Protecting Kyber and Saber Against Side-channel Analysis

Comparison and New Constructions

Master thesis by Yulia Kuzovkova (Student ID: 2720977)
Date of submission: January 30, 2022

1. Review: Prof. Dr. Juliane Krämer
2. Review: M.Sc. Thomas Aulbach
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

IT Security

Quantum and Physical attack resistant Cryptography

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Yulia Kuzovkova, die vorliegende Masterarbeit gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 30. Januar 2022

Y. Kuzovkova

Abstract

Quantum computers running Shor's algorithm will be able to break most current public-key cryptosystems in the near future. To address this threat, the National Instutute of Standards and Technology (NIST) has initiated a standardization competition for selecting a new generation of public-key primitives secure in the post-quantum world. Recently, three lattice-based Key Encapsulation Mechanisms (KEMs) were selected for the third and final round of the NIST post-quantum cryptography (PQC) standardization process. While the theoretical security of these schemes has been well understood, their practical security - for instance, the resistance against side-channel attacks and the overhead of deploying the countermeasures in the implementations still remains an open question.

In this thesis we aim to bridge this gap and investigate the cost of protecting embedded implementations of the two main candidates for standardization - lattice-based KEMs Kyber and Saber. First, we review the state-of-the art masking techniques for both schemes and propose a new countermeasure for protecting one of the main building blocks of Kyber - the comparison. Next, the most promising configurations of countermeasures are implemented and optimized for the 32-bit ARM Cortex-M4 architecture. Finally, we evaluate and compare the performance and code size impacts of masking for Kyber and Saber on a common platform.

Contents

List of Algorithms

List of Figures

List of Listings

List of Tables

1 Introduction

1.1 Motivation

In the last decades, the technological advancements in the semiconductor industry made the mass expansion of small embedded devices possible. From smart cards to networking sensors to mobile phones, these devices continuously generate and exchange sensitive data, communicating both locally with peers as well as globally over the Internet. Securing these communications as well as the embedded products themselves has become a challenging task for software designers, as due to the device's accessibility not only are remote attack scenarios practical, but also the devices themselves can be tampered with. These risks can be mitigated by deploying security protocols based on public-key cryptography like TLS, IPSec/IKE or DNSSEC to ensure authentication, privacy, data integrity and confidentiality.

Today, most applications and security protocols deploy RSA- and ECC-based public-key cryptosystems [69, 57] relying on hardness of number-theoretic assumptions, such as integer factorization and discrete logarithm. With the parameters chosen to be sufficiently large, these schemes are believed to be secure in the presence of classical computers, as exponential time is required for breaking the underlying assumptions. However, an algorithm invented by Peter Shor in 1994 allows to factor integers and compute a discrete logarithm efficiently in polynomial time, given a powerful quantum computer with a sufficiently large number of qubits [72]. While initially quantum computers were considered merely a theoretical construct, continuous advancements in physics and quantum computing technology [6, 77] made the quantum threat real and prompted the standardization bodies and the cryptographic community towards developing a new generation of cryptographic algorithms to replace potentially vulnerable RSA- and ECC-based schemes.

Back in 2016, the National Institute of Standards and Technology (NIST) has initiated a standardization competition for evaluating quantum-resistant public-key primitives, such as public-key encryption (PKE), key encapsulation mechanisms (KEM), and digital signatures (DS) [28]. For the first round 69 proposals were submitted, out of which 26 algorithms made their way into a second round [2]. These algorithms are based on five families of quantum-hard cryptographic problems - lattices, linear codes, hash functions, multivariate quadratic equations and isogenies. In October 2020, 7 finalists and 8 alternate candidates were selected for the third round, the majority of which are lattice-based [58]. It is expected that in early 2022 the winners will be announced which become the new standard. While the theoretical security and the implementation aspects of the candidate algorithms were the main selection criteria in the first two rounds of the competition, the main focus of the final round is on the side-channel and fault attack resistance of the schemes. In particular, the practical aspects, such as the cost and efficiency of deploying the side-channel countermeasures in the implementations, require further investigation. Our work aims to contribute to NIST's efforts by evaluating the impact of efficient side-channel countermeasures on the two most promising candidates for standardization - lattice-based KEMs Kyber [23] and Saber [34] - and providing a fair comparison of both schemes.

1.2 Contributions

This thesis investigates the cost of including efficient side-channel countermeasures into embedded implementations of Kyber and Saber KEMs, which are finalists in the third round of the NIST standardization competition. Our main contributions are as follows:

- We propose a novel countermeasure for a polynomial comparison, which is one of the core building blocks of Kyber. Our new masked comparison is based on a bitsliced binary search method; it compares the original compressed ciphertext with the re-encrypted uncompressed ciphertext and requires one arithmetic-to-Boolean conversion per ciphertext coefficient.
- We efficiently implement and compare three configurations of masking countermeasures for polynomial comparison and compression components of Kyber, proposed by authors of [37] and [24] and in this work. We optimize the first-order masked implementations for the ARM Cortex-M4 platform and evaluate the performance for the first- and higher-order masking.
- We present an efficient first-order masked implementation of the complete decapsulation operation of Kyber, given three configurations of countermeasures. To provide a fair comparison with the already existing side-channel resistant implementation of Saber from [12], we optimize the same building blocks of Kyber in ARM Cortex-M4 assembly, as were optimized in [12]. We evaluate the masked implementations on a common platform - the STM32F407 evaluation board - and analyze the performance and the code size as well as assess the overhead introduced by side-channel countermeasures for both lattice-based KEMs.

This allows us to fairly compare both KEMs regarding their cost of side-channel protection and derive a conclusion regarding their efficiency.

1.3 Structure

The remainder of this thesis is organized as follows:

- In Chapter 2 we describe the theoretical aspects used in our work. After introducing the notations, we briefly introduce the side-channel attacks and the possible countermeasures, and provide the necessary background on key encapsulation mechanisms (KEMs) Kyber and Saber.
- Once the relevant theory has been covered, Chapter 3 reviews state-of-the art masking techniques proposed in the literature for protecting the core building blocks of both KEMs.
- Next, in Chapter 4 we propose a new masking countermeasure for one of the most vulnerable components of Kyber - the polynomial comparison. Our new approach is based on a bitsliced binary search method and allows to test uncompressed masked polynomials and compressed public polynomials for equality in a secure fashion.

- In Chapter 5, we provide an efficient first-order masked implementation of Kyber optimized for ARM Cortex-M4 architecture. In particular, we implement and analyze different configurations of countermeasures for polynomial comparison and compression and estimate their impacts in the higher-order scenario. Following, we analyze the practical results of masked implementations of Kyber and Saber on a common platform - the STM32F407 evaluation board.
- Finally, in Chapter 6 we conclude, giving a summary of our findings and suggesting further improvements and future work.

2 Background

This chapter describes the theoretical background necessary for understanding our work. We begin by introducing the notations that will be used throughout the thesis in Section 2.1. Next, in Section 2.2 we give a brief introduction to side-channel attacks and countermeasures, such as masking and shuffling. Then, in Sections 2.3 and 2.4 we review the cryptographic primitives and the hard problems Kyber and Saber are based on. Finally, in Section 2.5 we describe how to use these building blocks to construct IND-CCA-secure key encapsulation mechanisms Kyber and Saber.

2.1 Notation

Polynomial rings, vectors and matrices. Let $\mathbb{Z}_q$ denote the ring of integers modulo q. We denote by R and R_q the rings of polynomials $\mathbb{Z}[X]/(\phi_n(x))$ and $\mathbb{Z}_q[X]/(\phi_n(x))$ respectively, where $\phi_n(x) = \sum_{k=0}^{n-1} x^k$ is the n-th cyclotomic polynomial. For any ring R, let $R^{l_1 \times l_2}$ be the ring of $l_1 \times l_2$ matrices over R. We use regular font letters (e.g. x) to denote elements in R or R_q; bold lower-case letters (e.g. $\boldsymbol{b}$) represent vectors and bold upper-case letters (e.g. $\boldsymbol{A}$) represent matrices with coefficients in R and R_q. We use subscript (e.g. x_i) to access the i-th coefficient of a polynomial $x \in R_q$ or the i-th bit of a bitstring $x \in \mathbb{Z}_2^k$.

Modular reductions. For an integer z, we denote $z' = z \bmod q$ the modular reduction of z in $[0, q)$ and $z' = z \bmod^{\pm} q$ the centered modular reduction of z in $[-\frac{q-1}{2}, \frac{q-1}{2}]$.

Rounding. For an element $x \in \mathbb{Q}$, let $\lfloor x \rfloor$ be the flooring operation, which returns the greatest integer smaller or equal to x and let $\lfloor x \rceil$ be the rounding operation that rounds x to the closest integer with ties being rounded up, i.e. $\lfloor x \rceil = \lfloor x + 0.5 \rfloor$.

Shifting. We denote by $x \ll b$ and $x \gg b$ shifting an integer x by b positions to the left and to the right respectively, which corresponds to $x \cdot 2^b$ and $\lfloor x/2^b \rfloor$. For polynomial and matrices, the shifting operation is applied coefficient-wise.

Distributions. Let $x \leftarrow \chi$ denote sampling of x according to a distribution χ. Sampling of matrices of polynomials is represented by $\boldsymbol{X} \leftarrow \chi(R^{l_1 \times l_2})$, where all the coefficients of $\boldsymbol{X}$ are sampled independently from the distribution χ. In some cases, we explicitly specify the randomness σ, used to generate the distribution: $\boldsymbol{X} \leftarrow \chi(R^{l_1 \times l_2}; \sigma)$ indicates that all the coefficients of $\boldsymbol{X}$ are sampled from the distribution χ given seed σ. The uniform distribution is denoted by $\mathcal{U}$.

Binomial distribution. We denote the centered binomial distribution as β_η, for a positive integer η. We define β_η as $\sum_{i=0}^{\eta-1}(x_i - y_i)$, where $\{(x_i, y_i)\}_{i=0}^{\eta-1} \leftarrow \mathcal{U}(\{0,1\}^{2\eta})$.

Masking. A sensitive variable x split into s shares is denoted as $x^{(\cdot)}$. We denote a Boolean sharing of $x \in \mathbb{Z}_2^k$ as $x^{(\cdot)_B}$, where $x^{(i)_B} \in \mathbb{Z}_2^k$ is the i-th share of x for $0 \leq i < s$ and $x^{(0)_B} \oplus ... \oplus x^{(s-1)_B} = x$. In case arithmetic masking is used, we denote an arithmetic sharing of $x \in \mathbb{Z}_q$ as $x^{(\cdot)_A}$, where $x^{(i)_A} \in \mathbb{Z}_q$ denotes the i-th share of x for $0 \leq i < s$ and $x^{(0)_A} + ... + x^{(s-1)_A} = x \bmod q$.

2.2 Side-channel Attacks and Countermeasures

In the recent years, side-channel analysis (SCA) has shown to be a very powerful mean for attacking cryptographic implementations. It is a class of physical attacks that exploit physical properties of the target device, such as timing behavior [52], power consumption [53], electromagnetic (EM) emanation [39]. When correlated with the internal state of the device, these implementation-specific characteristics could leak information on the executed instructions, intermediate state of the algorithm and manipulated data. Consequently, by using standard statistical tools, a side-channel attacker may extract the secrets and break unprotected implementations of cryptographic algorithms with minimal effort.

In this section, we briefly review the most common attack types and introduce countermeasures which help mitigate these attacks.

2.2.1 Attacks

Since the first publication by Kocher [52], various types of side-channel attacks were proposed in the literature:

Timing analysis. If cryptographic operations take different amount of time depending on the value of input and the secret parameters, those timing differences can be exploited to recover the sensitive data involved in the computations. Conditional branches [52], cache memory accesses [62] or CPU optimizations (for instance, branch prediction [55] and speculative execution techniques [51]) can result in variable time behavior which can be potentially misused. Usual countermeasures consist of eliminating data-dependent memory accesses and branches, based on sensitive data and verifying that all operations run in constant time.

Simple analysis. In this type of attack, the secret information is obtained by examining a pattern of the single power or electromagnetic trace [53, 39]. Since different instructions exhibit different power or electromagnetic leakages, one can determine what kind of operation is executed at a given time. This poses a serious threat if a sequence of operations depends on a secret key bit value, as the secret key can then be read directly from the trace.

Differential analysis. Compared to simple analysis, differential analysis [53] is a more powerful type of attack. This approach requires multiple power or electromagnetic traces to identify data-dependent correlations. For each key candidate, the set of traces is divided into two subsets, based on the hypothetical intermediate values. If the sets are correlated, then the difference of the averages of these subsets will be non-zero and can be identified from the peaks in the differential trace. Given enough traces, the resulting peaks will identify the correct key hypotheses.

Template analysis. Template analysis [26] is a class of profiled side-channel attacks that works in two stages. During profiling stage, a large number of traces are recorded, using a copy of the target device.

Dependent on the number of points of interests, the leakage at each point is modeled according to the Gaussian or multivariate distribution. The parameters of the distribution are then estimated for each possible value of the target variable. In the attack stage, the traces from the target device are compared to the template traces, based on the principle of maximum likelihood. This results in a set of most probable secret keys.

2.2.2 Countermeasures

Protection against side-channel attacks has become an ultimate goal for designers of cryptographic applications. A general countermeasure against timing analysis is a constant-time execution. By ensuring that all operations are independent on the sensitive variables and take the same amount of time, these attacks can be prevented. However, deployment of additional countermeasures or combinations of countermeasures is essential to thwart more sophisticated kinds of attacks. In the following, we introduce two commonly used techniques - masking [1, 27, 41] and shuffling [43].

Masking. A popular method for preventing side-channel analysis is secret sharing, or masking. The key idea is to make information leakages statistically independent of secrets by splitting each sensitive intermediate value x into s random shares $x_0, ..., x_{s-1}$, satisfying $x = x_0 \odot ... \odot x_{s-1}$, and executing the algorithm on the randomized inputs. By doing so, an attacker requires to combine at least s shares to reconstruct the secret; all the sets of less than s shares leak no sensitive information, which was formalized as probing security in the seminal work of Ishai et al [46]. The group operation $\odot$ depends on the type of masking, for instance arithmetic masking relies on the modular addition and Boolean masking utilizes exclusive OR (XOR) as the group operation. Typically, the shares $x_1, ..., x_{s-1}$ are generated by sampling $s - 1$ random masks, and the remaining share is computed as $x_0 = x - x_1 - ... - x_{s-1}$ for arithmetic masking or $x_0 = x \oplus x_1 \oplus ... \oplus x_{s-1}$ for Boolean masking. The value $s - 1$ is called a masking order, which can be viewed as a security parameter.

Shuffling. In contrast to masking, shuffling does not randomize the intermediate values, hence the secret-dependent operations still leak sensitive information. Instead, the execution time is randomized by generating a random permutation, and changing the order of the group of identical operations according to the generated permutation. This increases the measurement noise and reduces the signal-to-noise ratio (SNR) of the operations, therefore a strongly increased number of measurements is required to launch a successful attack.

2.3 Cryptographic Definitions

This section introduces two core cryptographic primitives used in public-key cryptosystems: public-key encryption (PKE) and key-encapsulation mechanism (KEM).

2.3.1 Public-Key Encryption

A public-key encryption scheme (PKE) [32] is a triple of probabilistic polynomial-time algorithms (KeyGen, Enc, Dec) defined over the key space $\mathcal{K}$, the message space $\mathcal{M}$ and the ciphertext space $\mathcal{C}$.

1. KeyGen(): a key-generation algorithm that returns a pair of keys $(pk, sk) \in \mathcal{K}$, where pk is a public key and sk is the corresponding private key.

2. $\mathsf{Enc}(pk, m)$: a probabilistic encryption algorithm that constructs a ciphertext $c \in \mathcal{C}$, given a public key pk and a message $m \in \mathcal{M}$.
3. $\mathsf{Dec}(sk, c)$: a deterministic decryption algorithm takes the private key sk and a ciphertext $c \in \mathcal{C}$ and outputs either a message $m' \in \mathcal{M}$ upon successful decryption, or a failure symbol $\perp$ indicating rejection.

Security

The security of public-key encryption is defined in the sense of indistinguishability under chosen-plaintext attacks (IND-CPA) [13]. Formally, security in terms of indistinguishability is presented as a cryptographic game [73, 14], where a cryptosystem is considered secure, if no adversaries can win the game with the probability significantly greater than of random guessing. Let A be a probabilistic polynomial-time adversary, that runs in two stages and aims to win the $\mathsf{IND\text{-}CPA}^{\mathsf{A}}_{\mathsf{PKE}}$ game, described below. In a first stage, A is given access to an encryption oracle $\mathsf{Enc}()$ to encrypt arbitrary (polynomially bounded) number of messages of its choice. In the second stage, A submits two distinct fresh messages m_0, m_1, and gets an encryption of one of the messages, c_b. The adversary's goal is to decide which message m_b is encrypted in a given ciphertext:

```
Game $\mathsf{IND\text{-}CPA}^{\mathsf{A}}_{\mathsf{PKE}}$ :
    $(pk, sk) \leftarrow \mathsf{KeyGen}()$
    $b \leftarrow \{0, 1\}$
    $(m_0, m_1) \leftarrow \mathsf{A}(pk)$
    $c_b \leftarrow \mathsf{Enc}(pk, m_b)$
    $b' \leftarrow \mathsf{A}^{\mathsf{Enc}()}(pk, c_b)$
    return $b \overset{?}{=} b'$.
```

A PKE is considered IND-CPA-secure, if for all efficient adversaries A there exists some negligible function $\mathsf{negl}(n)$ of the security parameter n, such that the advantage of A in winning the $\mathsf{IND\text{-}CPA}^{\mathsf{A}}_{\mathsf{PKE}}$ game is given by:

$$\textbf{Adv}\,^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathsf{A}) = \left| \Pr\left(\mathsf{IND\text{-}CPA}^{\mathsf{A}}_{\mathsf{PKE}} = 1\right) - \frac{1}{2} \right| < \mathsf{negl}(n).$$

Correctness

The PKE is called δ-correct [44], if for all (even unbounded) adversaries A, the probability of decryption failure is smaller or equal to δ, namely

$$\mathbb{E}[\max_{m \in \mathcal{M}} \Pr\left[\mathsf{Dec}(sk, c) \neq m \mid c \leftarrow \mathsf{Enc}(pk, m)\right]] \leq \delta,$$

where the expectation is taken over the randomness of $(pk, sk) \leftarrow \mathsf{KeyGen}()$.

2.3.2 Key-Encapsulation Mechanism

A key-encapsulation mechanism (KEM) [32] is a triple of probabilistic polynomial-time algorithms (KeyGen, Encaps, Decaps) together with the key space $\mathcal{K}$, the message space $\mathcal{M}$ and the ciphertext space $\mathcal{C}$, defined as follows:

1. KeyGen(): a key-generation algorithm that returns a pair of keys $(pk, sk) \in \mathcal{K}$, where pk is a public key and sk is the corresponding private key.
2. Encaps(pk): a probabilistic encapsulation algorithm that produces a ciphertext $c \in \mathcal{C}$ and a key $K \in \mathcal{K}$, given a public key pk.
3. Decaps(sk, c): a deterministic decapsulation algorithm that takes the private key sk and a ciphertext $c \in \mathcal{C}$ and outputs either a key $K \in \mathcal{K}$ upon successful decapsulation, or a failure symbol $\perp$ upon rejection.

Security

The standard security notion for key encapsulation mechanisms is the indistinguishability under (adaptive) chosen-ciphertext attacks (IND-CCA) [65]. Similarly to IND-CPA case, an adversary is given the access to an encapsulation oracle Encaps() throughout the attack, such that it can encapsulate an arbitrary number of keys of its choice. In addition, the attacker is given access to a decapsulation oracle Decaps(). IND-CCA-security model provides stronger security guarantees compared to IND-CPA model and is formalized in the following game:

```
Game $\mathsf{IND\text{-}CCA}^{\mathsf{A}}_{\mathsf{KEM}}$ :
    $(pk, sk) \leftarrow \mathsf{KeyGen}()$
    $b \leftarrow \{0, 1\}$
    $K'_1 \leftarrow \mathcal{K}$
    $(c', K'_0) \leftarrow \mathsf{Encaps}(pk)$
    $b' \leftarrow \mathsf{A}^{\mathsf{Decaps}()}(pk, c', K'_b)$
    return $b \overset{?}{=} b'$.
```

A KEM is considered IND-CCA-secure, if for all efficient adversaries A the probability of winning the $\mathsf{IND\text{-}CCA}^{\mathsf{A}}_{\mathsf{KEM}}$ game is negligible. More precisely, given some negligible function $\mathsf{negl}(n)$ of the security parameter n,

$$\mathbf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}}(\mathsf{A}) = \left| \Pr\left(\mathsf{IND\text{-}CCA}^{\mathsf{A}}_{\mathsf{KEM}} = 1\right) - \frac{1}{2} \right| < \mathsf{negl}(n).$$

Correctness

Similarly to the PKE case, the KEM is called δ-correct [44], if

$$\Pr\left[\mathsf{Decaps}(sk, c) \neq K \mid (pk, sk) \leftarrow \mathsf{KeyGen}; (c, K) \leftarrow \mathsf{Encaps}(pk)\right] \leq \delta.$$

2.4 Hard problems on Lattices

In this section we introduce a class of average-case hard cryptographic assumptions based on lattices. We focus on the Learning With Errors (LWE) problem and its variations, since a number of cryptographic primitives, including Kyber and Saber, rely on these assumptions. In contrast to other cryptographic hardness assumptions, such as integer factorization or discrete logarithm problems, LWE-based problems are believed to be quantum-hard, as no quantum attacks are known against them.

2.4.1 Learning With Errors

The LWE (Learning With Errors) problem was first introduced by Regev [68] and serves as a basis for many lattice-based cryptographic schemes. Let l, q, η positive integers, χ_η a probability distribution over $\mathbb{Z}_q$ parameterized by η, $\boldsymbol{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{l\times l})$ uniformly random matrix, $\boldsymbol{s} \leftarrow \chi_\eta(\mathbb{Z}_q^l)$ secret vector with small coefficients, fixed for all samples, $e \leftarrow \chi_\eta(\mathbb{Z}_q^l)$ small error, fresh for every sample. An instance of a LWE problem, $\mathsf{LWE}_{l,q,\eta}$, comes in two flavours - search-LWE and decision-LWE, defined below:

Search-LWE Problem.

- Given $(\boldsymbol{A}, \boldsymbol{b}) \in (\mathbb{Z}_q^{l\times l} \times \mathbb{Z}_q^l)$, recover the secret vector $\boldsymbol{s}$, such that $\boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{s} + e$.

Without the error term $\boldsymbol{e}$, this problem could be trivially solved for $\boldsymbol{s}$ by Gaussian elimination, however the addition of errors makes the search-LWE problem computationally hard. The search version of LWE problem is not suitable for cryptography, since it is important, that the adversary will not be able to obtain any information about the encrypted message - and not just to guess the message correctly. For cryptography, a decisional variant of LWE, described below, is preferable.

Decision-LWE Problem.

- Given $(\boldsymbol{A}, \boldsymbol{b}) \in (\mathbb{Z}_q^{l\times l} \times \mathbb{Z}_q^l)$, decide, whether $(\boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{s} + e) \in \mathbb{Z}_q^l$ is a LWE sample, or $\boldsymbol{b} \leftarrow \mathcal{U}(\mathbb{Z}_q^l)$ is chosen uniformly at random.

The $\mathsf{LWE}_{l,q,\eta}$ problem assumes that decision-LWE problem is computationally hard. More precisely, for any probabilistic polynomial-time adversaries A, it holds that

$$\mathbf{Adv}\,^{\mathsf{LWE}}_{l,q,\eta}(\mathsf{A}) = \left| \Pr\left(b = b' : \begin{array}{l} \boldsymbol{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{l\times l}); \\ (\boldsymbol{s}, \boldsymbol{e}) \leftarrow \chi_\eta(\mathbb{Z}_q^l) \times \chi_\eta(\mathbb{Z}_q^l); \\ \boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{s} + e;\ b' \leftarrow \mathsf{A}(\boldsymbol{A}, \boldsymbol{b})) \end{array} \right) \text{-} \Pr\left(b = b' : \begin{array}{l} \boldsymbol{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{l\times l}); \\ \boldsymbol{b} \leftarrow \mathcal{U}(\mathbb{Z}_q^l); \\ b' \leftarrow \mathsf{A}(\boldsymbol{A}, \boldsymbol{b}) \end{array} \right) \right| < \mathsf{negl}(n).$$

2.4.2 Module Learning With Errors

The MLWE (Module Learning With Errors) problem was introduced by Langlois and Stehlé [54]. It is a variant of LWE problem which replaces the ring $\mathbb{Z}_q$ in the LWE samples by a quotient ring R_q of dimension n with error distribution $\chi_\eta(R_q^l)$. An instance of a MLWE problem, $\mathsf{MLWE}_{l,q,\eta}$ is parameterized by positive integers l, q, η and comes in two variants - search-MLWE and decision-MLWE, defined as follows:

Search-MLWE Problem.

