrix

$\mathbf{S}$ satisfying $\widehat{\mathbf{H}} = \mathbf{SHP}$ can be obtained simultaneously. If there is no invertible matrix $\mathbf{S}$ that makes it a partial systematic form, (Phase 1) is performed again.

(Phase 2) Obtain $\mathsf{SDP}(\mathbf{H}_2, \widehat{s}_L, p)$ using the improved birthday-type decoding. If the root does not exist, go back to (Phase 1). If the Hamming weight of the vector $x := \widehat{s}_I + \mathbf{H}_1 y$ for $y \in \mathsf{SDP}(\mathbf{H}_2, \widehat{s}_L, p)$ is $t - p$, return $\mathbf{P}\widehat{e}$ because it is $\widehat{e} = (x \| y) \in \mathsf{SDP}(\widehat{\mathbf{H}}, \widehat{s}, t)$. If not, go back to (Phase 1).

Alg. 19 presents the BJMM-ISD process in detail.

Algorithm 19 Finding a root of SDP: BJMM-ISD

Input: $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$, $s \in \mathbb{F}_2^{n-k}$, and t
Output: $e \in \mathbb{F}_2^n$ such that $\mathbf{H}e = s$ and $w_H(e) = t$

while TRUE **do**
  $\mathbf{P} \xleftarrow{\$} \mathcal{P}_n$
  $\widehat{\mathbf{H}} = \mathbf{SHP} \leftarrow$ partial RREF($\mathbf{HP}$) // num. of bit operations $= (n-k-l)(n-k)n$
  if $\widehat{\mathbf{H}}_{I\times I} = \mathbf{I}_{n-k-l}$ **then**
    $\mathbf{H}_1, \mathbf{H}_2 \leftarrow \mathbf{H}_{J\times I}, \mathbf{H}_{J\times L}$ ▷ $\widehat{\mathbf{H}} = \left(\begin{array}{c|c} \mathbf{I}_{n-k-l} & \mathbf{H}_1 \\ \hline \mathbf{0} & \mathbf{H}_2 \end{array}\right)$
    $\widehat{s} \leftarrow \mathbf{S}s$
    for y in $\mathsf{SDP}(\mathbf{H}_2, \widehat{s}_L, p)$ **do** // $|\mathsf{SDP}(\mathbf{H}_2, \widehat{s}_L, p)| \approx \frac{\binom{k+l}{p}}{2^l}$ ▷ improved birthday-type decoding
      $x \leftarrow \widehat{s}_I + \mathbf{H}_1 y$ // num. of bit operations $= p(n-k-l)$
      if $w_H(x) = t - p$ **then**
        $\widehat{e} \leftarrow (x \| y)$
        return $\mathbf{P}\widehat{e}$
      end if
    end for
  end if
end while

The probability that $\widehat{e} = \mathbf{P}^{-1}e$ satisfies the Hamming weight condition for $e \in \mathsf{SDP}(\mathbf{H}, s, t)$ in BJMM-ISD is as follows.

$$\Pr[\mathbf{P} \xleftarrow{\$} \mathcal{P}_n,\ (w_H(\widehat{e}_I) = t - p) \wedge (w_H(\widehat{e}_J) = p)] = \frac{\binom{n-k-l}{t-p}\binom{k+l}{p}}{\binom{n}{t}}.$$

Therefore, the calculation amount for the bit operation in the BJMM-ISD is as follows.

$$\frac{\binom{n}{t}}{\binom{n-k-l}{t-p}\binom{k+l}{p}}\left((n-k-l)(n-k)n + \frac{p(n-k-l)\binom{k+l}{p}}{2^l} + T\right),$$

where $T :=$ num. of bit operations for $\mathsf{SDP}(\mathbf{H}_2, \widehat{s}_L, p)$. In this process, ε and r are set as follows for the computation of $\mathsf{SDP}(\mathbf{H}_2, \widehat{s}_L, p)$.

$$\varepsilon = \frac{\sqrt{(k+l)^2 - 2(k+l)p} + (k+l-p)}{2}, \quad r = \log_2\left(\binom{p}{p/2}\binom{k+l-p}{\varepsilon}\right).$$

The required amount of bit operations of BJMM-ISD for PALOMA parameters is as follows.

$$\begin{cases} O(2^{166.21})\ (l = 67,\ p = 14), & \text{PALOMA-128}, \\ O(2^{267.77})\ (l = 105,\ p = 22), & \text{PALOMA-192}, \\ O(2^{289.66})\ (l = 126,\ p = 26), & \text{PALOMA-256}. \end{cases}$$

Based on the above results, PALOMA claims that PALOMA-128, PALOMA-192, and PALOMA-256 have security strengths of 128-bit, 192-bit, and 256-bit, respectively. Tab. 5.1 is a comparison of the computational complexity of the exhaustive search, (improved) birthday-type decoding, and BJMM-ISD for PALOMA and Classic McEliece.

Table 5.1: Complexity of Several Attacks on PALOMA and Classic McEliece

Security	Algorithm	BJMM-ISD	Improved Birthday-type	Birthday-type	Exhaustive Search
128	PALOMA-128	$2^{166.21}$ $(l = 67,\ p = 14)$	$2^{225.78}$	$2^{244.11}$	$2^{476.52}$
	mceliece348864	$2^{161.97}$ $(l = 66,\ p = 14)$	$2^{220.26}$	$2^{238.75}$	$2^{465.91}$
192	PALOMA-192	$2^{267.77}$ $(l = 105,\ p = 22)$	$2^{399.67}$	$2^{448.91}$	$2^{885.11}$
	mceliece460896	$2^{215.59}$ $(l = 86,\ p = 18)$	$2^{311.80}$	$2^{345.58}$	$2^{678.88}$
256	PALOMA-256	$2^{289.66}$ $(l = 126,\ p = 26)$	$2^{415.59}$	$2^{464.66}$	$2^{916.62}$
	mceliece6688128	$2^{291.56}$ $(l = 126,\ p = 26)$	$2^{416.95}$	$2^{466.01}$	$2^{919.32}$
	mceliece6960119	$2^{289.92}$ $(l = 136,\ p = 28)$	$2^{402.41}$	$2^{443.58}$	$2^{874.57}$
	mceliece8192128	$2^{318.34}$ $(l = 157,\ p = 32)$	$2^{436.05}$	$2^{484.90}$	$2^{957.10}$

