Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography

Tim Fritzmann[1], Michiel Van Beirendonck[2], Debapriya Basu Roy[1], Patrick Karl[1], Thomas Schamberger[1], Ingrid Verbauwhede[2] and Georg Sigl[13]

[1] Technical University of Munich, TUM Department of Electrical and Computer Engineering, Chair of Security in Information Technology, Munich, Germany
{tim.fritzmann,debapriya.basu-roy,patrick.karl,t.schamberger,sigl}@tum.de
[2] imec-COSIC KU Leuven
Kasteelpark Arenberg 10 - bus 2452, 3001 Leuven, Belgium
{michiel.vanbeirendonck,ingrid.verbauwhede}@esat.kuleuven.be
[3] Fraunhofer Institute for Applied and Integrated Security (AISEC), Germany

Abstract. Side-channel attacks can break mathematically secure cryptographic systems leading to a major concern in applied cryptography. While the cryptanalysis and security evaluation of Post-Quantum Cryptography (PQC) have already received an increasing research effort, a cost analysis of efficient side-channel countermeasures is still lacking. In this work, we propose a masked HW/SW codesign of the NIST PQC finalists Kyber and Saber, suitable for their different characteristics. Among others, we present a novel masked ciphertext compression algorithm for non-power-of-two moduli. To accelerate linear performance bottlenecks, we developed a generic Number Theoretic Transform (NTT) multiplier, which, in contrast to previously published accelerators, is also efficient and suitable for schemes not based on NTT. For the critical non-linear operations, masked HW accelerators were developed, allowing a secure execution using RISC-V instruction set extensions. Our experimental results show a cycle count reduction factor of 3.18 for Kyber (K:245k/E:319k/D:339k) and 2.66 for Saber (K:229k/E:308k/D:347k) compared to the latest optimized ARM Cortex-M4 implementations. While Saber performs slightly better for the key generation and encapsulation, Kyber has slight performance advantages for the decapsulation. The masking overhead for the first-order secure decapsulation operation including randomness generation is around 4.14 for Kyber (D:1403k) and 2.63 for Saber (D:915k).

Keywords: Post-quantum cryptography · Kyber · Saber · masking · RISC-V · accelerators · instruction set extensions

Introduction

The fast progress in the area of quantum computers drives the need for new cryptographic algorithms resistant against quantum attacks. While classical public-key cryptography, such as RSA and Elliptic Curve Cryptography (ECC), will be broken with a large-scale quantum computer, Post-Quantum Cryptography (PQC) refers to a set of algorithms that are supposed to be secure against cryptanalytic quantum attacks. To accelerate the transition from classical to quantum-secure cryptography, the National Institute of Standards and Technology (NIST) started a standardization process [Nat16] and recently selected seven algorithms as finalists and eight alternate candidates [AASA+20]. Out of the seven finalists, five schemes are based on the hardness of structured lattice problems.

Lattice-based cryptography has become one of the most important PQC categories as it is characterized by a very high performance and relatively small ciphertext and key sizes.

In the last years, there has been a high focus on efficient implementations of PQC on constrained devices, with the ARM Cortex-M4 microcontroller as the main target platform [KRSS18, AABCG20, MKV20]. To increase the performance of such devices but keeping the flexibility of a SW solution, HW/SW codesign strategies were recently introduced for PQC [FSM+19, FSF+19, FDNG19]. A new trend is to use platforms based on RISC-V, which is an open Instruction Set Architecture (ISA) constructed using the reduced instruction set principles. The RISC-V initiative has led to a new processor design era, fostering the development of open-source processors and the integration of ISA extensions. In this context, related works developed ISA extensions for PQC achieving high performance and flexibility [AEL+20, FSS20].

While current PQC implementations claim to be resistant against timing Side-Channel Attacks (SCA), more advanced attacks (e.g. power or electromagnetic attacks) where the attacker has physical access to the device have been mostly neglected during the design of optimized implementations. In particular, Differential Power Attacks (DPA) are of major concern [KJJ99]. Simple Power Attacks (SPA) are very sensitive to noise, but DPA reduces the influence of noise by utilizing information from multiple measurements. Therefore, this work focuses on DPA protected PQC implementations.

Related works. Prior works already discuss DPA countermeasures for PQC and in particular for lattice-based cryptography. The first masked lattice scheme based on the Learning with Errors Problem (LWE) was proposed in [RRVV15], and subsequently improved in [RdCR+16]. Protection mechanisms for the GLP signature scheme were analyzed in [BBE+18] and blinding countermeasures for the signature scheme BLISS were developed in [Saa18]. A protected implementation of a predecessor of the NIST PQC second round scheme NewHope was proposed in [OSPG18], but security flaws were found in some of the underlying algorithms [VBDV21, BDH+21]. Later, the second round scheme qTESLA [GR19], the signature finalist Dilithium [MGTF19], and the Public-Key Encryption (PKE) / Key-Encapsulation Mechanism (KEM) finalist Saber [VBDK+20] were protected. Further works proposed higher-order conversions with prime modulus for the binomial sampling [SPOG19] and a fast protection mechanism for polynomial multiplications [HP21].

Contributions. Although there has been good progress due to all of these works, a clear picture of efficient countermeasures and related implementation costs is still not available for the NIST PQC finalists. In particular, NIST assumes that only one of the lattice-based PKE/KEM finalists Kyber, Saber, or NTRU will be standardized after the third round. NTRU is considered less efficient than the other two competitors but has a longer history [AASA+20]. Kyber and Saber have many similarities, but use prime and power-of-two moduli, respectively. To the best of our knowledge, no masked implementation of Kyber has been proposed so far, and as a result, Kyber and Saber have never been directly compared from this perspective. Moreover, it has been postulated that Saber is more efficient to mask [VBDK+20], precisely because prime moduli seem to complicate the introduction of countermeasures.

To provide a first comparison in this regard, we propose a masked implementation of Kyber and Saber on a RISC-V microcontroller. At the same time, we propose the first hardware accelerators and ISA extensions for masking lattice-based cryptography. In many ways, RISC-V is an ideal platform for masking. On one hand, we accelerate computational bottlenecks but keep the flexibility of software solutions. On the other hand, masking critical routines in open-sourced hardware gives us tight control over any side-channel leakage related to the microarchitecture of the processor [BGG+15].

In contrast to the non-masked ISA extensions for NewHope/Kyber/Saber in [FSS20] and NewHope/Kyber in [AEL+20], our HW/SW codesign provides accelerators for the bottlenecks of masked implementations, as well as more generic and powerful accelerators usable for a variety of schemes. In a masked implementation, many operations have to be duplicated, and performance bottlenecks become more present. Although the polynomial multiplication was already thoroughly studied, to the best of our knowledge, a generic multiplier covering a wide range of schemes is still missing. Prior works analyzed the usage of the Number Theoretic Transform (NTT) for schemes that originally use other multiplication methods [FSS20, CHK+21], but an efficient hardware solution is still missing.

Our specific contributions can be summarized as follows. We provide

- A generic NTT-based hardware multiplier suitable for a variety of lattice-based PKE/KEM and signature schemes supporting positive/negative wrapped convolutions, incomplete NTTs, and prime lifts;
- New cycle count records for Saber and Kyber on a RISC-V platform with ISA extensions;
- A novel algorithm for masked compression with prime moduli;
- The first masked implementation of Kyber;
- The first masked hardware implementations or masked HW/SW codesigns for PQC with Saber and Kyber as case study;
- Masked hardware accelerators for hashing, binomial sampling, A2B/B2A conversions, and compression. The considered methods are suitable for higher-order masking;
- Measures towards secure system design with share separation.

Paper organization. Section 1 provides an overview about mathematical hard problems used to construct lattice-based cryptography, the theoretical background of masking as one of the main countermeasures against SCA, and the decapsulation operation of the PQC finalists Kyber and Saber. Section 2 presents how the masking method can be applied for Kyber and Saber. In Section 3, the mathematical background, design exploration, and architecture description of the proposed hardware accelerator for the polynomial ring arithmetic is provided. In Section 4, the accelerators for the non-linear operations are presented. This includes the Keccak, bit-slicing, binomial sampling, and secure adder accelerators. In Section 5, the integration of the proposed accelerators into a RISC-V platform together with an architectural leakage reduction on system level is described. The experimental results of the leakage assessment of our accelerators and the performance evaluation for Kyber and Saber are given in Section 6. The work is summarized in Section 7.

1 Preliminaries

1.1 Module Learning With Errors and Module Learning with Rounding

The NIST PQC finalists Kyber and Saber are based on the Module Learning with Errors (MLWE) and Module Learning with Rounding (MLWR) problems, respectively. The MLWE and MLWR problems are both variants of the Ring Learning with Errors (RLWE) problem [LPR10].

Let $\lfloor x \rfloor$ denote the flooring operation, i.e. rounding towards negative infinity. The rounding operation $\lfloor x \rceil$ rounds towards the nearest integer with ties being rounded up, in other words, it holds that $\lfloor x \rceil = \lfloor x + 0.5 \rfloor$. We also reserve the notations $\lfloor x \rfloor_f$ and

$\lfloor x \rceil_f$ which implies flooring (resp. rounding) x up to f fractional digits. We reserve bold notation for matrices and vectors (of polynomials).

Let $\mathcal{R}_q = \mathbb{Z}_q/\langle\phi(x)\rangle$ be a polynomial ring with the integer q and the cyclotomic polynomial $\phi(x)$. An MLWE instance is defined by $(\boldsymbol{A}, \boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e})$ with the public matrix $\boldsymbol{A} \in \mathcal{R}_q^{k_1 \times k_2}$ sampled from a uniform distribution $\mathcal{U}$, the secret $\boldsymbol{s} \in \mathcal{R}_q^{k_2}$ sampled from a binomial distribution Ψ_{η_1} with parameter η_1, and the error $\boldsymbol{e} \in \mathcal{R}_q^{k_1}$ sampled from Ψ_{η_2} with parameter η_2. In contrast, the MLWR instance is defined by $(\boldsymbol{A}, \boldsymbol{b} = \lfloor \frac{p}{q}(\boldsymbol{A} \cdot \boldsymbol{s}) \rceil)$, replacing the error by a deterministic rounding function that scales the product by p/q and rounds the result to the nearest integer modulo p. As it is known to be a hard problem to distinguish MLWE/MLWR samples from a uniform sample pair and to recover the secret from $\boldsymbol{b}$, these samples are well suited to build cryptographic schemes.

1.2 Masking

Masking [CJRR99] is a well-known countermeasure against SCA. It splits a secret variable into multiple parts called shares. A first-order masking uses two shares, and aims to protect against SCA that extract information from the first-order statistical moment. The algorithm is executed on these shares individually to hide any power consumption that would be correlated with the original secret.

As we deal with matrices and vectors of polynomials, which are further split into shares, we use several separate indices. To index a matrix or a vector, we use square brackets, e.g. $\mathbf{A}[i][j]$. We use a subscript, e.g. $\mathbf{b}[0]_i$, to access the i-th coefficient of the polynomial $\mathbf{b}[0]$ (or also the i-th bit of a variable depending on the context). Finally, we reserve a superscript, e.g. $\mathbf{b}[0]^i$, to access the i-th share of $\mathbf{b}[0]$. In algorithms and figures, we highlight shared variables with s shares, by explicitly writing them as $\mathbf{b}^{\{0:s-1\}}$.

Threshold Implementations (TI). TI is an effective method to prevent side-channel attacks and leakages caused by glitches [NRR06]. The concept is based on multi-party computations. For a TI the following properties must hold: correctness (the result after the computations remains correct), non-completeness (the partial functions and computations are at least independent of one input share), and uniformity (input and output are uniformly distributed). For a security order of d and a function of algebraic degree t, the number of input shares must be $s \geq td + 1$. Thus, first-order TIs require a minimum of three input shares for non-linear functions.

Domain Oriented Masking (DOM). DOM is another masking method that provides side-channel resistance and glitch protection [GMK16]. In contrast to the function-oriented nature of TI, DOM operates on share domains. Operations that process shares from a single domain are uncritical as they can only leak information from one particular share domain. Without the information of the remaining domains, an attacker gains no advantage. Non-linear operations that combine shares from different domains require additional randomness to refresh the cross-domain operations. Figure 1 shows a first-order DOM representation of an AND-gate. Two inputs x and y are independently shared such that $x = x^0 \oplus x^1$ and $y = y^0 \oplus y^1$, where 0 and 1 denote the corresponding share domains and $\oplus$ denotes the Boolean XOR operation. The two cross-domain terms x^0y^1 and y^0x^1 are refreshed by adding a random bit r to the result. Registers in the cross-domain paths make sure that the terms are refreshed before being combined to the resulting shares z^0 and z^1 and thus, prevent glitches.

An advantage of DOM is the scalability for higher-order protection. For a protection order d, DOM results in $(d+1)^2$ intermediate terms of which $d(d+1)$ terms are domain crossing. Because there are always two cross-domain terms combining shares from two given domains, the randomness consumption can be reduced to $d(d+1)/2$ fresh shares.

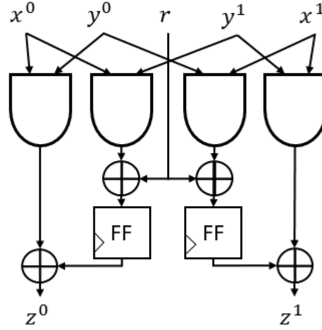


Figure 1: First-order DOM representation of an AND gate [GMK16].

Algorithm 1: SABER.CPA.DEC.	**Algorithm 2:** KYBER.CPA.DEC.
Input: Secret key $\mathbf{s}$	**Input:** Secret key $\mathbf{s}$
Output: Message m	**Output:** Message m
1 $\mathbf{u} \leftarrow \mathbf{c}_1$	1 $\mathbf{u} \leftarrow \texttt{Decompress}_q(\mathbf{c}_1, d_u)$
2 $v \leftarrow h_2 - 2^{\epsilon_p - \epsilon_T} \cdot c_2$	2 $v \leftarrow \texttt{Decompress}_q(c_2, d_v)$
3 $m = (v + (\mathbf{s} \bmod p) \cdot \mathbf{u}^T) \gg (\epsilon_p - 1)$	3 $m = \texttt{Compress}_q(v - \mathbf{s} \cdot \mathbf{u}^T, 1)$

1.3 Kyber and Saber Decapsulation

Both Kyber and Saber include a CPA-secure encryption scheme, from which they build a CCA-secure KEM. Since the plain encryption scheme can be already broken without DPA using CCA-style attacks, we choose to mask the CCA-secure KEM. Assuming that keys are pre-generated in a secure environment, the CCA-secure decapsulation is the operation that involves the long-term secret key and must be masked appropriately.

In Algorithms 1–6, we recall the decapsulation of Kyber and Saber, which uses the CPA-secure encryption and decryption as subroutines. We use a simplified notation that highlights both the similarities and differences between the two schemes. The listings use common symbols and operators, they hide the encodings into byte arrays, and they abstract away from the various transformations into and out of NTT domain. We note that Kyber uses a prime modulus $q = 3329$, whereas Saber chooses power-of-two moduli $q = 2^{13}$ and $p = 2^{10}$. For a full description of Kyber and Saber, we refer to their respective round 3 specification documents [SAB+20, DKR+20]. Both Kyber and Saber use a variety of symmetric primitives, all of which are based on the SHA3 standard: the hash functions $\mathcal{G}$ and $\mathcal{H}$, an extendable output functions XOF, and a key-derivation function KDF.

2 Masking Kyber and Saber

In this section, we describe the algorithms and methods necessary to create masked implementations of Kyber and Saber. These algorithms define the hardware architectures of our secure accelerators. Our masked implementations of the decapsulation operation in Saber and Kyber are illustrated in Figures 2 and 3, respectively.

As MLWE/MLWR-based schemes, Saber and Kyber use polynomial arithmetic as their main computational building block. Linear operations, such as ring multiplications with an unmasked input, additions, and subtractions, can be duplicated and performed on each arithmetic share individually. As a result, expensive operations such as polynomial multiplications become increasingly attractive to accelerate in hardware. Our developed

Algorithm 3: SABER.CPA.ENC.

Input: Public key $pk = (seed_A, \mathbf{b})$
Input: Message m
Input: Seed r
Output: Ciphertext $\mathbf{c} = (\mathbf{c_1}, c_2)$

1. $\mathbf{A} \xleftarrow{seed_A} \mathcal{U}(\mathcal{R}_q^{k \times k})$
2. $\mathbf{s}' \xleftarrow{XOF(r)} \Psi_\eta$
3. $\mathbf{u} \leftarrow \mathbf{A} \cdot \mathbf{s}'$
4. $v \leftarrow \mathbf{b}^T \cdot (\mathbf{s}' \bmod p) - 2^{\epsilon_p - 1} \cdot m$
5. $\mathbf{c}_1 \leftarrow (\mathbf{u} + \mathbf{h}) \gg (\epsilon_q - \epsilon_p)$
6. $c_2 \leftarrow (v + h_1) \gg (\epsilon_p - \epsilon_T)$

Algorithm 4: KYBER.CPA.ENC.

Input: Public key $pk = (seed_A, \mathbf{b})$
Input: Message m
Input: Seed r
Output: Ciphertext $\mathbf{c} = (\mathbf{c_1}, c_2)$

1. $\mathbf{A} \xleftarrow{seed_A} \mathcal{U}(\mathcal{R}_q^{k \times k})$
2. $(\mathbf{s}', \mathbf{e}_1, e_2) \xleftarrow{XOF(r)} \Psi_{\eta_1} \times \Psi_{\eta_2} \times \Psi_{\eta_2}$
3. $\mathbf{u} \leftarrow \mathbf{A} \cdot \mathbf{s}' + \mathbf{e}_1$
4. $v \leftarrow \mathbf{b}^T \cdot \mathbf{s}' + e_2 + \lceil \frac{q}{2} \rfloor \cdot m$
5. $\mathbf{c}_1 \leftarrow \mathtt{Compress}_q(\mathbf{u}, d_u)$
6. $c_2 \leftarrow \mathtt{Compress}_q(v, d_v)$

Algorithm 5: SABER.CCAKEM.DECAPS.

Input: Ciphertext $\mathbf{c}$
Input: Secret key $\mathbf{sk} = (\mathbf{s}, \mathbf{pk}, \mathcal{H}(pk), z)$
Output: Key K

1. $m' :=$ SABER.CPA.DEC$(\mathbf{s}, \mathbf{c})$
2. $(\bar{K}', r') := \mathcal{G}(m' || \mathcal{H}(pk))$
3. $\mathbf{c}' :=$ SABER.CPA.ENC$(\mathbf{pk}, m', r')$
4. **if** $\mathbf{c} = \mathbf{c}'$ **then**
5. | $K := \text{KDF}(\bar{K}' || \mathcal{H}(\mathbf{c}))$
6. **else**
7. | $K := \text{KDF}(z || \mathcal{H}(\mathbf{c}))$
8. **end**

Algorithm 6: KYBER.CCAKEM.DECAPS.