- Given $(\boldsymbol{A}, \boldsymbol{b}) \in (R_q^{l\times l} \times R_q^l)$, recover the secret vector $\boldsymbol{s}$, such that $\boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{s} + e$.

Decision-MLWE Problem.

- Given $(\boldsymbol{A}, \boldsymbol{b}) \in (R_q^{l\times l} \times R_q^l)$, decide, whether $(\boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{s} + e) \in R_q^l$ is a MLWE sample, or $\boldsymbol{b} \leftarrow \mathcal{U}(R_q^l)$ is chosen uniformly at random.

The $\mathsf{MLWE}_{l,q,\eta}$ problem assumes that decision-MLWE problem is hard to solve. More precisely, any probabilistic polynomial-time adversaries A can distinguish the MLWE samples from the uniform samples with a negligible advantage:

$$\mathbf{Adv}\,{}^{\mathsf{MLWE}}_{l,q,\eta}(\mathsf{A}) = \left| \Pr\left(b = b' : \begin{array}{l} \boldsymbol{A} \leftarrow \mathcal{U}(R_q^{l\times l}); \\ (\boldsymbol{s}, \boldsymbol{e}) \leftarrow \chi_\eta(R_q^l) \times \chi_\eta(R_q^l); \\ \boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{s} + e;\ b' \leftarrow \mathsf{A}(\boldsymbol{A}, \boldsymbol{b})) \end{array} \right) \text{- Pr} \left(b = b' : \begin{array}{l} \boldsymbol{A} \leftarrow \mathcal{U}(R_q^{l\times l}); \\ \boldsymbol{b} \leftarrow \mathcal{U}(R_q^l); \\ b' \leftarrow \mathsf{A}(\boldsymbol{A}, \boldsymbol{b}) \end{array} \right) \right| < \mathsf{negl}(n).$$

The security of MLWE instance can be increased by increasing the dimension l of the module. The case $l = 1$ corresponds to the Ring Learning With Errors (RLWE) problem, which was analyzed in [56]. As the dimension l is fixed, the security of RLWE can be adjusted by varying the dimension of the ring.

2.4.3 Learning With Rounding

The LWR (Learning With Rounding) problem was introduced by Banerjee, Peikert and Rosen [8]. This problem can be considered a ”derandomized” variant of the LWE problem, as it does not contain implicit noise terms, sampled from the error distribution. Instead, an instance of a LWR problem includes implicit errors caused by scaling and rounding the polynomial coefficients modulo q to modulo p. Let l, q, η, p positive integers with $q > p$, χ_η a probability distribution over $\mathbb{Z}_q$ parameterized by η, $\boldsymbol{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{l\times l})$ uniformly random matrix, $\boldsymbol{s} \leftarrow \chi_\eta(\mathbb{Z}_q^l)$ secret vector with small coefficients, fixed for all samples. There are two variants of the LWR instance, $\mathsf{LWR}_{l,q,p,\eta}$, defined as follows:

Search-LWR Problem.

- Given $(\boldsymbol{A}, \boldsymbol{b}) \in (\mathbb{Z}_q^{l\times l} \times \mathbb{Z}_p^l)$, recover the secret vector $\boldsymbol{s}$, such that $\boldsymbol{b} = \lfloor (p/q)\, \boldsymbol{A} \cdot \boldsymbol{s} \rceil$.

Decision-LWR Problem.

- Given $(\boldsymbol{A}, \boldsymbol{b}) \in (\mathbb{Z}_q^{l\times l} \times \mathbb{Z}_p^l)$, decide, whether $(\boldsymbol{b} = \lfloor (p/q)\, \boldsymbol{A} \cdot \boldsymbol{s} \rceil) \in \mathbb{Z}_p^l$ is a LWR sample, or $\boldsymbol{b} \leftarrow \mathcal{U}(\mathbb{Z}_p^l)$ is chosen uniformly at random.

It was shown that the LWR problem is as hard as the LWE problem - a reduction was given by Banerjee et al. [8], and further refined in several works [4, 22, 3]. The $\mathsf{LWR}_{l,q,p,\eta}$ problem assumes that decision-LWR problem is computationally hard. The advantage of the adversary is given by

$$\mathbf{Adv}\,{}^{\mathsf{LWR}}_{l,q,p,\eta}(\mathsf{A}) = \left| \Pr\left(b = b' : \begin{array}{l} \boldsymbol{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{l\times l}); \\ \boldsymbol{s} \leftarrow \chi_\eta(\mathbb{Z}_q^l); \\ \boldsymbol{b} = \lfloor (p/q) \boldsymbol{A} \cdot \boldsymbol{s} \rceil; \\ b' \leftarrow \mathsf{A}(\boldsymbol{A}, \boldsymbol{b}) \end{array} \right) \text{- } \Pr\left(b = b' : \begin{array}{l} \boldsymbol{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{l\times l}); \\ \boldsymbol{b} \leftarrow \mathcal{U}(\mathbb{Z}_p^l); \\ b' \leftarrow \mathsf{A}(\boldsymbol{A}, \boldsymbol{b}) \end{array} \right) \right| < \mathsf{negl}(n).$$

2.4.4 Module Learning With Rounding

The MLWR (Module Learning With Rounding) problem is the module version of LWR problem. As in the case of MLWE, the MLWR instance can be obtained from the regular LWR by computing the operations over the quotient ring R_q. There are two variants of the MLWR instance, $\mathsf{MLWR}_{l,q,p,\eta}$, defined as follows:

Search-MLWR Problem.

- Given $(\boldsymbol{A}, \boldsymbol{b}) \in (R_q^{l\times l} \times R_p^l)$, recover the secret vector $\boldsymbol{s}$, such that $\boldsymbol{b} = \lfloor (p/q)\, \boldsymbol{A} \cdot \boldsymbol{s} \rceil$.

Decision-MLWR Problem.

- Given $(\boldsymbol{A}, \boldsymbol{b}) \in (R_q^{l\times l} \times R_p^l)$, decide, whether $(\boldsymbol{b} = \lfloor (p/q)\, \boldsymbol{A} \cdot \boldsymbol{s} \rceil) \in R_p^l$ is a MLWR sample, or $\boldsymbol{b} \leftarrow \mathcal{U}(R_p^l)$ is chosen uniformly at random.

The $\mathsf{MLWR}_{l,q,p,\eta}$ problem assumes that decision-MLWR problem is computationally hard. The advantage of the adversary is given by

$$\mathbf{Adv}\,{}^{\mathsf{MLWR}}_{l,q,p,\eta}(\mathsf{A}) = \left| \Pr\left(b = b' : \begin{array}{l} \boldsymbol{A} \leftarrow \mathcal{U}(R_q^{l\times l}); \\ \boldsymbol{s} \leftarrow \chi_\eta(R_q^l); \\ \boldsymbol{b} = \lfloor (p/q) \boldsymbol{A} \cdot \boldsymbol{s} \rceil; \\ b' \leftarrow \mathsf{A}(\boldsymbol{A}, \boldsymbol{b}) \end{array} \right) \text{- } \Pr\left(b = b' : \begin{array}{l} \boldsymbol{A} \leftarrow \mathcal{U}(R_q^{l\times l}); \\ \boldsymbol{b} \leftarrow \mathcal{U}(R_p^l); \\ b' \leftarrow \mathsf{A}(\boldsymbol{A}, \boldsymbol{b}) \end{array} \right) \right| < \mathsf{negl}(n).$$

The security of Kyber and Saber relies on the hardness of solving the MLWE and MLWR problems, respectively.

2.5 The Kyber and Saber IND-CCA-secure KEM

Kyber and Saber are IND-CCA-secure key-encapsulation mechanisms and finalists in the third round of the NIST PQC standardization process [58]. Both schemes are constructed following a two-step approach: they include an IND-CPA-secure public-key encryption scheme (PKE), from which they build an IND-CCA secure key encapsulation mechanism (KEM) using a variant of the Fujisaki-Okamoto (FO) transform [38].

2.5.1 The IND-CPA-secure PKE

We describe an IND-CPA-secure public key encryption of Kyber (Kyber.CPAPKE) and Saber (Saber.CPAPKE) in Algorithms 1-6 using a simplified notation, which omits the low-level functions, such as encoding of the polynomials into the byte arrays, the details of the sampling and the transformations into and out of NTT (number-theoretic transform) domain.

Compression and Decompression

Both Kyber and Saber include compression operation which discards low-order bits of the public key and ciphertext components.

- The Kyber compression function takes an element $x \in R_q$ and returns an integer in $\{0, ..., 2^d - 1\}$, where $d < \lceil \log_2(q) \rceil$ is the bit-length of the result:

  $$\mathsf{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod 2^d.$$

  The decompression function is the reverse operation satisfying $x' = \mathsf{Decompress}_q(\mathsf{Compress}_q(x, d), d)$, such that x' is an element close to x with $|x' - x \bmod q| \leq \lceil q/2^{d+1} \rfloor$:

  $$\mathsf{Decompress}_q(x, d) = \lceil (q/2^d) \cdot x \rfloor.$$

- In case of Saber, the compression function can be expressed as a logical shift, since Saber uses a power-of-two modulus $q = 2^{13}$.

Kyber PKE

The public-key encryption of Kyber consists of three operations: key generation, encryption and decryption, given in Algorithms 1, 3 and 5.

- $\mathsf{Kyber.CPAPKE.KeyGen}()$: The key generation operation generates a public key pk and a secret key sk. It starts off by sampling the random seeds $seed_{\boldsymbol{A}}$ and σ from the uniform distribution $\mathcal{U}$ and expanding them into a public matrix $\boldsymbol{A}$, the secret vector $\boldsymbol{s}$, the error vector $\boldsymbol{e}$, according to uniform distribution $\mathcal{U}$ and binomial distributions β_{η_1} and β_{η_2} respectively. The public key is then obtained by computing a MLWE sample $\boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{s} + e$.

- Kyber.CPAPKE.Enc() : Similarly to key generation, the encryption operation first generates a public matrix $\boldsymbol{A}$, secret vector $\boldsymbol{s'}$, the error vector $\boldsymbol{e}_1$, from which a MLWE sample $\boldsymbol{u}$ is computed. After the error polynomial e_2 is sampled according to distribution β_{η_3}, the message m is encrypted by adding a product of public key pk with the secret $\boldsymbol{s'}$ and the error polynomial e_2. The result is then compressed into ciphertext $\boldsymbol{c}$.
- Kyber.CPAPKE.Dec() : The decryption function performs a reverse operation by reconstructing the message m from the ciphertext $\boldsymbol{c}$ using the private key sk.

	n	k	q	η_1	η_2	(d_u, d_v)	δ
Kyber512	256	2	3329	3	2	(10, 4)	2^{-123}
Kyber768	256	3	3329	2	2	(10, 4)	2^{-164}
Kyber1024	256	4	3329	2	2	(11, 5)	2^{-174}

Table 1: Parameter sets for Kyber

Parameter set. Kyber.CPAPKE is parameterized by the set of the positive integers $\{n, k, q, \eta_1, \eta_2, (d_u, d_v), \delta\}$ and has three versions, called Kyber512, Kyber768 and Kyber1024 in order of increasing security. In this thesis we focus on the Kyber768 parameter set, which provides medium level of security. It chooses the prime modulus $q = 3329$, the ring dimension $n = 256$, the module dimension $k = 3$, the parameters for centered binomial distribution $\eta_1 = \eta_2 = 2$ and for compression $(d_u, d_v) = (10, 4)$. These parameters make the probability of decryption error negligible, i.e. Kyber.CPAPKE is δ-correct with $\delta = 2^{-164}$ [7].

```
Algorithm 1  Kyber.CPAPKE.KeyGen()
Output: Public key $pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$,
        secret key $sk = \boldsymbol{s}$
1: $(seed_{\boldsymbol{A}}, \sigma) \leftarrow \mathcal{U}(\{0,1\}^n) \times \mathcal{U}(\{0,1\}^n)$
2: $\boldsymbol{A} \leftarrow \mathcal{U}(R_q^{k\times k}; seed_{\boldsymbol{A}})$
3: $(\boldsymbol{s}, \boldsymbol{e}) \leftarrow \beta_{\eta_1}(R_q^k; \sigma) \times \beta_{\eta_1}(R_q^k; \sigma)$
4: $\boldsymbol{b} := \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e}$
5: return $(pk, sk) = ((seed_{\boldsymbol{A}}, \boldsymbol{b}), \boldsymbol{s})$
```

```
Algorithm 2  Saber.CPAPKE.KeyGen()
Output: Public key $pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$,
        secret key $sk = \boldsymbol{s}$
1: $(seed_{\boldsymbol{A}}, \sigma) \leftarrow \mathcal{U}(\{0,1\}^n) \times \mathcal{U}(\{0,1\}^n)$
2: $\boldsymbol{A} \leftarrow \mathcal{U}(R_q^{k\times k}; seed_{\boldsymbol{A}})$
3: $\boldsymbol{s} \leftarrow \beta_{\eta}(R_q^k; \sigma)$
4: $\boldsymbol{b} := (\boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{h}) \gg (\epsilon_q - \epsilon_p)$
5: return $(pk, sk) = ((seed_{\boldsymbol{A}}, \boldsymbol{b}), \boldsymbol{s})$
```

Saber PKE

The Saber public-key encryption scheme Saber.CPAPKE = (KeyGen, Enc, Dec) is described in Algorithms 2, 4, 6.

- Saber.CPAPKE.KeyGen() : During key generation, the random seed $seed_{\boldsymbol{A}}$ is sampled from the uniform distribution $\mathcal{U}$ and expanded into a uniform public matrix $\boldsymbol{A}$. The secret vector $\boldsymbol{s}$ of the dimension k is generated according to distribution β_η using random seed σ. The public key is then obtained by computing the MLWR sample $\boldsymbol{A} \cdot \boldsymbol{s}$, where the result of the multiplication is rounded by performing the logical shift and adding the constant $\boldsymbol{h}$.

- Saber.CPAPKE.Enc() : Similarly to key generation, the encryption operation starts off by sampling a uniform matrix $\boldsymbol{A}$, another secret vector $\boldsymbol{s'}$ and computing a MLWR instance $\boldsymbol{u}$. The message m is then encrypted by adding the product of public key $\boldsymbol{b}$ with $\boldsymbol{s'}$ and another constant h_1.
- Saber.CPAPKE.Dec() : The decryption is the reverse operation, which recovers a secret m from the ciphertext $\boldsymbol{c}$ using secret key sk.

	n	k	q	p	T	η	δ
LightSaber	256	2	2^{12}	2^{10}	2^3	5	2^{-123}
Saber	256	3	2^{12}	2^{10}	2^4	4	2^{-164}
FireSaber	256	4	2^{12}	2^{10}	2^6	3	2^{-174}

Table 2: Parameter sets for Saber

Parameter set. In the following, we use the Saber parameter set $\{n, k, q, p, T, \eta, \delta\}$, which offers medium level of security (as opposite to LightSaber and FireSaber parameter sets). It chooses three power-of-two moduli $q = 2^{13}$, $p = 2^{10}$ and $T = 2^4$, the ring dimension $n = 256$ and the module dimension $k = 3$. The centered binomial distribution is parameterized by $\eta = 10$ and the constants h_1, h_2 and $\boldsymbol{h}$ are added to center the errors introduced by rounding. The parameter set above bounds the decryption error probability by at most 2^{-136}, i.e. Saber.CPAPKE is δ-correct with $\delta = 2^{-136}$ [11].

Algorithm 3 Kyber.CPAPKE.Dec(sk, c)

```
Input:  Secret key $sk = \boldsymbol{s}$, ciphertext $\boldsymbol{c} = (\boldsymbol{c_1}, c_2)$
Output: Message $m$
1: $\boldsymbol{u} \leftarrow \mathsf{Decompress}_q(\boldsymbol{c_1}, d_u)$
2: $v \leftarrow \mathsf{Decompress}_q(c_2, d_v)$
3: $m = \mathsf{Compress}_q(v - \boldsymbol{s} \cdot \boldsymbol{u}^T, 1)$
4: return $m$
```

Algorithm 4 Saber.CPAPKE.Dec(sk, c)

```
Input:  Secret key $sk = \boldsymbol{s}$, ciphertext $\boldsymbol{c} = (\boldsymbol{c_1}, c_2)$
Output: Message $m$
1: $\boldsymbol{u} \leftarrow \boldsymbol{c_1}$
2: $v \leftarrow h_2 - 2^{\epsilon_p - \epsilon_T} \cdot c_2$
3: $m = (v + (\boldsymbol{s} \bmod p) \cdot \boldsymbol{u}^T) \gg (\epsilon_p - 1)$
4: return $m$
```

Algorithm 5 Kyber.CPAPKE.Enc(pk, m, σ)

```
Input:  Public key $pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$, message $m$,
        random coins $\sigma$
Output: Ciphertext $\boldsymbol{c} = (\boldsymbol{c_1}, c_2)$
1: $\boldsymbol{A} \leftarrow \mathcal{U}(R_q^{k \times k}; seed_{\boldsymbol{A}})$
2: $\boldsymbol{s'} \leftarrow \beta_{\eta_1}(R_q^k; \sigma)$
3: $(\boldsymbol{e_1}, e_2) \leftarrow \beta_{\eta_2}(R_q^k; \sigma) \times \beta_{\eta_3}(R_q; \sigma)$
4: $\boldsymbol{u} \leftarrow \boldsymbol{A} \cdot \boldsymbol{s'} + \boldsymbol{e_1}$
5: $v \leftarrow \boldsymbol{b}^T \cdot \boldsymbol{s'} + e_2 + \mathsf{Decompress}_q(m)$
6: $\boldsymbol{c_1} \leftarrow \mathsf{Compress}_q(\boldsymbol{u}, d_u)$
7: $c_2 \leftarrow \mathsf{Compress}_q(v, d_v)$
8: return $\boldsymbol{c} = (\boldsymbol{c_1}, c_2)$
```

Algorithm 6 Saber.CPAPKE.Enc(pk, m, σ)

```
Input:  Public key $pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$, message $m$,
        random coins $\sigma$
Output: Ciphertext $\boldsymbol{c} = (\boldsymbol{c_1}, c_2)$
1: $\boldsymbol{A} \leftarrow \mathcal{U}(R_q^{k \times k}; seed_{\boldsymbol{A}})$
2: $\boldsymbol{s'} \leftarrow \beta_{\eta}(R_q^k; \sigma)$
3:
4: $\boldsymbol{u} \leftarrow \boldsymbol{A} \cdot \boldsymbol{s'} + \boldsymbol{h}$
5: $v \leftarrow \boldsymbol{b}^T \cdot (\boldsymbol{s'} \bmod p) + h_1 - 2^{\epsilon_p - 1} \cdot m$
6: $\boldsymbol{c_1} \leftarrow \boldsymbol{u} \gg (\epsilon_q - \epsilon_p)$
7: $c_2 \leftarrow v \gg (\epsilon_q - \epsilon_T)$
8: return $\boldsymbol{c} = (\boldsymbol{c_1}, c_2)$
```

2.5.2 The IND-CCA-secure KEM

The IND-CPA secure public key encryption schemes in the previous section can be converted into an IND-CCA secure key encapsulation mechanisms by applying an appropriate transformation. Both Kyber and Saber are utilizing a post-quantum variant of the Fujisaki-Okamoto (FO) transform by Hofheinz, Hövelmanns and Kiltz [44], however other transformations could be used to achieve chosen-ciphertext security [19, 65].

Kyber and Saber KEMs

The resulting KEMs consist of a triple of operations (KeyGen, Encaps, Decaps), given in Algorithms 7-12. At a high-level, the CCA-transformations for both schemes are identical and require access to three hash functions $\mathcal{G}$, $\mathcal{H}$, $\mathcal{H}'$, modeled as random oracles, as well as the public key encryption scheme CPAPKE = (KeyGen, Enc, Dec). The only difference is the instantiation of these functions. Kyber and Saber rely on Kyber.CPAPKE and Saber.CPAPKE operations, respectively, and — for the parameter sets we have chosen — hash-functions are instantiated with the different symmetric primitives, based on the SHA3 standard. The key generation for Kyber and Saber is similar to the ones from the previous section, with the difference that the secret key sk also includes the public key pk, the hash of pk and a secret random seed z. During encapsulation, a ciphertext $\boldsymbol{c}$ is returned together with shared key K, where $\boldsymbol{c}$ is obtained by encrypting a random message m, sampled from the uniform distribution, and K is derived by hashing together the message, the public key and the ciphertext. The decapsulation operation is described in detail in the following section.

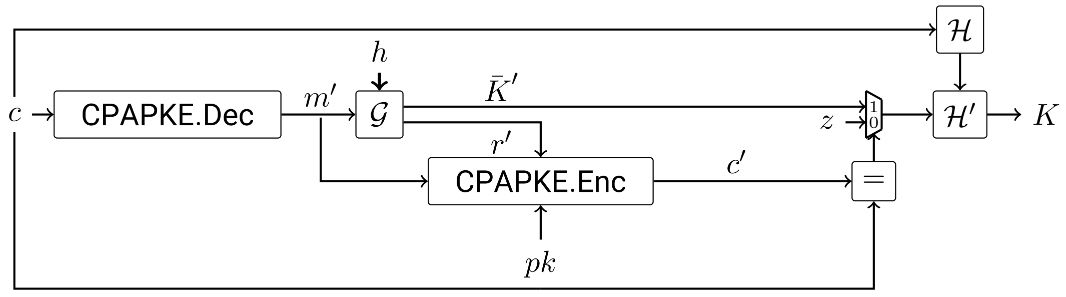

Figure 1: IND-CCA-secure decapsulation in Kyber and Saber

Fujisaki-Okamoto Transform

We illustrate the basic working of the FO transform in Figure 1. The core idea is to check the validity of a decrypted message m' in the decapsulation phase by performing re-encryption. The obtained candidate ciphertext $\boldsymbol{c'}$ is then compared with the original ciphertext $\boldsymbol{c}$. If both ciphertexts are equal, the session key K is derived from the message m', the ciphertext c and the hash of the public key pk, otherwise a pseudo-random string $K = \mathcal{H}'(z, \mathcal{H}(c))$ is returned. Since the decapsulation never indicates failure $\perp$, the rejection of malformed ciphertexts is implicit. Following [23] and [34], the resulting KEMs are proved to be IND-CCA secure if the hash functions $\mathcal{G}$, $\mathcal{H}$, $\mathcal{H}'$ are modeled as random oracles. As described in [44], Kyber.CCAKEM and Saber.CCAKEM are δ-correct, since the underlying public-key encryption schemes Kyber.CPAPKE and Saber.CPAPKE are δ-correct.