5.2 IND-CCA2-secure KEM = (GenKeyPair, Encap, Decap)

In the IND-CCA2 security experiment, which stands for INDistinguishability against Adaptive Chosen-Ciphertext Attack, for KEM = (GenKeyPair, Encap, Decap), the challenger $\mathcal{C}$ sends a challenge (ciphertext, key) pair to the adversary $\mathcal{A}$, who guesses whether the pair is correct or not. Here, "correct" means that the pair (ciphertext, key) is a valid output of the Encap. The adversary is allowed to query the decapsulation oracle $\mathcal{O}^{\mathsf{Decap}_{sk}}$ except for the challenge during the experiment. Fig. 5.1 depicts the security experiment $\mathsf{Exp}^{\mathsf{IND\text{-}CCA2}}_{\mathsf{KEM},\lambda}(\mathcal{A})$ for IND-CCA2-secure KEM in ROM. Here, λ is the security parameter and H is the random oracle.

We define the advantage $\mathsf{Adv}^{\mathsf{IND\text{-}CCA2}}_{\mathsf{KEM},\lambda}(\mathcal{A})$ of $\mathcal{A}$ as follows.

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA2}}_{\mathsf{KEM},\lambda}(\mathcal{A}) := \left| \Pr[\mathsf{Exp}^{\mathsf{IND\text{-}CCA2}}_{\mathsf{KEM},\lambda}(\mathcal{A}) = 1] - \frac{1}{2} \right|.$$

We say that KEM is IND-CCA2-secure in ROM when the advantage $\mathsf{Adv}^{\mathsf{IND\text{-}CCA2}}_{\mathsf{KEM},\lambda}(\mathcal{A})$ of $\mathcal{A}$ is negligible in terms of λ for any probabilistic polynomial-time attacker $\mathcal{A}$.

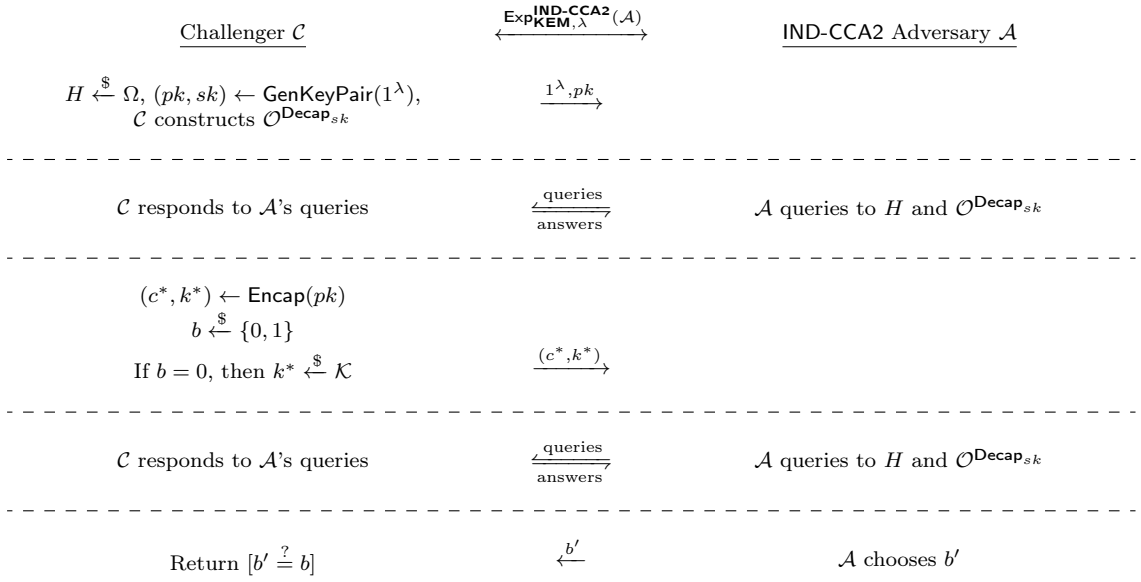


Figure 5.1: Security Experiment $\mathsf{Exp}^{\mathsf{IND\text{-}CCA2}}_{\mathsf{KEM},\lambda}(\mathcal{A})$ for IND-CCA2-secure KEM in ROM

5.2.1 OW-CPA-secure PKE

Based on the analysis results in Section 5.1, it is assumed that the underlying deterministic $\mathsf{PKE} = (\mathsf{GenKeyPair}, \mathsf{Encrypt}, \mathsf{Decrypt})$ of PALOMA is OW-CPA-secure. PKE has the following properties. For all key pairs $(pk, sk) \in \mathcal{PK} \times \mathcal{SK}$,

(1) (Injectivity) if $\mathsf{Encrypt}(pk; \widehat{e}_1) = \mathsf{Encrypt}(pk; \widehat{e}_2)$, then $\widehat{e}_1 = \widehat{e}_2$, and

(2) (Correctness) $\Pr[\widehat{e} \neq \mathsf{Decrypt}(sk; \mathsf{Encrypt}(pk; \widehat{e}))] = 0$ for all $\widehat{e} \in \mathcal{E}^n_t$.