Input: Ciphertext $\mathbf{c}$
Input: Secret key $\mathbf{sk} = (\mathbf{s}, \mathbf{pk}, \mathcal{H}(pk), z)$
Output: Key K

1. $m' :=$ KYBER.CPA.DEC$(\mathbf{s}, \mathbf{c})$
2. $(\bar{K}', r') := \mathcal{G}(m' || \mathcal{H}(pk))$
3. $\mathbf{c}' :=$ KYBER.CPA.ENC$(\mathbf{pk}, m', r')$
4. **if** $\mathbf{c} = \mathbf{c}'$ **then**
5. | $K := \text{KDF}(\bar{K}' || \mathcal{H}(\mathbf{c}))$
6. **else**
7. | $K := \text{KDF}(z || \mathcal{H}(\mathbf{c}))$
8. **end**

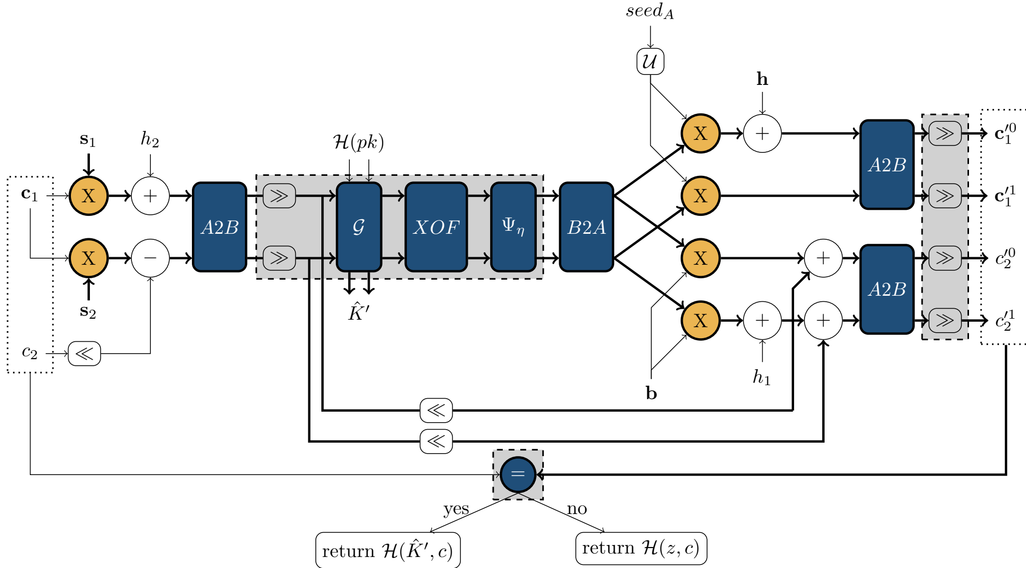


Figure 2: Masked decapsulation of Saber [VBDK+20]. Linear performance bottlenecks highlighted in yellow, non-linear masked routines highlighted in blue, operations that require Boolean masking grouped in light grey.

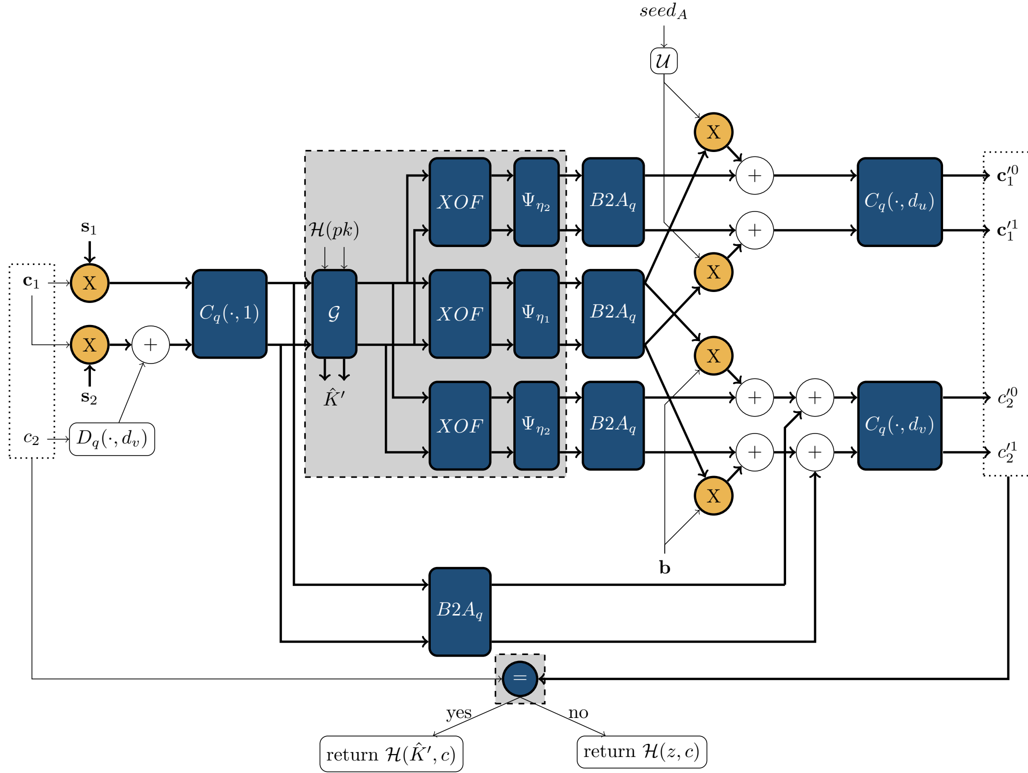


Figure 3: Masked decapsulation of Kyber. Linear performance bottlenecks highlighted in yellow, non-linear masked routines highlighted in blue, operations that require Boolean masking grouped in light grey.

generic hardware accelerator for polynomial arithmetic is presented in Section 3. The polynomial multiplications that we accelerate are highlighted in yellow in Figures 2 and 3.

Non-linear operations are more complex to mask. These operations combine information from both of the shares, and special care must be taken such that they do not jointly leak the secret unmasked value. In Figures 2 and 3, these operations are highlighted in blue. Typically, these operations are expressed in terms of bit-operations, and it is often more natural to fall back to methods based on Boolean masking. The combination of both arithmetic and Boolean masking in Saber and Kyber requires the use of mask conversion algorithms to switch from either Boolean to Arithmetic (B2A) or Arithmetic to Boolean (A2B) masking.

A masked implementation of Saber decapsulation targeting the ARM Cortex-M4 has been proposed in [VBDK+20]. The authors of [VBDK+20] show that Saber is relatively efficient to mask, and argue that this is due to Saber's choice for a power-of-two modulus and the deterministic rounding of MLWR. For the non-linear masked routines, they use a masked Keccak implementation [BDPVA10], a masked binomial sampler [SPOG19], and a masked comparison algorithm [OSPG18], and these exact same methods can be reused for a masked implementation of Kyber. We integrate them into our secure masked accelerators and discuss their hardware architectures in Section 4. To implement B2A and A2B conversions, the authors of [VBDK+20] adopt an algorithm due to Goubin [Gou01] and a table-based algorithm [CT03], respectively. Both of these algorithms are specialized for power-of-two moduli and can therefore not directly be reused for Kyber. Motivated by this observation, we choose to implement different B2A and A2B techniques. In the remainder of this section, we first outline the B2A and A2B conversions that we implement, and we subsequently use them to propose a novel method for masked ciphertext compression.

2.1 B2A and A2B Conversions

B2A and A2B conversions allow to securely convert between an arithmetic masking $x = A^0 + A^1$ and a Boolean masking $x = B^0 \oplus B^1$. These methods may choose to keep a single random mask $A^1 = B^1 = R$, in which case the conversions compute either

$$B^0 = (A^0 + R) \oplus R \text{ , or}$$
$$A^0 = (B^0 \oplus R) - R$$

without unmasking x.

A2B and B2A conversion methods were first proposed by Goubin [Gou01]. In software implementations, A2B conversions can efficiently be implemented using table-based methods [CT03, Deb12, VBDV21]. This is the approach taken in [OSPG18] and [VBDK+20]. The drawbacks of table-based methods are that they do not extend to higher-order security, require work-arounds to handle prime moduli [OSPG18], and that they are relatively difficult to translate to a hardware implementation that also resists glitches. B2A conversions, on the other hand, are typically not table-based. In [VBDK+20], Goubin's B2A method is used, which is specialized for power-of-two moduli. Some ad-hoc methods for prime-modulus B2A$_q$ and A2B$_q$ conversion were proposed in [OSPG18], and subsequently formalized in [BBE+18] and [SPOG19].

In contrast to the previous masked implementations [OSPG18] and [VBDK+20], we employ A2B and B2A conversions that are based on secure masked arithmetic addition over Boolean shares (`SecAdd`) [CGV14]. Our reasoning is many-fold. First, since both A2B and B2A conversion can be expressed in terms of `SecAdd`, we are able to accelerate both operations with a single hardware block. Second, in [BBE+18] this secure adder was extended to work with prime moduli. `SecAdd`$_q$ essentially makes two calls to `SecAdd`, such

that $\mathrm{B2A}_q$ and $\mathrm{A2B}_q$ can additionally be accelerated with the same `SecAdd` hardware. Third, `SecAdd` only requires the masked implementation of a binary adder, and efficient TI implementations of ripple-carry or Kogge-Stone variants have already been proposed [SMG15]. Finally, the A2B and B2A approaches based on secure addition are readily extensible to higher-order security, which is not the case for table-based A2B or Goubin's B2A algorithms. We now describe the conversion based on `SecAdd` in detail. Our focus is on univariate, first-order side-channel security, and wherever possible we simplify the algorithms to focus on this case. For a general description focusing on arbitrary orders and (multivariate) composability, we refer to the original works [CGV14, BBE+18].

`SecAdd` takes as inputs the Boolean maskings $\mathbf{x}^{\{0:1\}} = (x^0, x^1)$ and $\mathbf{y}^{\{0:1\}} = (y^0, y^1)$ and outputs a Boolean masking $\mathbf{s}^{\{0:1\}} = (s^0, s^1)$ such that $(s^0 \oplus s^1) = (x^0 \oplus x^1) + (y^0 \oplus y^1)$. $\texttt{SecAdd}_q$ [BBE+18] can be constructed from `SecAdd` by securely computing a second sum $(s'^0 \oplus s'^1) = (s^0 \oplus s^1) + (q'^0 \oplus q'^1)$, where (q'^0, q'^1) is a Boolean masking of $-q$ in two's complement form. If $x + y \geq q$, then $s' = (x + y - q)$ is the correct sum $(x + y) \bmod q$, and it also holds that $s' \geq 0$. Alternatively, if $x + y < q$, then $s = (x + y)$ is the correct sum and $s' < 0$. Since s' is negative in one case and positive in the other, the masked sign bit $c = \mathrm{sign}(s')$ can be used to select the correct sum:

$$\texttt{SecMux}(\mathbf{s}^{\{0:1\}}, \mathbf{s'}^{\{0:1\}}, \mathbf{c}^{\{0:1\}}) = \texttt{SecAnd}((\mathbf{c}\|...\|\mathbf{c})^{\{0:1\}}, \mathbf{s}^{\{0:1\}}) \oplus \texttt{SecAnd}(\neg(\mathbf{c}\|...\|\mathbf{c})^{\{0:1\}}, \mathbf{s'}^{\{0:1\}}).$$

Having a distinct sign bit requires that s' is computed up to at least $w = \lceil \log_2(q) \rceil + 1$ bits, i.e. one bit larger than the initial masks. $\texttt{SecAdd}_q$ is illustrated in Algorithm 7.

Algorithm 7: $\texttt{SecAdd}_q$ [BBE+18]

Input: $\mathbf{x}^{\{0:1\}} = (x^0, x^1), \mathbf{y}^{\{0:1\}} = (y^0, y^1)$ such that $x = x^0 \oplus x^1$, $y = y^0 \oplus y^1$
Result: $\mathbf{z}^{\{0:1\}} = (z^0, z^1)$ such that $z = z^0 \oplus z^1 = x + y \bmod q$
1 $\mathbf{q'}^{\{0:1\}} \leftarrow (2^w - q, 0)$
2 $\mathbf{s}^{\{0:1\}} \leftarrow \texttt{SecAdd}(\mathbf{x}^{\{0:1\}}, \mathbf{y}^{\{0:1\}})$
3 $\mathbf{s'}^{\{0:1\}} \leftarrow \texttt{SecAdd}(\mathbf{s}^{\{0:1\}}, \mathbf{q'}^{\{0:1\}})$
4 $\mathbf{c}^{\{0:1\}} \leftarrow (\mathbf{s'}^{\{0:1\}} \gg (w - 1))$
5 $\mathbf{z}^{\{0:1\}} \leftarrow \texttt{SecMux}(\mathbf{s}^{\{0:1\}}, \mathbf{s'}^{\{0:1\}}, \mathbf{c}^{\{0:1\}})$

We propose a new simplified version of $\texttt{SecAdd}_q$, which assumes that the input shares already satisfy $(x^0 \oplus x^1) + (y^0 \oplus y^1) = x + y - q$ in two's complement[1]. In this case, it is possible to directly compute $s' = x + y - q$ through $\texttt{SecAdd}(\mathbf{x}^{\{0:1\}}, \mathbf{y}^{\{0:1\}})$. If $s' < 0$, q must be added again to find the correct sum. This time, rather than using $c = \mathrm{sign}(s')$ to multiplex between s' and $s' + q$, we compute the correct sum as $s = s' + c \cdot q$. This is easily possible, since the multiplication with q distributes over the masking $c^0 \oplus c^1 = c$, i.e. $q \cdot c^0 \oplus q \cdot c^1 = q \cdot c$. Our simplified $\texttt{SecAdd}_q$ routine is shown in Algorithm 8. It avoids the masked multiplexer altogether.

A2B conversion follows directly from `SecAdd`. Given an arithmetic masking $x = A^0 + A^1$, the secure addition of A^0 and A^1 with outputs in Boolean masked form is exactly an A2B conversion. A2B and $\mathrm{A2B}_q$ based on this idea are illustrated in Algorithms 9 and 10. In these algorithms, the shares A^0 and A^1 are first themselves shared as a Boolean masking, before being fed into `SecAdd`. As we hinted at before, we have full control over this initial Boolean masking. Therefore, for $\mathrm{A2B}_q$, we create an initial masking of $A^0 + A^1 - q$, and use our simplified version of $\texttt{SecAdd}_q$.

[1] In what follows, we'll see that this is easily possible for the A2B and B2A conversions based on `SecAdd`.

Algorithm 8: $\texttt{SecAdd}_q$ simplified

Input: $\mathbf{x}^{\{0:1\}} = (x^0, x^1), \mathbf{y}^{\{0:1\}} = (y^0, y^1)$ such that $x + y + (2^w - q) = (x^0 \oplus x^1) + (y^0 \oplus y^1)$
Result: $\mathbf{z}^{\{0:1\}} = (z^0, z^1)$ such that $z = z^0 \oplus z^1 = x + y \bmod q$

1. $\mathbf{s'}^{\{0:1\}} \leftarrow \texttt{SecAdd}(\mathbf{x}^{\{0:1\}}, \mathbf{y}^{\{0:1\}})$
2. $\mathbf{c}^{\{0:1\}} \leftarrow (\mathbf{s'}^{\{0:1\}} \gg (w-1))$
3. $\mathbf{c'}^{\{0:1\}} \leftarrow (c^0 \cdot q,\, c^1 \cdot q)$
4. $\mathbf{z}^{\{0:1\}} \leftarrow \texttt{SecAdd}(\mathbf{s'}^{\{0:1\}}, \mathbf{c'}^{\{0:1\}})$

Algorithm 9: A2B [CGV14]

Input: $\mathbf{x}^{\{0:1\}} = (A^0, A^1)$ such that $x = A^0 + A^1 \mod 2^k$
Result: $\mathbf{x}^{\{0:1\}} = (B^0, B^1)$ such that $x = B^0 \oplus B^1$

1. $R^0, R^1 \xleftarrow{\$} \mathbb{Z}_{2^k}$
2. $\mathbf{B}_1^{\{0:1\}} \leftarrow (A^0 \oplus R^0, R^0)$
3. $\mathbf{B}_2^{\{0:1\}} \leftarrow (A^1 \oplus R^1, R^1)$
4. $\mathbf{x}^{\{0:1\}} \leftarrow \texttt{SecAdd}(\mathbf{B}_1^{\{0:1\}}, \mathbf{B}_2^{\{0:1\}})$

Algorithm 10: A2B_q [BBE+18]

Input: $\mathbf{x}^{\{0:1\}} = (A^0, A^1)$ such that $x = A^0 + A^1 \mod q$
Result: $\mathbf{x}^{\{0:1\}} = (B^0, B^1)$ such that $x = B^0 \oplus B^1$

1. $R^0, R^1 \xleftarrow{\$} \mathbb{Z}_q$
2. $\mathbf{B}_1^{\{0:1\}} \leftarrow (A^0 \oplus R^0, R^0)$
3. $\mathbf{B}_2^{\{0:1\}} \leftarrow ((A^1 + (2^w - q)) \oplus R^1, R^1)$
4. $\mathbf{x}^{\{0:1\}} \leftarrow \texttt{SecAdd}_q(\mathbf{B}_1^{\{0:1\}}, \mathbf{B}_2^{\{0:1\}})$

B2A conversion uses a similar idea. Given a Boolean masking $x = B^0 \oplus B^1$, the first arithmetic share A^0 is simply sampled randomly. The second arithmetic share can then be computed from the first one by securely computing $A^1 = (B^0 \oplus B^1) - A^0 \bmod 2^k$. Like in the A2B case, first a Boolean masking is created of $-A^0 \bmod 2^k$ and subsequently this is fed into SecAdd. The result is a Boolean masking $A^1 = B^0 \oplus B^1$, which can be decoded[2] to find the second share A^1. B2A and B2A_q are illustrated in Algorithms 11 and 12. Again, to utilize our simplified and more efficient version of $\texttt{SecAdd}_q$, we simply create an initial two's complement Boolean sharing of $(-A^0 \bmod q) - q$ instead.

2.2 Masked Compression

Both Saber and Kyber include a compression operation that rounds away some low-order bits of a ring element. In Line 3 of the decryption step, the compression operation is used for message decoding, i.e. mapping $(\lceil \frac{q}{2} \rfloor \cdot m + e)$ back to m. In the encryption step, Lines 5 and 6, the same operation is used to compress the ciphertext components $\boldsymbol{u}$ and v.

[2] This requires a secure FullXor(**B**) for composability proofs [CGV14]

Algorithm 11: B2A [CGV14]

Input: $\mathbf{x}^{\{0:1\}} = (B^0, B^1)$ such that $x = B^0 \oplus B^1$
Result: $\mathbf{x}^{\{0:1\}} = (A^0, A^1)$ such that $x = A^0 + A^1 \mod 2^k$

1. $A^0, R \xleftarrow{\$} \mathbb{Z}_{2^k}$
2. $\mathbf{B}_1^{\{0:1\}} \leftarrow ((2^k - A^0) \oplus R, R)$
3. $\mathbf{B}_2^{\{0:1\}} \leftarrow \texttt{SecAdd}(\mathbf{x}^{\{0:1\}}, \mathbf{B}_1^{\{0:1\}})$
4. $\mathbf{x}^{\{0:1\}} \leftarrow (A^0, B_2^0 \oplus B_2^1)$

Algorithm 12: B2A_q [BBE+18]

Input: $\mathbf{x}^{\{0:1\}} = (B^0, B^1)$ such that $x = B^0 \oplus B^1$
Result: $\mathbf{x}^{\{0:1\}} = (A^0, A^1)$ such that $x = A^0 + A^1 \mod q$

1. $A^0, R \xleftarrow{\$} \mathbb{Z}_q \times \mathbb{Z}_{2^w}$
2. $\mathbf{B}_1^{\{0:1\}} \leftarrow (((q - A^0) + (2^w - q)) \oplus R, R)$
3. $\mathbf{B}_2^{\{0:1\}} \leftarrow \texttt{SecAdd}_q(\mathbf{x}^{\{0:1\}}, \mathbf{B}_1^{\{0:1\}})$
4. $\mathbf{x}^{\{0:1\}} \leftarrow (A^0, B_2^0 \oplus B_2^1)$

For Saber, ciphertext compression is inherently tied to the security of its MLWR instance, whereas Kyber initially[3] only compressed the ciphertext components to reduce their size.

The Kyber compression function takes an input $x \in \mathbb{Z}_q$ and outputs an integer in $\{0, \ldots, 2^d - 1\}$, where $d < \lceil \log_2(q) \rceil$:

$$\texttt{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod 2^d \tag{1}$$

For Saber, where $q = 2^{13}$ is a power of two, $\texttt{Compress}_{2^k}$ can be expressed as a more simple logical shift. In order to round the result instead of flooring, the constant h_1, h_2 or $\mathbf{h}$ are added before the shift.