Algorithm 7 Kyber.CCAKEM.KeyGen()

```
Output: Public key $pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$,
        secret key $sk = (\boldsymbol{s}, \mathcal{H}(pk), pk, z)$
1: $z \leftarrow \mathcal{U}(\{0,1\}^n)$
2: $((seed_{\boldsymbol{A}}, \boldsymbol{b}), \boldsymbol{s}) \leftarrow$ Kyber.CPAPKE.KeyGen()
3: $sk = (s, pk, \mathcal{H}(pk), z)$
4: return $(pk, sk) = ((seed_{\boldsymbol{A}}, \boldsymbol{b}), (\boldsymbol{s}, pk, \mathcal{H}(pk), z))$
```

Algorithm 8 Saber.CCAKEM.KeyGen()

```
Output: Public key $pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$,
        secret key $sk = (\boldsymbol{s}, \mathcal{H}(pk), pk, z)$
1: $z \leftarrow \mathcal{U}(\{0,1\}^n)$
2: $((seed_{\boldsymbol{A}}, \boldsymbol{b}), \boldsymbol{s}) \leftarrow$ Saber.CPAPKE.KeyGen()
3: $sk = (s, pk, \mathcal{H}(pk), z)$
4: return $(pk, sk) = ((seed_{\boldsymbol{A}}, \boldsymbol{b}), (\boldsymbol{s}, pk, \mathcal{H}(pk), z))$
```

Algorithm 9 Kyber.CCAKEM.Encaps(pk)

```
Input: Public key $pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$
Output: Ciphertext $\boldsymbol{c}$, key $K$
1: $m \leftarrow \mathcal{U}(\{0,1\}^n)$
2: $(\bar{K}, r) := \mathcal{G}(m \mid\mid \mathcal{H}(pk))$
3: $\boldsymbol{c} =$ Kyber.CPAPKE.Enc$(pk, m; r)$
4: $K := \mathcal{H}'(\bar{K} \mid\mid \mathcal{H}(\boldsymbol{c}))$
5: return $(\boldsymbol{c}, K)$
```

Algorithm 10 Saber.CCAKEM.Encaps(pk)

```
Input: Public key $pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$
Output: Ciphertext $\boldsymbol{c}$, key $K$
1: $m \leftarrow \mathcal{U}(\{0,1\}^n)$
2: $(\bar{K}, r) := \mathcal{G}(m \mid\mid \mathcal{H}(pk))$
3: $\boldsymbol{c} =$ Saber.CPAPKE.Enc$(pk, m; r)$
4: $K := \mathcal{H}'(\bar{K} \mid\mid \mathcal{H}(\boldsymbol{c}))$
5: return $(\boldsymbol{c}, K)$
```

Algorithm 11 Kyber.CCAKEM.Decaps(sk, c)

```
Input: Ciphertext $\boldsymbol{c}$,
       secret key $sk = (\boldsymbol{s}, pk, \mathcal{H}(pk), z)$
Output: Key $K$
1: $m' :=$ Kyber.CPAPKE.Dec$(\boldsymbol{s}, \boldsymbol{c})$
2: $(\bar{K}', r') := \mathcal{G}(m' \mid\mid \mathcal{H}(pk))$
3: $\boldsymbol{c}' :=$ Kyber.CPAPKE.Enc$(\boldsymbol{pk}, m'; r')$
4: if $\boldsymbol{c} = \boldsymbol{c}'$ then
5:     return $K := \mathcal{H}'(\bar{K}' \mid\mid \mathcal{H}(c))$
6: else
7:     return $K := \mathcal{H}'(z \mid\mid \mathcal{H}(\boldsymbol{c}))$
```

Algorithm 12 Saber.CCAKEM.Decaps(sk, c)

```
Input: Ciphertext $\boldsymbol{c}$,
       secret key $sk = (\boldsymbol{s}, pk, \mathcal{H}(pk), z)$
Output: Key $K$
1: $m' :=$ Saber.CPAPKE.Dec$(\boldsymbol{s}, \boldsymbol{c})$
2: $(\bar{K}', r') := \mathcal{G}(m' \mid\mid \mathcal{H}(pk))$
3: $\boldsymbol{c}' :=$ Saber.CPAPKE.Enc$(\boldsymbol{pk}, m'; r')$
4: if $\boldsymbol{c} = \boldsymbol{c}'$ then
5:     return $K := \mathcal{H}'(\bar{K}' \mid\mid \mathcal{H}(c))$
6: else
7:     return $K := \mathcal{H}'(z \mid\mid \mathcal{H}(\boldsymbol{c}))$
```

3 Masking Kyber and Saber

This chapter describes the state-of-the-art techniques necessary to construct side-channel resistant implementations of Kyber and Saber. While both KEMs are considered to be secure against theoretical cryptanalysis, they will be used in the real-world applications where the biggest threats will appear from the physical attackers. Side-channel attacks on unprotected implementations of lattice-based cryptography have appeared in some recent papers. It was shown, that timing leakages of Gaussian samplers [25, 36], FO transformation [42] or cache behavior [20] can reveal information about secret values. By attacking the number theoretic transform [64, 76], modular reduction [74], message encoding [66, 5] and decoding [70], polynomial multiplication [45] or FO transform [67], power analysis attacks have been proven to be effective even for masked implementations of lattice-based schemes.

To achieve side-channel security, all the modules that process sensitive data, should be protected. In the literature, various mitigation techniques have been proposed, one of the most popular of which is masking [1, 27, 41]. In the remainder of this chapter, we review the most promising approaches, relevant for our implementation, to mask the vulnerable components of Kyber and Saber. These components are highlighted in grey in Figures 2 and 3. We focus on the first-order masking of the IND-CCA-secure decapsulation operation, Kyber.CCAKEM.Decaps and Saber.CCAKEM.Decaps, which is particularly vulnerable, as it directly involves the long-term secret key $\boldsymbol{s}$. The masking of IND-CPA-secure public-key encryption of Kyber and Saber is implicit, as both encryption and decryption operations are included in the IND-CCA decapsulation.

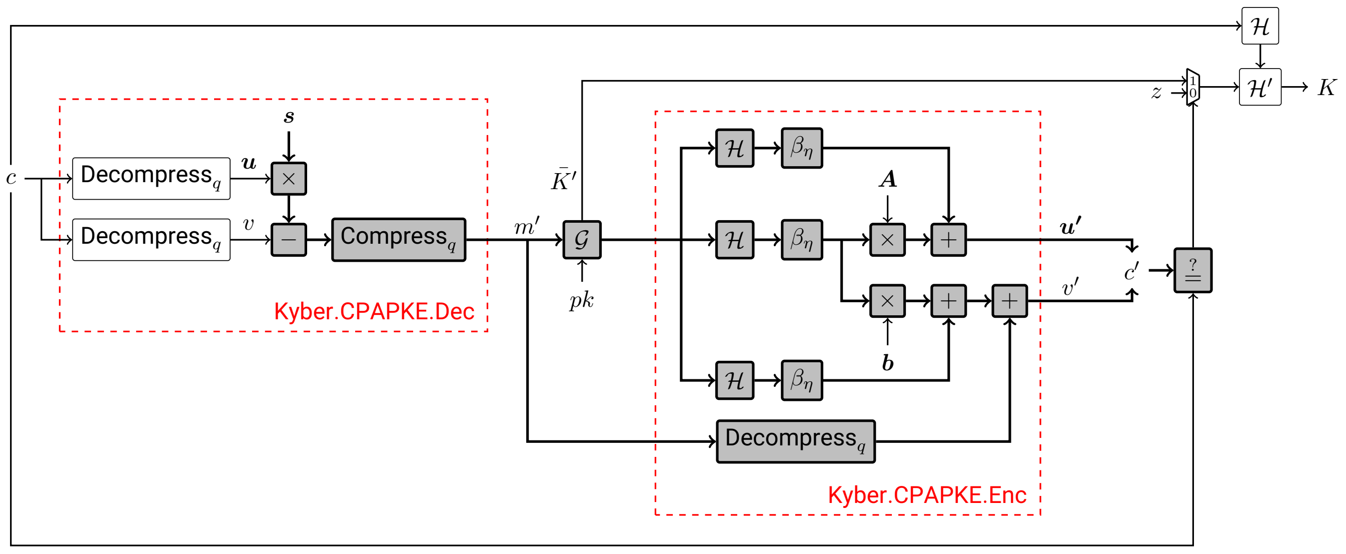

Figure 2: IND-CCA-secure decapsulation of Kyber. The components in grey are influenced by the long-term secret key $\boldsymbol{s}$ and should be masked. The operations which correspond to IND-CPA-secure encryption and decryption are grouped in red.

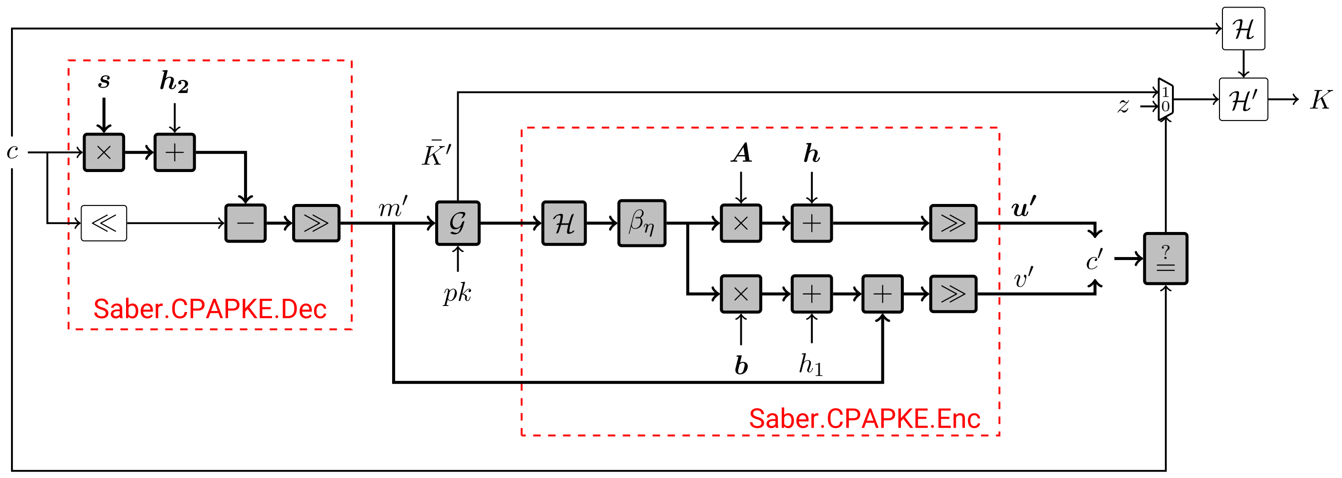

Figure 3: IND-CCA-secure decapsulation of Saber. The components in grey are influenced by the long-term secret key s and should be masked. The operations which correspond to IND-CPA-secure encryption and decryption are grouped in red.

3.1 Masked Polynomial Arithmetic

Since Kyber and Saber are based on MLWE and MLWR problems, respectively, they require polynomial arithmetic for encryption and decryption. Both schemes include polynomial multiplication, addition and subtraction, which should be protected when they process sensitive values. As these are arithmetic operations, the natural choice is to use arithmetic masking. In a first-order secure arithmetically masked scheme, a sensitive polynomial x is split into two random shares $(x^{(0)_A}, x^{(1)_A})$ such that $x = x^{(0)_A} + x^{(1)_A} \pmod q$. This is done by randomly choosing the one of the shares $x^{(1)_A} = r$ and computing the remaining share as $x^{(0)_A} = x - r \pmod q$.

3.1.1 Polynomial multiplication

In Kyber and Saber, a secret polynomial is multiplied with a public polynomial in CPAPKE.Enc and CPAPKE.Dec. The product $z = x \cdot y$ of an arithmetically masked polynomial x with an unmasked polynomial y can be computed by performing the operation on each share separately:

$$z^{(0)_A} = x^{(0)_A} \cdot y,$$
$$z^{(1)_A} = x^{(1)_A} \cdot y.$$

The obtained masked result is correct, since $z = x \cdot y = (x^{(0)_A} + x^{(1)_A}) \cdot y = (x^{(0)_A} \cdot y) + (x^{(1)_A} \cdot y) = z^{(0)_A} + z^{(1)_A} = z$. In case of Saber, the $\cdot$ operation corresponds to the coefficient-wise Toom-Cook multiplication carried out on both shares as described above. However, Kyber uses a different technique, which requires additional steps before and after the multiplication.

3.1.2 Number Theoretic Transform

As Kyber operates on polynomials in R_q, the Number Theoretic Transform (NTT) method is used to speed up the multiplications. Given two polynomials $x, y \in R_q$ in the so-called normal domain, the product $z = x \cdot y$ can be computed very efficiently by first mapping the polynomials to their alternative representation in the NTT domain, performing the multiplication in the NTT domain and converting the result back into normal domain using the inverse transformation:

$$z = x \cdot y = \mathsf{NTT}^{-1}(\mathsf{NTT}(x) \cdot \mathsf{NTT}(y)).$$

Since NTT transformations are linear operations, they are masked by splitting the sensitive variable into arithmetic shares and applying the transformations to each share individually. The multiplication in the NTT domain is then masked the same way as a regular polynomial multiplication.

3.1.3 Polynomial addition and subtraction

Similarly to polynomial multiplication, the arithmetic masking can also be applied to polynomial addition. Let x be a secret polynomial, y a secret or a public polynomial. Dependent on the value of y, the masking of the sum $z = x + y$ can be done in two possible ways. If a public value y is added to a secret value x, then only the sensitive value should be masked. The unmasked public value is then added to one of the shares:

$$\begin{aligned} z^{(0)_A} &= x^{(0)_A} + y, \\ z^{(1)_A} &= x^{(1)_A}. \end{aligned}$$

By computing $z = x + y = (x^{(0)_A} + x^{(1)_A}) + y = (x^{(0)_A} + y) + x^{(1)_A} = z^{(0)_A} + z^{(1)_A} = z$, the correctness of the result can be proven. The second option is to add two sensitive values x, y together. In this case both x and y should be masked and the computation is performed on both shares:

$$\begin{aligned} z^{(0)_A} &= x^{(0)_A} + y^{(0)_A}, \\ z^{(1)_A} &= x^{(1)_A} + y^{(1)_A}. \end{aligned}$$

The obtained result is correct, since $z = x + y = (x^{(0)_A} + x^{(1)_A}) + (y^{(0)_A} + y^{(1)_A}) = (x^{(0)_A} + y^{(0)_A}) + (x^{(1)_A} + y^{(1)_A}) = z^{(0)_A} + z^{(1)_A} = z$. As subtraction is a special case of addition, namely $z = x - y = x + (-y)$, the same techniques can be applied to mask polynomial subtraction.

3.2 Masked Boolean Operations

Besides polynomial arithmetic, both Kyber and Saber contain Boolean instructions, such as logical AND, exclusive OR (XOR), logical NOT, denoted as $\wedge$, $\oplus$ and $\neg$, respectively. These operations are part of binomial sampling and hashing routines and should be protected, as they process sensitive inputs. In a first-order secure Boolean masking scheme, every sensitive variable x is split into two Boolean shares ($x^{(0)_B}$, $x^{(1)_B}$), such that $x = x^{(0)_B} \oplus x^{(1)_B}$. By randomly choosing the one of the shares $x^{(1)_B} = r$ and computing the second share as $x^{(0)_B} = x \oplus r$, both shares remain independent of x. The $\wedge$, $\oplus$ and $\neg$ operations are replaced with their masked versions SecAND, SecXOR and SecNOT which compute on these shared inputs.

3.2.1 Linear Operations

SecXOR. Since the XOR is a linear operation, it can be masked by performing the computation on input shares separately. Given two Boolean-masked values x and y, the secure version of the operation $z = x \oplus y$ is computed as:

$$\begin{aligned} z^{(0)_A} &= x^{(0)_A} \oplus y^{(0)_A}, \\ z^{(1)_A} &= x^{(1)_A} \oplus y^{(1)_A}. \end{aligned}$$

By expanding the expression $z = x \oplus y = (x^{(0)_A} \oplus x^{(1)_A}) \oplus (y^{(0)_A} \oplus y^{(1)_A}) = (x^{(0)_A} \oplus y^{(0)_A}) \oplus (x^{(1)_A} \oplus y^{(1)_A}) = z^{(0)_A} \oplus z^{(1)_A} = z$, the correctness of the result can be proven.

SecNOT. The logical negation operation $\neg x$ of a Boolean-masked value x can be performed by negating just the first share of the masked input, since $\neg x = \neg(x^{(0)_A} \oplus x^{(1)_A}) = \neg x^{(0)_A} \oplus x^{(1)_A}$. In detail, the resulting Boolean shares are given by

$$\begin{aligned} x^{(0)_A} &= \neg x^{(0)_A}, \\ x^{(1)_A} &= x^{(1)_A}. \end{aligned}$$

3.2.2 Non-linear Operations

SecAND. As the logical AND operation is non-linear, it should be replaced with its secure counterpart, SecAND. To achieve the desired functionality, various algorithms can be used - in the following, we instantiate the SecAND component with masked algorithm from [71], given in Algorithm 13.

Algorithm 13 SecAND [71]

```
Input:   1. A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, x^{(1)_B})$, where $x = x^{(0)_B} \oplus x^{(1)_B}$.
         2. A Boolean sharing $\boldsymbol{y} = (y^{(0)_B}, y^{(1)_B})$, where $y = y^{(0)_B} \oplus y^{(1)_B}$..
Output:  A Boolean sharing $\boldsymbol{z} = (z^{(0)_B}, z^{(1)_B})$, such that $z = z^{(0)_B} \oplus z^{(1)_B} = x \wedge y$.
1: $\boldsymbol{z} \leftarrow \boldsymbol{x} \wedge \boldsymbol{y}$
2: $r_0 \leftarrow \mathcal{U}(\{0,1\}^w)$
3: $r_1 \leftarrow (x^{(0)_B} \wedge y^{(1)_B}) \oplus r_0$
4: $r_1 \leftarrow r_1 \oplus (x^{(1)_B} \wedge y^{(0)_B})$
5: $z^{(0)_B} \leftarrow z^{(0)_B} \wedge r_0$
6: $z^{(1)_B} \leftarrow z^{(1)_B} \wedge r_1$
7: return $(z^{(0)_B}, z^{(1)_B})$
```

SecREF. Some masking techniques described in the following sections require the Boolean or arithmetic shares to be refreshed, which has to be done in a masked fashion. In our implementation, we instantiate the SecREF component with masked algorithms from [71], given in Algorithms 14 and 15 for Boolean and arithmetic masking, respectively. However, other algorithms can be used to achieve the desired functionality.

Algorithm 14 Boolean SecREF [71]

Input: 1. A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, x^{(1)_B})$, where $x = x^{(0)_B} \oplus x^{(1)_B}$,
2. Bit size κ.

Output: A Boolean sharing $\boldsymbol{z} = (z^{(0)_B}, z^{(1)_B})$, such that $z = z^{(0)_B} \oplus z^{(1)_B} = x$.

```
1: $r \leftarrow \mathcal{U}(\{0,1\}^\kappa)$
2: $z^{(0)_B} \leftarrow x^{(0)_B} \oplus r$
3: $z^{(1)_B} \leftarrow x^{(1)_B} \oplus r$
4: return $(z^{(0)_B}, z^{(1)_B})$
```

Algorithm 15 Arithmetic SecREF [71]

Input: 1. An arithmetic sharing $\boldsymbol{x} = (x^{(0)_A}, x^{(1)_A})$, where $x = x^{(0)_A} + x^{(1)_A} \bmod q$,
2. Modulus q.

Output: An arithmetic sharing $\boldsymbol{z} = (z^{(0)_A}, z^{(1)_A})$, such that $z = z^{(0)_A} + z^{(1)_A} = x \bmod q$.

```
1: $r \leftarrow \mathcal{U}(\mathbb{Z}_q)$
2: $z^{(0)_A} \leftarrow x^{(0)_A} + r \bmod q$
3: $z^{(1)_A} \leftarrow x^{(1)_A} - r \bmod q$
4: return $(z^{(0)_A}, z^{(1)_A})$
```

3.3 Masked $\mathcal{G}$, $\mathcal{H}$, $\mathcal{H}'$

Both Kyber and Saber use the random oracles $\mathcal{G}$, $\mathcal{H}$ and $\mathcal{H}'$ for creating hash outputs and pseudo-random number generation. These modules are instantiated with the Keccak-based [63] primitives SHA3 and SHAKE, defined in the SHA3 standard. In [15], the authors proposed a way to securely implement Keccak algorithms, based on Boolean masking. Their approach is very efficient for the first-order masking schemes and we rely on it in our work. A higher-order masked software implementation of Keccak was presented in [10].

The core operation of the Keccak is the Keccak-f[1600] state permutation, each round of which contains the linear functions θ, ρ, π and ι and the non-linear operation χ. The linear operations consist of the rotation and exclusive OR (XOR) instructions that can be efficiently masked by splitting the variables into Boolean shares and duplicating the operation for each share. However, the χ step includes the AND calls which are non-linear and require special treatment:

$$x_i \leftarrow x_i \oplus (x_{i+1} \oplus 1) \wedge x_{i+2}.$$

As proposed in [15], the equation above can be masked by first converting the intermediate values x_i, x_{i+1}, x_{i+2} into Boolean shares. The output shares $x_i = x_i^{(0)_B} \oplus x_i^{(1)_B}$ are then computed as

$$x_i^{(0)_B} \leftarrow x_i^{(0)_B} \oplus (x_{i+1}^{(0)_B} \oplus 1) \wedge x_{i+2}^{(0)_B} \oplus x_{i+1}^{(0)_B} \wedge x_{i+2}^{(1)_B},$$
$$x_i^{(1)_B} \leftarrow x_i^{(1)_B} \oplus (x_{i+1}^{(1)_B} \oplus 1) \wedge x_{i+2}^{(1)_B} \oplus x_{i+1}^{(1)_B} \wedge x_{i+2}^{(0)_B}.$$

3.4 Masked A2B and B2A Conversions

Most lattice-based schemes combine two different types of masking. Arithmetic masking is used to protect arithmetic operations, for instance polynomial multiplications and additions. Other components like Keccak-based hash functions, contain Boolean instructions which can be efficiently protected with Boolean masking. To achieve side-channel resistance, it is essential to securely convert from one masking to another. The first secure B2A (Boolean to arithmetic) and A2B (arithmetic to Boolean) conversion methods were proposed by Goubin [40]. Over time, more efficient approaches for prime and power-of-two moduli were developed, which can be grouped into two main categories: table-based methods [31, 35], that use pre-computed look-up tables (LUT), and addition-based methods [30, 9], that require masked secure addition over Boolean shares (SecADD).

3.4.1 Masked Addition

Masked addition over Boolean shares is the core of addition-based masking conversion methods. Given two Boolean-shared inputs $x = x^{(0)_B} \oplus x^{(1)_B}$ and $y = y^{(0)_B} \oplus y^{(1)_B}$, the secure adder (SecADD) computes their arithmetic sum and outputs the Boolean sharing $z = z^{(0)_B} \oplus z^{(1)_B}$, such that $z = x + y = (x^{(0)_B} \oplus x^{(1)_B}) + (y^{(0)_B} \oplus y^{(1)_B})$. A generic higher-order secure addition method for power-of-two moduli was first proposed in [30]. However, for first-order masking schemes, this approach is not the most efficient. In [50], Karroumi et al. introduced the concept of blinded addition, which replaces the addition by a sequence of Boolean operations. The core observation is that one bit of the sum z can be computed as

$$z_i = x_i + y_i + \mathsf{carry}_i(x_i, y_i),$$

where the carry bit can be derived recursively from the preceding bits. A complete description of this method is given in Algorithm 16.

Algorithm 16 SecADD [50]

```
Input:  1. A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, x^{(1)_B})$, where $x = x^{(0)_B} \oplus x^{(1)_B}$,
        2. A Boolean sharing $\boldsymbol{y} = (y^{(0)_B}, y^{(1)_B})$, where $y = y^{(0)_B} \oplus y^{(1)_B}$,
        3. A random value $r \leftarrow \mathcal{U}(\{0,1\}^w)$.
Output: A Boolean sharing $\boldsymbol{z} = (z^{(0)_B}, z^{(1)_B})$, such that $z = z^{(0)_B} \oplus z^{(1)_B} = x + y \pmod{2^w}$.
 1: $T = x^{(0)_B} \wedge y^{(0)_B}$;  $\Omega = T \oplus r$
 2: $T = x^{(0)_B} \wedge y^{(1)_B}$;  $\Omega = \Omega \oplus T$
 3: $T = x^{(1)_B} \wedge y^{(0)_B}$;  $\Omega = \Omega \oplus T$
 4: $T = x^{(1)_B} \wedge y^{(1)_B}$;  $\Omega = \Omega \oplus T$
 5: $B = \Omega \ll 1$;  $r = r \ll 1$
 6: $z^{(0)_B} = x^{(0)_B} \wedge y^{(0)_B}$;  $z^{(1)_B} = x^{(1)_B} \wedge y^{(1)_B}$
 7: $T = r \wedge z^{(0)_B}$;  $\Omega = \Omega \oplus T$
 8: $T = r \wedge z^{(1)_B}$;  $\Omega = \Omega \oplus T$
 9: for $i = 0$ to $w - 1$ do
10:     $T = B \wedge z^{(0)_B}$;  $B = B \wedge z^{(1)_B}$
11:     $B = B \oplus \Omega$;  $B = B \oplus T$
12:     $B = B \ll 1$
13: $z^{(0)_B} = z^{(0)_B} \oplus B$;  $z^{(0)_B} = z^{(0)_B} \oplus r$
14: return $(z^{(0)_B}, z^{(1)_B})$
```

The algorithms for power-of-two moduli can be adapted for prime moduli. In [9], the authors construct the masked addition for the prime modulus (SecADD$_q$) from SecADD by first calculating $s = (x + y)$ in a masked fashion and then securely computing the second masked sum $s' = (x + y - q)$. If the value s' is positive, then then the masked sign bit $c = \mathsf{sign}(s')$ is set to 0 and $s' = (x + y - q)$ is the correct sum modulo q. Otherwise, if

s' is negative, then the value of c is set to 1 and $s = x + y$ is the correct sum. Since the sign bit takes different value in each case, it can be used to securely select the correct sum:

$$z^{(\cdot)_B} = \mathsf{SecAND}((c||...||c)^{(\cdot)_B}, s^{(\cdot)_B}) \oplus \mathsf{SecAND}(\neg(c||...||c)^{(\cdot)_B}, s'^{(\cdot)_B}),$$

where $(c||...||c)$ denotes the extension of the sign bit. The SecADD_q approach is summarized in Algorithm 17.