The Fujisaki-Okamoto transformation is a method for designing an IND-CCA2-secure scheme from an OW-CPA-secure scheme in random oracle model. There are several variants of the Fujisaki-Okamoto transformation. Using the above properties, PALOMA is designed based on the implicit rejection $\mathsf{KEM}^{\not\perp} = \mathsf{U}^{\not\perp}[\mathsf{PKE}_1 = \mathsf{T}[\mathsf{PKE}_0, G], H]$ among FO-like transformations proposed by Hofheinz et al. [14]. This is combined with two modules: (1) T: converting OW-CPA-secure PKE_0 to OW-PCA(Plaintext-Checking Attack)-secure PKE_1 and (2) $\mathsf{U}^{\not\perp}$: converting it to IND-CCA2-secure KEM as follows.

$$\begin{aligned}
&\text{OW-CPA-secure } \mathsf{PKE}_0 = (\mathsf{GenKeyPair}, \mathsf{Encrypt}_0, \mathsf{Decrypt}_0)\\
&\xrightarrow[\text{with a random oracle } G]{\mathsf{T}} \quad \text{OW-PCA-secure } \mathsf{PKE}_1 = (\mathsf{GenKeyPair}, \mathsf{Encrypt}_1, \mathsf{Decrypt}_1)\\
&\xrightarrow[\text{with a random oracle } H]{\mathsf{U}^{\not\perp}} \quad \text{IND-CCA2-secure } \mathsf{KEM}^{\not\perp} = (\mathsf{GenKeyPair}, \mathsf{Encap}, \mathsf{Decap}).
\end{aligned}$$

5.2.2 OW-CPA-secure PKE_0

PKE_0 is defined with the PKE and Perm/PermInv of PALOMA as follows, and Alg. 20 shows the detailed process of PKE_0.

$$\begin{aligned}
\mathsf{Encrypt}_0(pk; \widehat{r}; e^*) &:= (\widehat{r}, \mathsf{Encrypt}(pk; \mathsf{Perm}(e^*, \widehat{r}))),\\
\mathsf{Decrypt}_0(sk; (\widehat{r}, \widehat{s})) &:= \mathsf{PermInv}(\mathsf{Decrypt}(sk; \widehat{s}), \widehat{r}).
\end{aligned}$$

Algorithm 20 PALOMA: PKE_0

Input: A public key $pk \in \mathcal{PK}$, a random coin $\widehat{r} \in \{0,1\}^{256}$, $e^* \in \mathcal{E}_t^n$
Output: A ciphertext $(\widehat{r}, \widehat{s}) \in \{0,1\}^{256} \times \{0,1\}^{13t}$

1: **procedure** $\mathsf{Encrypt}_0(pk; \widehat{r}; e^*)$
2: $\quad \widehat{e} \leftarrow \mathsf{Perm}(e^*, \widehat{r})$
3: $\quad \widehat{s} \leftarrow \mathsf{Encrypt}(pk; \widehat{e})$
4: $\quad$ **return** $(\widehat{r}, \widehat{s})$
5: **end procedure**

Input: A secret key $sk \in \mathcal{SK}$, a ciphertext $(\widehat{r}, \widehat{s}) \in \{0,1\}^{256} \times \{0,1\}^{13t}$
Output: $e^* \in \mathcal{E}_t^n$

1: **procedure** $\mathsf{Decrypt}_0(sk; (\widehat{r}, \widehat{s}))$
2: $\quad \widehat{e} \leftarrow \mathsf{Decrypt}(sk; \widehat{s})$
3: $\quad e^* \leftarrow \mathsf{PermInv}(\widehat{e}, \widehat{r})$
4: $\quad$ **return** e^*
5: **end procedure**

As PKE is assumed to be OW-CPA-secure, it follows that PKE_0 is also OW-CPA-secure. Fig. 5.2 shows the OW-CPA adversary $\mathcal{A}$ for PKE using an arbitrary OW-CPA adversary $\mathcal{A}_0$ for PKE_0.

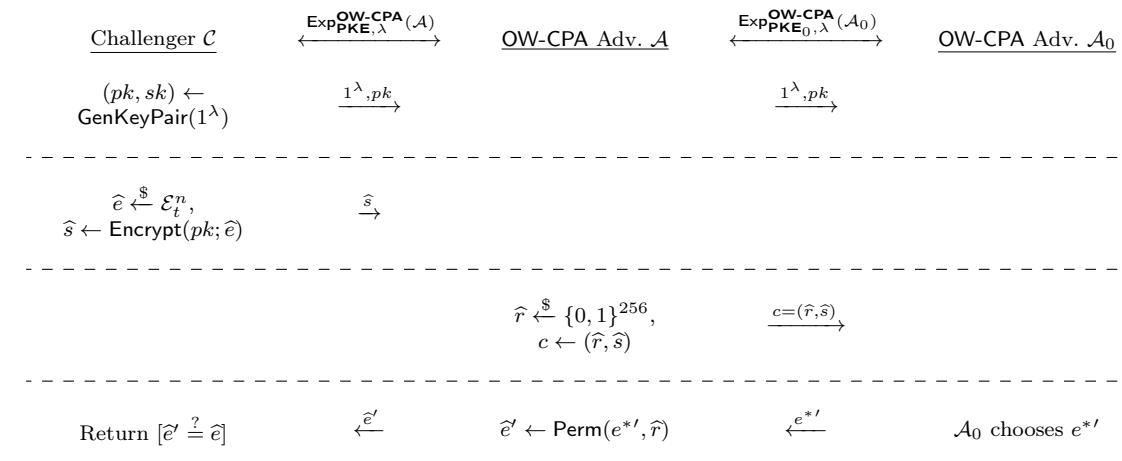


Figure 5.2: Construction of an OW-CPA Adversary for PKE using an OW-CPA Adversary for PKE_0

5.2.3 OW-PCA-secure PKE_1

The transform T for converting OW-CPA-secure PKE_0 to OW-PCA-secure PKE_1 is defined by

$$\mathsf{Encrypt}_1(pk; e^*) := \mathsf{Encrypt}_0(pk; \mathsf{RO}_G(e^*); e^*).$$

Alg. 21 shows the PKE_1 constructed by the transformation T and a random oracle RO_G.