Compression must discard some lower-order bits of arithmetically masked ring elements. Discarding these bits is inherently a Boolean operation, and A2B conversion can help to mask this operation effectively. In [VBDK+20], a new technique is proposed to optimize A2B conversion for masked logical shifting in Saber. Compression for prime moduli has so far only been treated in the context of message decoding in [RRVV15] and [OSPG18]. We first review these existing approaches and show that they do not extend efficiently to ciphertext compression. Subsequently, we outline a novel method to mask $\texttt{Compress}_q$, which is simple and efficient to implement.

2.2.1 $\texttt{MaskedCompress}_{2^k}$

For power-of-two moduli, ciphertext compression constitutes a simple rounded logical shift. In an arithmetic sharing, $(x_{msb} \| x_{lsb}) = (A^0_{msb} \| A^0_{lsb}) + (A^1_{msb} \| A^1_{lsb})$, this shift needs special consideration because the lower bits $A^0_{lsb} + A^1_{lsb}$ might contain a carry that must be added to the upper bits before they are shifted out. A straightforward solution is to first perform A2B conversion, since a Boolean masking $(x_{msb} \parallel x_{lsb}) = (B^0_{msb} \parallel B^0_{lsb}) \oplus (B^1_{msb} \parallel B^1_{lsb})$ doesn't have any carries.

The masked Saber implementation of [VBDK+20] optimizes table-based A2B conversion to only compute the carry for the lower bits, rather than a full conversion. This carry is subsequently added to the higher bits, leaving them as an arithmetic sharing. The full procedure is termed A2A conversion.

The A2A optimization also applies to the A2B conversion based on `SecAdd`. In this setting, when only the carry-out is required, all the summation logic can be pruned from the binary adder. Furthermore, since the carry is only needed at the final position, any carry look-ahead logic can be implemented maximally sparse. However, this optimization would also prevent us from supporting B2A conversion with the same `SecAdd` hardware. Hence, we implement the more simple solution, i.e. a full A2B conversion and subsequent share-wise logical shift.

2.2.2 $\texttt{MaskedDecode}_q$

Masked decoders have been proposed in [RRVV15] and [OSPG18]. Rather than dividing by the modulus, the decoding step is expressed as an interval comparison:

$$\texttt{Decode}_q(m) = \texttt{Compress}_q(m, 1) = \begin{cases} 0 \text{ for } m \in [\lceil 3q/4 \rceil, \lfloor q/4 \rfloor] \\ 1 \text{ for } m \in [\lceil q/4 \rceil, \lfloor 3q/4 \rfloor] \end{cases} . \tag{2}$$

For an arithmetic masking $x = A^0 + A^1$, [RRVV15] proposes a probabilistic decoder that uses information on the quadrants of A^0 and A^1 in a masked table lookup. A different approach is taken in [OSPG18], where a series of A2B-related transformations are used to create a masked decoder. Lacking an existing A2B_q transform, the authors propose another conversion, `TransformPower2`, that transforms an arithmetic masking mod q to

[3] Since round 3, Kyber-512 relies to a small extent on the rounding noise to add security as well.

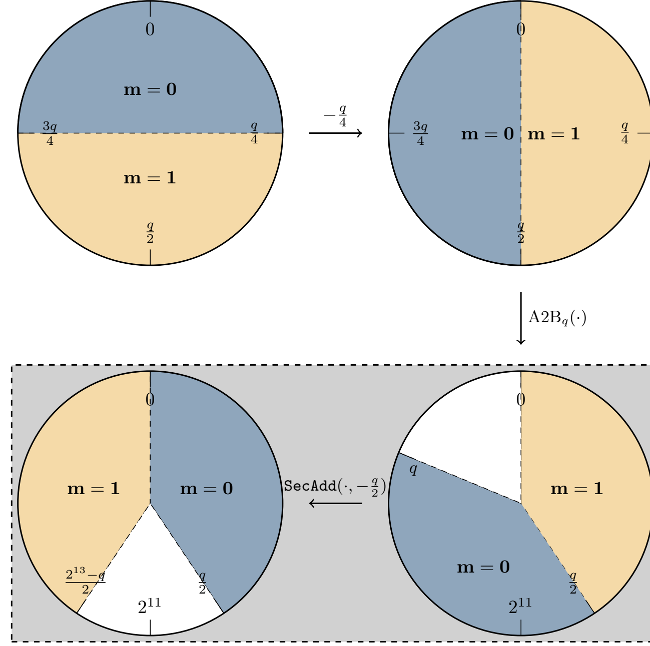


Figure 4: $\texttt{MaskedDecode}_q$. The final result is a Boolean masking where the most significant bit is a masking $(m^0 \oplus m^1) = m$. Adapted from [OSPG18] to use $\mathrm{A2B}_q$ instead of `TransformPower2`. Operations that require Boolean masking grouped in light grey.

a masking mod 2^k. Nevertheless, we do have an $\mathrm{A2B}_q$ conversion available, and use it to simplify the masked decoder of [OSPG18]. The resulting process is shown in Figure 4. The final result is a Boolean masking where the most significant bit is a masking of the decoded result: $(m^0 \oplus m^1) = m$.

2.2.3 $\texttt{MaskedCompress}_q$

Unfortunately, the techniques of masked decoding do not extend to masked compression. When dealing with 2 intervals of width $\frac{q}{2}$, it is possible to position their boundary exactly at a power of two as in Figure 4. However, already for $d = 2$ we have 4 intervals of width $\frac{q}{4}$, and this technique is no longer applicable.

We propose a substantially different masked compression technique. Rather than expressing it as an interval comparison, we analyze and mask the required division by the modulus q. The key idea is simple. First, we observe that the compression tolerates an approximate quotient $x' \approx (2^d/q) \cdot x$. In other words,

$$\texttt{Compress}_q(x, d) = \lceil (2^d/q) \cdot x + e \rfloor \bmod 2^d \quad , \tag{3}$$

remains correct for a small bounded error e. The reason for this is apparent if we express $(2^d/q) \cdot x$ as a binary fraction:

$$(2^d/q) \cdot x = \underbrace{\overbrace{\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}}^{\lfloor \frac{2^d \cdot x}{q} \rfloor}}_{d} \bullet \underbrace{\overbrace{\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}\boxed{\cdot}}^{\frac{2^d \cdot x \bmod q}{q}}}_{\infty} \quad . \tag{4}$$

The fractional part equals $\frac{2^d \cdot x \bmod q}{q}$, and it is strictly limited to the set of values $\frac{\{0,\ldots,q-1\}}{q}$. This fractional part is never exactly 0.5, but instead the edge-case values are $\frac{\lfloor \frac{q}{2} \rfloor}{q} = \frac{1664}{3329}$ and $\frac{\lceil \frac{q}{2} \rceil}{q} = \frac{1665}{3329}$, which should be rounded down and up, respectively. These values are still rounded correctly, even when subject to a small error $-\frac{1}{2q} \leq e < \frac{1}{2q}$:

$$\lceil \frac{\lfloor \frac{q}{2} \rfloor}{q} + e \rfloor = 0 \tag{5}$$

$$\lceil \frac{\lceil \frac{q}{2} \rceil}{q} + e \rfloor = 1 \ . \tag{6}$$

As a result, the approximate quotient $\lceil (2^d/q) \cdot x + e \rfloor$ is rounded correctly, given the same bound on e.

Our simple but crucial observation to build $\texttt{MaskedCompress}_q$ is that we can compute such an approximate quotient individually from the shares of $x = x^0 + x^1 \mod q$, using only finite-precision arithmetic. For example, using integer division, we can compute

$$\lfloor (2^d/q) \cdot x^0 \rfloor_f + \lfloor (2^d/q) \cdot x^1 \rfloor_f = ((2^d/q) \cdot x + e) \bmod 2^d \ , \tag{7}$$

which is a strict underestimate of the real quotient $(2^d/q) \cdot x \bmod 2^d$ with $e < 0$. More generally, we can compute rounded share-wise quotients,

$$\lfloor (2^d/q) \cdot x^0 \rceil_f + \lfloor (2^d/q) \cdot x^1 \rceil_f = ((2^d/q) \cdot x + e) \bmod 2^d \ , \tag{8}$$

with a bounded error e. In both cases, the rounding error e can be arbitrarily lowered by increasing the number of fractional bits f. As a result, for an appropriately large choice of f that fixes $-\frac{1}{2q} \leq e < \frac{1}{2q}$, the share-wise 'fixed-point' quotients of Equations 7 and 8 can be used to correctly retrieve the output of $\texttt{Compress}_q$.

We now analyze the requirements on f in detail. The share-wise quotients of Equations 7 and 8 consist of d integer and f fractional bits, with the remaining bits being truncated or rounded, respectively:

$$(2^d/q) \cdot x^i = \underbrace{\overbrace{\boxed{\cdot|\cdot|\cdot|\cdot}}^{\lfloor \frac{2^d \cdot x^i}{q} \rfloor}}_{d} \bullet \underbrace{\overbrace{\boxed{\cdot|\cdot|\cdot|\cdot|\cdot|\cdot|\cdot}}^{\lfloor \frac{2^d \cdot x^i \bmod q}{q} \rfloor_f}}_{f} \underbrace{\overbrace{\boxed{\cdot|\cdot|\cdot|\cdot|\cdot|\cdot|\cdot}}^{\frac{2^{d+f} \cdot x^i \bmod q}{q}}}_{\infty} \ . \tag{9}$$

When the quotients are truncated as in Equation 7, $\lfloor (2^d/q) \cdot x^0 \rfloor_f$ and $\lfloor (2^d/q) \cdot x^1 \rfloor_f$ produce a strict underestimate of the real quotient $(2^d/q) \cdot x$. This underestimate has the effect of truncating the actual quotient $(2^d/q) \cdot x$, and possibly omits a carry-in from the additive shares at the f-th fractional bit:

$$\lfloor (2^d/q) \cdot x^0 \rfloor_f + \lfloor (2^d/q) \cdot x^1 \rfloor_f = \lfloor (2^d/q) \cdot x \rfloor_f - \frac{\{0,1\}}{2^f} \bmod 2^d \ . \tag{10}$$

Nevertheless, this underestimate can still be rounded correctly, if f is chosen such that fractional values larger than 0.5 do not underflow below 0.5. Specifically, when $(2^d/q) \cdot x$ takes the edge-case fractional value $\frac{\lceil \frac{q}{2} \rceil}{q}$, it must hold that $\lfloor \frac{\lceil \frac{q}{2} \rceil}{q} \rfloor_f - \frac{1}{2^f} \geq 0.5$. For Kyber, this holds for $f \geq 13$.[4]

We can similarly analyze the rounded quotients of Equation 8. By rounding at the $(f+1)$-th binary digit, the worst-case rounding error is $|e_i| < \frac{1}{2^{f+1}}$ for each share-wise

[4] Since $\frac{1665}{3329} = {'b}0.1000000000001...$ as a binary fraction, an underflow is allowed at the 13-th position.

quotient.[5] The total rounding error for two shares therefore remains strictly bounded by $|e| < 2 \cdot \frac{1}{2^{f+1}}$. To satisfy $-\frac{1}{2q} \leq e < \frac{1}{2q}$, it suffices that $f > \log_2(2q)$, which again results in $f \geq 13$ for Kyber. As truncation is easier to implement than rounding and results in the same bound, we choose to implement it in our algorithm.

After computing share-wise quotients with a certain precision f, we obtain a 'fixed-point' arithmetic sharing:

$$x'^0 = \lfloor (2^d/q) \cdot x^0 \rfloor_f \tag{11}$$

$$x'^1 = \lfloor (2^d/q) \cdot x^1 \rfloor_f \tag{12}$$

$$(2^d/q) \cdot x \approx x'^0 + x'^1 \bmod 2^d \ , \tag{13}$$

with d integer bits and f fractional bits. For an appropriately large choice of f, this 'fixed-point' arithmetic sharing allows us to recover the output of $\texttt{Compress}_q(x, d)$:

$$\texttt{Compress}_q(x, d) = \lceil x'^0 + x'^1 \rfloor \bmod 2^d \tag{14}$$

$$= \lfloor x'^0 + x'^1 + 0.5 \rfloor \bmod 2^d \ . \tag{15}$$

Somewhat surprisingly, we have reduced $\texttt{MaskedCompress}_q$ exactly to the problem of $\texttt{MaskedCompress}_{2^k}$. The final output of $\texttt{Compress}_q(x, d)$ constitutes the upper d bits of the $(d + f)$-bit arithmetic sharing $(x'^0, x'^1 + 0.5)$, which we compute with a $(d + f)$-bit A2B conversion and subsequent share-wise logical shift. As before, the A2A conversion of [VBDK+20] is applicable to optimize the computation of the carry-in, but prevents unified hardware in our case.

We illustrate our $\texttt{MaskedCompress}_q$ routine in Algorithm 13 and also graphically in Figure 5, using only integer arithmetic and flooring divisions.[6] The simplicity is apparent, requiring only a single A2B call that combines information from the shares. For higher-order security, f must be chosen to tolerate rounding errors from an increasing number of shares. As a result, the required bit-size of the A2B grows logarithmically with the number of shares. For first-order security with $f = 13$, the largest value for d is $d_u = 11$ in Kyber-1024, requiring a 24-bit power-of-two A2B conversion. Using our novel $\texttt{MaskedCompress}_q$ algorithm, masked Kyber doesn't require any actual A2B_q conversion.

Algorithm 13: $\texttt{MaskedCompress}_q$

Input: $\mathbf{x}^{\{0:1\}} = (x^0, x^1)$ such that $x = x^0 + x^1$, d, $f > \log_2(2q)$
Result: $\mathbf{z}^{\{0:1\}} = (z^0, z^1)$ such that $z = z^0 \oplus z^1 = \texttt{Compress}_q(x, d)$

1. $x'^0 \leftarrow \lfloor (2^{d+f} \cdot x^0)/q \rfloor \bmod 2^{d+f}$
2. $x'^1 \leftarrow (\lfloor (2^{d+f} \cdot x^1)/q \rfloor + 2^f \cdot 0.5) \bmod 2^{d+f}$
3. $\mathbf{z}^{\{0:1\}} \leftarrow \texttt{A2B}(\mathbf{x}'^{\{0:1\}})$ // $(d + f)$-bit A2B
4. $\mathbf{z}^{\{0:1\}} \leftarrow \mathbf{z}^{\{0:1\}} \gg f$

2.3 Masked Equality Test

At the end of the decapsulation, the re-encrypted ciphertext $\mathbf{c}' = \mathbf{c}'^0 \oplus \mathbf{c}'^1$ must be checked for equality against the input ciphertext $\mathbf{c}$. The end result of the check is no longer sensitive, but the re-encrypted ciphertext itself mustn't be unmasked.

[5] More precisely, $|e_i| \leq \frac{\lfloor \frac{q}{2} \rfloor}{\frac{q}{2^f}}$.

[6] As an implementational note, for $d > 7$, $(2^{d+f}) \cdot x^i$ can grow larger than 32-bit. The result must be placed in a `uint64_t`, and special care must be exercised that the division of a `uint64_t` by the constant q compiles to a constant-time operation.

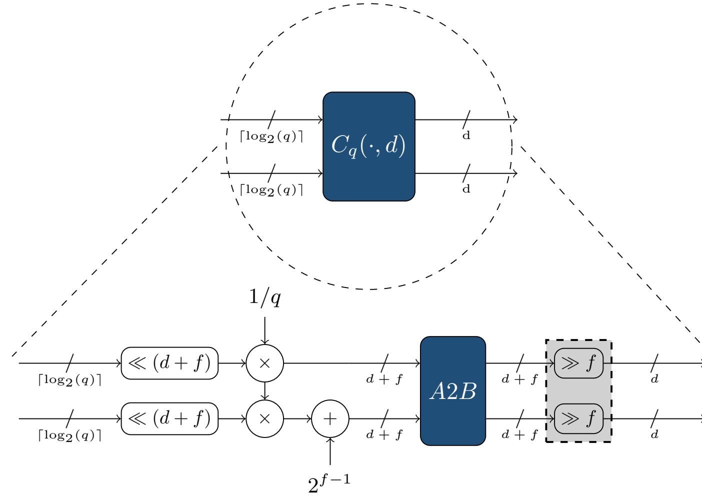


Figure 5: `MaskedCompress`$_q$

Both first-order and higher-order secure algorithms for masked equality testing have been proposed in [OSPG18] and [BPO+20], respectively. In [OSPG18], the main idea is to use an additional hashing step and check whether $\mathcal{H}(\mathbf{c} \oplus \mathbf{c}'^0)$ equals $\mathcal{H}(\mathbf{c}'^1)$[7]. The collision-resistance of $\mathcal{H}$ guarantees that the two hashes are only equal for a valid ciphertext, and the pre-image resistance ensures that the hashes no longer contain exploitable information about $\mathbf{c}'$.

Recently, it was shown that both the [OSPG18] and [BPO+20] methods leak some information on $\mathbf{c}'$, and that this information can be used to significantly decrease the security of the underlying MLWE instance [BDH+21]. The method of [OSPG18] contains a flaw because it checks the equality of the two masked ciphertext components $\mathbf{c}_1'^{\{0:1\}}$ and $\mathbf{c}_2'^{\{0:1\}}$ separately. The individual equalities are still sensitive, which was already noted by the authors of the masked Saber implementation [VBDK+20]. Luckily, the method permits a simple fix, by performing the test atomically for both ciphertext components. We take the same approach as [VBDK+20] and check whether

$$\mathcal{H}(\mathbf{c}_1 \oplus \mathbf{c}_1'^0 \parallel \mathbf{c}_2 \oplus \mathbf{c}_2'^0) \stackrel{?}{=} \mathcal{H}(\mathbf{c}_1'^1 \parallel \mathbf{c}_2'^1) \ . \tag{16}$$

By performing the hash atomically on the concatenation of both ciphertext components, the leakage present in [OSPG18] can be prevented [BDH+21].

2.4 Comparing Masking for Kyber vs Saber

In Figures 2 and 3, it can be seen that Kyber and Saber have highly similar masked architectures. The difference between MLWE and MLWR is apparent, in the extra XOF, Ψ_{η_2}, and B2A$_q$ calls required to sample the error terms $\mathbf{e}_1$, and e_2 for Kyber. B2A$_q$ has roughly twice the complexity of B2A, essentially because `SecAdd`$_q$ makes two calls to `SecAdd`. Kyber further needs an additional B2A$_q$ conversion to convert the Boolean masking $\mathbf{m}^{\{0:1\}}$ back to an arithmetic sharing mod q. This operation is ‘free’ for Saber, since the required share-wise left-shift $2^{\epsilon_p - 1} \cdot \mathbf{m}^{\{0:1\}}$ already has the added effect of converting to an arithmetic sharing mod p implicitly.

Using our new `MaskedCompress`$_q$ algorithm, masked ciphertext compression is remarkably similar for Saber and Kyber. For both, the involved non-linear operation is

[7] The XOR becomes a subtraction if $\mathbf{c}'^{\{0:1\}}$ is arithmetically shared as in [OSPG18].

a power-of-two A2B conversion, where only the high-order bits of the resulting Boolean masking must be kept. However, Saber only requires a 13-bit A2B conversion, whereas Kyber-1024 requires a 24-bit conversion. Moreover, the conversion width for Kyber grows (logarithmically) with the number of shares.

Specialized hardware could be used to favor masking methods for either Saber or Kyber. In this work, our aim is to be generic and support masking for both schemes with identical hardware. Therefore, in Section 4.3, we implement a 32-bit Kogge-Stone `SecAdd` that supports A2B conversion for both Saber and Kyber. Especially Saber could benefit from a smaller and faster adder, or from A2B/B2A algorithms specialized for power-of-two moduli [BCZ18]. In Section 4.2, we describe a generic hardware architecture for masked binomial sampling. This architecture could in turn be optimized for Kyber, which uses smaller η than Saber.