Algorithm 17 Masked SecADD_q from [50]

```
Input:  1. A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, x^{(1)_B})$, where $x = x^{(0)_B} \oplus x^{(1)_B}$,
        2. A Boolean sharing $\boldsymbol{y} = (y^{(0)_B}, y^{(1)_B})$, where $y = y^{(0)_B} \oplus y^{(1)_B}$.
Output: A Boolean sharing $\boldsymbol{z} = (z^{(0)_B}, z^{(1)_B})$, such that $z = z^{(0)_B} \oplus z^{(1)_B} = x + y \pmod q$.
1: $q^{(\cdot)_B} = (2^w - q, 0)$
2: $s'^{(\cdot)_B} = \mathsf{SecADD}(x^{(\cdot)_B}, y^{(\cdot)_B})$
3: $s^{(\cdot)_B} = \mathsf{SecADD}(s^{(\cdot)_B}, q'^{(\cdot)_B})$
4: $c^{(\cdot)_B} = (s'^{(\cdot)_B} \gg (w-1))$
5: $c^{(\cdot)_B} = \mathsf{SecREF}((c||...||c)^{(\cdot)_B}, w)$
6: $z^{(\cdot)_B} = \mathsf{SecAND}(s^{(\cdot)_B}, (c||...||c)^{(\cdot)_B}, w)$
7: $c^{(\cdot)_B} = \mathsf{SecREF}((c||...||c)^{(\cdot)_B}, w)$
8: $z^{(\cdot)_B} = z^{(\cdot)_B} \oplus \mathsf{SecAND}(s'^{(\cdot)_B}, \neg(c||...||c)^{(\cdot)_B}, w)$
9: return $(z^{(0)_B}, z^{(1)_B})$
```

3.4.2 Masked A2B Conversion

A2B conversion allows to securely transform the arithmetic shares $x = x^{(0)_A} + x^{(1)_A}$ into Boolean shares $x = x^{(0)_B} \oplus x^{(1)_B}$ without unmasking x. In the first-order masking scenario, the conversion is computed as

$$\begin{aligned} x^{(0)_B} &= (x^{(0)_A} + x^{(1)_A}) \oplus x^{(1)_A}, \\ x^{(1)_B} &= x^{(1)_A}. \end{aligned}$$

In the following, we describe two main masked conversion techniques - a table-based approach and an A2B conversion method based on secure masked arithmetic addition.

Table-based A2B Conversion

Table-based A2B conversion methods convert the arithmetic masking into a Boolean masking using a divide-and-conquer approach. The key idea is to split the sensitive variables into smaller chunks and convert each individual chunk using precomputed look-up tables. First-order secure A2B conversion algorithm was initially proposed by Coron and Tchulkine [31]. Their method uses two precomputed tables - one for transforming each chunk i into a Boolean shares $(i + r) \oplus r$, masked with a random value r, while the second table contains the carry bits, resulting from the modular addition $(i + r)$. In [35], Debraize has combined two separate tables into a unique table T, that computes $((i + r) \oplus r) + \gamma$. Thus, the outputs of T contain information about both the update of each chunk i and a carry bit. For k-bit chunks, the improved method requires 2^k table entries and is given in Algorithms 18 and 19.

Algorithm 18 Table T precomputation [31, 35].

```
Input: An integer $k$.
Output: A look-up table $T$, an integer $r$, an integer $\gamma$.
1: $r \leftarrow \mathcal{U}(\{0,1\}^k),\ \gamma \leftarrow \mathcal{U}(\{0,1\}^{n\cdot k})$
2: for $i = 0$ to $2^k - 1$ do
3:     $T[i] = ((i + r) \oplus r) + \gamma \bmod 2^{n\cdot k}$
4: return $(T, r, \gamma)$
```

Algorithm 19 A2B conversion of a $(n \cdot k)$-bit variable [31, 35].

```
Input: 1. An arithmetic sharing $\boldsymbol{x} = (x^{(0)_A}, x^{(1)_A})$, such that $x = x^{(0)_A} + x^{(1)_A} \pmod{2^{n\cdot k}}$,
       2. Table $T$, integers $r$, $\gamma$, generated during precomputation phase.
Output: A Boolean sharing $\boldsymbol{z} = (z^{(0)_B}, z^{(1)_B})$, such that $z = z^{(0)_B} \oplus z^{(1)_B} = x^{(0)_A} + x^{(1)_A} \pmod{2^{n\cdot k}}$.
1: for $i = 0$ to $n - 1$ do
2:     Split $x^{(0)_A}$ into $(A_h||A_l)$ and $x^{(1)_A}$ into $(R_h||R_l)$, such that $|A_l| = |R_l| = k$
3:     $x^{(0)_A} \leftarrow x^{(0)_A} - r \bmod 2^{(n-i)\cdot k}$
4:     $x^{(0)_A} \leftarrow x^{(0)_A} + R_l \bmod 2^{(n-i)\cdot k}$
5:     $x^{(0)_A} \leftarrow A_h||0 + T[A_l] \bmod 2^{n\cdot k}$
6:     $x^{(0)_A} \leftarrow x^{(0)_A} - \gamma \bmod 2^{n\cdot k}$
7:     $z^{(i)_B} \leftarrow A_l \oplus R_l,\ z^{(i)_B} \leftarrow A_l \oplus r$
8:     $x^{(0)_A} \leftarrow A_h,\ x^{(1)_A} \leftarrow R_h$
9: return $\boldsymbol{z}$
```

Addition-based A2B Conversion

Addition-based A2B conversion follows directly from secure masked arithmetic addition. Let $x = x^{(0)_A} + x^{(1)_A}$ be an arithmetic masking. After each of the shares $x^{(0)_A}$ and $x^{(1)_A}$ is transformed into a Boolean masking, they can be added securely using SecADD. Since the outputs of SecADD are Boolean-masked, this corresponds exactly to an A2B conversion. For prime moduli, a similar approach can be applied. Here, the addition for power-of-two moduli, SecADD, is replaced by the SecADD$_q$ algorithm as introduced earlier.

The A2B and A2B$_q$ conversion methods for power-of-two and prime moduli based on this idea are illustrated in Algorithms 20 and 21, respectively.

Algorithm 20 A2B [30]

Input: An arithmetic sharing $\boldsymbol{x} = (x^{(0)_A}, x^{(1)_A})$, where $x = x^{(0)_A} + x^{(1)_A} \bmod 2^k$.

Output: A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, x^{(1)_B})$, such that $x = x^{(0)_B} \oplus x^{(1)_B}$.

```
1: $r^{(0)_A}, r^{(1)_A} \leftarrow \mathcal{U}(\mathbb{Z}_{2^k})$
2: $\boldsymbol{z}_1^{(\cdot)_B} \leftarrow (x^{(0)_A} \oplus r^{(0)_A}, r^{(0)_A})$
3: $\boldsymbol{z}_2^{(\cdot)_B} \leftarrow (x^{(1)_A} \oplus r^{(1)_A}, r^{(1)_A})$
4: $\boldsymbol{x}^{(\cdot)_B} \leftarrow \mathsf{SecADD}(\boldsymbol{z}_1^{(\cdot)_B}, \boldsymbol{z}_2^{(\cdot)_B})$
5: return $\boldsymbol{x}^{(\cdot)_B}$
```

Algorithm 21 A2B$_q$ [9]

Input: An arithmetic sharing $\boldsymbol{x} = (x^{(0)_A}, x^{(1)_A})$, where $x = x^{(0)_A} + x^{(1)_A} \bmod q$.

Output: A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, x^{(1)_B})$, such that $x = x^{(0)_B} \oplus x^{(1)_B}$.

```
1: $r^{(0)_A}, r^{(1)_A} \leftarrow \mathcal{U}(\mathbb{Z}_q)$
2: $\boldsymbol{z}_1^{(\cdot)_B} \leftarrow (x^{(0)_A} \oplus r^{(0)_A}, r^{(0)_A})$
3: $\boldsymbol{z}_2^{(\cdot)_B} \leftarrow ((x^{(1)_A} + (2^w - q)) \oplus r^{(1)_A}, r^{(1)_A})$
4: $\boldsymbol{x}^{(\cdot)_B} \leftarrow \mathsf{SecADD}_q(\boldsymbol{z}_1^{(\cdot)_B}, \boldsymbol{z}_2^{(\cdot)_B})$
5: return $\boldsymbol{x}^{(\cdot)_B}$
```

3.4.3 Masked B2A Conversion

B2A conversion allows to securely transform the Boolean shares $x = x^{(0)_B} \oplus x^{(1)_B}$ into the arithmetic shares $x = x^{(0)_A} + x^{(1)_A}$ without unmasking x. Similarly to the A2B conversion, the B2A conversion for the first-order masking is computed as

$$x^{(0)_A} = (x^{(0)_B} \oplus x^{(1)_B}) - x^{(1)_B},$$
$$x^{(1)_A} = x^{(1)_B}.$$

However, in contrast to the A2B conversion, the B2A conversions are typically not table-based. The reason is that the B2A operation in general requires less execution time compared to A2B conversion, as it contains mostly the linear operations and can be implemented efficiently even without the precomputation. For a power-of-two moduli, B2A conversion methods were extensively researched [17, 21, 30]; the most efficient approach for the first-order masking scenario was proposed by Goubin [40] and is given in Algorithm 22. For prime moduli, the authors of [71] have presented an efficient B2A$_q$ conversion method, summarized in Algorithm 23.

Algorithm 22 B2A [40]

Input: A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, r)$, where $x = x^{(0)_B} \oplus r$.

Output: An arithmetic sharing $\boldsymbol{x} = (x^{(0)_A}, r)$, such that $x = x^{(0)_A} + r \pmod{2^w}$.

```
1: $\Gamma \leftarrow \mathcal{U}(\{0,1\}^w)$
2: $T = x^{(0)_B} \oplus \Gamma,\ T = T - \Gamma$
3: $T = T \oplus x^{(0)_B},\ \Gamma = \Gamma \oplus r$
4: $x^{(0)_A} = x^{(0)_B} \oplus \Gamma$
5: $x^{(0)_A} = x^{(0)_A} - \Gamma$
6: $x^{(0)_A} = x^{(0)_A} + T$
7: return $(x^{(0)_A}, r)$
```

Algorithm 23 B2A$_q$ [71]

Input: A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, x^{(1)_B})$, where $x^{(0)_B} \oplus x^{(1)_B} = x \in \mathbb{Z}_{2^k}$.

Output: An arithmetic sharing $\boldsymbol{z} = (z^{(0)_A}, z^{(1)_A})$, such that $z^{(0)_A} + z^{(1)_A} = x \bmod q$.

```
1: $A \leftarrow (x^{(0)_B}, 0)$
2: $B_1 \leftarrow \mathcal{U}(\mathbb{Z}_q)$
3: $B_0 \leftarrow A_0 - B_1 \bmod q$
4: $z^{(0)_A} \leftarrow B_0 - 2 \cdot (B_0 \cdot x^{(1)_B}) + x^{(1)_B} \bmod q$
5: $z^{(1)_A} \leftarrow B_1 - 2 \cdot (B_1 \cdot x^{(1)_B}) \bmod q$
6: $\mathsf{SecREF}(\boldsymbol{z}, q)$
7: return $\boldsymbol{z}$
```

3.5 Masked Binomial Sampler

Many LWE-based schemes include binomial samplers to generate secret and error vectors according to a given distribution. In Kyber and Saber, a centered binomial distribution is used, which computes a sample as

$$\beta_\eta = \sum_{i=0}^{\eta-1}(x_i - y_i),$$

where $|\beta_\eta| \leq \eta$ and the individual bits x_i, y_i are uniformly distributed. Since the generated values are sensitive, this operation should be masked. A first-order secure masked sampling method was proposed in [61]. Following, the authors of [71] presented a more efficient solution based on bit-slicing, which can generate multiple samples in parallel and works for arbitrary security orders. Given two Boolean-shared vectors $\boldsymbol{x}$, $\boldsymbol{y}$, a masked binomial sampler needs to compute the difference of the Hamming weights of these vectors and produce $\boldsymbol{z} = \mathsf{HW}(\boldsymbol{x}) - \mathsf{HW}(\boldsymbol{y})$ in a secure fashion. Since the result of subtraction $\boldsymbol{z}$ is Boolean-masked, it should be converted into arithmetic masking, as the sampled outputs are used in the subsequent polynomial arithmetic operations. For that, an additional B2A conversion is required to transform $\boldsymbol{z}$ into arithmetic shares. This conversion is instantiated with B2A$_q$ and B2A algorithms from the previous section for Kyber and Saber, respectively.

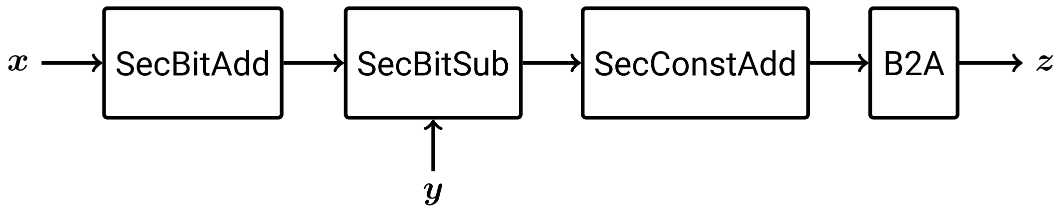

Figure 4: An overview of various components in the bit-sliced binomial sampler (Algorithm 27).

Algorithm 24 SecBitAdd [71]

Input: 1. A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, x^{(1)_B})$, where $x = x^{(0)_B} \oplus x^{(1)_B}$,
2. A bit size η.

Output: A Boolean sharing $\boldsymbol{z} = (z^{(0)_B}, z^{(1)_B})$, such that $z = z^{(0)_B} \oplus z^{(1)_B} = \mathsf{HW}(x)$.

```
1:  $\boldsymbol{t} \leftarrow 0,\ \boldsymbol{z} \leftarrow 0$
2:  for $j = 0$ to $\eta - 1$ do
3:      $\boldsymbol{t}_0 \leftarrow \boldsymbol{z}_0 \oplus \boldsymbol{x}_j$
4:      $\boldsymbol{w} \leftarrow \boldsymbol{x}_j$
5:      for $l = 1$ to $\lceil \log_2(\eta + 1) \rceil$ do
6:          $\boldsymbol{w}_0 \leftarrow \boldsymbol{z}_0 \oplus \boldsymbol{x}_j$
7:          $\boldsymbol{w} \leftarrow \mathsf{SecAND}(\boldsymbol{w}, \boldsymbol{z}_l)$
8:          $\boldsymbol{t}_l \leftarrow \boldsymbol{z}_l \oplus \boldsymbol{w}$
9:      $\boldsymbol{z} \leftarrow \boldsymbol{t}$
10: return $(z^{(0)_B}, z^{(1)_B})$
```

Algorithm 25 SecBitSub [71]

```
Input:  1. A Boolean sharing $\boldsymbol{z} = (z^{(0)_B}, z^{(1)_B})$, where $z = z^{(0)_B} \oplus z^{(1)_B}$,
        2. A Boolean sharing $\boldsymbol{y} = (y^{(0)_B}, y^{(1)_B})$, where $y = y^{(0)_B} \oplus y^{(1)_B}$,
        3. A bit size $\eta$.
Output: A Boolean sharing $\boldsymbol{z} = (z^{(0)_B}, z^{(1)_B})$, such that $z = z^{(0)_B} \oplus z^{(1)_B} = z - \mathsf{HW}(x)$.
 1: $\boldsymbol{t} \leftarrow 0$
 2: for $j = 0$ to $\eta - 1$ do
 3:     $\boldsymbol{t}_0 \leftarrow \boldsymbol{z}_0 \oplus \boldsymbol{y}_j$
 4:     $\boldsymbol{w} \leftarrow \boldsymbol{y}_j$
 5:     for $l = 1$ to $\lceil \log_2(\eta + 1) \rceil$ do
 6:         $\boldsymbol{u} \leftarrow \boldsymbol{z}_l,\ u^{(0)_B} \leftarrow \neg u^{(0)_B}$
 7:         $\boldsymbol{w} \leftarrow \mathsf{SecAND}(\boldsymbol{w}, \boldsymbol{u})$
 8:         $\boldsymbol{t}_l \leftarrow \boldsymbol{z}_l \oplus \boldsymbol{w}$
 9:     $\boldsymbol{z} \leftarrow \boldsymbol{t}$
10: return $(z^{(0)_B}, z^{(1)_B})$
```

Algorithm 26 SecConstAdd [71]

```
Input:  1. A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, x^{(1)_B})$, where $x = x^{(0)_B} \oplus x^{(1)_B}$,
        2. A bit size $\eta$.
Output: A Boolean sharing $\boldsymbol{y} = (y^{(0)_B}, y^{(1)_B})$, such that $y = y^{(0)_B} \oplus y^{(1)_B} = x + \eta$.
 1: $\boldsymbol{t} \leftarrow \eta$
 2: $\boldsymbol{t} \leftarrow \mathsf{SecREF}(\boldsymbol{t}, \lceil \log_2(\eta + 1) \rceil)$
 3: $\boldsymbol{y} \leftarrow \mathsf{SecADD}(\boldsymbol{x}, \boldsymbol{t})$
 4: return $(y^{(0)_B}, y^{(1)_B})$
```

Algorithm 27 SecSampler [71]

```
Input:  1. A Boolean sharing $\boldsymbol{x} = (x^{(0)_B}, x^{(1)_B})$, where $x = x^{(0)_B} \oplus x^{(1)_B}$,
        2. A Boolean sharing $\boldsymbol{y} = (y^{(0)_B}, y^{(1)_B})$, where $y = y^{(0)_B} \oplus y^{(1)_B}$,
        3. A bit size $\eta$.
Output: An arithmetic sharing $\boldsymbol{z} = (z^{(0)_A}, z^{(1)_A})$, such that $z = z^{(0)_A} + z^{(1)_A} = \mathsf{HW}(x) - \mathsf{HW}(y) \bmod q$.
 1: $\boldsymbol{z} \leftarrow \mathsf{SecBitAdd}(\boldsymbol{x}, \eta)$
 2: $\boldsymbol{z} \leftarrow \mathsf{SecBitSub}(\boldsymbol{z}, \boldsymbol{y}, \eta)$
 3: $\boldsymbol{z} \leftarrow \mathsf{SecConstAdd}(\boldsymbol{z}, \eta)$
 4: $\boldsymbol{z} \leftarrow \mathsf{B2A}_q(\boldsymbol{z}, \eta)$
 5: $z^{(0)_A} \leftarrow z^{(0)_A} - \eta (\bmod q)$
 6: return $(z^{(0)_A}, z^{(1)_A})$
```

The masked bitsliced sampler is given in Algorithm 27 and illustrated in Figure 4. It consists of three functions SecBitAdd, SecBitSub, SecConstAdd, described in Algorithms 24-26, and a B2A$_q$ or a B2A conversion for the case of Kyber and Saber, respectively. The first operation, SecBitAdd, takes the Boolean-shared input $\boldsymbol{x}$ and securely computes its Hamming weight $\boldsymbol{z} = \mathsf{HW}(\boldsymbol{x})$. Following, the SecBitSub operation takes the Boolean shares $\boldsymbol{z}$, $\boldsymbol{y}$ and outputs $\boldsymbol{z} = \boldsymbol{z} - \mathsf{HW}(\boldsymbol{y}) = \mathsf{HW}(\boldsymbol{x}) - \mathsf{HW}(\boldsymbol{y})$. To avoid negative results after subtraction and correctly covert the result into arithmetic shares, a constant η is added to $\boldsymbol{z}$ in a masked fashion in the SecConstAdd operation. This addition is later compensated by subtracting η from the arithmetic shares.

3.6 Masked Logical Shift

Since Saber uses power-of-two moduli $q = 2^{13}$ and $p = 2^{10}$, the ciphertext compression can be replaced by a simple logical shift operation, applied to each polynomial coefficient separately. For arithmetically masked coefficients, this operation requires special treatment, as the lower bits of the coefficients might contain a carry that should be added to the upper bits before shifting them out. A straightforward solution is to perform the A2B conversion before the shift, since the bits in a Boolean masking are independent of carries.

Algorithm 28 Table C precomputation [31, 35].

```
Input: An integer $k$.
Output: A look-up table $C$, an integer $r$, an integer $\gamma$.
1: $r \leftarrow \mathcal{U}(\{0,1\}^k),\ \gamma \leftarrow \mathcal{U}(\{0,1\}^k)$
2: for $i = 0$ to $2^k - 1$ do
3:     $C[i] = \begin{cases} \gamma, & \text{if } i + r < 2^k, \\ \gamma + 1,\ \text{mod} 2^k & \text{if } i + r \geq 2^k. \end{cases}$
4: return $(C, r, \gamma)$
```

Algorithm 29 A2A conversion of a $m + (n \cdot k)$-bit variable [12].

```
Input: 1. An arithmetic sharing $\boldsymbol{x} = (x^{(0)_A}, x^{(1)_A})$, such that $x = x^{(0)_A} + x^{(1)_A} \pmod{2^{m+n\cdot k}}$,
       2. Table $C$, integers $r$, $\gamma$, generated during precomputation phase.
Output: A Boolean sharing $\boldsymbol{z} = (z^{(0)_B}, z^{(1)_B})$, such that
        $z = z^{(0)_B} \oplus z^{(1)_B} = x \gg (n \cdot k) = x^{(0)_A} + x^{(1)_A} \bmod 2^m$.
1: $\Gamma \leftarrow \sum_{i=1}^{n} 2^{i \cdot k} \cdot \gamma \bmod 2^{m+(n \cdot k)}$
2: $P \leftarrow \sum_{i=0}^{n-1} 2^{i \cdot k} \cdot r$
3: $x^{(0)_A} \leftarrow x^{(0)_A} - P \bmod 2^{m+(n \cdot k)}$
4: $x^{(0)_A} \leftarrow x^{(0)_A} - \Gamma \bmod 2^{m+(n \cdot k)}$
5: for $i = 0$ to $n - 1$ do
6:     Split $x^{(0)_A}$ into $(A_h || A_l)$ and $x^{(1)_A}$ into $(R_h || R_l)$, such that $|A_l| = |R_l| = k$
7:     $x^{(0)_A} \leftarrow x^{(0)_A} + R_l \bmod 2^{m+(n-i) \cdot k}$
8:     $A_h \leftarrow A_h + C[A_l] \bmod 2^{m+(n-i-1) \cdot k}$
9:     $x^{(0)_A} \leftarrow A_h,\ x^{(1)_A} \leftarrow R_h$
10: return $\boldsymbol{z}$
```

To optimize this approach, the authors of [12] have proposed a new primitive, named A2A (arithmetic to arithmetic) conversion. It is based on a table-based A2B conversion method, described in Section 3.4.2, however instead of performing a full conversion, only the carry for the lower bits is computed. This carry is then added to the higher bits, leaving them as an arithmetic sharing. Algorithms 28 and 29 give the full details of the A2A conversion method.

3.7 Masked Compression

Both Kyber and Saber include compression operation that discards low-order bits of the polynomial coefficients. This operation is sensitive and should be implemented in a secure fashion. Since Saber uses a power-of-two moduli, the compression step can be replaced by a masked logical shift, described in the previous section. In the following, we consider the compression algorithms for Kyber, which uses a prime modulus. In the decryption step of Kyber (line 3, Algorithm 3), 1-bit compression is used to decode the message, i.e. an arithmetically masked polynomial coefficient is mapped to a Boolean-masked bit-string m. Furthermore, in the re-encryption step of Kyber (lines 6 and 7, Algorithm 5) d_u- and d_v-bit compression is used to compare the re-encrypted ciphertext with the original compressed ciphertext. In case of the Kyber768 parameter set, these parameters are equal to $d_u = 10$ and $d_v = 4$.

3.7.1 Masked 1-bit compression

For prime moduli, the one-bit compression can be expressed as an interval comparison:

$$\mathsf{Compress}_q(x,1) = \left\lceil (2^d/q)\cdot x \right\rfloor \mod 2 = \begin{cases} 1 & \text{for } \frac{q}{4} < x < \frac{3q}{4}, \\ 0 & \text{otherwise.} \end{cases}$$

This interval check can be easily computed if the implementation is unprotected, however it poses a challenge when a prime modulus q and masking is used. For arithmetic sharing, Bos et al. [24] proposed a masked binary search approach, which requires one A2B conversion per polynomial coefficient. First, the bounds of the interval above are shifted by subtracting $\frac{q}{4}$ from the arithmetic shares of the coefficient, such that only one interval comparison needs to be performed. This shifted function can be expressed as

$$\mathsf{Compress}_q(x,1) = \mathsf{Compress}^s_q\left(x - \left\lfloor \frac{q}{4} \right\rfloor \mod q\right) = \begin{cases} 0 & \text{for } x < \frac{q}{2}, \\ 1 & \text{otherwise.} \end{cases}$$

After the result is converted to k-bit Boolean shares, it suffices to check if the masked value is smaller than $\frac{q}{2}$. To improve performance, all the coefficients are compressed in parallel by transforming the Boolean shares into a bitsliced representation. The masked binary search is then performed as follows. Starting from the most-significant bit (MSB), the Boolean-shared coefficient is processed bit-by-bit. For instance, if the MSB is set to 1, all the remaining bits can be ignored and the coefficient can be set to 1, since $2^{\mathsf{MSB}} = 2^{k-1} > \frac{q}{2}$. If the MSB is set to 0, all the subsequent bits need to be considered. This process is repeated for the remaining bits until all possible values of the coefficients have been mapped to a single bit value. In case of Kyber, where $k = \lceil \log_2(q) \rceil = 12$ and $\lfloor \frac{q}{2} \rfloor = 1664$, $\mathsf{Compress}^s_q$ is computed as

$$\mathsf{Compress}^s_q(x) = x_{11} \oplus (\neg x_{11} \wedge x_{10} \wedge x_9 \wedge (x_8 \oplus (\neg x_8 \wedge x_7))).$$

In a masked implementation, the regular $\oplus$ and $\wedge$ operations should be replaced with the secure gadgets SecXOR and SecAND. The complete masked one-bit compression is summarized in Algorithm 30.