In OW-PCA for PKE_1, the adversary can query to the plaintext-checking orcle $\mathcal{O}^{\mathsf{PC}}$, described in Alg. 22, during the OW-CPA security experiment. For any OW-PCA-attackers $\mathcal{B}$ on PKE_1, there exists an OW-CPA-attacker $\mathcal{A}$ on PKE_0 satisfying the inequality below [14, Theorem 3.1].

$$\mathsf{Adv}^{\mathsf{OW\text{-}PCA}}_{\mathsf{PKE}_1,\lambda}(\mathcal{B}) \leq (q_G + q_P + 1)\mathsf{Adv}^{\mathsf{OW\text{-}CPA}}_{\mathsf{PKE}_0,\lambda}(\mathcal{A}), \tag{5.1}$$

where q_G and q_P are the number of queries to the random oracle RO_G and the plaintext-checking oracle $\mathcal{O}^{\mathsf{PC}}$, which can be implemented by re-encryption. Note that PALOMA cannot implement a ciphertext validity oracle because it generates error vectors as messages from a 256-bit string.

From Eq. (5.1), if PKE_0 is OW-CPA-secure and q_G, q_P are polynomials in terms of λ, then $\mathsf{Adv}^{\mathsf{OW\text{-}PCA}}_{\mathsf{PKE}_1,\lambda}(\mathcal{B})$ is negligible, so we have PKE_1 is OW-PCA-secure.

Algorithm 21 PALOMA: PKE_1

Input: A public key $pk \in \mathcal{PK}$, $e^* \in \mathcal{E}_t^n$
Output: A ciphertext $(\widehat{r}, \widehat{s}) \in \{0,1\}^{256} \times \{0,1\}^{13t}$

procedure $\mathsf{Encrypt}_1(pk; e^*)$
  $\widehat{r} \leftarrow \mathsf{RO}_G(e^*)$
  $(\widehat{r}, \widehat{s}) \leftarrow \mathsf{Encrypt}_0(pk; \widehat{r}; e^*)$
  return $(\widehat{r}, \widehat{s})$
end procedure

Input: A secret key $sk \in \mathcal{SK}$, a ciphertext $(\widehat{r}, \widehat{s}) \in \{0,1\}^{256} \times \{0,1\}^{13t}$
Output: $e^* \in \mathcal{E}_t^n$ or $\perp$

procedure $\mathsf{Decrypt}_1(sk; (\widehat{r}, \widehat{s}))$
  $e^* \leftarrow \mathsf{Decrypt}_0(sk; \widehat{s})$
  if $w_H(e^*) \neq t$ **then**
    return $\perp$
  end if
  $\widehat{r}' \leftarrow \mathsf{RO}_G(e^*)$
  if $\widehat{r}' \neq \widehat{r}$ **then**
    return $\perp$
  end if
  return e^*
end procedure

Algorithm 22 PALOMA: Plaintext Checking Oracle $\mathcal{O}^{\mathsf{PC}}$ for PKE_1

Input: A message e^*, a ciphertext $(\widehat{r}, \widehat{s})$
Output: 1 or 0

procedure $\mathcal{O}^{\mathsf{PC}}(e^*, (\widehat{r}, \widehat{s}))$
  $\widehat{r}' \leftarrow \mathsf{RO}_G(e^*)$
  if $\widehat{r}' \neq \widehat{r}$ **then**
    return 0
  end if
  $\widehat{e} \leftarrow \mathsf{Perm}(e^*, \widehat{r})$
  $\widehat{s}' \leftarrow \mathsf{Encrypt}(pk; \widehat{e})$
  if $\widehat{s}' \neq \widehat{s}$ **then**
    return 0
  end if
  return 1
end procedure

5.2.4 IND-CCA2-secure $\mathsf{KEM}^{\not\perp}$

The transform $\mathsf{U}^{\not\perp}$ for converting OW-PCA-secure PKE_1 to IND-CCA2-secure $\mathsf{KEM}^{\not\perp}$ is as follows.

$$\mathsf{Encap}(pk) := (\underbrace{\mathsf{Encrypt}_1(pk; e^*)}_{=:(\widehat{r},\widehat{s})}), \underbrace{\mathsf{RO}_H(e^* \| \widehat{r} \| \widehat{s})}_{=:\kappa}).$$

e^* is determined by $\mathsf{GenErrVec}(n, t, r^*)$ with uniformly chosen $r^* \in \{0,1\}^{256}$. Alg. 23 shows $\mathsf{KEM}^{\not\perp}$ of PALOMA constructed by using the transformation $\mathsf{U}^{\not\perp}$ and a random oracle RO_H.

For any IND-CCA2-attackers $\mathcal{B}$ on $\mathsf{KEM}^{\not\perp}$, there exists an OW-PCA-attacker $\mathcal{A}$ on PKE_1 satisfying the inequality below [14, Theorem 3.4].

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA2}}_{\mathsf{KEM}^{\not\perp},\lambda}(\mathcal{B}) \leq \frac{q_H}{2^{256}} \mathsf{Adv}^{\mathsf{OW\text{-}PCA}}_{\mathsf{PKE}_1,\lambda}(\mathcal{A}),$$

where q_H is the number of queries to the plaintext-checking oracle. Therefore, if PKE_1 is OW-PCA-secure and q_H are polynomials in terms of λ, then $\mathsf{Adv}^{\mathsf{IND\text{-}CCA2}}_{\mathsf{KEM}^{\not\perp},\lambda}(\mathcal{B})$ is negligible. Consequently, we have that $\mathsf{KEM}^{\not\perp}$ is IND-CCA2-secure.