3 HW Accelerators for Linear Operations

Ring arithmetic is one of the major performance bottlenecks of ideal lattice-based cryptography. In particular, the polynomial multiplication is frequently the optimization target for performance improvements. As already mentioned, the acceleration of ring operations is even more important for a masked design as these operations are usually performed on each share individually. Therefore, in this section, a generic hardware architecture for fast polynomial arithmetic based on the NTT is proposed.

3.1 Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is an efficient method to reduce the complexity of the polynomial multiplication from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log_2(n))$. It is a variant of the Fast Fourier Transform (FFT) with operations in the field $\mathbb{Z}_q$ instead of the complex numbers.

Let $a, s \in \mathbb{Z}_q/\phi(x)$ be two ring polynomials of degree $n-1$. Then the polynomial multiplication using the forward and inverse NTT can be computed with $c = INVNTT(NTT(a) \odot NTT(s))$, where $\odot$ denotes the coefficient-wise multiplication.

In lattice-based cryptography, the product of a polynomial multiplication of length $2n$ is usually reduced by the cyclotomic polynomial $\phi(x)$ (frequently $x^n - 1$ or $x^n + 1$). The polynomial reduction by $x^n - 1$ is also referred to as positive wrapped convolution and the reduction by $x^n + 1$ as negative wrapped convolution. Let $\omega_n \in \mathbb{Z}_q$ be the n-th root of unity with $\omega_n^n = 1 \mod q$ and $\omega_n^i \neq 1 \mod q$ for $\forall i \in [0, n-1]$. The forward transform of the coefficients a_i and the inverse transform of $\hat{a}_i$ are computed with

$$\hat{a}_i = \sum_{j=0}^{n-1} \gamma^j \cdot \omega_n^{ij} \cdot a_j \ , \qquad a_i = \frac{1}{n} \cdot \gamma^{-i} \sum_{j=0}^{n-1} \omega_n^{-ij} \cdot \hat{a}_j \ , \tag{17}$$

where γ is the $2n$-th root of unity γ_n for negative wrapped convolutions and $\gamma = 1$ for positive wrapped convolutions. With pre- and postprocessing using the powers of γ a length-$2n$ NTT with zero-padding can be avoided and a length-n NTT is sufficient.

3.2 Design Rationale - Number Theoretic Transform (NTT)

Table 1 summarizes polynomial arithmetic parameters used in several lattice-based algorithms. While some schemes already use parameters suitable for the NTT, others choose a prime not suitable for a direct application of the NTT.

Table 1: NTT parameters of several lattice-based algorithms.

Scheme	n	q	$\phi(x)$	**NTT-based**	$\lceil log_2(q') \rceil$
NewHope-512	512	12289	x^n+1	yes	14
NewHope-1024	1024	12289	x^n+1	yes	14
Kyber	256	3329	x^n+1	yes	12
Dilithium	256	8380417	x^n+1	yes	23
Falcon-512	512	12289	x^n+1	yes	14
Falcon-1024	1024	12289	x^n+1	yes	14
Saber	256	8192	x^n+1	no	34
ntruhps2048509	509	2048	x^n-1	no	31
ntruhps2048677	677	2048	x^n-1	no	32
ntruhps4096821	821	4096	x^n-1	no	34
ntruhrss701	701	8192	x^n-1	no	36
LAC-128	512	251	x^n+1	no	25
LAC-192/256	1024	251	x^n+1	no	26

NTT with prime lift. The original prime q can be lifted to any 'NTT friendly' prime $q' > n \cdot q^2$ for an NTT-based polynomial multiplication. The intermediate values and result of the polynomial multiplication have coefficients not larger than $n \cdot q^2$. If q' is set sufficiently large, precision errors caused by the modular arithmetic are avoided [PNPM15]. After polynomial multiplication with the NTT, the coefficients can be reduced by the original prime q. Using signed arithmetic, the maximum absolute value of the coefficients during the computation is $n \cdot q^2/4$ when the coefficients are represented in $[-q/2, q/2)$. Some schemes always multiply large polynomials with small polynomials sampled from the error distribution, allowing to further decrease the value of q'. However, for schemes as NTRU, large polynomials with coefficients in $[0, q)$ are multiplied with polynomials having the same coefficient range. As in this work all schemes of Table 1 shall be supported by the same hardware architecture and unsigned arithmetic is more suitable for hardware circuits, the rule $q' > n \cdot q^2$ is applied.

All NTT-based schemes of Table 1 have primes smaller than 23 bits. To cover all ranges, in this work, we develop a flexible Montgomery multiplier for any prime up to 24 bits. For algorithms that are not NTT-based, a lifted prime q' has to be found that covers the remaining algorithms. To allow an easy reduction, the Solinas prime $q' = 2^{39} - 2^{12} + 1 = 549755809793$ is chosen. For this prime the condition $q' \equiv 1 \mod 2n$ (the prime has the form $q' = 2^k p + 1$) holds and the n-th as well as the $2n$-th root of unity exists (e.g., for $n \in [256, 512, 1024, 2048]$).

Positive and negative wrapped convolution. Choosing $\gamma = 1$ or $\gamma = \gamma_n = \sqrt{\omega_n}$ with $\omega_n^n = 1 \mod q$, $\omega_n^{n-1} = \gamma_n^n = -1 \mod q$, and $n = 2^k$ leads to positive and negative wrapped convolutions for NTT-based schemes, respectively. Lifting to a higher prime only works if no reduction errors are introduced during the convolution. Negacyclic convolutions involve negative intermediate results that lead to an erroneous output when reduced by q'. These reductions can be avoided using signed arithmetic. For unsigned arithmetic, polynomial multiplications with polynomials of length $n' = 2n$, zero-padding, and consecutive polynomial reduction by $\phi(x)$ can be used. Positive wrapped convolutions can still be realized with an NTT of length $n' = n$.

Incomplete NTT. The prime q is usually chosen such that $\phi(x)$ can be factored into $n = 2^k$ linear terms $\phi(x) = \prod_{i=0}^{n-1} \phi_i(x) \mod q$. This allows the full application of the NTT and the basecase multiplication of two transformed polynomials corresponds to a

simple coefficient-wise multiplication. The concept of the incomplete NTT for lattice-based cryptography was first proposed in [LS19] and a similar concept was later adopted to the second round Kyber specification. Kyber reduced its prime value (consequently key and ciphertext sizes) and chose a value where the n-th root of unity exists but not the $2n$-th root of unity. This prevents applying a full NTT and only $l-1$ layers of the NTT are applied resulting into $n/2$ polynomials of degree two. More precisely, the cyclic polynomial is factored to $\phi(x) = x^n + 1 = \prod_{i=0}^{n/2-1}(x^2 - \omega_n^{2i+1}) = \prod_{i=0}^{n/2-1}(x^2 - \omega_n^{2br(i)+1})$ with br denoting the bit reversal function.

NTT algorithms. When exploiting symmetry, periodicity, and scale properties of the Fourier transformation, the complexity of Equation 17 can be reduced with an divide-and-conquer approach from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log_2(n))$. The two most common methods for splitting a large Fourier transform into smaller pieces are the Cooley-Tukey (CT) [CT65] and the Gentleman-Sande (GS) [GS66] algorithms. The butterfly operation, which is the main operation of these algorithms, consists of simple arithmetic in $\mathbb{Z}_q$. The Cooley-Tukey decimation-in-time (DIT) approach computes $x' \leftarrow x + y \cdot \omega$ and $y' \leftarrow x - y \cdot \omega$ with $\omega, x, y \in \mathbb{Z}_q$ and ω usually a power of ω_n (also known as Twiddle factor). The Gentleman-Sande decimation-in-frequency (DIF) approach computes $x' \leftarrow x + y$ and $y' \leftarrow (x - y) \cdot \omega$.

Different in-place variants of the Cooley-Tukey and Gentleman-Sande algorithms exist, denoted as $NTT^{CT}_{br\to no}$, $NTT^{CT}_{no\to br}$, $NTT^{GS}_{br\to no}$, and $NTT^{GS}_{no\to br}$, where, e.g., $no \to br$ indicates that the input is in normal and the output in bit-reversed order. The bit-reversal can be completely avoided with a combination of the different variants $NTT^{CT}_{no\to br}$ and $INVNTT^{GS}_{br\to no}$ [POG15].

Likewise to previous works, we use different algorithms for the forward and inverse NTT to avoid the bit-reversal step, although the bit-reversal operation is simple in hardware. Using a DIT algorithm for the forward transform and a DIF algorithm for the inverse transform has the further advantage that the multiplications by the powers of γ_n can be integrated into precomputed tables for the Twiddle factors.

Algorithms 14 and 15 illustrate the operations for our flexible NTT. Starting with the original NTT/INVNTT algorithms, we modify the algorithms to support an early abort for an incomplete NTT, as required by Kyber. The incomplete NTT can be activated using the `EARLY_ABORT` signal. Moreover, we integrated support for either positive or negative wrapped convolutions. The wrapping method can be switched using the `NEGACYCLIC` signal. Thus, all schemes of Table 1 can use the same algorithms. Note that the INVNTT requires a final scaling by n^{-1}. For NTT-based schemes, the Twiddle table is stored in Montgomery domain in order to make use of a flexible Montgomery multiplier. In negacyclic NTT-based schemes, the Twiddle table contains n ($n/2$ at early aborts) merged values for the powers of ω_n and γ_n in bit-reversed order and the same amount of precomputed values for the inverse transform. For schemes with positive wraparound or schemes not based on NTT, n precomputed values of the powers of ω_n are stored in the Twiddle table.

3.3 Architecture - Number Theoretic Transform (NTT)

Designing an efficient and flexible NTT with support of all mentioned lattice-based schemes requires new design approaches and multiple components. Figure 6 illustrates the hardware architecture of our proposed NTT unit. It consists of seven different main modules: two RAM blocks (*NTT* and *Twiddle RAM*), four address units (*NTT*, *INVNTT*, *Point*, and *Wrap*), and a *Modular Arithmetic Unit*.

Algorithm 14: NTT transform

```
Input: a ∈ Z_q/φ(x), twiddle_table,
       EARLY_ABORT, NEGACYCLIC
Result: â ∈ Z_q/φ(x)
1  if EARLY_ABORT then
2  |  stop ← n/2
3  else
4  |  stop ← n
5  end
6  t ← n
7  for m = 1 to stop − 1 by m = 2m do
8  |  t ← t/2
9  |  for i = 0 to m − 1 by 1 do
10 |  |  j1 ← 2it,  j2 ← j1 + t
11 |  |  if NEGACYCLIC then
12 |  |  |  m′ ← m
13 |  |  else
14 |  |  |  m′ = 0
15 |  |  end
16 |  |  ω ← twiddle_table[m′ + i]
17 |  |  for j = j1 to j2 − 1 by 1 do
18 |  |  |  z1 ← a_{j+t} · ω  mod q
19 |  |  |  a_j ← a_j + z1  mod q
20 |  |  |  a_{j+t} ← a_j − z1  mod q
21 |  |  end
22 |  end
23 end
```

Algorithm 15: INVNTT transform

```
Input: â ∈ Z_q/φ(x), invtwiddle_table,
       EARLY_ABORT, NEGACYCLIC
Result: a ∈ Z_q/φ(x)
1  if EARLY_ABORT then
2  |  m ← n/2,  t ← 2,  k ← 0
3  else
4  |  m ← n,  t ← 1
5  end
6  for m to m > 1 by m = m/2 do
7  |  j1 ← 0
8  |  if NEGACYCLIC then
9  |  |  m′ ← m/2
10 |  else
11 |  |  m′ = 0
12 |  end
13 |  for i = 0 to m/2 − 1 by 1 do
14 |  |  j2 ← j1 + t
15 |  |  if EARLY_ABORT then
16 |  |  |  ω ← invtwiddle_table[k++]
17 |  |  else
18 |  |  |  ω ← invtwiddle_table[m′ + i]
19 |  |  end
20 |  |  for j = j1 to j2 − 1 by 1 do
21 |  |  |  z1 ← a_j,  z2 ← a_{j+t}
22 |  |  |  a_j ← z1 + z2  mod q
23 |  |  |  a_{j+t} ← (z1 − z2) · ω  mod q
24 |  |  end
25 |  |  j1 ← j1 + 2t
26 |  end
27 |  t ← 2t
28 end
```

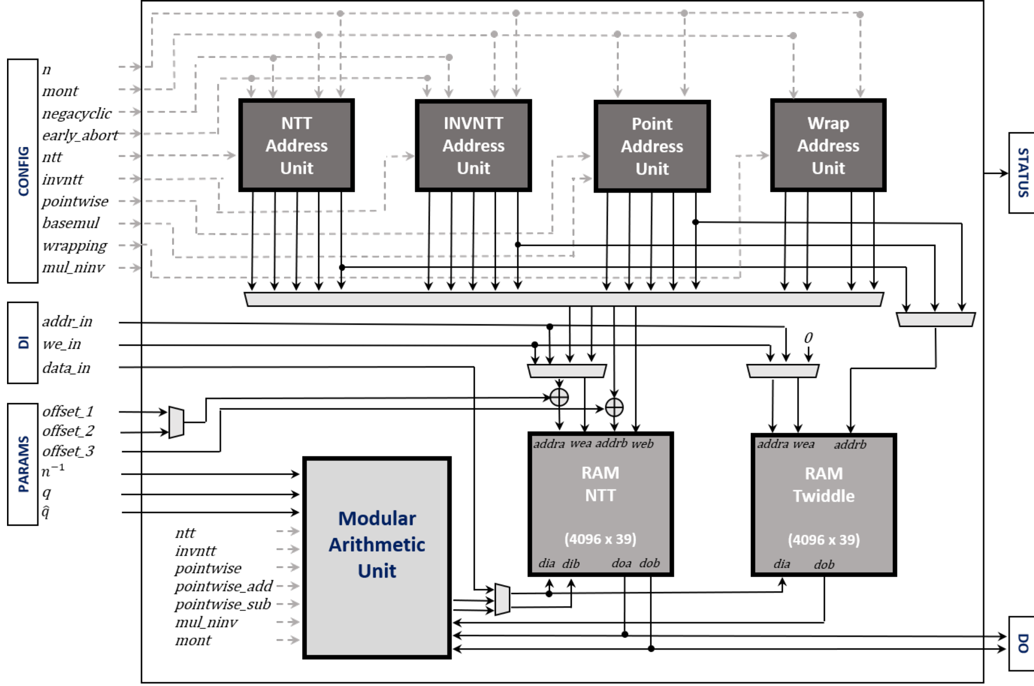


Figure 6: Loosely coupled *NTT Unit*.

NTT and Twiddle RAM. The two memory blocks are used for storing the input/output coefficients and the precomputed Twiddle table, respectively. The size of these memory blocks is chosen large enough to support all parameter sets. To increase the efficiency, the dual-port capabilities of the RAM blocks are exploited. The PKE/KEM schemes usually have coefficients of less than 16 bits. As the deployed system is based on a 32-bit architecture, the input `data_in` (and the output) can contain two coefficients in one word.

NTT/INVNTT Address Unit. The *NTT/INVNTT Address Unit* generates the two read and write addresses to load and store two coefficients as well as the read address for the Twiddle factor according to Algorithms 14 and 15. The signals `ntt` and `invntt` trigger the corresponding address computations. Optionally, `early_abort` and `negacyclic` can be set. The signal `mont` is used to select the number of pipeline stages to delay the write signals according to the delay in the modular arithmetic unit.

Point Address Unit. It computes the addresses for pointwise multiplications, additions, and subtractions. The signal `basemul` is used to select basecase multiplications at schemes with early abort. Let $f, g \in \mathbb{Z}_q/\phi(x)$ and let $\mathrm{NTT}(f) \circ \mathrm{NTT}(g) = \hat{f} \circ \hat{g} = \hat{h}$ denote the basecase multiplication with $n/2$ products. These products are computed with

$$\hat{h}_{2i} + \hat{h}_{2i+1}x = (\hat{f}_{2i} + \hat{f}_{2i+1}x)(\hat{g}_{2i} + \hat{g}_{2i+1}x) \mod x^2 - \omega_n^{2br(i)+1} \ . \tag{18}$$

To ideally exploit the NTT hardware architecture, we split the basecase computation into four parts according to Algorithm 16. Each multiplication and addition step can be carried out in $n/4$ cycles (plus pipeline slack), whereas the address is incremented always by four.

Wrap Address Unit. This address unit is used for schemes not based on NTT to reduce the length-n' polynomial product by $\phi(x) = x^n + 1$. At this negative wrapping, the lower part of the polynomial is subtracted by the higher part.

Algorithm 16: Basecase multiplication (incomplete NTT)

Input: $\hat{f}, \hat{g} \in \mathbb{Z}_q/\phi(x)$, *twiddle_table*, *invtwiddle_table*
Result: $\hat{h} = \hat{f} \circ \hat{g} \in \mathbb{Z}_q/\phi(x)$

1 **for** $i = 0$ **to** $n/4$ **by** 4 **do**
2 $\quad \omega \leftarrow twiddle_table[n/4 + i]$
3 $\quad \hat{h}_{4i} \leftarrow \hat{f}_{4i+1} \cdot \hat{g}_{4i+1} \cdot \omega + \hat{f}_{4i} \cdot \hat{g}_{4i} \mod q$
4 $\quad \hat{h}_{4i+1} \leftarrow \hat{f}_{4i} \cdot \hat{g}_{4i+1} + \hat{f}_{4i+1} \cdot \hat{g}_{4i} \mod q$
5 $\quad \omega \leftarrow invtwiddle_table[n/4 - i - 1]$
6 $\quad \hat{h}_{4i+2} \leftarrow \hat{f}_{4i+3} \cdot \hat{g}_{4i+3} \cdot \omega + \hat{f}_{4i+2} \cdot \hat{g}_{4i+2} \mod q$
7 $\quad \hat{h}_{4i+3} \leftarrow \hat{f}_{4i+2} \cdot \hat{g}_{4i+3} + \hat{f}_{4i+3} \cdot \hat{g}_{4i+2} \mod q$
8 **end**

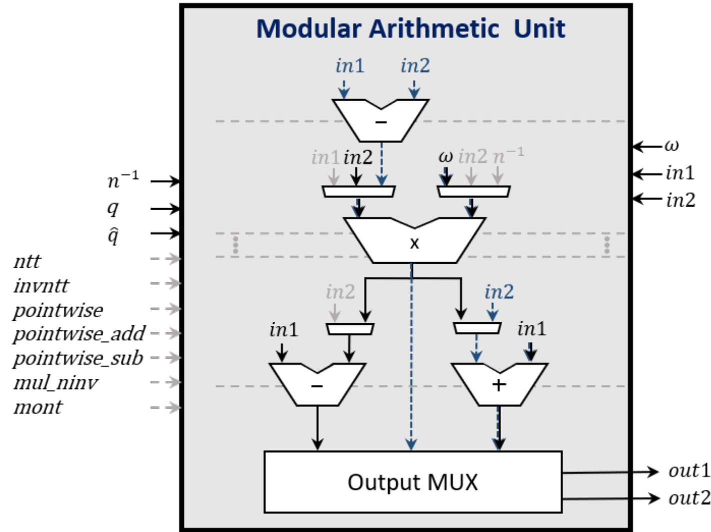


Figure 7: *Modular Arithmetic Unit.* Black: decimation in time, blue dashed: decimation in frequency, grey: pipeline stages and other functionalities.