Algorithm 30 $\mathsf{MaskedCompress}_q(x, 1) = \mathsf{Compress}^s_q(x - \lfloor \frac{q}{4} \rfloor \mod q)$ [24]

```
Input: An arithmetic sharing $a = a^{(\cdot)_A} = (a^{(0)_A}, a^{(1)_A})$ of a polynomial $a \in \mathbb{Z}_q[X]$.
Output: A Boolean sharing $m = m^{(\cdot)_B} = (m^{(0)_B}, m^{(1)_B})$ of the message $m^{(\cdot)_B} = \mathsf{Compress}_q(a, 1) \in \mathbb{Z}_{2^{256}}$.
 1: for $i = 0$ to $n$ do
 2:     $a_i^{(0)_A} = a_i^{(0)_A} - \lfloor \frac{q}{4} \rfloor \mod q$
 3:     $a_i^{(\cdot)_B} = \mathsf{A2B}_q(a_i^{(\cdot)_A})$
 4: $x^{(\cdot)_B} = \mathsf{Bitslice}(a^{(\cdot)_B})$
 5: $m^{(\cdot)_B} = \mathsf{SecAND}(\mathsf{SecREF}(\neg x_8^{(\cdot)_B}, x_7^{(\cdot)_B}))$
 6: $m^{(\cdot)_B} = \mathsf{SecREF}(\mathsf{SecXOR}(m^{(\cdot)_B}, x_8^{(\cdot)_B}))$
 7: $m^{(\cdot)_B} = \mathsf{SecAND}(m^{(\cdot)_B}, x_9^{(\cdot)_B})$
 8: $m^{(\cdot)_B} = \mathsf{SecAND}(m^{(\cdot)_B}, x_{10}^{(\cdot)_B})$
 9: $m^{(\cdot)_B} = \mathsf{SecAND}(m^{(\cdot)_B}, \neg x_{11}^{(\cdot)_B})$
10: $m^{(\cdot)_B} = \mathsf{SecXOR}(m^{(\cdot)_B}, x_{11}^{(\cdot)_B})$
11: return $m^{(\cdot)_B}$
```

3.7.2 Masked d-bit compression

The one-bit compression technique described above does not scale up to d-bit compression. In case $d = 1$, there are two intervals of width $\frac{q}{2}$ and the coefficient can be decoded efficiently by performing a single comparison. However, for $d = n$ with $n > 1$, there are 2^n intervals of width $\frac{q}{2^n}$, which makes this method impractical. A different masked compression technique was proposed in [37]. Instead of performing masked interval check, the division by modulus q is masked. The key observation is that the compression operation tolerates approximate results for division on binary fraction, namely

$$\mathsf{Compress}_q(x, d) = \lfloor (2^d/q) \cdot x + e \rceil \mod 2^d$$

remains correct for a small error e. In case of masked compression, such an approximate quotient can be computed individually from the shares of $x = x^0 + x^1 \mod q$, using integer division:

$$\mathsf{Compress}_q(x, d) = \lfloor (2^d/q) \cdot x^0 \rfloor_f + \lfloor (2^d/q) \cdot x^1 \rfloor_f = \lfloor (2^d/q) \cdot x + e \rceil \mod 2^d,$$

where the error e can be corrected by using extra precision, namely by increasing the number of fractional bits f. For first-order security, it suffices that $f \geq 13$ to bound the error e by $-\frac{1}{2q} \leq e < \frac{1}{2q}$. The final output consists of d upper bits of the $d + f$ Boolean sharing $(\lfloor (2^d/q) \cdot x^0 \rfloor_f, \lfloor (2^d/q) \cdot x^1 + e \rfloor_f)$, computed with a $(d + f)$-bit A2B conversion and share-wise logical shift. This method requires one A2B call per coefficient and is illustrated in Algorithm 31.

Algorithm 31 $\mathsf{MaskedCompress}_q(x, d)$ [37]

```
Input:  1. An arithmetic sharing $\boldsymbol{a}^{(\cdot)_A} = (a^0, a^1)$, where $a = a^{(0)_A} + a^{(1)_A}$,
        2. The bit width of integer and fractional parts $d$ and $f$, where $f >\ \log_2(2q)$
Output: A Boolean sharing $\boldsymbol{m}^{(\cdot)_B} = (m^{(0)_B}, m^{(1)_B})$, where $m = m^{(0)_B} + m^{(1)_B} = \mathsf{Compress}_q(a, 1)$.
1: $a^{(0)_A} = \lfloor (2^{d+f} \cdot a^{(0)_A})/q \rfloor \bmod 2^{d+f}$
2: $a^{(1)_A} = (\lfloor (2^{d+f} \cdot a^{(1)_A})/q \rfloor + 2^f \cdot 0.5) \bmod 2^{d+f}$
3: $\boldsymbol{m}^{(\cdot)_B} = \mathsf{A2B}(\boldsymbol{a}^{(\cdot)_A})$
4: $\boldsymbol{m}^{(\cdot)_B} = (\boldsymbol{m}^{(\cdot)_A} \gg f)$
5: return $\boldsymbol{m}^{(\cdot)_B}$
```

3.8 Masked Comparison

At the end of the decapsulation, the public ciphertext $\boldsymbol{c} = (\boldsymbol{c}_1, c_2)$ is compared with the masked re-encrypted ciphertext $\boldsymbol{c'} = (\boldsymbol{c}_1'^{(0)_B} \oplus \boldsymbol{c}_1'^{(1)_B},\ c_2'^{(0)_B} \oplus c_2'^{(1)_B})$. To prevent leakage of sensitive variables, the ciphertext $\boldsymbol{c'}$ shouldn't be unmasked, as its value depends on the secret $\boldsymbol{s}$.

3.8.1 Hash-based Comparison

Since the masked comparison tests both ciphertexts for equality, it assumes prior ciphertext compression after re-encryption. In case of Kyber, the comparison checks whether

$$\boldsymbol{c'} = (\mathsf{Compress}_q(\boldsymbol{u'}, d_u), \mathsf{Compress}_q(v', d_v)) \overset{?}{=} c.$$

after the compression step in Kyber.CPAPKE.Enc. Similarly, in Saber the rounded logical shift is performed in Saber.CPAPKE.Enc in order to verify whether

$$\boldsymbol{c'} = ((\boldsymbol{u'} \gg (\epsilon_q - \epsilon_p)), (v' \gg (\epsilon_q - \epsilon_T))) \overset{?}{=} c.$$

As the masked ciphertext $\boldsymbol{c'}$ depends on the secret $\boldsymbol{s}$, the comparison operation should be performed securely, without unmasking $\boldsymbol{c'}$. A first-order secure algorithm for masked ciphertext comparison has been proposed in [61]. The core idea is to include an additional hashing step before the comparison and check whether $\mathcal{H}(c_1'^1) = \mathcal{H}(c_1 \oplus c_1'^0)$ and $\mathcal{H}(c_2'^1) = \mathcal{H}(c_2 \oplus c_2'^0)$, which is only true if the ciphertext $\boldsymbol{c'}$ is valid due to collision-resistance of $\mathcal{H}$. Recently, this approach was shown to be vulnerable [18], because the two separate equality checks of the masked components $\boldsymbol{c}_1, c_2$ leak sensitive information about $\boldsymbol{c}$. By performing a single check on the concatenated ciphertext components, the authors of [12] have shown that this leakage can be prevented. In detail, the vulnerability in [61] is fixed by computing

$$\mathcal{H}(c_1' \,||\, c_2') \overset{?}{=} \mathcal{H}(c_1 \oplus c_1'^0 \,||\, c_2 \oplus c_2'^0).$$

3.8.2 Masked Decompressed Comparison

The authors of [24] have suggested another way to check Kyber ciphertexts for the equality. The key idea is to skip the costly masked compression step inside the Kyber.CPAPKE.Enc, and to decompress the public ciphertext c instead. This changes the equality test above to

$$(\boldsymbol{u'}, v') \stackrel{?}{=} \mathsf{Decompress}_q(c, d),$$

where $d = d_u$ and $d = d_v$ for the first and second parts of the ciphertext, respectively.
The new approach compares a public compressed ciphertext with a masked uncompressed ciphertext coefficient by coefficient. The results for all individual comparisons are then combined into one masked output bit. This method requires two A2B conversions per coefficient and is described in Algorithm 32.

Algorithm 32 Masked DecompressedComparison [24]

Input: 1. An arithmetic sharing $\boldsymbol{u'}^{(\cdot)_A} = (u^{(0)_A}, u^{(1)_A})$, where $u = u^{(0)_A} + u^{(1)_A}$,
2. An arithmetic sharing $v'^{(\cdot)_A} = (v^{(0)_A}, v^{(1)_A})$, where $v = v^{(0)_A} + v^{(1)_A}$,
3. A ciphertext c,
4. Two public functions S and E defined by Kyber, which specify the start- and end-points of the intervals in compression

Output: A Boolean sharing $b^{(\cdot)_B} = (b^0, b^1)$, where $b = b^{(0)_B} \oplus b^{(1)_B} = 1$ if $(\mathsf{Compress}_q(\boldsymbol{u'}, d_u), \mathsf{Compress}_q(v', d_v) = c$, otherwise $b = 0$.

```
1:  function DecompressedComparison
2:      $(\boldsymbol{u''}, v'') = \mathsf{Decompress}_q(c)$
3:      $t_{\boldsymbol{w}}^{(\cdot)_B}, t_{\boldsymbol{x}}^{(\cdot)_B} = \mathsf{PolyCompare}(\boldsymbol{u'}^{(\cdot)_A}, \boldsymbol{u''})$
4:      $t_y^{(\cdot)_B}, t_z^{(\cdot)_B} = \mathsf{PolyCompare}(v'^{(\cdot)_A}, v'')$
5:      $b^{(\cdot)_B} = \mathsf{SecAND}(\mathsf{SecAND}(t_{\boldsymbol{w}}^{(\cdot)_B}, t_{\boldsymbol{x}}^{(\cdot)_B}),\ \mathsf{SecAND}(t_y^{(\cdot)_B}, t_z^{(\cdot)_B}))$
6:      for $i = \log_2(n-1)$ to $0$ do
7:          $t_b^{(\cdot)_B} = b^{(\cdot)_B} \gg 2^i$
8:          $b^{(\cdot)_B} = b^{(\cdot)_B} \bmod (2^{2^i} - 1)$
9:          $b^{(\cdot)_B} = \mathsf{SecAND}(b^{(\cdot)_B}, t_b^{(\cdot)_B})$
10:     return $b^{(\cdot)_B}$
11:
12: function PolyCompare($u'^{(\cdot)_A}_i, u''$)
13:     for $i = 0$ to $n-1$ do
14:         $s_{u''} = \mathsf{S}(u''_i)$; $e_{u''} = \mathsf{E}(u''_i)$
15:         $w_i^{(\cdot)_A} = x_i^{(\cdot)_A} = u'^{(\cdot)_A}_i$
16:         $w_i^{(0)_A} = (w_i^{(0)_A} + 2^{\lceil \log_2(q) \rceil - 1} - s_{u''}) \bmod q$
17:         $x_i^{(0)_A} = (x_i^{(0)_A} - e_{u''}) \bmod q$
18:         $w_i^{(\cdot)_B} = \mathsf{MSB}(\mathsf{A2B_q}(w_i^{(\cdot)_A}))$
19:         $x_i^{(\cdot)_B} = \mathsf{MSB}(\mathsf{A2B_q}(x_i^{(\cdot)_A}))$
20:     return $\mathsf{Bitslice}(w^{\cdot}_B), \mathsf{Bitslice}(x^{\cdot}_B)$
```

Let us consider one public coefficient b as an example. The ciphertext comparison $a' \stackrel{?}{=} \mathsf{Decompress}_q(b,d)$ is computed as follows. Since the coefficient b is compressed, it contains only approximate value, i.e., only the d most-significant bits of b are kept, the other lower bits are discarded. Given that, the public value b cannot be mapped to a single value of a; instead it should be decompressed to an interval $\mathsf{S}(b) \leq a \leq \mathsf{E}(b) - 1$, where $\mathsf{Compress}_q(a) = b$ holds for every a. We denote by $\mathsf{S}(b)$ and $\mathsf{E}(b)$ the public functions which compute the lower (resp. upper bounds) of this interval. Next, one has to check if the private uncompressed coefficient a' falls within the interval $[\mathsf{S}(b), \mathsf{E}(b-1)]$. Since the value a' is sensitive, this should be done in a masked fashion. The core idea is first to subtract the interval bounds from a' and then extract the MSB by converting the arithmetically masked coefficients into Boolean masking. If a' is indeed within the given interval, then the MSB of $a' - \mathsf{S}(b)$ should be 0, while the MSB of $a' - \mathsf{E}(b)$ should be 1. The results of all individual comparisons are then combined by computing the SecAND operation until one output bit remains.

3.9 Comparing Masking for Kyber and Saber

As can be seen in Figures 2 and 3, the design of Kyber is very similar to Saber, which allows to apply the similar masking techniques. The main difference between both schemes is due to the MLWE and MLWR problems, used in Kyber an Saber, respectively. In Kyber, additional calls to functions $\mathcal{H}$ and β_η for sampling the error terms $\boldsymbol{e_1}$, e_2, are required. The sampling step includes extra $\mathsf{B2A}_q$ conversions as the outputs of the sampler should be arithmetically masked. Moreover, Kyber includes a Decompression step during the re-encryption (line 5 of the Algorithm 5), which can be implemented with another $\mathsf{B2A}_q$ conversion and a multiplication with a constant. In Saber, this operation is implicit, since the logical left shift (line 5 of the Algorithm 6) results in an arithmetic sharing mod p. Another difference is in the arithmetic operations modulo prime q in Kyber versus modulo power-of-two in Saber. For power-of-two moduli, the compression operation corresponds to a share-wise logical shift. Furthermore, the computations modulo q are more expensive. For instance, the $\mathsf{A2B}_q$ conversion based on masked addition has roughly twice the complexity of A2B, since SecADD_q, makes two calls to SecADD.

4 New Decompressed Comparison

In this chapter, we introduce a new countermeasure for masked comparison based on [24]. Our new algorithm compares the uncompressed masked polynomials with compressed public polynomials using a bitsliced binary search approach. It can be applied in first- and higher-order settings and is faster compared to the original method, since it requires only one A2B conversion per coefficient.

4.1 Construction

In the original algorithm from [24], the authors presented a way to test a masked uncompressed ciphertext and a public compressed ciphertext for equality. Their method is repeated for each pair of polynomial coefficients (a', b) and works as follows. Given a public coefficient b compressed to d bits, the first step is to determine the interval $[\mathsf{S}(b), \mathsf{E}(b) - 1]$ containing the possible values of b after decompression. The lower and upper bounds of this interval are computed with the public functions $\mathsf{S}(b)$ and $\mathsf{E}(b)$, defined by Kyber, respectively. The next step is to verify, whether the masked uncompressed coefficient a' is indeed within the range $\mathsf{S}(b) \leq a' \leq \mathsf{E}(b) - 1$. In other words, the comparison

$$a' \overset{?}{=} \mathsf{Decompress}_q(b, d)$$

returns true, if both interval checks $a' - \mathsf{S}(b) < 0$ and $a' - (\mathsf{E}(b) - 1) > 0$ return false. The results of the individual checks are then converted to Boolean shares and combined into a single masked output bit by performing the subsequent calls to SecAND. This method requires two A2B conversions per polynomial coefficient and is given in Algorithm 32.

In the following, we describe our new algorithm to perform the comparison between a masked re-encrypted ciphertext and unmasked public ciphertext based on a bitsliced binary search approach. The core idea is to shift the interval of the possible decompressed values $[\mathsf{S}(b), \mathsf{E}(b) - 1]$ and the uncompressed masked coefficient a' by the lower bound $\mathsf{S}(b)$ and instead to verify whether the shifted coefficient $a' - \mathsf{S}(b)$ lies within the shifted interval $[0,\ \mathsf{E}(b) - \mathsf{S}(b) - 1]$. After the shifted value $a' - \mathsf{S}(b)$ is converted to Boolean shares, this check can be done by performing a single comparison $a' - \mathsf{S}(b) < \mathsf{E}(b) - \mathsf{S}(b)$. We denote the function which computes the comparison as

$$\mathsf{Compare}_q(a', d) = \begin{cases} 0 & \text{for } a' - \mathsf{S}(b) < \mathsf{E}(b) - \mathsf{S}(b), \\ 1 & \text{otherwise.} \end{cases}$$

The size of the shifted interval depends on the bit width of the compressed coefficient. For d-bit compressed values, the possible interval sizes are $\lfloor \frac{q}{2^d} \rfloor$ and $\lceil \frac{q}{2^d} \rceil$. As in the Kyber768 parameter set two values of d, namely $(d_u, d_v) = (10, 4)$, are defined, we need distinguish two cases for the comparison of 10-bit and 4-bit values. In both cases, the $\mathsf{Compare}_q$ operation can be computed in a masked fashion by performing a masked binary search on the Boolean-shared bits of the shifted coefficient. The results of all individual comparisons can then be combined together by performing the subsequent SecAND calls until a single masked output bit remains.

4.2 10-bit comparison

The first important building block of our new countermeasure is the comparison operation, that verifies whether the Boolean-masked coefficient $a = a' - \mathsf{S}(b)$ is within the range $0 \leq a' - \mathsf{S}(b) < \mathsf{E}(b) - \mathsf{S}(b)$. For 10-bit values, the interval $[0,\ \mathsf{E}(b) - \mathsf{S}(b) - 1]$ can take two possible sizes, dependent on the value of the public coefficient b:

$$size_a = \mathsf{E}(b) - \mathsf{S}(b) = \begin{cases} 4 = \left\lceil \frac{q}{2^{10}} \right\rceil & \text{if } b \bmod 4 = 1 \text{ and } 128 \leq \mathsf{Decompress}_q(b, 10) < 384, \\ & \text{if } b \bmod 4 = 0 \text{ and } 384 \leq \mathsf{Decompress}_q(b, 10) < 640, \\ & \text{if } b \bmod 4 = 3 \text{ and } 640 \leq \mathsf{Decompress}_q(b, 10) < 896, \\ & \text{if } b \bmod 4 = 2 \text{ and } 896 \leq \mathsf{Decompress}_q(b, 10) < 128, \\ 3 = \left\lfloor \frac{q}{2^{10}} \right\rfloor & \text{otherwise.} \end{cases}$$

We illustrate the case $size_a = 4$ graphically in Figure 5. In the following, we describe the detailed steps to derive the algorithm, which computes $a' - \mathsf{S}(b) \geq \mathsf{E}(b) - \mathsf{S}(b)$, using OR, AND, XOR and negation operations. These operations are denoted by $\vee$, $\wedge$, $\oplus$ and $\neg$, respectively.

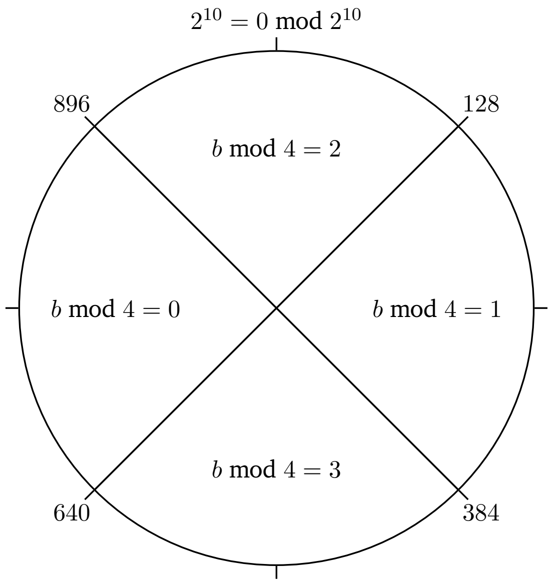

Figure 5: $size_a = \mathsf{E}(b) - \mathsf{S}(b) = 4$

4.2.1 Boolean Expressions for 10-bit Comparison

Let $a = a' - \mathsf{S}(b)$ denote a Boolean-masked shifted coefficient. We define the comparison operation as

$$\mathsf{Compare}_q(a, 10) = \begin{cases} 0 & \text{if } a \leq size_a, \\ 1 & \text{otherwise.} \end{cases}$$

This operation can be computed in a masked fashion by performing a bitsliced binary search on the bits of a. Since there are two possible sizes of the interval $[0,\ \mathsf{E}(b) - \mathsf{S}(b) - 1]$, we need to securely decide whether the masked value a is greater than $size_a = \left\lceil \frac{q}{2^{10}} \right\rceil = 4$ or $size_a = \left\lfloor \frac{q}{2^{10}} \right\rfloor = 3$. To do so, we first derive the equations for computing the two comparisons separately. Then, we combine both cases into a single comparison.

Case 1: $size_a = \mathsf{E}(b) - \mathsf{S}(b) = 4$

First, we consider the intervals of the size $\mathsf{E}(b) - \mathsf{S}(b) = 4$. In this case, the Boolean expression for the interval comparison is computed as:

$$\begin{aligned}\mathsf{Compare}_q(a, 10) = (&a_{11} \vee a_{10} \vee a_9 \vee a_8 \vee a_7 \vee a_6 \vee a_5 \vee a_4 \vee a_3) \oplus \\ &(\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8 \vee a_7 \vee a_6 \vee a_5 \vee a_4 \vee a_3) \wedge (a_2 \wedge (a_1 \oplus (\neg a_1 \wedge a_0)))).\end{aligned}$$

Correctness. The equation above is derived as follows. We compare if a Boolean-shared masked coefficient a is greater than the interval size $\mathsf{E}(b) - \mathsf{S}(b) = 4$ starting from the most-significant bit (MSB). The relation $a > 4$ holds, when:

1. If at least one of the bits $a_{11}, ..., a_3$ is set, the smallest possible value of $a = 2^3 = 8$ is greater than 4. Otherwise the lower bits should be taken into consideration: $\mathsf{Compare}_q(a, 4) = (a_{11} \vee a_{10} \vee a_9 \vee a_8 \vee a_7 \vee a_6 \vee a_5 \vee a_4 \vee a_3) \oplus (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8 \vee a_7 \vee a_6 \vee a_5 \vee a_4 \vee a_3) \wedge (...))$.
2. If the bits a_2, a_1 are set, the value of $a = 2^2 + 2^1 = 6$ is greater than 4. Otherwise the least significant bits should be considered: $\mathsf{Compare}_q(a, 4) = (a_{11} \vee a_{10} \vee a_9 \vee a_8 \vee a_7 \vee a_6 \vee a_5 \vee a_4 \vee a_3) \oplus (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8 \vee a_7 \vee a_6 \vee a_5 \vee a_4 \vee a_3) \wedge (a_2 \wedge (a_1 \oplus (\neg a_1 \wedge (...)))))$.
3. If the bits a_2, a_0 are set, then the value of $a = 2^2 + 2^0 = 5$ is greater than 4, therefore: $\mathsf{Compare}_q(a, 10) = (a_{11} \vee a_{10} \vee a_9 \vee a_8 \vee a_7 \vee a_6 \vee a_5 \vee a_4 \vee a_3) \oplus (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8 \vee a_7 \vee a_6 \vee a_5 \vee a_4 \vee a_3) \wedge (a_2 \wedge (a_1 \oplus (\neg a_1 \wedge a_0))))$.
4. All the remaining bit configurations are smaller or equal to 4.

Case 2: $size_a = \mathsf{E}(b) - \mathsf{S}(b) = 3$

Next, let us consider the remaining case of the intervals of the size $\mathsf{E}(b) - \mathsf{S}(b) = 3$. We compute the Boolean expression for the interval comparison $a > 3$ as

$$\mathsf{Compare}_q(a, 10) = (a_{11} \vee a_{10} \vee a_9 \vee a_8 \vee a_7 \vee a_6 \vee a_5 \vee a_4 \vee a_3 \vee a_2).$$

Correctness. If at least one of the bits $a_{11}, ..., a_2$ is set, the smallest possible value of $a = 2^2 = 4$ is greater than 3, therefore: $\mathsf{Compare}_q(a, 10) = (a_{11} \vee a_{10} \vee a_9 \vee a_8 \vee a_7 \vee a_6 \vee a_5 \vee a_4 \vee a_3 \vee a_2)$. All the remaining bit configurations are smaller or equal to 3.