Algorithm 23 PALOMA: $\mathsf{KEM}^{\not\perp}$

Input: A public key $pk \in \mathcal{PK}$
Output: A ciphertext $(\widehat{r}, \widehat{s}) \in \{0,1\}^{256} \times \{0,1\}^{13t}$ and a key $\kappa \in \{0,1\}^{256}$

procedure $\mathsf{Encap}(pk)$
  $r^* \xleftarrow{\$} \{0,1\}^{256}$
  $e^* \leftarrow \mathsf{GenErrVec}(n, t, r^*)$
  $(\widehat{r}, \widehat{s}) \leftarrow \mathsf{Encrypt}_1(pk; e^*)$
  $\kappa \leftarrow \mathsf{RO}_H(e^* \| \widehat{r} \| \widehat{s})$
  return $(\widehat{r}, \widehat{s})$ and κ
end procedure

Input: A secret key $sk \in \mathcal{SK}$, a ciphertext $(\widehat{r}, \widehat{s}) \in \{0,1\}^{256} \times \{0,1\}^{13t}$
Output: $e^* \in \mathcal{E}_t^n$ or $\perp$

procedure $\mathsf{Decap}(sk; (\widehat{r}, \widehat{s}))$
  $e^* \leftarrow \mathsf{Decrypt}_1(sk; (\widehat{r}, \widehat{s}))$
  $\widetilde{e} \leftarrow \mathsf{GenErrVec}(n, t, r)$ // $r \leftarrow sk$
  if $e^* = \perp$ **then**
    $\kappa \leftarrow \mathsf{RO}_H(\widetilde{e} \| \widehat{r} \| \widehat{s})$
  else
    $\kappa \leftarrow \mathsf{RO}_H(e^* \| \widehat{r} \| \widehat{s})$
  end if
  return κ
end procedure

Chapter 6

Conclusion

In this proposal, we introduce PALOMA, an IND-CCA2-secure KEM based on an SDP with a binary separable Goppa code. While the components and mechanisms used in PALOMA have been studied for a long time, no critical attacks have been found. Many cryptographic communities believe that the scheme constructed by these is secure. The Classic McEliece, which is the round 4 cipher of the NIST PQC competition, was also designed based on similar principles [4]. Both PALOMA and Classic McEliece have similar public key sizes. However, PALOMA is designed with a focus on deterministic algorithms for constant-time operations, making it more efficient in terms of implementation speed compared to Classic McEliece. We give the feature comparison between PALOMA and Classic McEliece in Tab. 6.1.

Table 6.1: Comparison between PALOMA and Classic McEliece

	PALOMA	Classic McEliece
Structure	Fujisaki-Okamoto-structure KEM (implicit rejection)	SXY-structure KEM (implicit rejection)
Problem	SDP	SDP
Trapdoor type	Niederreiter	Niederreiter
Finite Field $\mathbb{F}_{q^m}$	$\mathbb{F}_{2^{13}}$	$\mathbb{F}_{2^{12}}$, $\mathbb{F}_{2^{13}}$
Linear code $\mathcal{C}$	Binary separable Goppa code	Binary irreducible Goppa code
Goppa polynomial $g(X)$	Separable (not irreducible)	Irreducible
Time for generating $g(X)$	Constant	Non-constant
Parity-check matrix $\mathbf{H}$ of $\mathcal{C}$	**ABC**	**BC**
Parity-check matrix $\widehat{\mathbf{H}}$ of $\widehat{\mathcal{C}}$	Systematic form	Systematic form
Decoding algorithm	Extended Patterson	Berlekamp-Massey
Probability of decryption failure (correctness)	0	0

A primary role of post-quantum cryptography is to serve as an alternative to current cryptosystems that are vulnerable to quantum computing attacks. Therefore, we have designed PALOMA with a conservative approach, and thus, we firmly believe that PALOMA can serve as a dependable alternative to existing cryptosystems in the era of quantum computers.

Bibliography

[1] Becker, A., Joux, A., May, A., Meurer, A.: Decoding random binary linear codes in 2n/20: How $1+1=0$ improves information set decoding. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology – EUROCRYPT 2012. pp. 520–536. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[2] Berlekamp, E.: Nonbinary bch decoding (abstr.). IEEE Transactions on Information Theory **14**(2), 242–242 (1968). https://doi.org/10.1109/TIT.1968.1054109

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

[4] Bernstein, D., Chou, T., Lange, T., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Schwabe, P., Sendrier, N., Szefer, J., Wang, W.: Classic mceliece (2017)

[5] Bernstein, D.J., Lange, T., Peters, C.: Attacking and defending the mceliece cryptosystem. In: Buchmann, J., Ding, J. (eds.) Post-Quantum Cryptography. pp. 31–46. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

[6] Bernstein, D.J., Lange, T., Peters, C.: Smaller decoding exponents: Ball-collision decoding. In: Rogaway, P. (ed.) Advances in Cryptology – CRYPTO 2011. pp. 743–760. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[7] Bezzateev, S.V., Noskov, I.K.: Patterson algorithm for decoding separable binary goppa codes. In: 2019 Wave Electronics and its Application in Information and Telecommunication Systems (WECONF). pp. 1–5 (2019). https://doi.org/10.1109/WECONF.2019.8840650

[8] Bezzateev, S., Shekhunova, N.: Totally decomposed cumulative goppa codes with improved estimations. Designs, Codes and Cryptography **87**(2) (Mar 2019)