Modular Arithmetic Unit. It performs the butterfly operation of Algorithm 14 (Lines 18-20) and Algorithm 15 (Lines 21-23). Figure 7 illustrates the architecture of this unit. Its main components are a generic modular multiplier, a modular adder, and two modular subtractors. The signals `ntt`, `invntt`, `pointwise`, `pointwise_add`, `pointwise_sub`, `mul_ninv` are used to configure the multiplexers to either perform DIT or DIF butterfly operations, pointwise multiplications ($out1 = in1 \cdot in2 \mod q$), pointwise additions ($out1 = in1 + in2 \mod q$), pointwise subtractions ($out1 = in1 - in2 \mod q$) or multiplications by constants ($out1 = in1 \cdot n^{-1} \mod q$).

Generic Modular Multiplier. As stated previously, our proposed generic modular multiplier architecture supports Montgomery modular multiplications up to 24 bits. For multiplications with lifted primes at 'NTT unfriendly' schemes, it also supports modular multiplication using a reduction-friendly Solinas prime ($2^{39} - 2^{12} + 1$). While designing this dual multiplier, our objective has been to ensure that the architecture provides high operating frequency with pipelining support. Moreover, costly resources like FPGA DSP blocks are shared between the two multiplication modes.

The architecture of the proposed dual multiplier is shown in Figure 8 and the corresponding operational description in Algorithm 17. The multiplier takes a and b as input multiplicands. The design also requires the Montgomery modulus M and $M' = -M^{-1} \mod R$, where $R = 2^{24}$ was chosen. The control input `mont` determines whether a Montgomery multiplication or multiplication modulo Solinas prime is executed. In Figure 8, the modules with color blue are shared between both multiplication modes, modules with color dark grey are dedicated modules for multiplications modulo Solinas prime, and modules with color light grey are dedicated modules for Montgomery multiplications. As we can see, the DSP blocks and a few multiplexers are part of the shared resources, whereas the dedicated modules contain mainly adders and subtractors. The adders and subtractors are implemented by efficient usage of fast carry chains [KG16]. The reduction logic for multiplications modulo Solinas prime is implemented using the target prime structure and involves only two additions and three subtraction operations. To allow a high operating frequency, the Montgomery and Solinas multiplier have pipeline registers included (12 and 6 stages respectively). For simplicity, the pipelining registers are not shown in Figure 8.

3.4 Results - Number Theoretic Transform (NTT)

All resource utilization and frequency results of this work are extracted after place and route phase using Xilinx Vivado. The chosen platform of this work is the NewAE Technology Target Board CW305 equipped with an Artix 7 FPGA XC7A100T. Table 2 compares flexible NTT designs of previous works with our design. However, to the best of our knowledge, none of the previous works provides a similar level of flexibility. Our design supports the following features: 1) configurable on runtime; 2) the highest parameter range covering all mentioned lattice-based algorithms (n up to 4096, q up to 39 bits); 3) positive and negative convolutions; 4) early abort; 5) pointwise multiplications, additions, and subtractions.

The amount of clock cycles of our NTT architecture is $2n \cdot log(n)$ plus 14 or 8 cycles latency depending on whether Montgomery or Solinas prime reductions are performed. Previous works, such as [FS19, FSS20], take advantage of the small coefficient size of some schemes and pack two coefficients in one memory line. As we also want to support large coefficient widths, we decided to not store two coefficients in one word and also to not compute two parallel butterfly computations. The cycle count can be further reduced by using multiple data RAM blocks (e.g. 8 in [BUC19]) to reduce the memory access bottleneck. This allows to load and process multiple coefficients in parallel. As shown in [MKÖ+20], this can significantly reduce the cycle count. However, using multiple RAM blocks and butterfly units gets extremely expensive in terms of area and also increases

Algorithm 17: Dual multiplier operational description

```
   Input: a, b, mont, P = 2^39 − 2^12 + 1, Montgomery constants: M, M′, R = 2^24
   Result: If mont: c = a × b × R^−1 mod M    else: c = a × b mod P
 1 if (mont) then
       /* |a| = |b| = |M| = |M′| = 24 */
 2     p0 = a[23 : 0] × b[16 : 0]          // FPGA DSP block
 3     p1 = a[23 : 0] × b[23 : 17]         // FPGA DSP block
 4     t0 = (p0 + p1 << 17) mod R          // Partial product adder mont (mod R) block (Fig. 8)
 5     p2 = t0 × M′[16 : 0]                // FPGA DSP blocks
 6     p3 = t0 × M′[23 : 17]               // FPGA DSP block
 7     t1 = (p2 + p3 << 17)                // Partial product adder mont block (Fig. 8)
 8     p4 = t1 × M[16 : 0]                 // FPGA DSP block
 9     p5 = t1 × M[23 : 17]                // FPGA DSP block
10     t2 = (p4 + p5 << 17)                // Paritial product adder mont block (Fig. 8)
11     c0 = (t0 + t2) >> R                 // Adder and shifter module (Fig. 8)
12     c1 = (c0 − M)                       // Subtractor module (Fig. 8)
13     if (c1 < 0) then
14         c = c0
15     end
16     else
17         c = c1
18     end
19 end
20 else
       /* |a| = |b| = 39 */
21     p0 = a[23 : 0] × b[16 : 0]          // FPGA DSP block
22     p1 = a[23 : 0] × b[33 : 17]         // FPGA DSP block
23     p2 = a[23 : 0] × b[38 : 34]         // FPGA DSP block
24     p3 = a[38 : 24] × b[16 : 0]         // FPGA DSP block
25     p4 = a[38 : 24] × b[33 : 17]        // FPGA DSP block
26     p5 = a[38 : 24] × b[38 : 34]        // FPGA DSP block
27     s0 = p0 + p1 << 17 + p2 << 34       // Partial product adder Solinas block (Fig. 8)
28     s1 = p3 + p4 << 17 + p5 << 34       // Partial product adder Solinas block (Fig. 8)
29     c′0 = s0 + s1 << 24                 // Final product generator block (Fig. 8)
30     c = c′0 mod P                       // Reduction logic block (Fig. 8)
31 end
```

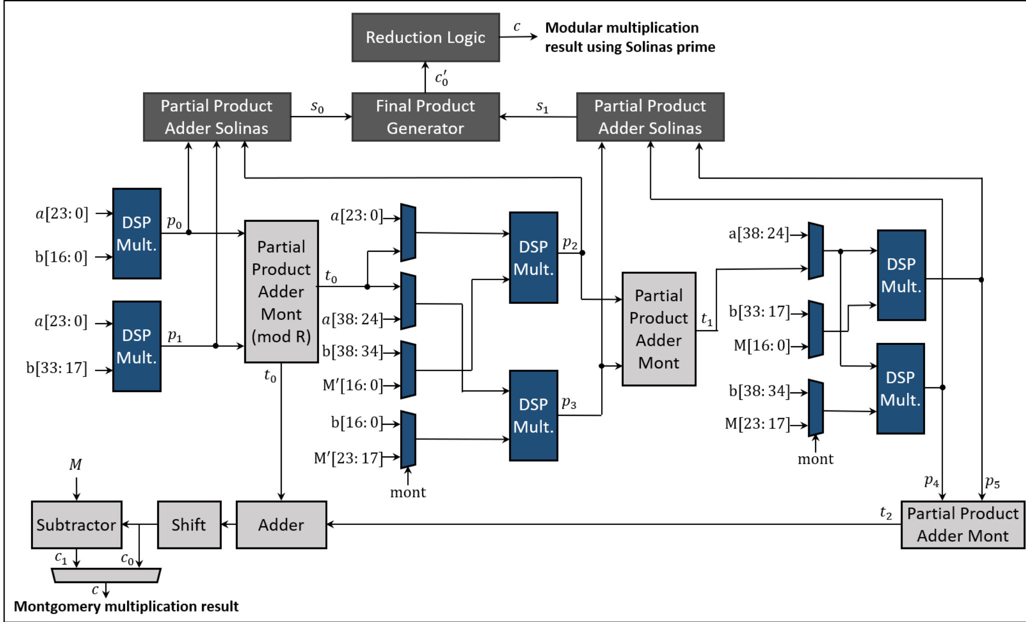


Figure 8: Architecture of dual multiplier supporting up to 24-bit Montgomery multiplications and 39-bit modular multiplications using Solinas prime $P = 2^{39} - 2^{12} + 1$.

the design complexity. Such extreme high-speed implementations are rather suitable for customized standalone co-processors of complete schemes. With a codesign, the extremely low latencies would not have such a strong influence on the overall performance. Therefore, our codesign focuses more on a very high flexibility and a good compromise between area and performance.

4 HW Accelerators for Non-Linear Operations

In this section, we describe hardware architectures for the non-linear operations of Kyber and Saber. These operations need to combine information from both shares and therefore require special treatment in a masked design.

4.1 Masking Keccak

Most lattice-based NIST schemes use the Keccak functions SHA3 and SHAKE to create hash outputs and pseudo-random numbers. Keccak hardware implementations have a particularly good energy efficiency for random number generation because Keccak generates a high amount of bits per round [BUC19]. The core operation of the Keccak algorithms is the Keccak state permutation function `f-1600`. One round of this function can be split into the following operations: *Theta* (θ), *Rho* (ρ), *Pi* (π), *Chi* (χ), and *Iota* (ι). While *Theta*, *Rho*, *Pi*, and *Iota* are linear functions consisting of *XOR* and rotation operations, the function *Chi* is a non-linear operation, which additionally requires *AND* as well as *NOT* operations. The linear functions can be performed on the shares individually as illustrated in Figure 9. Therefore, we discuss in the following only the non-linear function *Chi* in more detail.

***Chi* operation (χ).** The Keccak state can be represented as a three-dimensional array $A[x, y, z]$ with the coordinates $x \in [0, 4]$, $y \in [0, 4]$, and $z \in [0, 63]$. Let $A[x, y]$ determine

Table 2: Resource and performance overview for loosely coupled NTT.

Design	Device	LUTs	FFs	Slices	DSP	BRAM	Max. Freq.	NTT Cycles
[FS19]	FPGA	980	395	–	26	2	–	$n = 256$: 1,800 $n = 512$: 4,616 $n = 1024$: 10,248
[BUC19]	ASIC (40 nm)	–	–	–	–	–	72 MHz	$n = 256$: 1,289 $n = 512$: 2,826 $n = 1024$: 6,155
[MKÖ+20]	FPGA	7,400 8,100 16,000 22,000	5,000 5,200 14,000 17,000	– – – –	24 24 56 248	24 24 24 96	147 MHz 141 MHz 125 MHz 125 MHz	$n = 256$: 160 $n = 512$: 345 $n = 1024$: 490 $n = 4096$: 3,276
[FSS20]	FPGA / ASIC (65 nm)	2,908/–	170[a]/–	–	9/–	0/–	45 MHz	$n = 256$: 1,935 $n = 512$: 8,169[b] $n = 1024$: 18,537[b]
This work	FPGA	2,454	1,917	774	7	4.5	153 MHz	$n = 256$: 3,584(+14/8) $n = 512$: 8,192(+14/8) $n = 1024$: 20,480(+14/8) $n = 2048$: 45,056(+14/8) $n = 4096$: 98,304(+14/8)

[a] Does not include resources of register file.
[b] Does not include time for bit reversal.

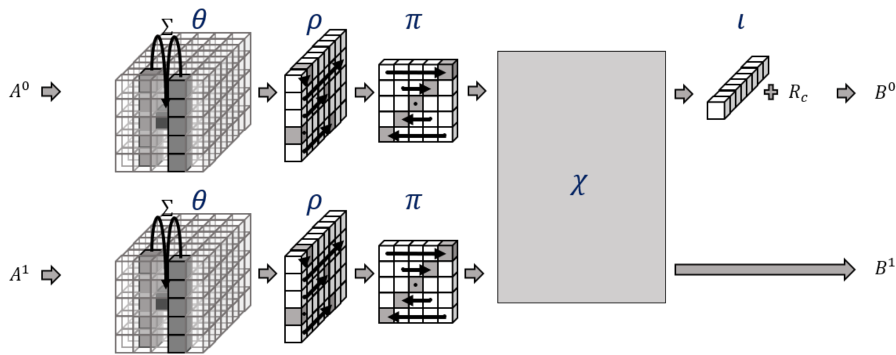


Figure 9: Masking Keccak.

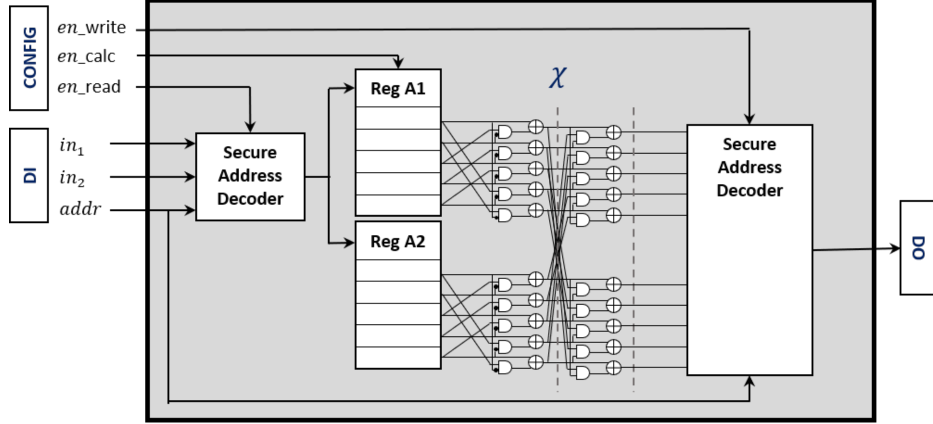


Figure 10: Masked Chi accelerator. Dotted lines illustrate register stages.

an input lane (a 64 bit word) and $B[x, y]$ the output lane of a specific operation. The χ operation is defined by $B[x, y] = A[x, y] \oplus ((A[x+1, y] + 1) \wedge A[x+2, y])$ for x and y in $[0, 4]$. As proposed in [BDPVA10], the output shares $B^0[x, y]$ and $B^1[x, y]$ of the masked input shares $A^0[x, y]$ and $A^1[x, y]$ can be computed with

$$B^0[x, y] \leftarrow A^0[x, y] \oplus (A^0[x+1, y] + 1)A^0[x+2, y] \oplus A^0[x+1, y]A^1[x+2, y] \text{ and } \tag{19}$$

$$B^1[x, y] \leftarrow A^1[x, y] \oplus (A^1[x+1, y] + 1)A^1[x+2, y] \oplus A^1[x+1, y]A^0[x+2, y] \ . \tag{20}$$

If the operations in Equations 19 and 20 are executed from left to right, the authors in [BDPVA10] argue that all intermediate computations are independent of native variables. Instead of using fresh randomness, different parts of the state are reused to form independent computations.

To accelerate the computations in Equations 19 and 20, we developed the hardware design of Figure 10. The accelerator consists of three steps. In the first step, for each share, five 32-bit lanes of a fixed y coordinate are loaded via a secure address decoder into two separated register files. Depending on the address value, the input in_1 and in_2 are either stored in the registers *Reg A1* or *Reg A2*. In the second step, the Chi operation is computed. Therefore, Equation 19 is split into two parts: $\hat{B}^0[x, y] = A^0[x, y] + (A^0[x+1, y]+1)A^0[x+2, y]$ and $B^0[x, y] = \hat{B}^0[x, y] + A^0[x+1, y]A^1[x+2, y]$. Equation 20 is split in the same way. While the first part contains only computations with a single share, the second part includes both shares. However, the critical shares are already blinded by independent state bits. To avoid leakages due to glitch effects, the computations are separated by registers. Finally, the result of the Chi operation is written to the output and the next 32-bit lanes can be loaded. This procedure requires 2×5 repetitions until the whole Chi operation is performed. Loading the complete states into an accelerator would only lead to a small performance improvement as the actual Chi computation of the proposed accelerator requires only two cycles. However, it would significantly increase the area costs.

4.2 Masking Binomial Sampling

Many efficient LWE-based schemes require sampling from a centered binomial distribution. A centered binomial sample can be retrieved by

$$\Psi_\eta = \sum_{i=0}^{\eta-1}(x_i - x'_i) \mod q \ , \tag{21}$$

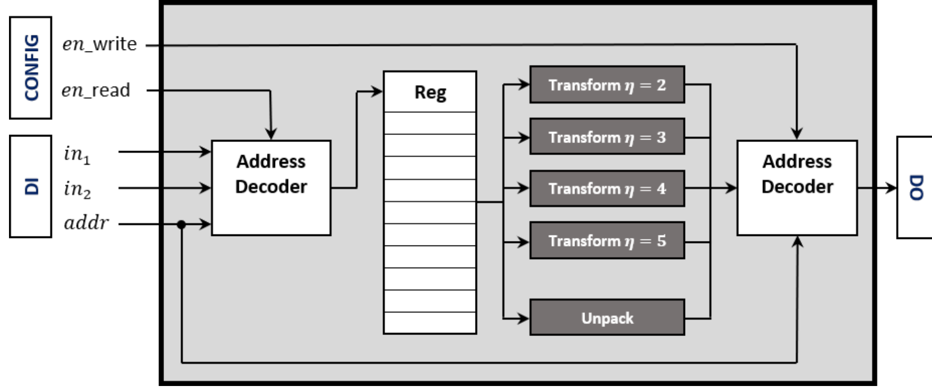


Figure 11: Bit-slicing accelerator.

with x_i and x'_i denoting the bits of uniformly distributed η-bit integers.

The authors in [SPOG19] proposed two different masked sampling methods for software implementations. The first method, which is based on [OSPG18], was specially designed for first-order masked implementations. The second method turns the input into a bit-sliced representation and computes the Hamming weight of x and x' using a secure *AND* function.

In this work, we develop a generic binomial sampling accelerator suitable for various values of η. More specifically, Kyber-512 ($\eta = 2$, $\eta = 3$), Kyber-768 ($\eta = 2$), Kyber-1024 ($\eta = 2$), Lightsaber ($\eta = 5$), Saber ($\eta = 4$), and Firesaber ($\eta = 3$) are supported with the same architecture. Our proposed architecture is based on the second software method presented in [SPOG19]. Two separate accelerators were developed for performing the binomial sampling. This includes a bit-slicing accelerator and a masked adder tree.

Bit-slicing accelerator. The bit-slicing method allows computing multiple samples in parallel. Although still more efficient than non-bit-slicing approaches, the conversion from the Keccak output into bit-sliced format turns out to be relatively costly in software if the sampling is performed according to the specification of, e.g., Kyber or Saber [VBDK+20]. However, in hardware turning the Keccak output into bit-sliced format corresponds to a simple rewiring. Figure 11 shows the top-level architecture of the bit-slicing accelerator. The uniformly distributed Keccak squeeze is stored in up to $2\eta_{max}$ registers within the accelerator with $\eta_{max} = 5$. The transformation in the accelerator is performed according to Algorithm 18 for different values of η. Depending on the value of `addr`, the address decoder at the output selects the desired 32-bit values of $x_i[0:31]$ or $x'_i[0:31]$ with $i \in [0, \eta - 1]$. The bit-slicing accelerator also supports the reverse operation for unpacking bit-sliced values of $x_i[0:31]$ into normal representation.

