IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 3, pp. 366–390. DOI:10.46586/tches.v2023.i3.366-390

Kavach: Lightweight masking techniques for polynomial arithmetic in lattice-based cryptography

Aikata Aikata[1], Andrea Basso[2,3], Gaetan Cassiers[1*], Ahmet Can Mert[1] and Sujoy Sinha Roy[1]

[1] University of Technology Graz, Graz, Austria
{aikata,gaetan.cassiers,ahmet.mert,sujoy.sinharoy}@iaik.tugraz.at
[2] University of Birmingham, Birmingham, UK
[3] University of Bristol, Bristol, UK
andrea.basso@bristol.ac.uk

Abstract. Lattice-based cryptography has laid the foundation of various modern-day cryptosystems that cater to several applications, including post-quantum cryptography. For structured lattice-based schemes, polynomial arithmetic is a fundamental part. In several instances, the performance optimizations come from implementing compact multipliers due to the small range of the secret polynomial coefficients. However, this optimization does not easily translate to side-channel protected implementations since masking requires secret polynomial coefficients to be distributed over a large range. In this work, we address this problem and propose two novel generalized techniques, one for the number theoretic transform (NTT) based and another for the non-NTT-based polynomial arithmetic. Both these proposals enable masked polynomial multiplication while utilizing and retaining the small secret property.
For demonstration, we used the proposed technique and instantiated masked multipliers for schoolbook as well as NTT-based polynomial multiplication. Both of these can utilize the compact multipliers used in the unmasked implementations. The schoolbook multiplication requires an extra polynomial accumulation along with the two polynomial multiplications for a first-order protected implementation. However, this cost is nothing compared to the area saved by utilizing the existing cheap multiplication units. We also extensively test the side-channel resistance of the proposed design through TVLA to guarantee its first-order security.

Keywords: Masking · Side-Channel Attacks · Lattice-based Cryptography · Post-Quantum Cryptography

1 Introduction

The development of new and fatal attacks against classical cryptographic schemes (e.g., RSA, ECC) seems unlikely, but the advent of quantum computers has threatened their security. It came to light in 1997 when Peter Shor proposed an efficient quantum algorithm to solve integer factorization and discrete logarithm problems [Sho97]. While quantum computers are still far from reaching the computational power needed to break the classical schemes, the recent technological developments [AAB+19, IBM] have been slowly bridging the gap. Experts believe that within 15 years, quantum computers will pose a substantial threat to public-key cryptography [MP21]. Hence, it is urgent to develop

*The work was partially done while being with Lamarr Security Research.

Licensed under Creative Commons License CC-BY 4.0.
Received: 2023-01-15 Accepted: 2023-03-15 Published: 2023-06-09

and migrate to quantum-resistant alternatives to prevent security breaches. As a first step, the American National Institute of Standards and Technology (NIST) initiated a standardization process in 2016 to find new quantum-resistant protocols, also known as Post Quantum Cryptography (PQC), for key encapsulation and digital signatures. In this standardization, the lattice-based schemes debuted as the most promising candidates. These schemes offered good performance, efficiency, low latency, and agility.

While the NIST standardization aimed at selecting a portfolio of post-quantum algorithms for *general applications*, there is a need for developing tailored post-quantum schemes that meet application-specific needs, for example, the constraints of the IoT and automotive applications. In recent years, several new post-quantum algorithms [DPPvW22, MKKV21] have emerged with better performance or security features (or both) than the candidate algorithms in the NIST standardization. Security against side-channel attacks [MOP07] has become an essential requirement in applications where an attacker can obtain side-channel information such as variations in the power consumption or electromagnetic emanation, or temperature of the cryptographic device.

Masking is a widely used countermeasure against differential power analysis-based side-channel attacks [MOP07]. Several masking schemes have been proposed in the literature [RRVV15, OSPG18, BDK+21b, AMD+21, KDB+22, FBR+22] for lattice-based public-key algorithms. Section 2.5 discusses masking works in more detail. Typically, masking schemes for lattice-based algorithms increase the computation overhead by approximately 2.5× to 6× (for first-order protection) with respect to unprotected implementations. Therefore, significant research is needed to investigate masking schemes that reduce the performance overhead. One of the computationally-intensive operations in lattice-based algorithms is the multiplication of polynomials [SR19]. The asymptotically fastest method for this is to use Number Theoretic Transform (NTT), which reduces polynomial multiplication to coefficient-wise multiplication. The NTT-based polynomial multiplication is however limited to polynomial rings where the modulus q is prime (we name such rings "NTT-friendly", while rings with no NTT are "NTT-unfriendly"). It is still possible to use NTT-based polynomial multiplication to implement cryptographic algorithms with NTT-unfriendly rings, but this comes at a performance overhead since it typically involves computing NTTs with a significantly larger modulus (see Section 2.5).