[9] Canteaut, A., Chabanne, H., national de recherche en informatique et en automatique (France). Unité de recherche Rocquencourt, I.: A Further Improvement of the Work Factor in an Attempt at Breaking McEliece's Cryptosystem. Rapports de recherche, Institut national de recherche en informatique et en automatique (1994), `https://books.google.co.kr/books?id=QMuRHAAACAAJ`

[10] Faugère, J.C., Gauthier-Umanã, V., Otmani, A., Perret, L., Tillich, J.P.: A distinguisher for high rate mceliece cryptosystems. In: 2011 IEEE Information Theory Workshop. pp. 282–286 (2011). https://doi.org/10.1109/ITW.2011.6089437

[11] Finiasz, M., Sendrier, N.: Security bounds for the design of code-based cryptosystems. In: Matsui, M. (ed.) Advances in Cryptology – ASIACRYPT 2009. pp. 88–105. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

[12] Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology. p. 537–554. CRYPTO '99, Springer-Verlag, Berlin, Heidelberg (1999)

[13] Goppa, V.D.: A new class of linear error-correcting codes. Probl. Inf. Transm. **6**, 300–304 (1970)

[14] Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the fujisaki-okamoto transformation. In: Kalai, Y., Reyzin, L. (eds.) Theory of Cryptography. pp. 341–371. Springer International Publishing, Cham (2017)

[15] Karp, R.M.: Reducibility among Combinatorial Problems, pp. 85–103. Springer US, Boston, MA (1972), https://doi.org/10.1007/978-1-4684-2001-2_9

[16] Kim, D.C., Hong, D., Lee, J.K., Kim, W.H., Kwon, D.: Lsh: A new fast secure hash function family. In: Lee, J., Kim, J. (eds.) Information Security and Cryptology - ICISC 2014. pp. 286–313. Springer International Publishing, Cham (2015)

[17] Knuth, D.E.: The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms. Addison-Wesley Longman Publishing Co., Inc., USA (1997)

[18] Lee, P.J., Brickell, E.F.: An observation on the security of mceliece's public-key cryptosystem. In: Barstow, D., Brauer, W., Brinch Hansen, P., Gries, D., Luckham, D., Moler, C., Pnueli, A., Seegmüller, G., Stoer, J., Wirth, N., Günther, C.G. (eds.) Advances in Cryptology — EUROCRYPT '88. pp. 275–280. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)

[19] Leon, J.S.: A probabilistic algorithm for computing minimum weights of large error-correcting codes. IEEE Transactions on Information Theory **34**(5), 1354–1359 (1988)

[20] Massey, J.: Shift-register synthesis and bch decoding. IEEE Transactions on Information Theory **15**(1), 122–127 (1969). https://doi.org/10.1109/TIT.1969.1054260

[21] May, A., Meurer, A., Thomae, E.: Decoding random linear codes in $o(2^{0.054n})$. In: Lee, D.H., Wang, X. (eds.) Advances in Cryptology – ASIACRYPT 2011. pp. 107–124. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[22] May, A., Ozerov, I.: On computing nearest neighbors with applications to decoding of binary linear codes. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. pp. 203–228. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)

[23] McEliece, R.J.: A Public-Key Cryptosystem Based On Algebraic Coding Theory. Deep Space Network Progress Report **44**, 114–116 (Jan 1978)

[24] Minder, L., Shokrollahi, A.: Cryptanalysis of the sidelnikov cryptosystem. In: Naor, M. (ed.) Advances in Cryptology - EUROCRYPT 2007. pp. 347–360. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)

[25] Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. In: Problems of Control and Information Theory 15. pp. 159–166 (1986)

[26] Patterson, N.: The algebraic decoding of goppa codes. IEEE Trans. Inf. Theor. **21**(2), 203–207 (Sep 2006). https://doi.org/10.1109/TIT.1975.1055350, http://dx.doi.org/10.1109/TIT.1975.1055350

[27] Prange, E.: The use of information sets in decoding cyclic codes. IRE Transactions on Information Theory **8**(5), 5–9 (1962)

[28] SIDELNIKOV, V.M., SHESTAKOV, S.O.: On insecurity of cryptosystems based on generalized reed-solomon codes. Discrete Mathematics and Applications **2**(4), 439 – 444 (1992). https://doi.org/https://doi.org/10.1515/dma.1992.2.4.439, `https://www.degruyter.com/view/journals/dma/2/4/article-p439.xml`

[29] Stern, J.: A method for finding codewords of small weight. In: Cohen, G., Wolfmann, J. (eds.) Coding Theory and Applications. pp. 106–113. Springer Berlin Heidelberg, Berlin, Heidelberg (1989)

[30] Targhi, E.E., Unruh, D.: Post-quantum security of the fujisaki-okamoto and oaep transforms. In: Hirt, M., Smith, A. (eds.) Theory of Cryptography. pp. 192–216. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)