Algorithm 18: Bit-slicing transform η

Input: Byte array $\boldsymbol{b} = \{b_0, b_1, \ldots, b_{8\eta-1}\} \in \mathbb{B}^{8\eta}$ with $\mathbb{B}$ denoting the set $[0, 255]$
Result: Bit-sliced 32-bit terms $x_i[0:31]$, and $x'_i[0:31]$ with $i \in [0, \eta - 1]$

1 $s \leftarrow$ BytesToBitstream($\boldsymbol{b}$)
2 **for** $i = 0$ **to** 31 **by** 1 **do**
3   **for** $j = 0$ **to** $\eta - 1$ **by** 1 **do**
4         $x_j[i] \leftarrow (s \gg (2\eta \cdot i + j)) \wedge 1$
5         $x'_j[i] \leftarrow (s \gg (2\eta \cdot i + j + \eta)) \wedge 1$
6   **end**
7 **end**

Masked adder tree. After transforming the Keccak squeeze into bit-sliced format, the two sums $\boldsymbol{s}[0:31] = \sum_0^{\eta-1} x_i[0:31]$ and $\boldsymbol{s'}[0:31] = \sum_0^{\eta-1} x'_i[0:31]$ must be computed and subtracted according to Equation 21, where this time i denotes the index of 32-bit variables and not the bit location. Note that these sums compute 32 binomial samples in parallel. These computations can be performed in hardware using the adder tree shown in Figure 12 (a). To simplify the subsequent description of the binomial sampler, the brackets are omitted, e.g., instead of $x_i[0:31]$ we write x_i. The adder tree consists of η stages with η half adders each. The computation of the binomial sampling can be split into two steps. In the first step, the sum $\boldsymbol{s}$ is computed using the input $\boldsymbol{x}$ and $\boldsymbol{z}$, whereas $\boldsymbol{z}$ is initially set to zero. Now, in each stage, the 32-bit values of $\boldsymbol{x}$ are subsequently added to the intermediate sum of the previous stage. The carries of these computations are always forwarded to the next half adder of the same stage and the intermediate sums are forwarded to the next stage. After η stages the circuit outputs the sum $\boldsymbol{s}$. In the second step, the sum of the previous step is assigned to the input $\boldsymbol{z} = \boldsymbol{s}$ and additions with the inverse of $\boldsymbol{x'}$ are performed within the stages. This corresponds to the desired subtraction of $\boldsymbol{s}$ and $\boldsymbol{s'}$.

Masked adder tree based on TI. Let the sum and carry computations in the adder tree be split into two functions $f1 : (x_0, z_0) \rightarrow (s_0)$ with $s_0 = x_0 \oplus z_0$ and $f2 : (c_{i-1}, z_{i-1}, z_i) \rightarrow (c_i, s_i)$ with $c_i = (c_{i-1} \wedge z_{i-1})$ and $s_i = z_i \oplus c_i$ for $i \neq 0$. The direct sharing approach presented in [BNN+12] can be used to construct functions that are in accordance to the TI principles. With the three input shares of $\boldsymbol{x}_i^{\{0:2\}}$, the sum $\boldsymbol{s}_i^{\{0:2\}}$ and carry $\boldsymbol{c}_i^{\{0:2\}}$ can be computed with the following splits:

$$s_0^0 = x_0^0 \oplus z_0^0, \quad s_0^1 = x_0^1 \oplus z_0^1, \quad s_0^2 = x_0^2 \oplus z_0^2 \tag{22}$$

$$c_i^0 = (c_{i-1}^1 \wedge z_{i-1}^1) \oplus (c_{i-1}^1 \wedge z_{i-1}^2) \oplus (c_{i-1}^2 \wedge z_{i-1}^1); \quad s_i^0 = z_i^0 \oplus c_i^0 \tag{23}$$

$$c_i^1 = (c_{i-1}^2 \wedge z_{i-1}^2) \oplus (c_{i-1}^0 \wedge z_{i-1}^2) \oplus (c_{i-1}^2 \wedge z_{i-1}^0); \quad s_i^1 = z_i^1 \oplus c_i^1 \tag{24}$$

$$c_i^2 = (c_{i-1}^0 \wedge z_{i-1}^0) \oplus (c_{i-1}^0 \wedge z_{i-1}^1) \oplus (c_{i-1}^1 \wedge z_{i-1}^0); \quad s_i^2 = z_i^2 \oplus c_i^2 \tag{25}$$

While the linear functions in these equations can always be computed with a single share, for the non-linear functions, at least one share is always missing during the computations (non-completeness property). When converting the proposed adder tree using TI principles and the functions $f1$ and $f2$, the architecture of Figure 12 (b) is obtained. It is not possible to fulfill the uniformity property of a non-linear Boolean operation that has two inputs and one output [NRS11]. Therefore, the uniformity property for each output of the function $f2$ needs to be recovered using fresh randomness. Changing the adder tree to use full adders where three-input operations are used to avoid the refreshing step is theoretically possible. However, such an architecture would lose the flexibility as for each η another circuit would be required. Therefore, another alternative to reduce the randomness requirements is investigated.

Masked adder tree based on DOM. When the uniformity property is preserved, secure TI implementations can be realized with a low amount of randomness. As this is not the case for our adder tree and the generation of fresh randomness is in most platforms expensive, we investigate the behaviour of the adder tree architecture with DOM principles. As shown in Figure 12 (c), the DOM approach significantly reduces the complexity. Instead of three instances of $f1$ and $3 \cdot (\eta_{max} - 1)$ instances of $f2$ in each level only two and $\eta_{max} - 1$ instances are required, respectively, when using the DOM approach. The computation $c_i = (c_{i-1} \wedge z_{i-1})$ in $f2$-*DOM* is realized with the secure *DOM-AND*. For the generation of 32 binomially distributed coefficients, the adder tree based on TI requires $4 \cdot \eta_{max} \cdot (\eta_{max} - 1)$

random 32-bit values plus $(2 \cdot \eta_{max})$ values for randomizing the zero-input of $\boldsymbol{z}$. In contrast, the DOM approach requires $2 \cdot \eta_{max} \cdot (\eta_{max} - 1)$ plus η_{max} random values. For instance, with $\eta_{max} = 5$ the amount of randomness reduces from $90 \times$ 32-bit to $45 \times$ 32-bit.

4.3 Secure Adder

The secure arithmetic addition for masked Boolean shares is an essential element for the generic B2A and A2B conversions. Two secure adder designs based on the ripple-carry adder and a pipelined Kogge-Stone adder were proposed in [SMG15]. The Kogge-Stone adder achieves a lower latency as it belongs to the class of carry-lookahead adders. It splits the carry computation into a generate and propagate part. Due to its good performance, the suggested Kogge-Stone adder suits well to our application and the proposed architecture was adopted for our design. The TI-based Kogge-Stone adder for three shares, shown in Figure 13, is constructed using three stages for performing 4-bit additions. The vertical stages create propagate bits $\boldsymbol{p}_i^{\{0:2\}}$ and generate bits $\boldsymbol{g}_i^{\{0:2\}}$. The first stage, requires the linear function $f1 : (\boldsymbol{x}_i^{\{0:2\}}, \boldsymbol{y}_i^{\{0:2\}}) \rightarrow \boldsymbol{p}_i^{\{0:2\}}$ with

$$p_i^0 = x_i^0 \oplus y_i^0, \quad p_i^1 = x_i^1 \oplus y_i^1, \quad p_i^2 = x_i^2 \oplus y_i^2 \tag{26}$$

and the non-linear function $f2 : (\boldsymbol{x}_i^{\{0:2\}}, \boldsymbol{y}_i^{\{0:2\}}, r_i) \rightarrow \boldsymbol{g}_i^{\{0:2\}}$ with

$$g_i^0 = (x_i^1 \wedge y_i^1) \oplus (x_i^1 \wedge y_i^2) \oplus (x_i^2 \wedge y_i^1) \oplus r_i \ , \tag{27}$$

$$g_i^1 = (x_i^2 \wedge y_i^2) \oplus (x_i^0 \wedge y_i^2) \oplus (x_i^2 \wedge y_i^0) \oplus (x_i^0 \wedge r_i) \oplus (y_i^0 \wedge r_i) \ , \tag{28}$$

$$g_i^2 = (x_i^0 \wedge y_i^0) \oplus (x_i^0 \wedge y_i^1) \oplus (x_i^1 \wedge y_i^0) \oplus (x_i^0 \wedge r_i) \oplus (y_i^0 \wedge r_i) \oplus r_i \ . \tag{29}$$

The remaining stages require $f2$ and $f3 : (\boldsymbol{g}_{i+j}^{\{0:2\}}, \boldsymbol{g}_i^{\{0:2\}}, \boldsymbol{p}_{i+j}^{\{0:2\}}) \rightarrow \boldsymbol{g}_{i+j}^{\{0:2\}}$ with $j = 2^{stage-1}$ and

$$g_{i+j}^0 = (g_i^1 \wedge p_{i+j}^1) \oplus (g_i^1 \wedge p_{i+j}^2) \oplus (g_i^2 \wedge p_{i+j}^1) \oplus g_{i+j}^1 \ , \tag{30}$$

$$g_{i+j}^1 = (g_i^2 \wedge p_{i+j}^2) \oplus (g_i^0 \wedge p_{i+j}^2) \oplus (g_i^2 \wedge p_{i+j}^0) \oplus g_{i+j}^2 \ , \tag{31}$$

$$g_{i+j}^2 = (g_i^0 \wedge p_{i+j}^0) \oplus (g_i^0 \wedge p_{i+j}^1) \oplus (g_i^1 \wedge p_{i+j}^0) \oplus g_{i+j}^0 \ . \tag{32}$$

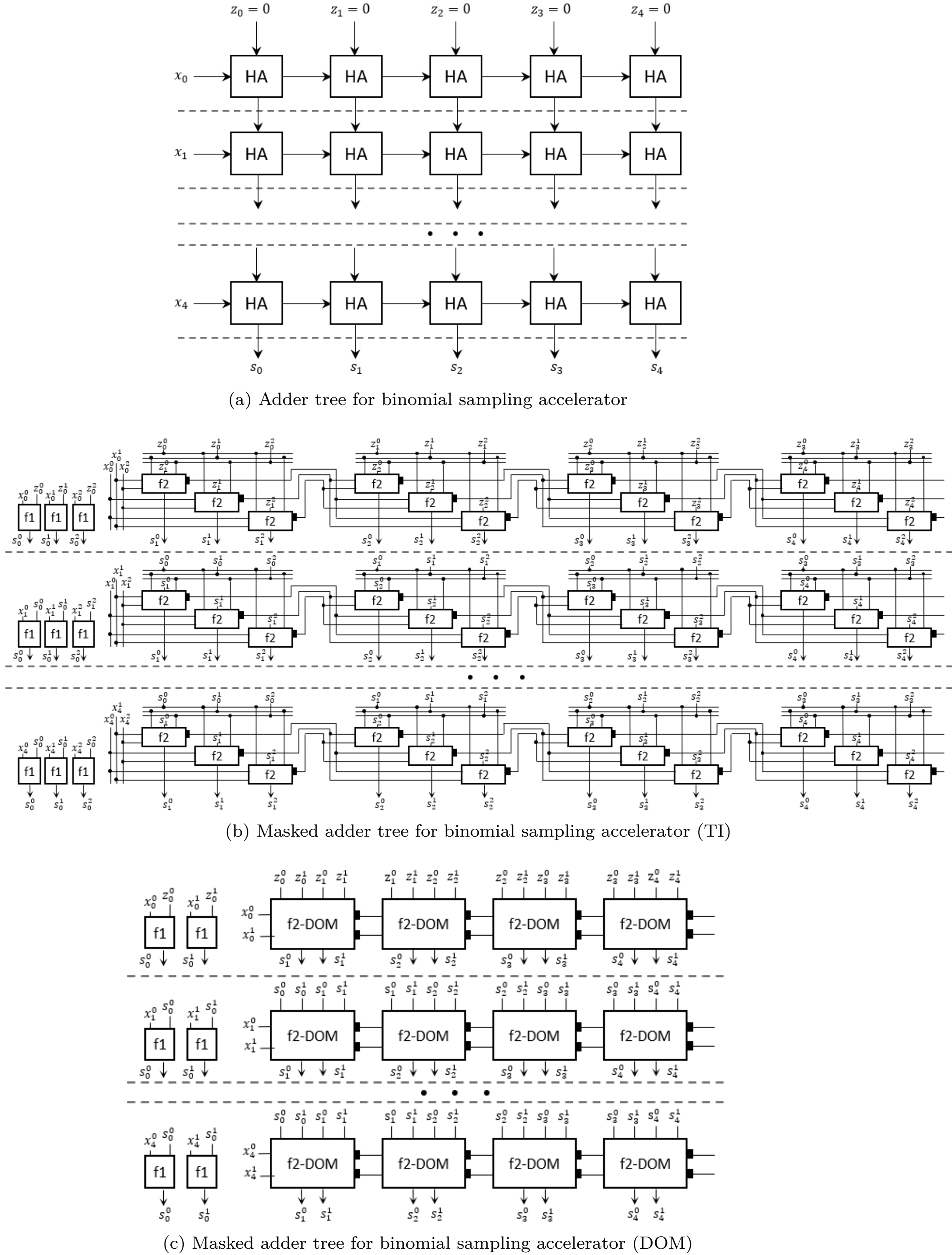
While the first stage requires further randomness for recovering the uniformity property after $f2$, the remaining stages can use the independent bit values of g_i^0 instead of r_i to keep uniformity.

4.4 Results - Non-Linear Accelerators

Table 3 summarizes the resource utilization and performance of our HW accelerators presented in this section evaluated for the Artix 7 FPGA XC7A100T. Critical signals and components that involve non-linear operations were defined with the Verilog `dont_touch` attribute, preventing the synthesis tool from optimizations. Without this attribute, a lower resource utilization and better performance can be expected. Nevertheless, we chose the safe option and accept these drawbacks.

The cycle counts in Table 3 are the latencies within the accelerator. Cycles for loading, storing, and clearing the input/output operands are excluded. From a system perspective, the accelerator latencies are partially hidden by the loading operations. As an example, consider the masked adder trees for the binomial sampling accelerator shown in Figure 12. If the shares associated with x_4 are the last operands to be loaded, all previous stages can already compute their results and only the last adder stage would account for an effective latency of 4 clock cycles.

To the best of our knowledge, no HW/SW codesign of Keccak that supports masked and non-masked operations was published so far. The fully masked HW designs [BDPVA10,



(a) Adder tree for binomial sampling accelerator

(b) Masked adder tree for binomial sampling accelerator (TI)

(c) Masked adder tree for binomial sampling accelerator (DOM)

Figure 12: Adder tree for binomial sampling (*Binom Tree*) with $\eta_{max} = 5$.

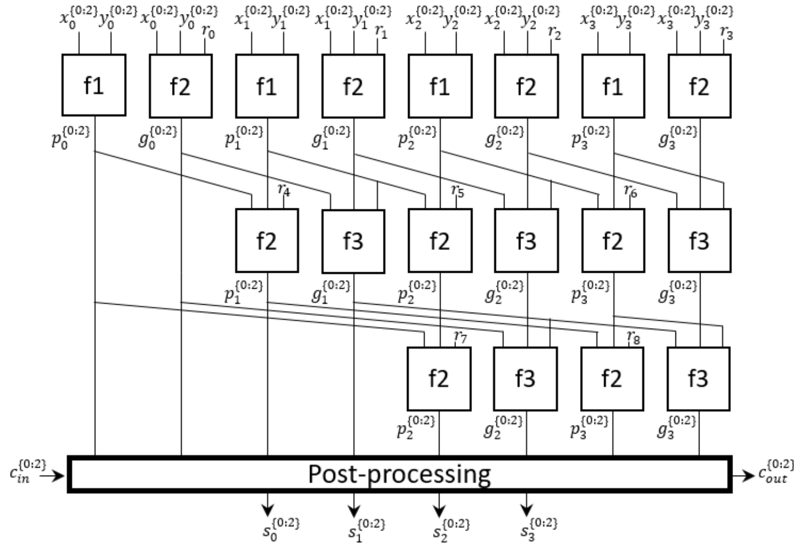


Figure 13: Secure Kogge-Stone adder (`SecAdd`) for 4-bit additions.

Table 3: Resource and performance overview for the non-linear accelerators.

Design	**LUTs**	**FFs**	**Slices**	**DSP**	**BRAM**	**Max. Freq.**	**Cycles**
Keccak (`f-1600`) [FSS20]	3,847	0	–	0	0	–	1/round
Keccak (`f-1600`)	3,100	0	920	0	0	303 MHz	1/round
Masked Chi	1,488	971	632	0	0	370 MHz	2
Bit-slice	277	320	123	0	0	357 MHz	0
Binom Tree (DOM)	5,352	2,570	1,706	0	0	112 MHz	9
Binom Tree (TI)	9,232	4,166	2,839	0	0	140 MHz	9
Secure Adder (TI) [SMG15]	937[a)]	1,330[a)]	–	0	0	62 MHz	6
Secure Adder (TI)	2,464	1,323	1,054	0	0	454 MHz	6 (7)

[a)] Does not contain resources for secure address decoder and no support for an input carry.

GSM17, ABP+18] report only ASIC results in gate equivalents making it difficult to compare with our FPGA results. The tightly-coupled `f-1600` accelerator in [FSS20] only supports non-masked computations. Our `f-1600` accelerator supports complete round computations for non-masked and incomplete round computations (only *Theta*, *Rho*, *Pi*) for masked operations. The Chi accelerator is used to securely accelerate the non-linear operation of Keccak.

The results show that the DOM variant of the binomial sampling accelerator (*Binom Tree*) does not only decrease the amount of randomness that is required but also leads to a significant area reduction when compared to the TI variant. Therefore, in the remainder of this article, only the DOM variant is considered for further measurements.

Compared to [SMG15], our secure adder is very similar. Both are designed for 32-bit operations. The higher resource consumption can be explained by the `dont_touch` attributes, an additional secure address decoder and an additional feature that allows to compute 32-bit additions with and without input carry. While Saber requires only 16-bit and Kyber 24-bit additions (in the compression, see Section 2.2), the computations with input carry can become important for schemes with a larger parameter set or higher-order masking. Note that if the input carry is enabled, our adder takes one additional clock cycle.

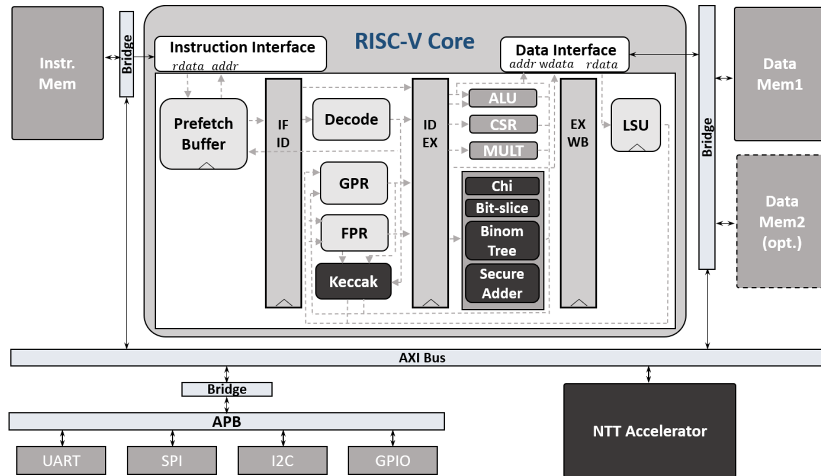


Figure 14: RISC-V system with masked post-quantum accelerators.

5 System Integration

RISC-V is an open Instruction Set Architecture (ISA) based on the Reduced Instruction Set Computer (RISC) principles. Due to its open-source character, RISC-V has meanwhile achieved a wide distribution in academia but also in industry. Several open-sourced RISC-V processor designs were proposed in the last years. One of the most popular processors is the 32-bit solution CV32E40P (formerly RI5CY) from the Parallel Ultra Low Power (PULP) project[8]. The CV32E40P core, originally developed by ETH Zürich and the University of Bologna, is an in-order execution core with four pipeline stages. It supports the complete base integer instruction set (I) and the extensions for compressed instructions (C) as well as multiplication instructions (M). Optionally, the extension for single-precision floating-point instructions (F) can be used. Additionally, the core features some custom ISA extensions such as hardware loops, post-incrementing load and store operations, and bit-manipulation operations in order to optimize the core for low-power signal processing applications.

Without further optimization for post-quantum applications, the processor's performance is significantly lower compared to the performance of the popular ARM Cortex-M4 [FSS20], which is probably the most used embedded evaluation platform for cryptography in academia. Nevertheless, the core is completely written in SystemVerilog and highly suitable for custom extensions and core modifications. This makes the core well suited for the evaluation of our accelerators developed in this project.