On the other hand, recent works have shown that when a structured lattice-based scheme uses a power-of-2 modulus, the masking countermeasure incurs the least performance overhead [BDK+21b, FBR+22, KDB+22] since, when using binary (masked or not) representations, a modular reduction is essentially free for such moduli. The downside of using a power-of-2 modulus is that the ring is NTT-unfriendly. In learning with errors (LWE) or learning with rounding (LWR)-based public-key schemes, this limitation is mitigated since one operand polynomial (typically a secret or an error polynomial) is always a small-coefficient polynomial, allowing significant optimizations in multiplication algorithms, both "classical" (e.g, schoolbook, Toom-Cook or Karatsuba) and NTT-based (allowing to use a smaller modulus [AMJ+22]). For example, [RB20] presents a very low latency schoolbook polynomial multiplier (unprotected implementation) by using many small-coefficient (hence low-cost) multipliers in parallel. Unfortunately, existing masking schemes for lattice-based cryptography prevent such optimizations since they split a secret or an error polynomial with small coefficients into uniformly random polynomials with large coefficients.

Our Contributions: In this work, we propose two techniques to extend the benefits of the small-coefficient property to masked implementations. These techniques are then instantiated in concrete hardware design, implementation, and t-test evaluation to verify real-life security guarantees.[1]

Our first method is an optimized technique for NTT-based polynomial multiplication in

[1] Our implementation is available at https://extgit.iaik.tugraz.at/sesys/kavach_artifacts.

masked implementations of algorithms with NTT-unfriendly rings. Through a new choice of group for the arithmetic masking, we are able to reduce the required size for the modulus of the NTT. Among methods that improve over the worst-case "large coefficient" masked NTT-based multiplication for NTT-unfriendly rings, our method is the first general one (i.e., it works with any modulus), and it surpasses all the existing works (in hardware, it uses two to three times less area for the same performance). This method works for any masking order and has a cost (in computation time or area) proportional to the number of shares. Another advantage of this method is that it can efficiently use hardware that implements *non-masked* NTT-based polynomial multiplication in the same NTT-unfriendly ring, allowing resource-sharing.

Our second method applies to "classical" multiplication algorithms. We propose a new *hybrid masking* that combines arithmetic and Boolean masking, which allows us to split small secret coefficients into small shares, thus preserving the small secret property. We show how to use masking when implementing lattice-based cryptography, including conversions between arithmetic and hybrid masking. Our new hybrid masking gadgets are proven to be secure against first-order leakage in the hardware glitch-robust probing model. Finally, we provide experimental validation by implementing the proposed algorithms in hardware and performing leakage assessment tests [GJJR11]. Regarding the performance, the use of hybrid masking in place of the state-of-the-art approach of arithmetic masking with large coefficients allows using small multipliers instead of large ones (4.5× area gain), at the cost of performing more additions (accumulators are anyway instantiated, hence this does not increase the area).

With these contributions, we improve the state-of-the-art for masking lattice-based schemes and provide improved insights on implementation costs for future PQC scheme designs.

Organization: We present notations and preliminary information about polynomial multiplications, lattice-based construction, generic countermeasures to side-channel attacks, and the research gap in Section 2. In Section 3, we present the methods to optimize side-channel countermeasures for an NTT-based polynomial arithmetic unit. In the next section, Section 4, we introduce the technique of hybrid masking to efficiently mask non-NTT-based polynomial arithmetic. We then implement the proposed techniques and provide implementation details in Section 5. Following this, Section 6 describes the t-test evaluation setup and results. Finally, in Section 7, we discuss the applications of the proposed techniques and conclude the work.