Appendix A

SAGE code for a Binary Separable Goppa code used in PALOMA

```
'''

Copyright 2024 FDL(Future cryptograph Design Laboratory, Kookmin University

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the Software
is furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES
OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE
OR OTHER DEALINGS IN THE SOFTWARE.

'''

'''
    Sage Code for Binary Separable Goppa Codes used in PALOMA
    Developed by KMU/FDL
    2024.02.23.
'''

'''
    F2m = GF(2^13) (i.e., m = 13)
    Separable Goppa Polymoial g(X) with degree t in F2m[X]
    (t-error correctable code)

    n + t <= q^m = 2^13 = 8192
    k >= n - mt = n - 13t

    [PALOMA parameter sets]
    PALOMA128:
        n = 3904( 61), k = 3072, n-k =  832(13), m = 13, t =  64,
```

```
        f = z^13 + z^7 + z^6 + z^5 + 1
    PALOMA192:
        n = 5568( 87), k = 3904, n-k = 1664(26), m = 13, t = 128,
        f = z^13 + z^7 + z^6 + z^5 + 1
    PALOMA256:
        n = 6592(103), k = 4928, n-k = 1664(26), m = 13, t = 128,
        f = z^13 + z^7 + z^6 + z^5 + 1

    [Toy parameters]
    n =  37, k =  19, n-k =  18, t =  3, m =  6, f = z^6 + z^4 + z^3 + z + 1
    n = 100, k =  72, n-k =  28, t =  4, m =  7, f = z^7 + z + 1
    n = 120, k =  64, n-k =  56, t =  8, m =  7, f = z^7 + z + 1
    n = 241, k = 121, n-k = 120, t = 15, m =  8, f = z^ 8 + z^4 + z^3 + z^2 + 1
    n =  53, k =  27, n-k =  26, t =  2, m = 13, f = z^13 + z^7 + z^6 + z^5 + 1
    n =  79, k =  40, n-k =  39, t =  3, m = 13, f = z^13 + z^7 + z^6 + z^5 + 1
'''

reset ()
var('z,X')

import random

#####################################################

def separator():    print("=======================================")
def separator2():    print("---------------------------------------")

#####################################################

#####################################################
# parameters: n, t, m, irr_poly
#####################################################

paloma_param_set = [
    [3904,  64, 13, z^13 + z^7 + z^6 + z^5 + 1], # PALOMA128
    [5568, 128, 13, z^13 + z^7 + z^6 + z^5 + 1], # PALOMA192
    [6592, 128, 13, z^13 + z^7 + z^6 + z^5 + 1], # PALOMA256

    [ 37,  3, 6, z^6 + z^4 + z^3 + z + 1],
    [100,  4, 7, z^7 + z + 1],
    [120,  8, 7, z^7 + z + 1],
    [241, 15, 8, z^8 + z^4 + z^3 + z^2 + 1],

    [53, 2, 13, z^13 + z^7 + z^6 + z^5 + z^0],
    [79, 3, 13, z^13 + z^7 + z^6 + z^5 + z^0],

    [216,  8, 13, z^13 + z^7 + z^6 + z^5 + 1],
    [424, 16, 13, z^13 + z^7 + z^6 + z^5 + 1]
]

#####################################################
# Set Parameter
#####################################################

def SetParameter(param_num):
    n, t, m, f = paloma_param_set[param_num]
    k = n - m*t
    R2.<z> = GF(2)[]
    F2m.<z> = GF(2^m, modulus = R2(f))
    R2m.<X> = PolynomialRing(F2m)