4.2.2 Combining Boolean Expressions for 10-bit Comparison

For efficiency, we combine the Boolean equations for two interval comparisons together. First, the Boolean shares of a are transformed into a bitsliced representation, such that the comparison operation can be computed for all coefficients in parallel. To distinguish between two possible interval sizes, we define an n-bit integer

$mask$, such that for $0 \leq i < n$, the i-th bit of $mask$ is set to 1, if the correct interval size for the i-th coefficient is equal to 4; otherwise the i-th bit of $mask$ is set to 0. In other words,

$$mask_i = \begin{cases} 1 & \text{if } size_{a_i} = 4, \\ 0 & \text{otherwise.} \end{cases}$$

The complete algorithm for $\mathsf{Compare}_q(a, mask, 10)$ is given in Algorithm 33. The operations & and | indicate bitwise logical AND and OR operations, respectively, applied to each share individually. In a masked implementation, the logical operations $\wedge$, $\vee$, $\oplus$ and $\neg$ should be replaced with their secure counterparts SecAND, SecOR, SecXOR and SecNOT, where the SecOR operation is computed as

$$\mathsf{SecOR}\left(a^{(\cdot)_B}, b^{(\cdot)_B}\right) = \mathsf{SecNOT}\left(\mathsf{SecAND}\left(\mathsf{SecNOT}\left(a^{(\cdot)_B}\right), \mathsf{SecNOT}\left(b^{(\cdot)_B}\right)\right)\right).$$

To combine the Boolean expressions for two interval comparisons into a single expression, the common parts of both equations are computed for all coefficients together; the parts of the expressions which are different, are computed separately and combined into a single output, dependent on the value of $mask$. For instance, two intermediate results x, y can be combined into a single output $z = x \mathbin{\&} mask \mid y \mathbin{\&} \neg mask$ using bitwise logical operations, where the values of x and y correspond to the cases $size = 4$ and $size = 3$, respectively.

Algorithm 33 Masked 10-bit comparison, $\mathsf{Compare}_q(a, mask, 10)$

Input: 1. A Boolean sharing $a^{(\cdot)_B} = (a^{(0)_B}, a^{(1)_B})$ of n bitsliced ciphertext coefficients,
2. An n-bit integer $mask$ indicating the size of the interval
containing possible values of i-th ciphertext coefficient.
For $0 \leq i < n$, $mask_i = 1$ if $\mathsf{E}(b) - \mathsf{S}(b) = 4$; $mask_i = 0$ if $\mathsf{E}(b) - \mathsf{S}(b) = 3$,
3. A bit width d of a compressed ciphertext coefficient.

Output: A Boolean sharing $b^{(\cdot)_B} = (b^{(0)_B}, b^{(1)_B})$ of b, where $b = 1$
if $a > \mathsf{E}(b) - \mathsf{S}(b)$, otherwise $b = 0$.

```
   /* For readability, here the secure operations SecAND, SecOR, SecXOR and SecNOT
   are denoted by ∧, ∨, ⊕ and ¬, respectively. */
1: $v^{(\cdot)_B} = (a_2^{(\cdot)_B} \wedge (a_1^{(\cdot)_B} \oplus (\neg a_1^{(\cdot)_B} \wedge a_0^{(\cdot)_B})))$
2: $w^{(\cdot)_B} = (a_{11}^{(\cdot)_B} \vee a_{10}^{(\cdot)_B} \vee a_9^{(\cdot)_B} \vee a_8^{(\cdot)_B} \vee a_7^{(\cdot)_B} \vee a_6^{(\cdot)_B} \vee a_5^{(\cdot)_B} \vee a_4^{(\cdot)_B} \vee a_3^{(\cdot)_B})$
3: $x^{(\cdot)_B} = \neg w^{(\cdot)_B} \wedge v^{(\cdot)_B}$
4: $y^{(\cdot)_B} = w^{(\cdot)_B} \vee a_2^{(\cdot)_B}$
5: $z^{(\cdot)_B} = w^{(\cdot)_B} \oplus x^{(\cdot)_B}$
6: $b^{(\cdot)_B} = (y^{(\cdot)_B} \mathbin{\&} \neg mask) \mid (z^{(\cdot)_B} \mathbin{\&} mask)$
7: return $b^{(\cdot)_B}$
```

4.3 4-bit comparison

The second important building block is the comparison of 4-bit values. Similarly to the 10-bit case, the 4-bit comparison verifies whether the Boolean-masked shifted coefficient $a = a' - \mathsf{S}(b)$ lies within the interval

$[0,\ \mathsf{E}(b) - \mathsf{S}(b) - 1]$. Dependent on the value of the public coefficient b, this interval can take two possible sizes:

$$size_a = \mathsf{E}(b) - \mathsf{S}(b) = \begin{cases} 209 = \left\lceil \frac{q}{2^4} \right\rceil & \text{if } 3225 \leq \mathsf{Decompress}_q(b, 4) < 105, \\ 208 = \left\lfloor \frac{q}{2^4} \right\rfloor & \text{otherwise,} \end{cases}$$

The interval sizes for 4-bit comparison are illustrated in Figure 6. In the following, we derive the algorithm, which securely computes the 4-bit comparison $a' - \mathsf{S}(b) \geq \mathsf{E}(b) - \mathsf{S}(b)$.

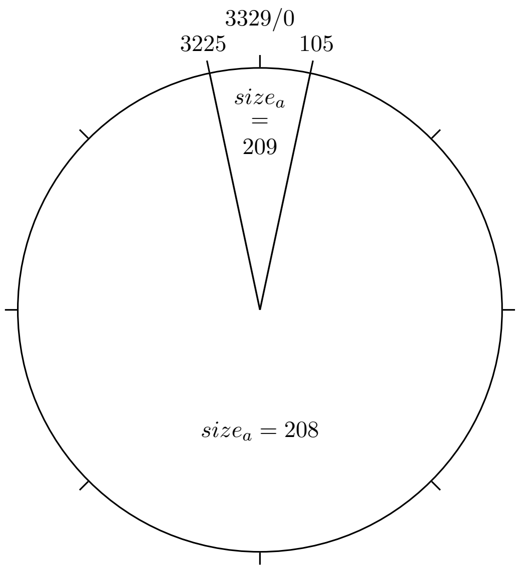

Figure 6: Interval sizes for 4-bit comparison

4.3.1 Boolean Expressions for 4-bit Comparison

Given a Boolean-masked shifted coefficient $a = a' - \mathsf{S}(b)$, the comparison operation is defined as

$$\mathsf{Compare}_q(a, 4) = \begin{cases} 0 & \text{if } a \leq size_a, \\ 1 & \text{otherwise.} \end{cases}$$

As in the 10-bit case, we perform a masked binary search on the bits of a. We begin by deriving the Boolean equations for computing two comparisons, $a > size_a = 209$ and $a > size_a = 208$ and then merge both equations together into a single comparison.

Case 1: $size_a = \mathsf{E}(b) - \mathsf{S}(b) = 209$

We start off with analyzing the intervals of the size $\mathsf{E}(b) - \mathsf{S}(b) = 209$. In this case, the comparison operation, $\mathsf{Compare}_q(a, 4)$, is computed as

$$\begin{aligned} \mathsf{Compare}_q(a, 4) = {} & (a_{11} \vee a_{10} \vee a_9 \vee a_8) \oplus \\ & (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8) \wedge a_7 \wedge a_6 \wedge (a_5 \oplus (\neg a_5 \wedge a_4 \wedge (a_3 \vee a_2 \vee a_1)))) \end{aligned}$$

Correctness. The equation derived above computes the comparison $a > 209$ correctly, since:

1. In case one of the bits $a_{11}, \ldots, a_8$ is set, the smallest possible value of $a = 2^8 = 256$ is greater than 209. Otherwise the lower bits should be taken into consideration: $\mathsf{Compare}_q(a, 4) = (a_{11} \vee a_{10} \vee a_9 \vee a_8) \oplus (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8) \wedge (\ldots))$.
2. If the bits a_7, a_6, a_5 are set, then the value of $a = 2^7 + 2^6 + 2^5 = 224$ is greater than 209. Otherwise the lower bits should be considered: $\mathsf{Compare}_q(a, 4) = (a_{11} \vee a_{10} \vee a_9 \vee a_8) \oplus (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8) \wedge a_7 \wedge a_6 \wedge (a_5 \oplus (\neg a_5 \wedge (\ldots))))$.
3. If the bits a_7, a_6, a_4 are set, as well as one of the bits a_3, a_2, a_1, then the smallest possible value of $a = 2^7 + 2^6 + 2^4 + 2^1 = 210$ is greater than 209, therefore: $\mathsf{Compare}_q(a, 4) = (a_{11} \vee a_{10} \vee a_9 \vee a_8) \oplus (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8) \wedge a_7 \wedge a_6 \wedge (a_5 \oplus (\neg a_5 \wedge a_4 \wedge (a_3 \vee a_2 \vee a_1))))$
4. All the remaining bit combinations are smaller or equal to 209.

Case 2: $\mathsf{E}(b) - \mathsf{S}(b) = 208$

Next, we consider the second case of the intervals of the size $\mathsf{E}(b) - \mathsf{S}(b) = 208$. The Boolean expression for the interval comparison $a > 208$ is given by

$$\begin{aligned}\mathsf{Compare}_q(a, 4) = (a_{11} \vee a_{10} \vee a_9 \vee a_8) \oplus \\ (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8) \wedge a_7 \wedge a_6 \wedge (a_5 \oplus (\neg a_5 \wedge a_4 \wedge (a_3 \vee a_2 \vee a_1 \vee a_0))))\end{aligned}$$

Correctness. The comparison operation defined above computes the correct comparison $a > 208$ since:

1. In case one of the bits $a_{11}, \ldots, a_8$ is set, the smallest possible value of $a = 2^8 = 256$ is greater than 208. Otherwise the lower bits should be taken into consideration: $\mathsf{Compare}_q(a, 4) = (a_{11} \vee a_{10} \vee a_9 \vee a_8) \oplus (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8) \wedge (\ldots))$.
2. If the bits a_7, a_6, a_5 are set, then the value of $a = 2^7 + 2^6 + 2^5 = 224$ is greater than 208. Otherwise the lower bits should be considered: $\mathsf{Compare}_q(a, 4) = (a_{11} \vee a_{10} \vee a_9 \vee a_8) \oplus (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8) \wedge a_7 \wedge a_6 \wedge (a_5 \oplus (\neg a_5 \wedge (\ldots))))$.
3. If the bits a_7, a_6, a_4 are set, as well as one of the bits a_3, a_2, a_1, a_0, then the smallest possible value of $a = 2^7 + 2^6 + 2^4 + 2^0 = 209$ is greater than 208, therefore: $\mathsf{Compare}_q(a, 4) = (a_{11} \vee a_{10} \vee a_9 \vee a_8) \oplus (\neg(a_{11} \vee a_{10} \vee a_9 \vee a_8) \wedge a_7 \wedge a_6 \wedge (a_5 \oplus (\neg a_5 \wedge a_4 \wedge (a_3 \vee a_2 \vee a_1 \vee a_0))))$
4. All the remaining bit combinations are smaller or equal to 208.

4.3.2 Combining Boolean Expressions for 4-bit Comparison

To improve performance, we combine two interval comparisons into a single comparison - the masked algorithm for $\mathsf{Compare}_q(a, mask, 4)$ is given in Algorithm 34. As previously, the Boolean-masked coefficient a is converted into a bitsliced representation; the & and | operations denote bitwise logical AND and OR, respectively; the logical operations $\wedge$, $\vee$, $\oplus$ and $\neg$ should be replaced with their secure versions SecAND, SecOR, SecXOR and SecNOT. For $0 \le i < n$, we define the variable $mask$ as

$$mask_i = \begin{cases} 1 & \text{if } size_{a_i} = 208, \\ 0 & \text{otherwise.} \end{cases}$$

When comparing the Boolean equations for two interval comparisons, one can notice that all but one terms are identical. The only difference is the term a_1 vs. $a_1 \vee a_0$ for the expressions evaluating $a > 209$ and $a > 208$, respectively. These terms can be combined in a single term as $x = (a_1 \mathbin{\&} \neg mask) \mid ((a_1 \vee a_0) \mathbin{\&} mask)$. As before, the variable $mask$ is needed to distinguish between two possible interval sizes. The remaining computation is identical for all ciphertext coefficients.

Algorithm 34 Masked 4-bit comparison, $\mathsf{Compare}_q(a, mask, 4)$

```
Input:  1. A Boolean sharing $a^{(\cdot)_B} = (a^{(0)_B}, a^{(1)_B})$ of $n$ bitsliced ciphertext coefficients,
        2. An $n$-bit integer $mask$ indicating the size of the interval
        containing possible values of $i$-th ciphertext coefficient.
        For $0 \leq i < n$, $mask_i = 1$ if $\mathsf{E}(b) - \mathsf{S}(b) = 208$; $mask_i = 0$ if $\mathsf{E}(b) - \mathsf{S}(b) = 209$,
        3. A bit width $d$ of a compressed ciphertext coefficient.
Output: A Boolean sharing $b^{(\cdot)_B} = (b^{(0)_B}, b^{(1)_B})$ of $b$, where $b = 1$
        if $a > \mathsf{E}(b) - \mathsf{S}(b)$, otherwise $b = 0$.
   /* For readability, here the secure operations SecAND, SecOR, SecXOR and SecNOT
   are denoted by $\wedge$, $\vee$, $\oplus$ and $\neg$, respectively. */
1: $x^{(\cdot)_B} = (a_1^{(\cdot)_B} \mathbin{\&} \neg mask) \mid ((a_1^{(\cdot)_B} \vee a_0^{(\cdot)_B}) \mathbin{\&} mask)$
2: $y^{(\cdot)_B} = a_7^{(\cdot)_B} \wedge a_6^{(\cdot)_B} \wedge (a_5^{(\cdot)_B} \oplus (\neg a_5^{(\cdot)_B} \wedge a_4^{(\cdot)_B} \wedge (a_3^{(\cdot)_B} \vee a_2^{(\cdot)_B} \vee x^{(\cdot)_B})))$
3: $z^{(\cdot)_B} = (a_{11}^{(\cdot)_B} \vee a_{10}^{(\cdot)_B} \vee a_9^{(\cdot)_B} \vee a_8^{(\cdot)_B})$
4: $b^{(\cdot)_B} = z^{(\cdot)_B} \oplus (\neg z^{(\cdot)_B} \wedge y^{(\cdot)_B})$
5: return $b^{(\cdot)_B}$
```

4.4 Combining Everything Together

In this section we describe our new algorithm for ciphertext comparison, NewDecompressedComparison, given in Algorithm 35. It builds on the previously introduced $\mathsf{Compare}_q(a, 10)$ and $\mathsf{Compare}_q(a, 4)$ algorithms and securely computes

$$(\boldsymbol{u'}, v') \stackrel{?}{=} \mathsf{Decompress}_q(c, d).$$

As a first step, the coefficients of the public ciphertext c are decompressed with the function $\mathsf{Decompress}_q(c, d)$, where $d = d_u = 10$ for the first part of the ciphertext, and $d = d_v = 4$ for the second part, according to Kyber768 parameter set. These parameters are defined in the Kyber512 parameter set as well; however in case of Kyber1024, which uses different values of $d = (d_u, d_v) = (11, 5)$, our algorithm should be adjusted accordingly. Both public and private ciphertexts are then compared coefficient-wise for each pair of ciphertext coefficients. Let us consider one pair of public and private coefficients (w_i'', w_i') with $0 \leq i < n$ as an example. The function PolyCompare starts with computing the bounds $\mathsf{S}(w_i'')$ and $\mathsf{E}(w_i'')$ of the interval containing possible values of c_i after decompression. Next, the size of this interval, $size_{w''}$, is computed and the i-th bit of $mask$ is set to 1 if the interval size is equal to 4 and 208 for $d = 10$ and $d = 4$, respectively; otherwise the i-th bit of $mask$ is set to 0. The arithmetically masked private coefficients w_i' are then shifted by $size_{w''}$ and converted into a Boolean-masked bitsliced representation by applying the $\mathsf{A2B}_q$ conversion and a subsequent

Bitslice operation. By doing so, the interval comparison can be computed for all coefficients in parallel (limited by the register size of the target platform) using $\mathsf{Compare}_q(a,d)$ operation. After the comparison operation, bitsliced outputs $t_{\boldsymbol{u}}^{(\cdot)_B}$, $t_{\boldsymbol{v}}^{(\cdot)_B}$ corresponding to two parts of the ciphertext, are obtained. These results are negated and combined into a single bitsliced output $b^{(\cdot)_B}$ with SecAND. The individual bits of $b^{(\cdot)_B}$ are then combined into a final result using subsequent calls to SecAND.

Algorithm 35 Masked NewDecompressedComparison

Input: 1. An arithmetic sharing $\boldsymbol{u'}^{(\cdot)_A} = (u^{(0)_A}, u^{(1)_A})$ of a vector of polynomials $\boldsymbol{u'} \in \mathbb{Z}_q[X]^k$,
2. An arithmetic sharing $v'^{(\cdot)_A} = (v^{(0)_A}, v^{(1)_A})$ of a polynomial $v' \in \mathbb{Z}_q[X]$,
3. A ciphertext c,
4. Two public functions S and E defined by Kyber, which specify the start- and end-points of the intervals in compression.

Output: A Boolean sharing $b^{(\cdot)_B} = (b^{(0)_B}, b^{(1)_B})$ of b, where $b = 1$ if $(\mathsf{Compress}_q(\boldsymbol{u'}, d_u), \mathsf{Compress}_q(v', d_v)) = c$, otherwise $b = 0$.

```
1:  function NewDecompressedComparison
2:      $(\boldsymbol{u''}, v'') = \mathsf{Decompress}_q(c)$
3:      $t_{\boldsymbol{u}}^{(\cdot)_B} = \mathsf{PolyCompare}(\boldsymbol{u'}^{(\cdot)_A}, \boldsymbol{u''}, d_u)$
4:      $t_v^{(\cdot)_B} = \mathsf{PolyCompare}(v'^{(\cdot)_A}, v'', d_v)$
5:      $b^{(\cdot)_B} = \mathsf{SecAND}(\neg t_{\boldsymbol{u}}^{(\cdot)_B}, \neg t_v^{(\cdot)_B})$
6:      for $i = \log_2(n-1)$ to $0$ do
7:          $t_b^{(\cdot)_B} = b^{(\cdot)_B} \gg 2^i$
8:          $b^{(\cdot)_B} = b^{(\cdot)_B} \bmod (2^{2^i} - 1)$
9:          $b^{(\cdot)_B} = \mathsf{SecAND}(b^{(\cdot)_B}, t_b^{(\cdot)_B})$
10:     return $b^{(\cdot)_B}$
11:
12: function PolyCompare($w'^{(\cdot)_A}, w'', d_w$)
13:     $mask = \{0\}^n$
14:     for $i = 0$ to $n-1$ do
15:         $s_{w''} = \mathsf{S}(w''_i)$
16:         $e_{w''} = \mathsf{E}(w''_i)$
17:         $size_{w''} = e_{w''} - s_{w''}$
18:         $mask = mask \mid ((size_{w''} \overset{?}{=} SIZE) \ll i)$      ▷ $SIZE = 4$ if $d_w = 10$; $SIZE = 208$ if $d_w = 4$
19:         $w'^{(0)_A}_i = (w'^{(0)_A}_i - s_{w''}) \bmod q$
20:         $w'^{(\cdot)_B}_i = \mathsf{A2B_q}(w'^{(\cdot)_A}_i)$
21:         $x^{(\cdot)_B} = \mathsf{Bitslice}(w'^{(\cdot)_B})$
22:     $r^{(\cdot)_B} = \mathsf{Compare}_q(x^{(\cdot)_B}, mask, d_w)$
23:     return $r^{(\cdot)_B}$
```

5 Implementation Details and Performance Evaluation

This chapter evaluates the impact of masking on Kyber and Saber. After briefly describing the testing environment, we provide an overview of the first-order masked implementations of Kyber and Saber using the masking techniques from the previous chapters. In case of Kyber, we implement several algorithms for polynomial compression and comparison and estimate their efficiency in the higher-order settings. Then, we evaluate both schemes in terms of performance and code size and discuss the results.

5.1 Testing Environment

We integrated our code into the *pqm4* framework, a popular crypto library for testing and benchmarking of post-quantum cryptographic schemes for the ARM Cortex-M4 architecture [49]. All benchmarks were run on the STM32F407-DISCOVERY evaluation board, manufactured by ST Microelectronics. This embedded platform is supported by the *pqm4* library and features a 32-bit ARM Cortex-M4 CPU, 192 KB of SRAM, 1 MB of on-chip flash memory and an internal TRNG.

The software was compiled using the standard settings of *pqm4*, with minor modifications to measure the run time of the subroutines. For instance, the performance was evaluated at the operating frequency of 24 MHz for a core system clock; the randomness required for masking was sampled from the internal TRNG clocked at 48 MHz. Cycle counts were measured using the built-in 24-bit SysTick timer on the target board.

5.2 Masked Implementation of Kyber

In this section we summarize our first-order masked implementation of Kyber, optimized for the ARM Cortex-M4 architecture. We provide an overview of the masking techniques for protecting the core components of Kyber, describe the performance optimizations we made and present the results.

5.2.1 Compression

In the following, we focus on the polynomial compression of Kyber. We implemented the masked version of the $\mathsf{Compress}_q(x, 1)$ component in two different configurations:

1. A masked 1-bit compression from [24], based on a bitsliced binary search approach. The high-level description of this method is given in Section 3.7.1 and in Algorithm 30. We denote it as *bitsliced compression*.

2. A masked compression from [37], based on integer division. This method is described in Section 3.7.2 and in Algorithm 31. We denote it as *division-based compression*.

Bitsliced Compression

Let us recall the bitsliced polynomial compression, given in Algorithm 30. First, the arithmetically masked coefficients are shifted by $\frac{q}{4}$. The result is then converted into Boolean shares with the $\mathsf{A2B}_q$ conversion. For the first-order masking, this conversion can be efficiently implemented using the pre-computed look-up tables. In our implementation, we use a variant of the table-based A2B conversion from [35, 31], described in Section 3.4.2 and in Algorithms 18 and 19. As this approach was designed for the power-of-two moduli, it cannot be used directly and should be adapted for prime moduli q. To overcome this, similar to [24], we remove splitting sensitive values into chunks and use larger tables that operate on full-sized inputs to avoid carry propagation.

Listing 1 shows our implementation of the table-based $\mathsf{A2B}_q$ conversion. The function `Table_Gen()` initializes the entries of the look-up table T with $\mathsf{T}[i] = ((i + r_1) \bmod q) \oplus r_2$ for $i \in [0, q-1]$. The value i is arithmetically masked with a random value $r_1 \in [0, q-1]$ on the input side; on the output side, the Boolean mask with the randomness $r_2 \in [0, q-1]$ is applied. The actual conversion is performed in the function `A2B()`. Here, the function `rng()` samples the $\lceil \log_2(q) \rceil$ random bits; `csubq()` performs a constant-time conditional subtraction of a variable a by modulus q, i.e., if the value of a is less than q, it is not modified; otherwise a is set to $a - q$.

After the shifted coefficients are converted into Boolean shares, they are transformed into a bit-sliced representation, such that the compression operation can be computed for all the coefficients in parallel. The remaining Boolean expressions are computed using the secure Boolean operations SecAND, SecXOR, SecREF.

```c
uint16_t r1;
uint16_t r2;
uint16_t T[KYBER_Q];

void Table_Gen(void)
{
    r1 = sample_modq();
    r2 = rng(KYBER_Q_BITSIZE);        // BITS = bit size

    for(int i=0; i < KYBER_Q; i++) {
        T[i] = ((i + r1) % KYBER_Q) ^ r2;
    }
}

void A2B(boolean_shares x, arith_shares a)
{
    uint16_t R, a0;
    R = rng(KYBER_Q_BITSIZE);
    a0 = csubq(a[0] + KYBER_Q - r1);
    a0 = csubq(a0 + a[1]);
    x[0] = T(a0) ^ R;
    X[1] = r2 ^ R;
}
```

Listing 1: Table-based A2B conversion based on [24, 35].

Division-based Compression

We now return to the division-based polynomial compression, given in Algorithm 31. This operation is performed by dividing the shifted polynomial coefficients by q, rounding the result and reducing modulo 2^{d+f}, where $d = 1$ is the compression parameter and $f = 13$ is the bit width of the fractional part [37]. The output is then converted into Boolean shares and reduced modulo q by performing the subsequent share-wise logical shift.

Constant-time division. The integer division requires special treatment, as dependent on the value of inputs, this operation takes variable time on the ARM Cortex-M processors. To achieve constant-time behavior, we exploit the fact, that division by a constant q is equivalent to multiplication by the reciprocal of that constant $q^{-1} = \frac{1}{q}$. To replace the division by a sequence of multiplications, additions and shifts, we adapt the method of Jones [48], that works as follows. First, the reciprocal of q is converted into a binary representation:

$$\frac{1}{q} = \frac{1}{3329} = (0.0000000000010011101011111011011101101000000010111011000001010101)_2.$$

We now consider all the bits to the right of the binary point and shift them to the left until the bit on the right side of the binary point is 1. In our case, the required number of shifts is 11. For the first-order secure 1-bit compression, the size of the dividend $2^{d+f} \cdot x$ does not exceed 32 bits, therefore we take 32 most significant bits and two additional bits to ensure the correct rounding:

$$(1001\ 1101\ 0111\ 1101\ 1011\ 1011\ 0100\ 0000\ 01)_2.$$

This value is then incremented by 1, shifted by 1 position to the left and truncated to the required 32 bits:

$$(0011\ 1010\ 1111\ 1011\ 0111\ 0110\ 1000\ 0001)_2 = (\mathsf{3AFB\ 7681})_{16},$$

which corresponds to fractional inverse of q shifted to the left (or multiplied with a power of two). To obtain the correct result, we need to perform a reverse operation, i.e. divide by a power of two by performing right logical shifts. The division of an integer a by q can then be expressed as

$$\frac{a}{q} = ((((a \cdot (\mathsf{3AFB7681})_{16}) \gg 32) + a) \gg 11) \gg 1).$$

In detail, our implementation is given in Listing 2.

```
uint32_t cdivq(uint32_t a)
{
  uint64_t result;
  result = ((((uint64_t)a * 0x3AFB7681) >> 32) + a) >> 12);
  return result;
}
```

Listing 2: Constant-time division of 32-bit integer a by q based on [48, 47].