Figure 14 shows the architecture of our system. Its main components are a RISC-V processor including our tightly coupled accelerators (Keccak, Chi, Bit-slice, Binom Tree, Secure Adder), a loosely coupled NTT accelerator, an instruction memory, one data memory (optional second data memory), and a set of peripherals (UART, SPI, I2C, GPIO).

In addition to our custom accelerators, the RISC-V core includes the following components: prefetch buffer, instruction decoder, General Purpose Register (GPR), Floating Point Register (FPR), ALU, Control Status Register (CSR), multiplication unit, Load Store Unit (LSU).

[8]https://pulp-platform.org, https://github.com/pulp-platform

5.1 Architectural Leakage Reduction

Storing two shares in the same register file can lead to exploitable leakages, even if both shares are not accessed simultaneously [SR15]. The reason is that the registers can be connected to the same internal bus and combinatorial circuit. Although influencing the performance, in this work, only one share is located within the registers at one time step. Before processing the second share, the first share is cleared.

At the non-linear accelerators in the *EX* stage, the shares are stored always in different register files. Optimizations are turned off by `DONT_TOUCH` attributes. Register values are only accessed via a secure address decoder, which first computes a select signal before accessing one of the register banks. Moreover, addresses are one-hot encoded to avoid problems during address switches.

The pipeline registers between the *ID* and *EX* stage are another typical source of leakage at the transition of operations with another share. This affects three operand registers for the ALU, multiplier unit, and post-quantum accelerators, respectively. These pipeline registers must be cleared after critical operations. Moreover, the serial divider, which is capable of performing divisions and remainder computations, contains pipeline registers that have to be cleared to avoid leakages.

In an FPGA design, the instruction and data memories are constructed using BRAM resources. The main elements of a BRAM are an input register, memory array, output latch, and an optional output register to improve the critical path. Overwriting one of the registers/latches with another share can lead to exploitable leakages [BDGH15]. The routing nets in the memory array have buffers to improve the signal quality. Charging and discharging the nets can thus lead to amplified leakages. To avoid such effects a separated second data memory is placed in the design. It can be optionally used to clearly separate the shares for critical operations. Variables can be relocated using the `section` attribute of the compiler.

5.2 Accelerator Types

In the presented architecture, two different accelerator types are used. While the NTT accelerator is loosely coupled to the processor and connected to an AXI bus, the Keccak, Chi, Bit-slice, Binom Tree, and Secure Adder accelerators are tightly coupled and directly integrated into the RISC-V processor.

Loosely coupled accelerators. Loosely coupled accelerators are ideally suitable for large and computationally intensive tasks with a low communication between core and accelerator. The authors in [FSS20] and [AEL+20] have shown that the NTT is also well suited for tightly coupled accelerators. However, these previous works focused on schemes with 16-bit coefficients. In this work, bit sizes up to 39-bit are supported to cover all main lattice-based schemes. Computing the convolution using the Chinese Remainder Theorem (CRT) turned out to be less efficient [FSS20]. A loosely coupled approach is therefore the preferred solution to clearly separate the 39-bit operations within the NTT and the 32-bit operations of the processor.

The accelerator configuration registers and NTT memory are memory-mapped. Table 4 summarizes the memory map of the platform. The addresses starting at 0x1B10 8000 include: i) the parameters *offset_1*, *offset_2*, *offset_3*, n^{-1}, q, and $\hat{q}$, ii) a configuration register containing the polynomial length n and the configuration signals `mont`, `negacyclic`, `early_abort`, `ntt`, `invntt`, `pointwise`, `basemul`, `wrapping`, `mul_ninv` (see Section 3.3).

Tightly coupled accelerators. Tightly coupled accelerators have the advantage that no costly and complex bus communication is required to access data from the register files. This makes the accelerators ideally suitable for small tasks and operations. Usually, these

Table 4: Memory map RISC-V platform.

Address space	Interface
0x0000 0000 0x0008 0000	Instruction memory
0x0010 0000 0x0018 0000	Data memory
0x1A10 0000 0x1A11 0000	Peripherals (UART, GPIO, I2C, . . .)
0x1B10 0000 0x1B10 7FFF	NTT memory
0x1B10 8000 0x1B10 800A	NTT configuration and status
0x1C10 0000 0x1C18 0000	Data memory for 2nd share

Table 5: RISC-V instruction set extensions with R-type encoding.

Funct7	Funct3	Operation Name	Description
0x08	0	*keccak.f1600*	Keccak round with options: complete/ incomplete round, reset
0x14	0 1 2 3 4 5	*pq.mbinw* *pq.mbinc* *pq.mbincinv* *pq.mbinr* *pq.mbincpy* *pq.mbinrst*	Masked binomial sampler operations for BinomTree (write, compute, compute inverse, read, copy, reset)
0x15	0 1	*pq.slicew* *pq.slicer*	Bit-slice operations (write, read)
0x16	0 1 2	*pq.mchiw* *pq.mchic* *pq.mchir*	Masked Chi operations (write, compute, read)
0x17	0 1 2 3	*pq.maddw* *pq.maddc* *pq.maddcc* *pq.maddr*	Masked secure adder operations (write, compute, compute with carry in, read)

kinds of accelerators are much smaller compared to loosely coupled designs. This does not only decrease the area requirements but usually also leads to a high flexibility. Tightly coupled accelerators can be accessed using ISA extensions. Similar to [FSS20], the Keccak accelerator for the `f-1600` round function is placed in the *ID* stage because this accelerator requires parallel access of 50 registers, which are in the same processor stage. All remaining accelerators require at most three input and one output operand.

Table 5 provides an overview about the ISA extensions developed in this work. All instructions are mapped to the opcode 0x77. The instructions are all single-cycled except of *pq.mbinc*, *pq.mbincinv*, *pq.mchic*, *pq.maddc*, and *pq.maddcc* (see Section 4.4).

The Keccak instruction can be configured to perform complete and incomplete rounds. The register of $rs1$ controls this configuration together with the reset functionality. Register $rs2$ is used for the Keccak round selection. The remaining accelerators have write instructions (input in $rs1/rs2$, address in rd) and read instructions (output and address in rd) to securely copy the shares between register file and accelerator. In addition to the compute operation, the *Binom Tree* accelerator has instructions for resetting $\boldsymbol{z}^{\{0:2\}}$ and copying the sum $\boldsymbol{s}^{\{0:2\}}$ to the input $\boldsymbol{z}^{\{0:2\}}$. The instruction *pq.mbincinv* is used for computing the subtraction.

5.3 Results - System Integration

Table 6 states the resource consumption and performance of the whole RISC-V system as shown in Figure 14 for three different configurations. The standalone version consists

Table 6: Resource overview and estimated max. frequency for the system with and without accelerators.

Design	LUTs	FFs	Slices	DSP	BRAM	Max. Freq.
Standalone	13,412	9,344	4,946	6	32	62.5
Accelerated	20,697	11,833	6,852	13	36.5	62.5
Accelerated masked	29,889	17,152	9,641	13	52.5	58.8

only of the basic RISC-V system without any accelerators. This serves as baseline for comparison with our accelerator extensions.

The accelerated version includes the loosely coupled NTT and the tightly coupled Keccak accelerator. Compared to the standalone version, the LUT consumption increases by a factor of 1.54, the FF and Slice consumption by a factor of 1.27 and 1.39, respectively. As the longest path in the design lies not within the accelerators, the maximum frequency remains at 62.5 MHz.

The last configuration further enables the masked accelerators, i.e. the Secure Adder, Binom Tree, Chi, and Bit-slice accelerators. In addition to that, the optional second data memory is instantiated to allow domain separation of the data shares and thus, accounts for the increase of BRAM usage. Compared to the accelerated version, the LUT/FF/Slice consumption increases by a factor of 1.44/1.45/1.41. Although the accelerators still do not contain the longest path in the design, the maximum frequency slightly decreases. This is most likely caused by the reduced routing capabilities due to increased resource consumption.

6 Experimental Results

This section provides an overview of the performance results for the optimized non-masked and masked implementations of Kyber and Saber, and the leakage assessment of our routines and accelerators.

6.1 Performance Unmasked Implementations

In this work, we evaluated the cycle count for Kyber and Saber with the NIST security level III instantiations. Our source code for the accelerated implementations was compiled with optimization flag `-O3`. Table 7 summarizes our benchmark results and provides a comparison to related works. For the non-masked accelerated version, only the loosely coupled generic NTT unit and the Keccak `f-1600` accelerator are used.

When compared to the HW/SW codesign with RISC-V ISA extensions in [FSS20], we achieved a performance improvement factor of 3.35 for Saber and 1.04 for Kyber (whole algorithm execution). Due to the genericity of our NTT unit, a clear performance advantage for the non-NTT based scheme Saber gets visible. While the tightly coupled NTT design in [FSS20] can be used for Saber only with a costly CRT decomposition and is thus not faster as accelerated Karatsuba/Toom-Cook approaches, our work is directly suitable for a variety of lattice schemes without any hardware changes. Although [FSS20] tailored their design for the small coefficient size of Kyber and two butterfly operations are computed in parallel, we still achieved slightly better performance results. This is mainly achieved due to the flexible and efficient basecase multiplication for incomplete NTTs that is directly integrated within the accelerator. If optimizing for a single NTT-based scheme, like Kyber, the tightly coupled approach has also some advantages including the reduced communication overhead between core and accelerator and the better access to the system memory.

When comparing our results with the latest assembly-optimized implementations of

Table 7: Cycle count non-masked and optimized (`-O3`, GCC PULPino RISC-V compiler 7.1.1 20170509).

Algorithm	Device	Key gen.	Encaps.	Decaps.
Kyber-768 [KRSS18]	ARM M4	976,757 (×3.99)	1,146,556 (×3.60)	1,094,849 (×3.23)
Kyber-768 [AABCG20]	ARM M4	864,008 (×3.53)	1,032,540 (×3.24)	969,867 (×2.86)
Kyber-768 baseline [FSS20]	RISC-V	2,102,505 (×8.59)	2,625,824 (×8.24)	2,573,963 (×7.60)
Kyber-768 optimized [FSS20]	RISC-V	273,370 (×1.12)	325,888 (×1.02)	340,418 (×1.00)
Kyber-768 optimized **(this work)**	RISC-V	**244,660 (×1.00)**	**318,595 (×1.00)**	**338,746 (×1.00)**
Saber [KRSS18]	ARM M4	896,035 (×3.91)	1,161,849 (×3.77)	1,204,633 (×3.47)
Saber [MKV20]	ARM M4	853,000 (×3.72)	1,103,000 (×3.58)	1,127,000 (×3.24)
Saber [CHK+21]	ARM M4	658,000 (×2.87)	864,000 (×2.80)	835,000 (×2.40)
Saber baseline [FSS20]	RISC-V	2,110,283 (×9.20)	2,737,181 (×8.87)	2,797,400 (×8.05)
Saber optimized [FSS20]	RISC-V	760,893 (×3.32)	1,000,043 (×3.24)	1,201,524 (×3.46)
Saber optimized **(this work)**	RISC-V	**229,405 (×1.00)**	**308,430 (×1.00)**	**347,323 (×1.00)**

Table 8: Code size in bytes.

Algorithm	Device	Code size
Kyber-768 [KRSS18]	ARM M4	11,400
Kyber-768 baseline [FSS20]	RISC-V	17,266
Kyber-768 optimized [FSS20]	RISC-V	11,658
Kyber-768 optimized (this work)	**RISC-V**	**13,304**
Saber [KRSS18]	ARM M4	44,468
Saber baseline [FSS20]	RISC-V	17,912
Saber optimized [FSS20]	RISC-V	11,802
Saber optimized (this work)	**RISC-V**	**10,900**

Kyber and Saber on ARM Cortex-M4, we achieve performance improvement factors of 3.18 and 2.66 (whole algorithm execution), respectively.

It has to be noted that the matrix-vector multiplications in MLWE/MLWR schemes require to multiply different ring elements from the matrix with always the same vector. To optimize the AXI communication overhead and the NTT computation costs, we leave the transformed vector within the NTT memory. Moreover, we only load the result from the NTT memory when subsequent operations like polynomial additions/subtractions are completed. For Saber, the number of NTT calls could be further reduced when deviating from the specifications and test vectors. For example, the public matrix $\boldsymbol{A}$ in Kyber is after the sampling already assumed to be in the NTT domain and ring elements are transferred in the NTT domain.

The code size of the proposed implementation is provided in Table 8. Only small deviations of the code size are visible when compared to the ISA extensions in [FSS20]. When compared to a baseline implementation on RISC-V, the code size is still significantly smaller as more complex operations are performed with fewer instructions.

6.2 Performance Masked Implementations

This section provides an overview of our results for the masked Kyber and Saber implementations and compares to prior works [OSPG18] and [VBDK+20]. The masked RLWE implementation presented in [OSPG18] is based on the NewHope algorithm, which has many similarities to Kyber. Both are NTT-based and use a prime modulus, leading to similar masking requirements and approaches. While our implementation and the masked Saber implementation of [VBDK+20] were evaluated for *NIST Level III*, the masked RLWE scheme in [OSPG18] can be categorized to *NIST Level V*. To get a feeling about the performance difference for the different security levels, the authors in [FSS20] reported a factor of 1.26 (Kyber) and 1.63 (Saber) when comparing an unmasked RISC-V imple-

Table 9: Cycle count and code size in bytes masked (-O3, GCC PULPino RISC-V compiler 7.1.1 20170509).

Algorithm	Device	Decapsulation unmasked	Decapsulation masked	Generate randomness	Code size masked
Masked RLWE [OSPG18]	ARM M4	4, 416, 918	25, 334, 493 (×5.74)	+0 (×5.74)[a)]	–
Kyber (this work)	**RISC-V**	**338, 746**	**1, 235, 460 (×3.65)**	**+167, 190 (×4.14)**	**28, 554**
Saber [VBDK+20]	ARM M4	1, 123, 280	2, 833, 348 (×2.52)	+0 (×2.52)[a)]	–
Saber (this work)	**RISC-V**	**347, 323**	**905, 395 (×2.61)**	**+9, 530 (×2.63)**	**21, 042**

[a)] Randomness generation included in decapsulation measurement as onboard TRNG available.

mentation of *Level III* with *Level V*. Although still difficult to compare with [OSPG18], the measurements in Table 9 indicate that our design has a significantly lower cycle count even when the overhead is far above 1.26 in the masked setting for *Level V*.

The comparison to [VBDK+20] is much simpler as the same security level and scheme is considered. Including randomness generation, we achieve a cycle count improvement of factor 3.10 for the masked Saber implementation when using our proposed accelerators. It is also important to mention that our accelerators are designed for flexibility and the non-linear algorithms are easier to extend to higher-order masking schemes than in [VBDK+20]. More specialized accelerators might lead to further speed improvements. However, in this work, we focus on controlled executions of non-linear operations in hardware and a high flexibility.

For our target platform and implementation, Kyber proves more costly to mask than Saber. As explained in Section 2.4, this is partly due to the more complicated prime-moduli masking algorithms and additional masked error sampling of Kyber. However, our efficient masking accelerators compute these algorithms in minimal cycles (Table 3), thereby greatly reducing this algorithmic overhead. These non-linear algorithms can be expected to be significantly slower in a pure software implementation, and accordingly the overhead of masking schemes with prime moduli is expected to be higher. Another large contributing factor is the generation of the randomness required for the masking, for which we use a 32-byte seed and expand it with SHAKE-128 using our Keccak accelerator. While Saber can directly use the Keccak squeeze, Kyber requires partly an additional rejection sampling to obtain uniform randomness modulo q. As a result, Kyber requires roughly 17.5 times more cycles to generate the initial randomness compared to Saber.

6.3 Side-Channel Leakage Evaluation

In this section, we perform a side-channel leakage evaluation of all non-linear operations discussed in this article. These operations are critical as they need to process both shares at the same time. We describe the applied leakage evaluation method, namely the Test Vector Leakage Assessment (TVLA), give details about our measurement setup, and finally provide evaluation results for each operation given a total of 100, 000 side-channel measurements each.

Test Vector Leakage Assessment (TVLA). The resistance of an implementation against SCA can be evaluated by observing the presence of exploitable side-channel information, also called leakage, in a set of side-channel measurements. There are methods based on calculating the Signal-to-Noise Ratio (SNR) [MOP07] or the correlation [DS16] of intermediate variables in order to identify the amount of leakage and the corresponding measurement samples. A central downside of these methods is that an evaluator has to set some assumptions on an exploitable intermediate result or needs to assume a specific hypothesis model.

The Test Vector Leakage Assessment (TVLA) [GJJR11, SM15] methodology has been established to overcome this downside. It makes use of Welch's t-test in order

to statistically evaluate the presence of side-channel leakage without prior knowledge about the investigated implementation. Given two sets of data $\mathcal{Q}_0$ and $\mathcal{Q}_1$, Welch's t-test evaluates if the respective means μ_0 and μ_1 significantly differ from each other. The resulting metric of the TVLA, called *t-value*, is calculated as

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \quad , \tag{33}$$

with variances s_0^2, s_1^2 and n_0, n_1 denoting the cardinalities of the two sets. A high t-value indicates that the null hypothesis (both sets were drawn from the same distribution) is rejected, which implies that it is possible for an attacker to statistically distinguish both sets. This is taken as an indicator for side-channel leakage. In literature, a threshold of $|t| > 4.5$ is usually defined to reject the null hypothesis with a confidence greater than 99.999%.

In order to perform leakage evaluations, the 'non-specific' or 'fixed-vs.-random' t-test can be applied: The evaluator measures the power consumption of multiple algorithm executions with a Boolean or arithmetic masked fixed input $x_{fixed} = x^0 + x^1$ and with a randomly masked input $x_{rand} = x'^0 + x'^1$. Measurements are then split into a set $\mathcal{Q}_0$ with fixed input data and a set $\mathcal{Q}_1$ with random input data. Finally, given these two sets, the t-value according to Equation 33 is calculated for each point in time. A resulting t-value outside the confidence interval ($|t| > 4.5$) indicates that both sets can be distinguished and therefore the implementation exhibits side-channel leakage, which can potentially be used to mount an attack. Otherwise, the implementation can be considered to withstand first-order univariate attacks with the evaluated amount of measurements.

Measurement setup. We implemented our RISC-V design (cf. Section 5) on a NewAE CW305 target board that features an Artix-7 FPGA (XC7A100T). The RISC-V core clock frequency was set to 10 MHz for all side-channel measurements. The SPI interface of the RISC-V platform is used to load the different test programs into the instruction and data memory. The SPI stimuli were created using the GCC PULPino RISC-V compiler (version 7.1.12017050). For all side-channel and performance measurements, the non-linear routines and the accesses of the HW accelerators were manually optimized in assembly. This allows full control of the execution order of instructions and can ensure that the shares are correctly cleaned in order to have only one share at a time within the processor pipeline and register files. The input data according to the TVLA methodology is transferred from the measurement PC to the RISC-V platform through the UART interface.

We acquire side-channel measurements through the SMA connector of the CW305 board with a Picoscope 6402D USB oscilloscope at a sampling frequency of 156.25 MHz. These power measurements correspond to the FPGA's internal supply voltage measured over the integrated 100 mΩ shunt resistor amplified by a 20 dB low-noise amplifier. A dedicated trigger mapped to the RISC-V GPIO port is used to indicate the correct time frame for the measurements. For all TVLA evaluations, a total amount of 100, 000 traces were recorded.

Evaluation Results. To practically validate the first-order SCA resistance of our hardware architectures and the non-linear operations, we applied the TVLA method as described earlier in this section. In order to verify the measurement setup, each leakage test is performed twice: once with activated Random Number Generator (RNG) and once with deactivated RNG. The results are shown in Figure 15. It can be clearly seen that the resulting t-values contain high peaks far above the confidence boundary of $|t| > 4.5$ for the tested operations when turning the RNG off. This validates the setup and shows that all considered operations are leaking information in an unmasked setting or with deactivated RNG.