```

```
    separator()
    print(f"C=[{n},{k},>=2*{t}+1], m={m}")

    return n, k, t, m, F2m, R2m

##################################################
# function for hex representation
##################################################

def str_f2m_hex(x, F2m):
    return "0x{:04x}".format(ZZ(list(F2m(x).polynomial()), base = 2))
#    return hex(ZZ(list(F2m(x).polynomial()), base = 2))

def show_mat_hex(m, F2m):
    nrows, ncols = m.nrows(), m.ncols()
    for r in range(0, nrows):
        str = "["
        for c in range(0, ncols):
            if c%16 == 0:
                str += "\n"
            str += str_f2m_hex(m[r][c], F2m) + " "
        print (str , "\n]")

def show_poly_hex(f, F2m):
    show_mat_hex(matrix(list(f)), F2m)

def support_set(bin_vec):
    return [index for index, value in enumerate(bin_vec) if value == 1]

##################################################
# Generation of Random Goppa Code
##################################################

def GenGoppaCode(n, t, m, F2m, R2m):
    print("C is generated by L and g.")

    listF2m = list(F2m)
    mbitset = list(range(0,2^m,1))
    random.shuffle(mbitset)

    separator()
    # Support set L
    print("L is generating.. ")
    L = [listF2m[j] for j in mbitset[:n]];
    show_mat_hex(matrix(L), F2m);

    separator2()
    # Separable Goppa polynomial g(X)
    print("g(X) is generating.. ")
    g = prod([(R2m.parameter()+listF2m[j]) for j in mbitset[n:n+t]]);
    show_poly_hex(g, F2m); print("");

    separator2()
    # Matrix A, B, C
    print("A is generating..")
    coeffg = list(g) + [0]*(t-1)
    A = matrix([coeffg[i:i+t] for i in [1..t]]);

    separator2()
    print("B is generating..")
    B = matrix(F2m, t, n, lambda r, c: (L[c]^r));
```

```
163
164     separator2()
165     print("A*B is computing..")
166     H = A*B;
167
168     separator2()
169     # Matrix H = A*B*C
170     print("H=(A*B)*C is computing..")
171     for ind in [0..n-1]:
172         tt = g(L[ind])^-1
173         H.set_column(ind, H.column(ind) * tt)
174     #print("H =\n")
175     #show_mat_hex(H, F2m); print("");
176
177     return L, g, H
178
179 ##################################################
180
181 ''' Given f s.t gcd(f,g) = 1, find f^-1 such that f^-1*f = 1 (mod g) '''
182
183 def getInv(f, g, R2m):
184     t = g.degree()
185     d0, d1 = R2m(f), R2m(g)
186     a0, a1 = R2m(1), R2m(0)
187
188     while d1 != 0:
189         q, r = d0.quo_rem(d1)
190         d0, d1 = d1, r
191         a2 = a0 - q*a1
192         a0, a1 = a1, a2
193
194     return a0*d0.leading_coefficient()^-1
195
196
197 ''' Find a2, b1 such that b1*s_hat = a2 (mod g12) with deg condition '''
198
199 def EEA_for_keyeqn(s_hat, g12, dega, degb, R2m):
200     a0, a1 = R2m(s_hat), R2m(g12)
201     b0, b1 = R2m(1), R2m(0)
202
203     while a1 != 0:
204         q, r = a0.quo_rem(a1)
205         a0, a1 = a1, r
206         b2 = b0 - q*b1
207         b0, b1 = b1, b2
208
209         if a0.degree() <= dega and b0.degree() <= degb:
210             break
211
212     return a0 , b0
213
214
215 ''' Compute Square Root of f(X) mod g12(X) '''
216
217 def get_sqrt(f, g, m, R2m):
218     sqrtx = power_mod(R2m.parameter(), 2^(m-1), g) # precomputable value
219     degf = R2m(f).degree()
220     listf = list(f)
221     fe = [sqrt(listf[2*j]) for j in [0..floor(degf/2)]]
222     fo = [sqrt(listf[2*j+1]) for j in [0..floor((degf-1)/2)]]
223     sqrtf = (R2m(fe) + R2m(fo)*sqrtx)%g
```

```
224
225     return sqrtf
226
227
228 ''' Given f, find a(X), b(X) such that f = a^2(X) + b^2(X)*X '''
229
230 def get_a2b2x(f, R2m):
231     degf = R2m(f).degree()
232     listf = list(f)
233     fe = [sqrt(listf[2*j]) for j in [0..floor(degf/2)]]
234     fo = [sqrt(listf[2*j+1]) for j in [0..floor((degf-1)/2)]]
235     a, b = R2m(fe), R2m(fo)
236
237     return a, b
238
239 ####################################################
240 # Generate Random Error Vector with Hamming Weight t
241 ####################################################
242
243 def GenErrVec(n,t):
244     nset = list(range(0,n))
245     shuffle(nset)
246
247     e = [0]*n
248     for i in nset[0:t]:
249         e[i] = 1
250
251     return e
252
253 ####################################################
254 # Construct Key Equation
255 ####################################################
256
257 def ConstructKeyEqn(s, g, m, R2m):
258     s_ast = R2m(1) + R2m.parameter()*s
259     g1, g2 = gcd(g, s), gcd(g, s_ast)
260     g12 = R2m(g/g1/g2)
261     s2_ast, s1 = R2m(s_ast/g2), R2m(s/g1)
262
263     u = getInv(g2*s1, g12, R2m)
264     u = (g1 * s2_ast * u)%g12
265     v = get_sqrt(u, g12, m, R2m)
266
267     return v, g1, g2, g12
268
269 ####################################################
270 # Solve Key Equation
271 ####################################################
272
273 def SolveKeyEqn(v, g12, dega, degb, R2m):
274     a0, b0 = EEA_for_keyeqn(v, g12, dega, degb, R2m)
275     return a0, b0
276
277 ####################################################
278 # Find Error Vector
279 ####################################################
280
281 def FindErrVec(sigma, L, n):
282     err_support_set = []
283     for i in [0..n-1]:
284         if sigma(L[i]) == 0:
```

```
285             err_support_set += [i]
286
287     err_vector = [0]*n
288     for i in err_support_set:
289         err_vector[i] = 1
290
291     return err_vector
292
293 ####################################################
294 # Recovery Error Vector
295 ####################################################
296
297 def DecodeExtPatterson(n, t, L, g, s, m, R2m):
298     s = R2m(list(s))
299     # print (f"gcd(s, g)= {gcd(s,g)}\n")
300
301     # print (f"s(X) is irreducible? {s.is_irreducible()}")
302     # factors = factor(s)
303     # s_linear_factors = [factor[0] for factor in factors if factor[0].degree() ==
        1]
304     # print(s_linear_factors)
305     # print (f"gcd(s, g)= {gcd(s,g)}")
306     # print (s.factor())
307
308     separator2()
309     print("Constructing Key Equation..")
310     v, g1, g2, g12 = ConstructKeyEqn(s, g, m, R2m);
311
312     separator2()
313     print("Solving Key Equation..")
314     a2, b1 = SolveKeyEqn(v, g12, floor(t/2)-g2.degree(), floor((t-1)/2)-g1.degree()
        , R2m);
315     a, b = a2*g2, b1*g1
316     sigma = (a^2 + b^2*R2m.parameter()).monic()
317
318     separator2()
319     print("Finding Zeros of Error Locator Polynomial..")
320     rec_e = FindErrVec(sigma, L, n);
321
322     separator2()
323     print("Recovered error polynomial rec_e(X)")
324     print(R2m(rec_e))
325
326     return rec_e
327
328 ####################################################
329
330 def DoPALOMA(param_num):
331     n, k, t, m, F2m, R2m = SetParameter(param_num)
332
333     ####################################################
334     # Generating Goppa code
335     ####################################################
336
337     separator()
338     L, g, H = GenGoppaCode(n, t, m, F2m, R2m);
339
340     ####################################################
341     # Generating t-Hamming weight Error Vector
342     ####################################################
343
```

```
    separator()
    print("Generating t-Hamming weight Error Vector e..")
    e = GenErrVec(n, t);
    print("supp(e)")
    print(support_set(e))
    print("error polynomial e(X)")
    print(R2m(e))

    ####################################################
    # Decoding by Extended Patterson Decoding
    ####################################################

    separator()
    print("Computing the Syndrome Vector s of e..")
    s = H*vector(e)
    #print(s)
    show_mat_hex(matrix(s), F2m)

    separator()
    print("Recovering Error Vector e from s..")
    rec_e = DecodeExtPatterson(n, t, L, g, R2m(list(s)), m, R2m);
    print("supp(rec_e)")
    print(support_set(rec_e))

    separator()
    print(f"e == rec_e ? {e == rec_e}");

####################################################

DoPALOMA(0)
DoPALOMA(1)
DoPALOMA(2)
```