A2B conversion. After the actual compression step, the outputs modulo 2^{d+f} are converted into Boolean shares. For the power-of-two moduli, the A2B conversion can be efficiently computed using either pre-computed look-up tables or a masked addition over Boolean shares. In contrast to the case of prime moduli, the precomputed tables for power-of-two moduli cannot be reused for other building blocks, such as the polynomial comparison, since the compression parameters are different (in Kyber768, 1-, 4- and 10-bit compression is used, which corresponds to $d + f = 14$, $d + f = 17$ and $d + f = 23$-bit A2B conversions, respectively). This would require generation of different look-up tables, which would severely impact the stack utilization. Therefore, for power-of-two moduli, we implement an addition-based method [30], described in Algorithm 20. This approach can be computed efficiently when instantiated with a fast first-order secure addition from [50], given in Algorithm 16.

Performance Evaluation

To compare the running time of both algorithms for polynomial compression, we have performed an optimized first-order masked implementation in ARM Cortex-M4 assembly and C, and a generic higher-order masked implementation in C. In case of bitsliced compression, we work with modulo $q = 3329$. As described before, the table-based $\mathsf{A2B}_q$ conversion is used for the first-order masking; for the higher-order masked compression we use the addition-based $\mathsf{A2B}_q$ conversion (Algorithm 3 from [71]) instantiated with the masked addition, SecADD_q (Algorithm 10 from [9]). For division-based compression we work with modulo $2^{d+f} = 2^{14}$. Here, the A2B conversion based on the masked secure addition is used for all security orders. The first-order masking approach is described above; the higher-order masked compression is instantiated with the A2B conversion (Algorithm 3 from [71]) together with the masked addition, SecADD (Algorithm 9 from [9]). As can be seen in Table 3 below, the bitsliced compression method is more efficient in the first-order masking scenario. However, starting from the second order, the division-based compression performs best.

Masked Compression	**Order**					
	1	2	3	4	5	6
Bitsliced compression	45	3 359	10 005	44 822	82 896	137 427
Division-based compression	94	2 632	9 914	20 793	38 347	63 624

Table 3: Practical performance results for polynomial compression in thousands of clock cycles on ARM Cortex-M4. The first-order implementation is optimized in assembly; the generic higher-order masking is implemented in C.

5.2.2 Comparison

In this section we consider the polynomial comparison of Kyber. To protect this building block against side-channel analysis, we implemented and compared three different techniques:

1. Our new masked comparison is based on a bitsliced binary search method and does not involve polynomial compression. The detailed description of the new technique is given in Chapter 4. We denote it as *new bitsliced comparison*.

2. A masked comparison from [24] performs the ciphertext comparison by decompressing the public ciphertext coefficients and performing two interval checks. The description of this method is given in Section 3.8.2 and in Algorithm 32. We denote it as *decompressed comparison*.

3. A hash-based comparison is based on polynomial compression. The high-level view of this approach is given in Section 3.8.1; the description of masked 10- and 4-bit compression is given in Algorithm 31. We denote it as *compressed comparison*.

New Bitsliced Comparison

We recall our new approach for bitsliced polynomial comparison, summarized in Algorithm 35. After the public compressed ciphertext coefficients are decoded from a byte string into a polynomial respresentation, the next step is to determine the interval of the possible values of the polynomial coefficients after decompression. The bounds of this interval are computed by a public functions $\mathsf{E}()$, $\mathsf{S}()$, using the pre-computed look-up tables. These functions are predefined by Kyber, hence we omit their description here. The arithmetically masked uncompressed coefficients are then shifted by the size of this interval and converted into a Boolean shares. Throughout the algorithm, we work with a prime modulo $q = 3329$ and use the same table-based $\mathsf{A2B}_q$ conversion, as in the case of the bitsliced compression. Our C implementation of the $\mathsf{A2B}_q$ conversion is shown in Listing 1. The precomputed look-up tables are generated fresh for every Kyber invocation and can be reused in the first-order masked implementation for both our new bitsliced comparison and bitsliced compression. Once the shifted coefficients are converted into Boolean shares, they are transformed into a bitsliced representation, such that the remaining computation can be performed for all the coefficients in parallel. In our case, we process 32 coefficients simultaneously, as the register and the word size of our target platform, ARM Cortex-M4, is 32 bits. We replace the Boolean operations with $\wedge$, $\vee$, $\oplus$, $\neg$ with the secure functions SecAND, SecOR, SecXOR, SecNOT, respectively, where the logical OR operation is given by

$$a \vee b = \neg(\neg a \wedge \neg b).$$

Decompressed Comparison

The decompressed comparison method, given in Algorithm 32, also involves arithmetic modulo $q = 3329$. Its core building blocks are similar to those of the new bitsliced comparison. Both $\mathsf{A2B}_q$ conversions are based on the same table-based approach, described in Listing 1. The look-up tables are precomputed for every Kyber invocation and can be reused for both decompressed comparison and bitsliced compression, since the both components rely on arithmetic modulo q; the logical AND operations are computed securely using SecAND.

Compressed Comparison

We now return to the compressed comparison, described in Section 3.8.1. First, the masked ciphertext is compressed coefficient by coefficient. To do so, we use a division-based compression method, given in Algorithm 31 and in Section 5.2.1, with the size of the fractional part $f = 13$ [37] and the compression parameter set to $d = 10$ and $d = 4$ for the first and second parts of the ciphertext, respectively. The next step is then to check, whether

$$\mathcal{H}(c'^{(0)_B}) \stackrel{?}{=} \mathcal{H}(c \oplus c'^{(1)_B}).$$

Here, the first share of the masked ciphertext $\boldsymbol{c'}$ is combined with the public ciphertext c using SecXOR and fed into the hash function $\mathcal{H}$, while the second share of the masked ciphertext is hashed separately. Both outputs are then compared bit-by-bit using exclusive OR.

Constant-time division. The core building block of the division-based compression is a division of integers by a constant q. For the first-order secure 4-bit compression, the size of the divident $2^{d+f} \cdot x$ does not exceed 32 bits, hence we reuse our implementation, given in Listing 2. However, the 10-bit compression involves dividends with the sizes greater than 32 bits and requires special treatment. In this case, we use a larger division, that is constructed in a similar way except that we take 64 + 2 most significant bits into consideration. Our implementation is illustrated in Listing 3 below.

```
uint64_t cdivq(uint64_t a)
{
  uint64_t result;
  result = (((((uint128_t)a * 0x3AFB7680BB054E5D) >> 64) + a) >> 12);
  return result;
}
```

Listing 3: Constant-time division of 64-bit integer a by q based on [48, 47].

Performance Evaluation

We compare the running time of the algorithms for ciphertext comparison, first using our *new bitsliced comparison* method from the Chapter 4 and the *decompressed comparison* from [24], which work with a prime modulo $q = 3329$ and do not include the ciphertext compression. The third approach, the *compressed comparison*, works with modulo $2^{d+f} = 2^{14}$ and invokes an efficient masked compression from [37] as a subroutine.

For algorithms working with prime modulo q, we use the table-based A2B$_q$ conversion in a first-order masked implementation; for the higher-order masked implementation, the addition-based A2B$_q$ conversion (Algorithm 3 from [71]) instantiated with the masked addition, SecADD$_q$ (Algorithm 10 from [9]) is deployed. For the *compressed comparison* that works with power-of-two moduli, the A2B conversion based on the masked secure addition is applied for all security orders. The first-order masking approach is based on Algorithms 20 and 16; the higher-order masked compression is instantiated with the A2B conversion (Algorithm 3 from [71]) together with the masked addition, SecADD (Algorithm 9 from [9]). As the hash-based approach works only for the first-order masking, in the higher-order masked compressed comparison instead of using hash-functions, we apply SecAND operation to the values of $(c_1'||c_2')$, $(c_1 \oplus c_1'^0||c_2 \oplus c_2'^0)$ until a single masked output bit remains.

We have performed an optimized first-order masked implementation in ARM Cortex-M4 assembly and C, and a generic higher-order masked implementation in C. As can be seen in Table 4 below, the best approach for the first-order masking is the decompressed comparison. For the higher-order masking, the compressed comparison is significantly faster. We can see that starting from the second order, our new bitsliced comparison presents an improvement to the original method, the decompressed comparison, by roughly a factor of two. This is because our new method has just a single A2B$_q$ call, while the original approach includes two A2B$_q$ conversions.

Masked Comparison	**Order**					
	1	2	3	4	5	6
New bitsliced comparison	400	13 816	40 433	179 858	332 332	550 646
Decompressed comparison	349	26 746	79 869	358 289	662 738	1098 869
Compressed comparison	651	11 776	30 809	67 989	122 417	199 549

Table 4: Practical performance results for polynomial comparison in thousands of clock cycles on ARM Cortex-M4. The first-order implementation is optimized in assembly; the generic higher-order masking is implemented in C.

5.2.3 Decapsulation

This section provides an overview of the first-order masked implementation of Kyber using the masking techniques described in the previous chapters. Figure 7 illustrates the program flow of the masked decapsulation operation, Masked.Kyber.CCAKEM.Decaps, and serves as a visual representation of the information presented in this section. For simplicity, it omits the low-level details, such as encodings into and from the byte arrays, the generation of the public matrix $\boldsymbol{A}$ and transformations into and out of the NTT domain.

Our masked Kyber implementation builds on the unprotected reference implementation from the *pqm4* crypto library [49], optimized for speed. The linear arithmetic operations - polynomial multiplication, addition and subtraction, as well as NTT transformations, are duplicated on both shares, and can be implemented by reusing the respective functions from the reference implementation.

To mask the 1-bit compression operation, $\mathsf{Compress}_q(x, 1)$, of the Masked.Kyber.CPAPKE.Dec routine, we implement two recent proposals - the *bitsliced compression* [24] and the *division-based compression* [37] methods, the instantiation of which is described in detail in Section 5.2.1.

The masked decompression operation, $\mathsf{Decompress}_q$, of the Masked.Kyber.CPAPKE.Enc operation is implemented by converting the Boolean-masked output of the 1-bit compression into arithmetic shares with the $\mathsf{B2A}_q$ conversion [71], described in Algorithm 23. The result is then multiplied with a constant. We do not mask the decompression of the ciphertext in the Masked.Kyber.CPAPKE.Dec routine, as this is the operation on the public values known to the adversary, that does not depend on the long-term secret $\boldsymbol{s}$.

To protect symmetric components $\mathcal{G}$ and $\mathcal{H}$, based on the Keccak algorithm [63], we use the efficient first-order masked Keccak implementation from [15]. For binomial sampler, β_η, we adapt to the parameters of Kyber the efficient bitsliced sampler [71], described in Algorithm 27.

We implement three different techniques to compare re-encrypted masked ciphertext coefficients with public compressed ciphertext coefficients. Two of them - our *new bitsliced comparison* and the *decompressed comparison* [24] methods, do not apply ciphertext compression. Instead, the public ciphertext is decompressed. Furthermore, we implement the *compressed comparison* 3.8.1 method, together with the 10-bit and 4-bit ciphertext compression [37]. We describe these algorithms in detail in the Section 5.2.2.

While the ciphertext comparison is a sensitive operation, as it processes inputs derived from the long-term secret $\boldsymbol{s}$, the output of the comparison does not provide any valuable information to a side-channel attacker. As discussed in [12, 33], it is hard to construct a valid ciphertext that fails the comparison and it is not possible to generate an invalid ciphertext that verifies correctly up to a negligible probability. Even if the adversary obtains the return value, they does not learn any sensitive information about $\boldsymbol{s}$. Given that, there is no need to mask the Keccak-based component $\mathcal{H}'$, used for key derivation.

Assembly optimizations

The main goal of this work is to provide a fair comparison of our masked implementation of Kyber with the already existing masked implementation of Saber [12], described in the following section. To do so, we optimize roughly the same components in Kyber as were optimized in Saber using a modular approach. The low-level components, such as secure Boolean operations - SecAND, SecXOR, SecREF, SecNOT, as well as the arithmetic operations - SecADD, csubq, cdivq, are implemented as assembly macros. By using macros instead of functions, the overhead of the function calls can be eliminated at the cost of a small code size increase. The higher-level functions, such as A2B, A2B$_q$, B2A$_q$ conversions, binomial sampler and polynomial compression and comparison are implemented completely in assembly and include the low-level macros as subroutines. We also reuse efficient assembly implementations of the polynomial arithmetic operations - NTT, NTT^{-1} and pointwise multiplication from the unmasked *pqm4* reference implementation.

To further improve performance, several optimization techniques were applied. For instance, we fully unrolled the loop inside the SecADD routine. By doing so, the loop bookkeeping instructions, such as evaluating loop exit conditions, incrementing loop counters, performing pointer arithmetic or register moves can be removed and only the base instructions, such as memory accesses, Boolean and arithmetic operations have to be performed. Furthermore, we utilize the full register capacity whenever possible to avoid extra loads and stores and carefully schedule the assembly instructions to reduce data dependencies and to eliminate the pipeline stalls.

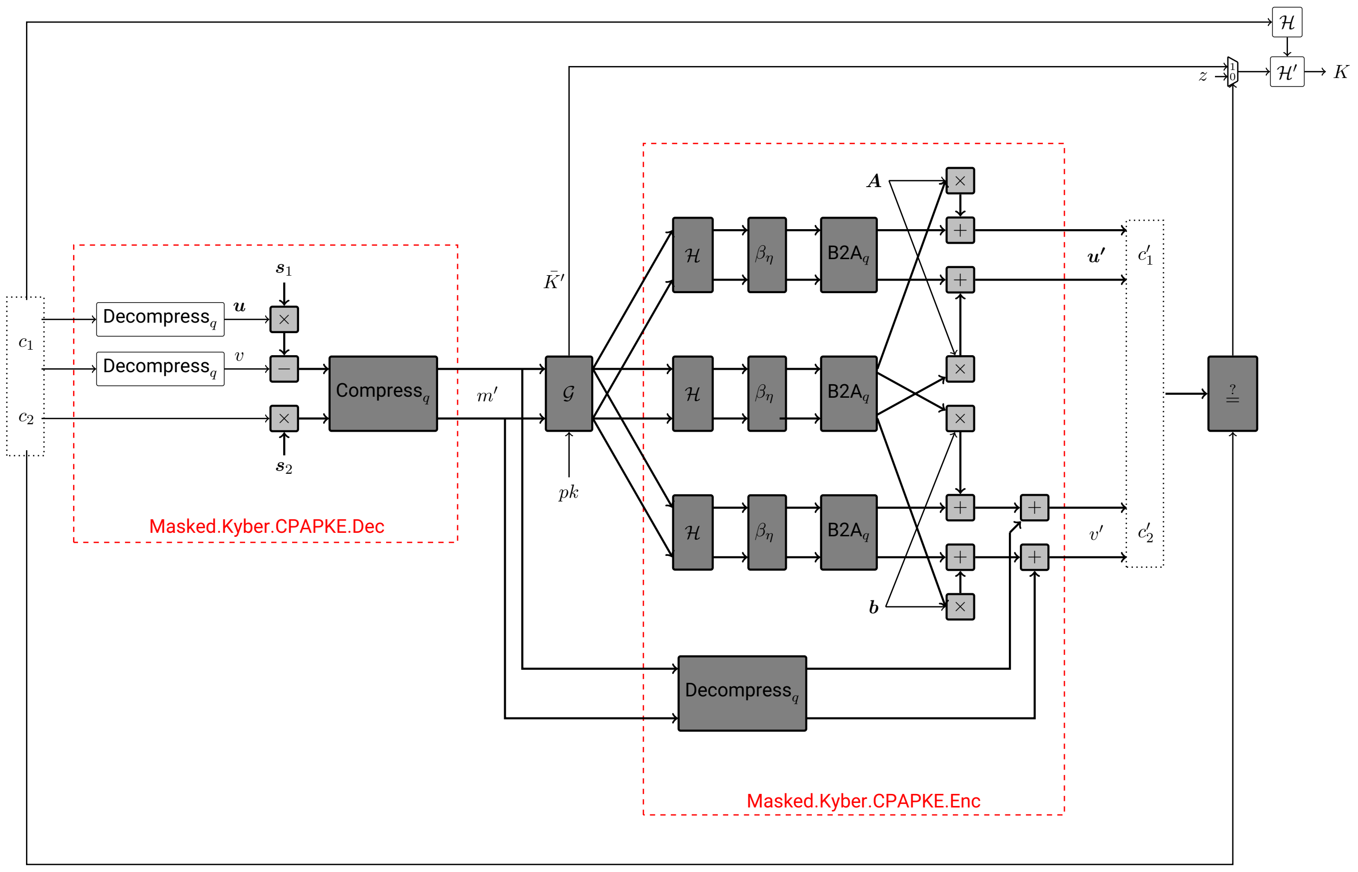

Figure 7: Masked IND-CCA-secure decapsulation of Kyber, Masked.Kyber.CCAKEM.Decaps. The components in grey are influenced by the long-term secret key s and should be masked. Linear components are highlighted in light grey; non-linear components are in dark grey. The operations which correspond to IND-CPA-secure encryption and decryption are grouped in red.

Practical Performance Results

To achieve the best results, we implemented and compared performance of the unmasked decapsulation operation, Kyber.CCAKEM.Decaps with three different configurations of the first-order masked implementations of Masked.Kyber.CCAKEM.Decaps, optimized in ARM Cortex-M4 assembly and in C:

- Configuration (A): As a basis for our masked implementation, we take the unmasked Kyber768 implementation from the *pqm4* crypto library.
- Configuration (B): In our most efficient masked design (B), we implement the masked decapsulation together with the bitsliced compression and the decompressed comparison. Since both algorithms work with the prime modulo $q = 3329$ and include the same table-based A2B$_q$ conversion as a subroutine, the precomputed look-up table needs to be generated just once for every Kyber invocation and can be reused by both operations.
- Configuration (C): In our masked design (C), we include the bitsliced compression and our new bitsliced comparison in the decapsulation operation. As in the previous configuration, both algorithms work with the prime modulo $q = 3329$ and use the table-based A2B$_q$ conversions, with the look-up tables reused by both components.
- Configuration (D): In our masked design (D), we substitute the polynomial compression and comparison with the division-based compression and compressed comparison. These algorithms work with the power-of-two moduli 2^{d+f} and include the A2B conversions, based on masked secure addition.

An overview of our performance results is given in Table 5. Our most efficient implementation (B) requires $2567 \cdot 10^3$ cycles for the decapsulation operation, which outperforms the other configurations (C) and (D) by more than $50 \cdot 10^3$ and $320 \cdot 10^3$ CPU cycles, respectively. The overall overhead factors of our masked design (B) compared to an unmasked implementation are 3.02x and 3.23x for CPU cycles and code size, respectively.

To better understand the overall impact of masking, we profile the CPU cycles of the most important components and summarize the results in Table 6. A significant portion of the performance overhead is attributed to a masked re-encryption step, indcpa_enc, that contains a masked binomial sampler, which is computationally expensive due to rejection sampling modulo $q = 3329$ and masked calls to a hash function $\mathcal{H}$. The non-linear operations have overhead factors ranging from 7.53x for hash function $\mathcal{G}$ to 15.00x, 21.32x and 116.33x for polynomial compression, binomial sampling and polynomial comparison, respectively. This slowdown was to be expected, as the non-linear components are expensive to mask. We cannot directly compare the cost of the linear operations, such as polynomial arithmetic and matrix generation, as in the reference implementation these building blocks are interleaved, while we separate them in our masked implementation.

Configuration		CPU Cycles	Code Size
Kyber.CCAKEM.Decaps	(A)	850 (1.00x)	11 296 (1.00x)
	(B)	2 567 (3.02x)	36 470 (3.23x)
Masked.Kyber.CCAKEM.Decaps	(C)	2 619 (3.08x)	36 590 (3.24x)
	(D)	2 889 (3.40x)	36 938 (3.27x)

Table 5: Cycle count in thousands of cycles and code size in bytes of Kyber.CCAKEM.Decaps compared to different implementations of Masked.Kyber.CCAKEM.Decaps. The overhead factor of the masked decapsulation is included in brackets.

Overall, our masked design (B) requires 13458 bytes of randomness, the majority of which are required for binomial sampling of the error polynomials. The compression and comparison operations are randomized as well and use 640 and 4404 random bytes, respectively.

Operation	(unmasked)	Masked.Kyber.CCAKEM.Decaps			
	(A)	(B)		(C)	(D)
`crypto_kem_dec`	850	2 567	(3.02x)	2 619	2 889
`table_gen`	-	40		40	-
`indcpa_dec`	63	156	(2.48x)	156	206
`hashg` (SHA3-512)	13	98	(7.53x)	98	98
`indcpa_enc`	647	1 796	(2.78x)	1 796	1 802
`comparison`	3	349	(116.33x)	400	651
`hashh` (SHA3-256)	113	113	(1.00x)	113	113
`kdf`	13	13	(1.00x)	13	13
#randombytes	-	13 458		12 718	22 640
`indcpa_enc`	647	1 796	(2.78x)	1 796	1 802
`decompression`	-	86		87	89
`gen_matrix`	-	391		391	391
`binomial_sampler`	48	1 023	(21.31x)	1 023	1 027
`poly_arith`	-	294		294	294
#randombytes	-	8 414		8 442	8 296
`indcpa_dec`	64	156	(2.48x)	156	206
`unpack`	-	23		23	23
`poly_arith`	-	87		87	87
`compression`	3	45	(15.00x)	45	94
#randombytes	-	640		640	2048

Table 6: Performance numbers and randomness consumption of unmasked Kyber768 decapsulation operation compared to different implementations of Masked.Kyber.CCAKEM.Decaps and its subroutines. The clock cycle counts are reported in thousands of cycles. The slowdown factor of the most efficient approach (B) compared to the unmasked decapsulation is included in brackets.

5.3 Masked Implementation of Saber

In this section we briefly summarize the first-order masked implementation of Saber from [12], that we use for the comparison with our first-order masked implementation of Kyber. To evaluate the performance of both schemes on a common platform, we integrate the masked decapsulation, Masked.Saber.CCAKEM.Decaps, into the *pqm4* framework with minor modifications. For instance, we removed the hardening instructions, such as randomization of the load and store memory buses and the destination buses of the ALU. These operations reduce the microarchitectural leakage caused by combinations of both shares; they are specific to particular implementation and device and do not affect the functionality of the scheme. Furthermore, we removed overwriting the registers and the stack memory with the randomness before and after use. This was done to make the comparison between masked implementations fairer, however to achieve practical security these operations should be included in both Kyber and Saber, as it is expected that both KEMs are affected by the microarchitectureal effects in roughly the same way.

In Figure 8, it can be seen that the masked decapsulation of Saber is very similar to that of Kyber. The linear arithmetic operations - polynomial multiplication, addition and subtraction are masked by duplicating the operation on both shares. The symmetric components $\mathcal{G}$ and $\mathcal{H}$ are masked as in Kyber by reusing efficient first-order masked Keccak implementation from [15]. For binomial sampler, β_η, the efficient bitsliced sampler [71], described in Algorithm 27, is adapted to the parameters of Saber.

To compare re-encrypted masked ciphertext coefficients with public ciphertext coefficients, the hash-based comparison method, given in Section 3.8.1, is used together with the ciphertext compression. Since Saber works with power-of-two moduli, this compression operation can be replaced by a logical shift operation, the masked version of which is implemented using the table-based A2A conversion method [12], described in Section 3.6. As we have argued in Section 5.2.3, the output of the comparison does not provide any additional information to a side-channel adversary, therefore there is no need to mask the key derivation component $\mathcal{H}'$. This reasoning also applies to Saber as both schemes include the FO transform in their design.
To achieve the best performance, several components of Saber are optimized in ARM Cortex-M4 assembly. For instance, secure Boolean operations are implemented as assembly macros; the higher-level functions, such as A2A conversions, polynomial comparison and binomial sampler are implemented completely in assembly and use the low-level macros as subroutines. The polynomial arithmetic operations, such as Toom-Cook, Karatsuba multiplications and a pointwise multiplication are further assembly components that are reused from the unmasked reference implementation.