To cover all accelerators and non-linear operations, which require to process two shares at the same time, we performed the following tests: i) masked Keccak `SHAKE-128` (includes `f-1600` and Chi accelerator), ii) masked binomial sampling Ψ_4 (includes Bit-slicing and Binom Tree accelerators), iii) masked B2A (includes Secure Adder accelerator), iv) masked B2A$_q$ (includes Secure Adder accelerator), v) `MaskedCompress`$_q$ (includes Secure Adder accelerator). Note that the experiment for the compression (Algorithm 13) includes the A2B conversion. Thus, all non-linear operations discussed in Section 2 for masking Kyber and Saber are covered by our experiments. Except for the masked Keccak, all experiments with non-linear operations were performed with 32 polynomial coefficients, which is one function call of the bit-sliced binomial sampler. The masked binomial sampling was measured with Saber parameters $\eta = 4$.

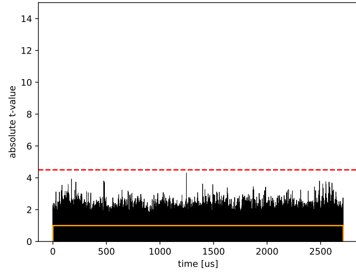
The evaluation results with the RNG turned on show that all implementations stay within the confidence boundary of $|t| < 4.5$. This validates the univariate first-order SCA resistance of the non-linear functions, and therefore all corresponding accelerators, for the given amount of measurement traces. For completeness, we provide the corresponding power measurements in Appendix A (Figure 16). The 24 Keccak rounds and 32 rounds for the coefficients can be partly distinguished by a visual inspection of the power traces. To cover the less critical linear polynomial arithmetic and our loosely coupled NTT accelerator, we provide TVLA results (Figure 17, Appendix B) and the corresponding power measurement (Figure 18, Appendix B) for the polynomial multiplication $\boldsymbol{s} \cdot \boldsymbol{u}^T$ using NTT and Kyber parameters.

7 Conclusion

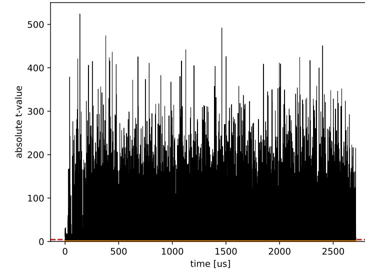
Attacks on the implementation of a cryptographic algorithm are a major concern in cryptography as these attacks allow to break mathematically secure algorithms using side-channel information. Masking methods can be a powerful countermeasure against SCA, even if the attacker has access to the physical device. In the last years, there have been some first works about masking methods for PQC. However, for most PQC finalists the design cost for a secure implementation is still missing. In this work, we presented generic hardware accelerators for the linear and non-linear operations of masked lattice-based cryptography, with a particular focus on Saber and Kyber. Although NTT designs have been a research target in the last years, so far, no generic HW solutions were proposed. Our novel NTT architecture supports positive/negative wraparounds, incomplete NTTs, and prime lifts for non-NTT based schemes, achieving fast polynomial arithmetic for a variety of lattice schemes. Non-linear operations, which involve the processing of two shares at the same time, were accelerated with tightly coupled design solutions for a controlled and efficient execution. These accelerators include the Keccak Chi, the binomial sampling with bit-slicing, and secure addition operations. All accelerators were integrated into a RISC-V platform and ISA extensions were developed to access the accelerators. Due to a novel masked ciphertext compression algorithm and the flexibility of our design, schemes with a power-of-two as well as a non-power-of-two modulus with quite different masking operations can be supported. As a proof of concept, we propose masked implementations of Kyber and Saber. Most of the implemented algorithms extend readily to higher-order side-channel security, which is therefore a clear next step for future research.

Acknowledgements

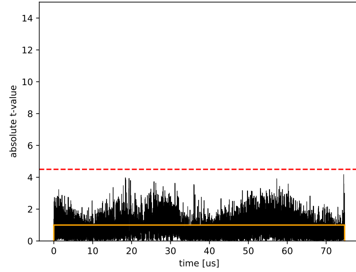
This work was partly funded by the German Ministry of Education, Research and Technology in the context of the project Aquorypt (reference number 16KIS1017K). Moreover, this work was supported in part by CyberSecurity Research Flanders with reference number



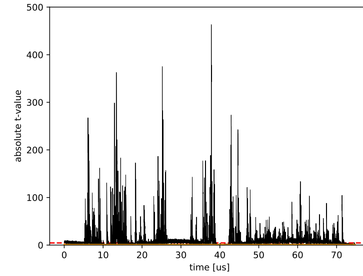
(a) Masked Keccak – `SHAKE-128` (RNG on)



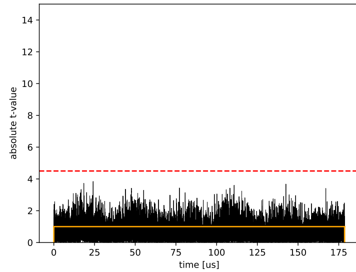
(b) Masked Keccak – `SHAKE-128` (RNG off)



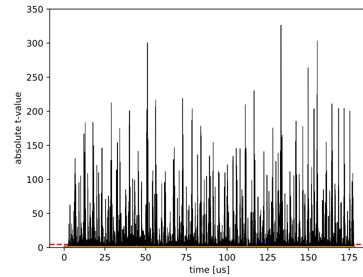
(c) Masked sampling – Ψ_4 (RNG on)



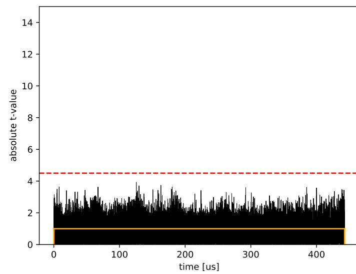
(d) Masked sampling – Ψ_4 (RNG off)



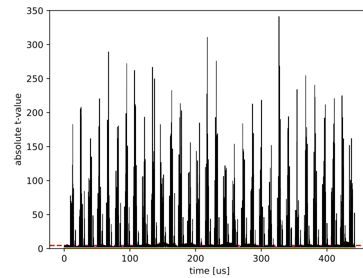
(e) Masked B2A(RNG on)



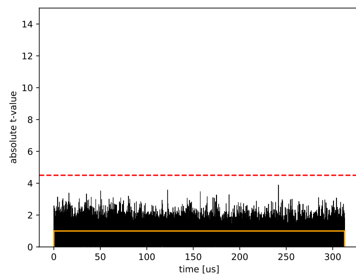
(f) Masked B2A(RNG off)



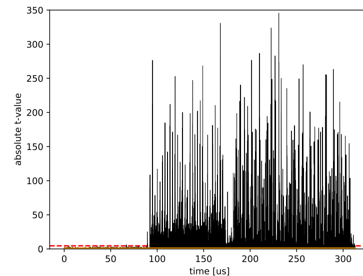
(g) Masked B2A$_q$(RNG on)



(h) Masked B2A$_q$(RNG off)



(i) `MaskedCompress`$_q$ (RNG on)



(j) `MaskedCompress`$_q$ (RNG off)

Figure 15: TVLA results for all non-linear operations with their accelerators given a total amount of 100, 000 traces. The confidence interval ($t = 4.5$) is shown as dotted red line, while the trigger interval is given in orange.

VR20192203, the Research Council KU Leuven (C16/15/058), and the Horizon 2020 ERC Advanced Grant (695305 Cathedral). Michiel Van Beirendonck is funded by an FWO PhD fellowship strategic basic research.

References

[AABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M}LWE schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):336–357, Jun. 2020.

[AASA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2020.

[ABP+18] Victor Arribas, Begül Bilgin, George Petrides, Svetla Nikova, and Vincent Rijmen. Rhythmic Keccak: SCA security and low latency in HW. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 269–290, 2018.

[AEL+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. ISA extensions for finite field arithmetic: Accelerating Kyber and NewHope on RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):219–242, Jun. 2020.

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 354–384. Springer, 2018.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. Improved high-order conversion from Boolean to arithmetic masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):22–45, May 2018.

[BDGH15] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Wei He. Exploiting FPGA block memories for protected cryptographic implementations. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(3):1–16, 2015.

[BDH+21] Shivam Bhasin, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and defending masked polynomial comparison for lattice-based cryptography. Cryptology ePrint Archive, Report 2021/104, 2021.

[BDPVA10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Building power analysis resistant implementations of Keccak. In *Second SHA-3 candidate conference*, volume 142. Citeseer, 2010.

[BGG+15] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications*, pages 64–81, Cham, 2015. Springer International Publishing.

[BNN+12] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, and Georg Stütz. Threshold implementations of all 3 × 3 and 4 × 4 S-boxes. In *International workshop on cryptographic hardware and embedded systems*, pages 76–91. Springer, 2012.

[BPO+20] Florian Bache, Clara Paglialonga, Tobias Oder, Tobias Schneider, and Tim Güneysu. High-speed masking for polynomial comparison in lattice-based KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):483–507, Jun. 2020.

[BUC19] Utsav Banerjee, Tenzin Ukyab, and Anantha Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):17–61, Aug. 2019.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between Boolean and arithmetic masking of any order. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 188–205. Springer, 2014.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 159–188, 2021.

[CJRR99] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Annual International Cryptology Conference*, pages 398–412. Springer, 1999.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. A new algorithm for switching from arithmetic to Boolean masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 89–97. Springer, 2003.

[Deb12] Blandine Debraize. Efficient and provably secure methods for switching from arithmetic to Boolean masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 107–121. Springer, 2012.

[DKR+20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. SABER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[DS16] François Durvaux and François-Xavier Standaert. From improved leakage detection to the detection of points of interests in leakage traces. In *Advances in Cryptology – EUROCRYPT 2016*, pages 240–262. Springer Berlin Heidelberg, 2016.

[FDNG19] Farnoud Farahmand, Viet B. Dang, Duc Tri Nguyen, and Kris Gaj. Evaluating the potential for hardware acceleration of four NTRU-based key encapsulation mechanisms using software/hardware codesign. In *PQCrypto*, volume 11505 of *Lecture Notes in Computer Science*, pages 23–43. Springer, 2019.

[FS19] Tim Fritzmann and Johanna Sepúlveda. Efficient and flexible low-power NTT for lattice-based cryptography. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 141–150. IEEE, 2019.

[FSF⁺19] Tim Fritzmann, Thomas Schamberger, Christoph Frisch, Konstantin Braun, Georg Maringer, and Johanna Sepúlveda. Efficient hardware/software co-design for NTRU. In Nicola Bombieri, Graziano Pravadelli, Masahiro Fujita, Todd Austin, and Ricardo Reis, editors, *VLSI-SoC: Design and Engineering of Electronics Systems Based on New Computing Paradigms*, pages 257–280, Cham, 2019. Springer International Publishing.

[FSM⁺19] Tim Fritzmann, Uzair Sharif, Daniel Müller-Gritschneder, Cezar Reinbrecht, Ulf Schlichtmann, and Johanna Sepúlveda. Towards reliable and secure post-quantum co-processors based on RISC-V. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1148–1153. IEEE, 2019.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):239–280, Aug. 2020.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[GMK16] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact masked hardware implementations with arbitrary protection order. In *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security*, TIS '16, page 3, New York, NY, USA, 2016. Association for Computing Machinery.

[Gou01] Louis Goubin. A sound method for switching between Boolean and arithmetic masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 3–15. Springer, 2001.

[GR19] François Gérard and Mélissa Rossi. An efficient and provable masked implementation of qTESLA. In *International Conference on Smart Card Research and Advanced Applications*, pages 74–91. Springer, 2019.

[GS66] W Morven Gentleman and Gordon Sande. Fast Fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of Keccak. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 205–212. IEEE, 2017.

[HP21] Daniel Heinz and Thomas Pöppelmann. Combined fault and DPA protection for lattice-based cryptography. Cryptology ePrint Archive, Report 2021/101, 2021.

[KG16] Petter Källström and Oscar Gustafsson. Fast and area efficient adder for wide data in recent Xilinx FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[KRSS18] Matthias J Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4, 2018.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23, 2010.

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly fast NTRU using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 180–201, 2019.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium: Efficient implementation and side-channel evaluation. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 344–362, Cham, 2019. Springer International Publishing.

[MKÖ+20] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, Michela Becchi, and Aydin Aysu. A flexible and scalable NTT hardware: applications from homomorphically encrypted deep learning to post-quantum cryptography. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 346–351. IEEE, 2020.

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 222–244, 2020.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: revealing the secrets of smart cards (Advances in Information Security)*. Springer, 2007.

[Nat16] National Institute of Standards and Technology. Announcing request for nominations for public-key post-quantum cryptographic algorithms, 2016. https://csrc.nist.gov/news/2016/public-key-post-quantum-cryptographic-algorithms.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *International conference on information and communications security*, pages 529–545. Springer, 2006.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, 2011.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure and masked ring-LWE implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 142–174, 2018.

[PNPM15] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrian Macias. Accelerating homomorphic evaluation on reconfigurable hardware (extended version). In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 143–163. Springer, 2015.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In *International Conference on Cryptology and Information Security in Latin America*, pages 346–365. Springer, 2015.

[RdCR+16] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively homomorphic ring-LWE masking. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography*, pages 233–244, Cham, 2016. Springer International Publishing.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-LWE implementation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 683–702. Springer, 2015.

[Saa18] Markku-Juhani O Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures. *Journal of Cryptographic Engineering*, 8(1):71–84, 2018.

[SAB+20] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology. In *Lecture Notes in Computer Science*, pages 495–513. Springer Berlin Heidelberg, 2015.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic addition over Boolean masking. In *International Conference on Applied Cryptography and Network Security*, pages 559–578. Springer, 2015.

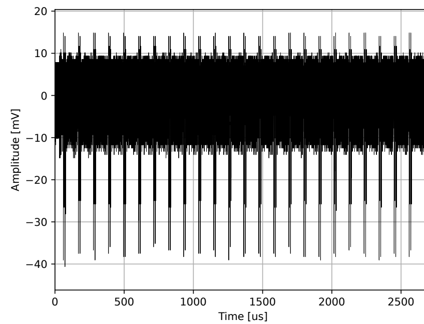
[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *IACR International Workshop on Public Key Cryptography*, pages 534–564. Springer, 2019.

[SR15] Hermann Seuschek and Stefan Rass. Side-channel leakage models for RISC instruction set architectures from empirical data. In *2015 Euromicro Conference on Digital System Design*, pages 423–430. IEEE, 2015.

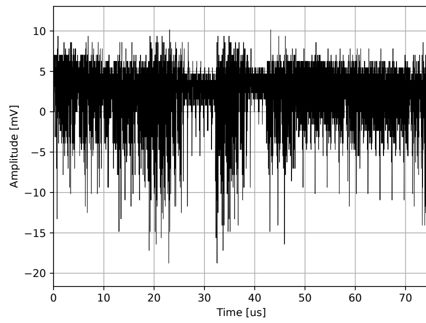
[VBDK+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel resistant implementation of SABER. *IACR Cryptol. ePrint Arch*, 733:2020, 2020.

[VBDV21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. Analysis and comparison of table-based arithmetic to Boolean masking. *IACR Cryptol. ePrint Arch.*, 2021:67, 2021.

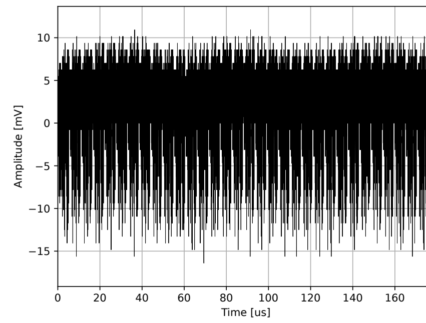
A Power Measurements of Non-Linear Operations



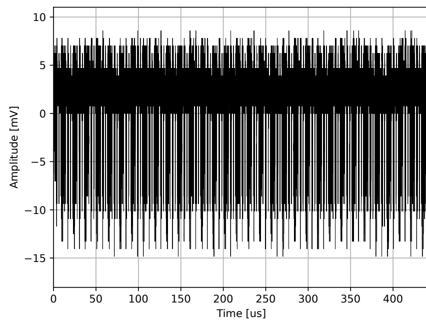
(a) Masked Keccak – SHAKE-128



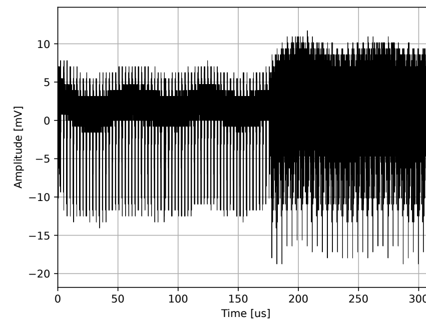
(b) Masked Sampling – Ψ_4



(c) Masked B2A



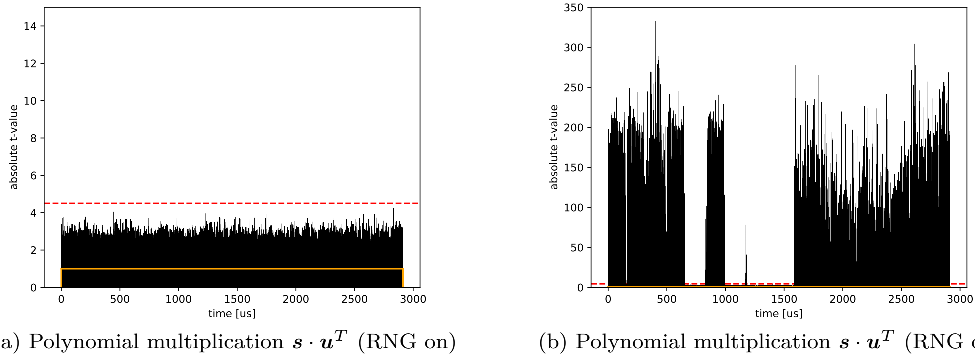
(d) Masked B2A$_q$



(e) MaskedCompress$_q$

Figure 16: Power measurements non-linear operations.

B Power Measurement and TVLA Linear Operations



(a) Polynomial multiplication $\boldsymbol{s} \cdot \boldsymbol{u}^T$ (RNG on)

(b) Polynomial multiplication $\boldsymbol{s} \cdot \boldsymbol{u}^T$ (RNG off)

Figure 17: TVLA results for the linear polynomial multiplication $\boldsymbol{s} \cdot \boldsymbol{u}^T$ given a total amount of 100k traces. The confidence interval ($t = 4.5$) is shown as dotted red line, while the is trigger interval is given in orange.

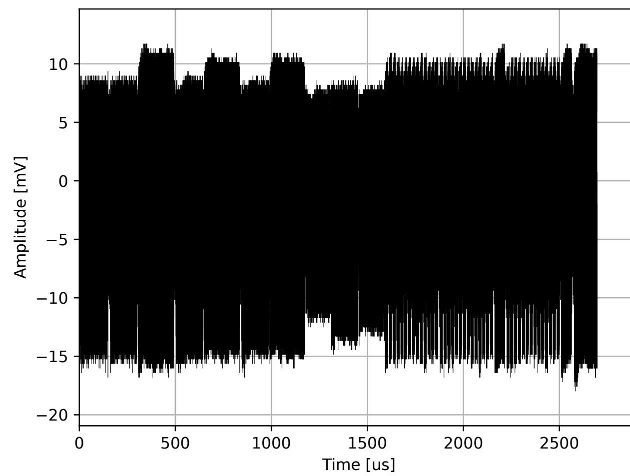


Figure 18: Power measurements linear polynomial multiplication $\boldsymbol{s} \cdot \boldsymbol{u}^T$.