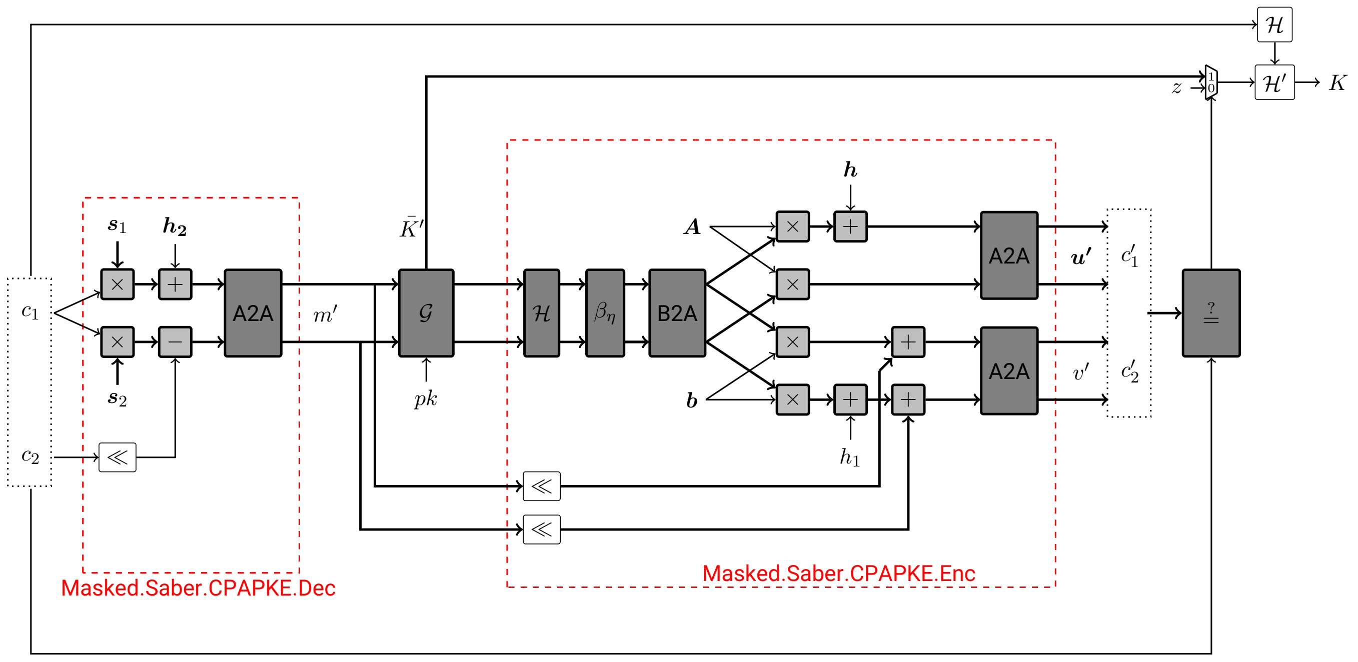

Figure 8: Masked IND-CCA-secure decapsulation of Saber, Masked.Saber.CCAKEM.Decaps. The components in grey are influenced by the long-term secret key s and should be masked. Linear components are highlighted in light grey; non-linear components are in dark grey. The operations which correspond to IND-CPA-secure encryption and decryption are grouped in red.

5.4 Comparing Masked Implementations

In Table 7, we compare our most efficient first-order masked implementation of Kyber with a first-order masked implementation of Saber from [12]. While both implementations correspond to the *NIST Security Level III*, their parameters are different - for instance, Kyber achieves classical and quantum security levels of 2^{182} and 2^{165}, respectively, with the error probability $\delta = 2^{-164}$ [7], while the corresponding parameters for Saber are 2^{189}, 2^{172} and $\delta = 2^{-136}$ [11], respectively. This makes an absolute comparison not possible. Therefore, we compare the overhead factors of masking, which are 3.02x and 2.50x for performance and 3.23x and 4.18x for code size of Kyber and Saber, respectively.

The performance slowdown of Kyber over Saber is mainly caused by two main characteristics of Kyber - the choice of prime modulo together with the MLWE as the underlying hard problem. Because of the use of prime modulo $q = 3329$, the low-level components such as A2B$_q$ and B2A$_q$ conversions and compression are more expensive for Kyber, compared to Saber. For instance, polynomial compression in Saber can be replaced by a simple logical shift, while Kyber requires dedicated compression operation, that includes A2B$_q$ conversions together with Boolean and arithmetic operations.

Furthermore, the MLWE-based schemes require binomial sampling of four error polynomials. In Saber this operation is replaced by rounding which corresponds to a simple logical shift, computed with the A2A conversion. Another contributing factor is the randomness generation. Both schemes generate the required randomness using the Keccak function $\mathcal{H}$. However, Kyber requires an additional rejection sampling step to obtain the required uniform random numbers modulo q, while Saber can use the output of $\mathcal{H}$ directly. Overall, Kyber utilizes 2.67x more random bytes than Saber.

The code size increase of Saber over Kyber is mainly attributed to the choice of different data structures and the implementation of the polynomial multiplication. Compared to the multiplications in the NTT domain, used in Kyber, the combination of Toom-Cook and Karatsuba multiplications of Saber is less compact and includes fully unrolled loops which severely impact the code size. By adapting the NTT transformation for power-of-two moduli of Saber as suggested in [29], the code size usage of Saber could potentially be improved.

For our target platform and implementation, first-order masking of Kyber is faster than that of Saber. However, to achieve real-world security, higher-order masking is essential. As Saber has lower performance overhead factor, compared to Kyber, it might significantly outperform Kyber in case of higher-order masking.

Configuration		CPU Cycles		Code Size		TRNG
Kyber.CCAKEM.Decaps	(A)	850	(1.00x)	11 296	(1.00x)	0
Masked.Kyber.CCAKEM.Decaps	(B)	2 567	(3.02x)	36 470	(3.23x)	13 458
Saber.CCAKEM.Decaps		1 123	(1.00x)	11 802	(1.00x)	0
Masked.Saber.CCAKEM.Decaps		2 802	(2.50x)	49 339	(4.18x)	5 048

Table 7: Cycle count, code size and TRNG usage of implementations of Masked.Kyber.CCAKEM.Decaps and Masked.Saber.CCAKEM.Decaps compared to their corresponding unmasked versions Kyber.CCAKEM.Decaps and Saber.CCAKEM.Decaps, respectively. The clock cycle counts are given in thousands of cycles; the code size and TRNG usage are reported in bytes.

6 Conclusion

In this thesis, we investigated masking techniques for protecting embedded implementations of Kyber and Saber against side-channel analysis and proposed a new countermeasure for one of the core building blocks of Kyber - the polynomial comparison. We combined different approaches from this and previous works in a first-order masked implementation of Kyber optimized for the ARM Cortex-M4. Our masked decapsulation operation features an overhead factor of 3.02x, compared to its unmasked version, which shows that Kyber is very efficient to mask. Finally, we evaluated the masked implementations of Kyber and Saber on a common platform - STM32F407 evaluation board and assessed the cost of a secure implementation for both schemes.

Further improvements of our work might be addressed in the future. Previous attacks [60, 59] have shown that first-order masking is not sufficient to achieve practical side-channel resistance. To protect against higher-order side-channel analysis, our implementation should be combined with other countermeasures, such as shuffling and extended to the higher-order masking.

Bibliography

[1] Mehdi-Laurent Akkar and Christophe Giraud. "An Implementation of DES and AES, Secure against Some Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Generators. 2001, pp. 309–318. DOI: 10.1007/3-540-44709-1_26. URL: https://doi.org/10.1007/3-540-44709-1%5C_26.

[2] Gorjan Alagic et al. *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*. en. 2019-01-31 00:01:00 2019. DOI: https://doi.org/10.6028/NIST.IR.8240. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=927303.

[3] Jacob Alperin-Sheriff and Daniel Apon. "Dimension-Preserving Reductions from LWE to LWR". In: *IACR Cryptol. ePrint Arch.* (2016), p. 589. URL: http://eprint.iacr.org/2016/589.

[4] Joël Alwen et al. "Learning with Rounding, Revisited - New Reduction, Properties and Applications". In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. 2013, pp. 57–74. DOI: 10.1007/978-3-642-40041-4_4. URL: https://doi.org/10.1007/978-3-642-40041-4%5C_4.

[5] Dorian Amiet et al. "Defeating NewHope with a Single Trace". In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*. 2020, pp. 189–205. DOI: 10.1007/978-3-030-44223-1_11. URL: https://doi.org/10.1007/978-3-030-44223-1%5C_11.

[6] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574 (Oct. 2019), pp. 505–510. DOI: 10.1038/s41586-019-1666-5.

[7] Roberto Maria Avanzi et al. "CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation". In: 2020.

[8] Abhishek Banerjee, Chris Peikert, and Alon Rosen. "Pseudorandom Functions and Lattices". In: *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Springer, 2012, pp. 719–737. DOI: 10.1007/978-3-642-29011-4_42. URL: https://doi.org/10.1007/978-3-642-29011-4%5C_42.

[9] Gilles Barthe et al. "Masking the GLP Lattice-Based Signature Scheme at Any Order". In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*. 2018, pp. 354–384. DOI: 10.1007/978-3-319-78375-8_12. URL: https://doi.org/10.1007/978-3-319-78375-8%5C_12.

[10] Gilles Barthe et al. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl et al. ACM, 2016, pp. 116–129. DOI: 10.1145/2976749.2978427. URL: https://doi.org/10.1145/2976749.2978427.

[11] Andrea Basso et al. *SABER: Mod-LWR based KEM (Round 3 Submission)*. 2020.

[12] Michiel Van Beirendonck et al. "A Side-Channel-Resistant Implementation of SABER". In: *ACM J. Emerg. Technol. Comput. Syst.* 17.2 (2021), 10:1–10:26. DOI: 10.1145/3429983. URL: https://doi.org/10.1145/3429983.

[13] Mihir Bellare and Phillip Rogaway. "Introduction to Modern Cryptography". In: *UCSD CSE 207 Course Notes*. 2005, p. 207.

[14] Mihir Bellare and Phillip Rogaway. "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs". In: *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*. 2006, pp. 409–426. DOI: 10.1007/11761679_25. URL: https://doi.org/10.1007/11761679%5C_25.

[15] Guido Bertoni et al. *Building power analysis resistant implementations of Keccak, Round 3 finalist of the Cryptographic Hash Algorithm Competition of NIST*. 2010.

[16] Guido Bertoni et al. "Keccak". In: *IACR Cryptol. ePrint Arch.* (2015), p. 389. URL: http://eprint.iacr.org/2015/389.

[17] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. "Improved High-Order Conversion From Boolean to Arithmetic Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 22–45. DOI: 10.13154/tches.v2018.i2.22-45. URL: https://doi.org/10.13154/tches.v2018.i2.22-45.

[18] Shivam Bhasin et al. "Attacking and Defending Masked Polynomial Comparison for Lattice-Based Cryptography". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.3 (2021), pp. 334–359. DOI: 10.46586/tches.v2021.i3.334-359. URL: https://doi.org/10.46586/tches.v2021.i3.334-359.

[19] Silvio Biagioni, Daniel Masny, and Daniele Venturi. "Naor-Yung paradigm with shared randomness and applications". In: *Theor. Comput. Sci.* 692 (2017), pp. 90–113. DOI: 10.1016/j.tcs.2017.06.019. URL: https://doi.org/10.1016/j.tcs.2017.06.019.

[20] Nina Bindel et al. "Bounding the Cache-Side-Channel Leakage of Lattice-Based Signature Schemes Using Program Semantics". In: *Foundations and Practice of Security - 10th International Symposium, FPS 2017, Nancy, France, October 23-25, 2017, Revised Selected Papers*. 2017, pp. 225–241. DOI: 10.1007/978-3-319-75650-9_15. URL: https://doi.org/10.1007/978-3-319-75650-9%5C_15.

[21] Alex Biryukov et al. "Optimal First-Order Boolean Masking for Embedded IoT Devices". In: *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*. Ed. by Thomas Eisenbarth and Yannick Teglia. Vol. 10728. Lecture Notes in Computer Science. Springer, 2017, pp. 22–41. DOI: 10.1007/978-3-319-75208-2_2. URL: https://doi.org/10.1007/978-3-319-75208-2%5C_2.

[22] Andrej Bogdanov et al. "On the Hardness of Learning with Rounding over Small Modulus". In: *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*. 2016, pp. 209–224. DOI: 10.1007/978-3-662-49096-9_9. URL: https://doi.org/10.1007/978-3-662-49096-9%5C_9.

[23] Joppe W. Bos et al. "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM". In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 353–367. DOI: 10.1109/EuroSP.2018.00032. URL: https://doi.org/10.1109/EuroSP.2018.00032.

[24] Joppe W. Bos et al. “Masking Kyber: First- and Higher-Order Implementations”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 173–214. DOI: 10.46586/tches.v2021.i4.173-214. URL: https://doi.org/10.46586/tches.v2021.i4.173-214.

[25] Leon Groot Bruinderink et al. “Flush, Gauss, and Reload - A Cache Attack on the BLISS Lattice-Based Signature Scheme”. In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. 2016, pp. 323–345. DOI: 10.1007/978-3-662-53140-2_16. URL: https://doi.org/10.1007/978-3-662-53140-2%5C_16.

[26] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. “Template Attacks”. In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. 2002, pp. 13–28. DOI: 10.1007/3-540-36400-5_3. URL: https://doi.org/10.1007/3-540-36400-5%5C_3.

[27] Suresh Chari et al. “Towards Sound Approaches to Counteract Power-Analysis Attacks”. In: *Advances in Cryptology - CRYPTO ’99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. 1999, pp. 398–412. DOI: 10.1007/3-540-48405-1_26. URL: https://doi.org/10.1007/3-540-48405-1%5C_26.

[28] Lidong Chen et al. *Report on Post-Quantum Cryptography*. en. 2016-04-28 2016. DOI: https://doi.org/10.6028/NIST.IR.8105.

[29] Chi-Ming Marvin Chung et al. “NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 159–188. DOI: 10.46586/tches.v2021.i2.159-188. URL: https://doi.org/10.46586/tches.v2021.i2.159-188.

[30] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. “Secure Conversion between Boolean and Arithmetic Masking of Any Order”. In: *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. 2014, pp. 188–205. DOI: 10.1007/978-3-662-44709-3_11. URL: https://doi.org/10.1007/978-3-662-44709-3%5C_11.

[31] Jean-Sébastien Coron and Alexei Tchulkine. “A New Algorithm for Switching from Arithmetic to Boolean Masking”. In: *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*. Ed. by Colin D. Walter, Çetin Kaya Koç, and Christof Paar. Vol. 2779. Lecture Notes in Computer Science. Springer, 2003, pp. 89–97. DOI: 10.1007/978-3-540-45238-6_8. URL: https://doi.org/10.1007/978-3-540-45238-6%5C_8.

[32] Ronald Cramer and Victor Shoup. “Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack”. In: *SIAM J. Comput.* 33.1 (2003), pp. 167–226. DOI: 10.1137/S0097539702403773. URL: https://doi.org/10.1137/S0097539702403773.

[33] Jan-Pieter D’Anvers et al. “Decryption Failure Attacks on IND-CCA Secure Lattice-Based Schemes”. In: *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*. Ed. by Dongdai Lin and Kazue Sako. Vol. 11443. Lecture Notes in Computer Science. Springer, 2019, pp. 565–598. DOI: 10.1007/978-3-030-17259-6_19. URL: https://doi.org/10.1007/978-3-030-17259-6%5C_19.

[34] Jan-Pieter D'Anvers et al. "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM". In: *IACR Cryptol. ePrint Arch.* (2018), p. 230. URL: http://eprint.iacr.org/2018/230.

[35] Blandine Debraize. "Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking". In: *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*. 2012, pp. 107–121. DOI: 10.1007/978-3-642-33027-8_7. URL: https://doi.org/10.1007/978-3-642-33027-8%5C_7.

[36] Thomas Espitau et al. "Side-Channel Attacks on BLISS Lattice-Based Signatures: Exploiting Branch Tracing against strongSwan and Electromagnetic Emanations in Microcontrollers". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2017, pp. 1857–1874. DOI: 10.1145/3133956.3134028. URL: https://doi.org/10.1145/3133956.3134028.

[37] Tim Fritzmann et al. "Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography". In: *IACR Cryptol. ePrint Arch.* (2021), p. 479. URL: https://eprint.iacr.org/2021/479.

[38] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 537–554. DOI: 10.1007/3-540-48405-1_34. URL: https://doi.org/10.1007/3-540-48405-1%5C_34.

[39] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. "Electromagnetic Analysis: Concrete Results". In: *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Generators. 2001, pp. 251–261. DOI: 10.1007/3-540-44709-1_21. URL: https://doi.org/10.1007/3-540-44709-1%5C_21.

[40] Louis Goubin. "A Sound Method for Switching between Boolean and Arithmetic Masking". In: *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. 2001, pp. 3–15. DOI: 10.1007/3-540-44709-1_2. URL: https://doi.org/10.1007/3-540-44709-1%5C_2.

[41] Louis Goubin and Jacques Patarin. "DES and Differential Power Analysis (The "Duplication" Method)". In: *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*. 1999, pp. 158–172. DOI: 10.1007/3-540-48059-5_15. URL: https://doi.org/10.1007/3-540-48059-5%5C_15.

[42] Qian Guo, Thomas Johansson, and Alexander Nilsson. "A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM". In: *IACR Cryptol. ePrint Arch.* (2020), p. 743. URL: https://eprint.iacr.org/2020/743.

[43] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. "An AES Smart Card Implementation Resistant to Power Analysis Attacks". In: *Applied Cryptography and Network Security, 4th International Conference, ACNS 2006, Singapore, June 6-9, 2006, Proceedings*. 2006, pp. 239–252. DOI: 10.1007/11767480_16. URL: https://doi.org/10.1007/11767480%5C_16.

[44] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation". In: *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. Lecture Notes in Computer Science. Springer, 2017, pp. 341–371. DOI: 10.1007/978-3-319-70500-2_12. URL: https://doi.org/10.1007/978-3-319-70500-2%5C_12.

[45] Wei-Lun Huang, Jiun-Peng Chen, and Bo-Yin Yang. "Power Analysis on NTRU Prime". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.1 (2020), pp. 123–151. DOI: 10.13154/tches.v2020.i1.123-151. URL: https://doi.org/10.13154/tches.v2020.i1.123-151.

[46] Yuval Ishai, Amit Sahai, and David A. Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. 2003, pp. 463–481. DOI: 10.1007/978-3-540-45146-4_27. URL: https://doi.org/10.1007/978-3-540-45146-4%5C_27.

[47] Douglas W. Jones. *Reciprocal Multiplication, a tutorial*. http://homepage.divms.uiowa.edu/~jones/bcd/divide.html. Accessed: 2022-01-10.

[48] Nigel Jones. *Division of integers by constants*. https://embeddedgurus.com/stack-overflow/2009/06/division-of-integers-by-constants/. Accessed: 2022-01-10.

[49] Matthias J. Kannwischer et al. "pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4". In: *IACR Cryptol. ePrint Arch.* (2019), p. 844. URL: https://eprint.iacr.org/2019/844.

[50] Mohamed Karroumi, Benjamin Richard, and Marc Joye. "Addition with Blinded Operands". In: *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*. 2014, pp. 41–55. DOI: 10.1007/978-3-319-10175-0_4. URL: https://doi.org/10.1007/978-3-319-10175-0%5C_4.

[51] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *meltdownattack.com* (2018). URL: https://spectreattack.com/spectre.pdf.

[52] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9. URL: https://doi.org/10.1007/3-540-68697-5%5C_9.

[53] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25. URL: https://doi.org/10.1007/3-540-48405-1%5C_25.

[54] Adeline Langlois and Damien Stehlé. "Worst-Case to Average-Case Reductions for Module Lattices". In: *IACR Cryptol. ePrint Arch.* (2012), p. 90. URL: http://eprint.iacr.org/2012/090.

[55] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018, pp. 973–990. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[56] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors Over Rings". In: *IACR Cryptol. ePrint Arch.* (2012), p. 230. URL: http://eprint.iacr.org/2012/230.

[57] Victor S. Miller. "Use of Elliptic Curves in Cryptography". In: *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*. 1985, pp. 417–426. DOI: 10.1007/3-540-39799-X_31. URL: https://doi.org/10.1007/3-540-39799-X%5C_31.

[58] Dustin Moody et al. *Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process*. en. 2020-07-22 2020. DOI: https://doi.org/10.6028/NIST.IR.8309.

[59] Kalle Ngo, Elena Dubrova, and Thomas Johansson. "Breaking Masked and Shuffled CCA Secure Saber KEM by Power Analysis". In: *ASHES@CCS 2021: Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security, Virtual Event, Republic of Korea, 19 November 2021*. Ed. by Chip-Hong Chang et al. ACM, 2021, pp. 51–61. DOI: 10.1145/3474376.3487277. URL: https://doi.org/10.1145/3474376.3487277.

[60] Kalle Ngo et al. "A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM Implementation". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 676–707. DOI: 10.46586/tches.v2021.i4.676-707. URL: https://doi.org/10.46586/tches.v2021.i4.676-707.

[61] Tobias Oder et al. "Practical CCA2-Secure and Masked Ring-LWE Implementation". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.1 (2018), pp. 142–174. DOI: 10.13154/tches.v2018.i1.142-174. URL: https://doi.org/10.13154/tches.v2018.i1.142-174.

[62] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and Countermeasures: the Case of AES". In: *IACR Cryptol. ePrint Arch.* (2005), p. 271. URL: http://eprint.iacr.org/2005/271.

[63] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. *The Keccak reference*. Round 3 submission to NIST SHA-3. 2011. URL: http://keccak.noekeon.org/Keccak-reference-3.0.pdf.

[64] Peter Pessl and Robert Primas. "More Practical Single-Trace Attacks on the Number Theoretic Transform". In: *IACR Cryptol. ePrint Arch.* (2019), p. 795. URL: https://eprint.iacr.org/2019/795.

[65] Charles Rackoff and Daniel R. Simon. "Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack". In: *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*. Ed. by Joan Feigenbaum. Vol. 576. Lecture Notes in Computer Science. Springer, 1991, pp. 433–444. DOI: 10.1007/3-540-46766-1_35. URL: https://doi.org/10.1007/3-540-46766-1%5C_35.

[66] Prasanna Ravi et al. "Drop by Drop you break the rock - Exploiting generic vulnerabilities in Lattice-based PKE/KEMs using EM-based Physical Attacks". In: *IACR Cryptol. ePrint Arch.* (2020), p. 549. URL: https://eprint.iacr.org/2020/549.

[67] Prasanna Ravi et al. "Generic Side-channel attacks on CCA-secure lattice-based PKE and KEMs". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 307–335. DOI: 10.13154/tches.v2020.i3.307-335. URL: https://doi.org/10.13154/tches.v2020.i3.307-335.

[68] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*. Ed. by Harold N. Gabow and Ronald Fagin. ACM, 2005, pp. 84–93. DOI: 10.1145/1060590.1060603. URL: https://doi.org/10.1145/1060590.1060603.

[69] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems (Reprint)". In: *Commun. ACM* 26.1 (1983), pp. 96–99. DOI: 10.1145/357980.358017. URL: https://doi.org/10.1145/357980.358017.

[70] Thomas Schamberger et al. "A Power Side-Channel Attack on the CCA2-Secure HQC KEM". In: *Smart Card Research and Advanced Applications - 19th International Conference, CARDIS 2020, Virtual Event, November 18-19, 2020, Revised Selected Papers*. 2020, pp. 119–134. DOI: 10.1007/978-3-030-68487-7_8. URL: https://doi.org/10.1007/978-3-030-68487-7%5C_8.

[71] Tobias Schneider et al. "Efficiently Masking Binomial Sampling at Arbitrary Orders for Lattice-Based Crypto". In: *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*. 2019, pp. 534–564. DOI: 10.1007/978-3-030-17259-6_18. URL: https://doi.org/10.1007/978-3-030-17259-6%5C_18.

[72] Peter W. Shor. “Algorithms for Quantum Computation: Discrete Logarithms and Factoring”. In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700. URL: https://doi.org/10.1109/SFCS.1994.365700.

[73] Victor Shoup. “Sequences of games: a tool for taming complexity in security proofs”. In: *IACR Cryptol. ePrint Arch.* (2004), p. 332. URL: http://eprint.iacr.org/2004/332.

[74] Bo-Yeon Sim, Aesun Park, and Dong-Guk Han. “Chosen-ciphertext Clustering Attack on CRYSTALS-KYBER using the Side-channel Leakage of Barrett Reduction”. In: *IACR Cryptol. ePrint Arch.* (2021), p. 874. URL: https://eprint.iacr.org/2021/874.

[75] Ehsan Ebrahimi Targhi and Dominique Unruh. “Post-Quantum Security of the Fujisaki-Okamoto and OAEP Transforms”. In: *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part II*. Ed. by Martin Hirt and Adam D. Smith. Vol. 9986. Lecture Notes in Computer Science. 2016, pp. 192–216. DOI: 10.1007/978-3-662-53644-5_8. URL: https://doi.org/10.1007/978-3-662-53644-5%5C_8.

[76] Zhuang Xu et al. “Magnifying Side-Channel Leakage of Lattice-Based Cryptosystems with Chosen Ciphertexts: The Case Study of Kyber”. In: *IACR Cryptol. ePrint Arch.* (2020), p. 912. URL: https://eprint.iacr.org/2020/912.

[77] Han-Sen Zhong et al. *Quantum computational advantage using photons*. Dec. 2020.