2 Background

2.1 Notation

We use $\mathbb{Z}_q$ to represent integers modulo q. We use $\mathcal{R}_q$ to represent polynomial ring $\mathbb{Z}_q/\phi(x)$ where $\phi(x)$ is a reduction polynomial of degree n. We use $\mathcal{B}^l$ to represent a byte array of size l. We use lowercase, bold lowercase, and bold uppercase letters to represent an integer (e.g., $a \in \mathbb{Z}_q$), a polynomial (e.g., $\boldsymbol{a} = \sum_{i=0}^{n-1} \boldsymbol{a}[i] \cdot x^i \in \mathcal{R}_q$) or vector (e.g., $\boldsymbol{a} \in \mathbb{Z}_q^n$), and a vector of polynomials (e.g., $\boldsymbol{A} \in \mathcal{R}_q^{k \times l}$), respectively. We use $\{a\}^l$ to represent a vector of length-l where each element is a. We use $a[i]$, $\boldsymbol{a}[i]$ and $\boldsymbol{A}[i]$ to represent the i-th bit of integer a, i-th of coefficient of polynomial $\boldsymbol{a}$ and i-th polynomial of polynomial vector $\boldsymbol{A}$. We use $\times$ and $\star$ to represent polynomial multiplication and coefficient-wise multiplication of two polynomials, respectively. We use $\cdot$ to represent integer multiplication (e.g., $a \cdot b$) or multiplication of an integer with coefficients of a polynomial (e.g., $a \cdot \boldsymbol{b}$ or $\boldsymbol{b} \cdot a$). We use `&`, `|`, $\oplus$, `!`, and $\gg$ / $\ll$ to represent logical AND, OR, XOR, NOT, and right/left shift operations, respectively. We represent the sampling of an integer a from uniform distribution as $a \xleftarrow{\$} \mathbb{Z}_q$. Similarly, we use $\boldsymbol{a} \xleftarrow{\$} \mathcal{R}_q$ and $\boldsymbol{a} \leftarrow \chi(\mathcal{R}_q, \sigma)$ to represent the

sampling of polynomial $\boldsymbol{a} \in \mathcal{R}_q$ from uniform distribution and χ with standard deviation σ, respectively. A polynomial $\boldsymbol{s} \in \mathcal{R}_q$ is *small* if all its coefficients belong to the range $[-\mu, +\mu] \mod q$ where μ is a k-bit positive integer and $K = 2^k$ with $\mu \ll q$.

2.2 Masking

Differential side-channel attacks are very powerful against cryptographic schemes. A strong countermeasure against them is the use of masking, which randomizes the power consumption or electromagnetic emanation of the cryptographic platform. Masking a computation means replacing every intermediate value x with a randomized tuple (named a sharing) that represents the value, and whose elements are the shares. Then, all computations are modified to operate on these shares: every computation is replaced by a gadget that performs the same computation on the shares. In this paper, we are mainly interested in first-order arithmetic masking, where the sharing of a value $x \in \mathbb{Z}_q$ is a tuple $(x_0, x_1) \in \mathbb{Z}_q^2$ such that $x_0 + x_1 = x \mod q$. The sharing (x_0, x_1) of x is *uniform* if its probability distribution is uniform over the set $\{(x_0, x_1) \in \mathbb{Z}_q^2 | x_0 + x_1 = x \mod \mathbb{Z}_q\}$, and a sharing is *fresh* if its distribution over that set is independent of any other value in the computation (excluding values that are computed from these shares). As a particular case, Boolean masking is arithmetic masking in $\mathbb{Z}_2$. We also extend the notion of sharing to polynomials (coefficient-wise): arithmetic shares of $\boldsymbol{s} \in \mathcal{R}_q$ are the tuple $(\boldsymbol{s}_0, \boldsymbol{s}_1) \in \mathcal{R}_q$ if $\boldsymbol{s}_0 + \boldsymbol{s}_1 = \boldsymbol{s}$.

The security of masking is most often studied in the t-probing model [ISW03]. A gadget is t-probing secure if any set of t ($t = 1$ for first-order masking) intermediate values in the computations (named probes) is independent of the secret values of the inputs of the gadget, assuming that the input shares are fresh. This abstract leakage model does however not cover the leakage caused by glitches (or other intra-cycle data-dependent timing) in hardware implementations, which led to the introduction of the glitch-robust probing model [FGP+18]. In this model, a probe on a wire observes not only the value of the wire but also all the inputs of the combinatorial circuit that computes that wire, therefore only the synchronization elements (i.e., the registers) stop the propagation of glitches.

2.3 Polynomial multiplication

Polynomial multiplication is one of the fundamental operations in lattice-based cryptography. There are different approaches to implementing polynomial multiplication. The selection of the proper implementation method depends on scheme parameters, target performance, and platform.

Schoolbook polynomial multiplication: For polynomials $\boldsymbol{a} \in \mathcal{R}_q$ and $\boldsymbol{b} \in \mathcal{R}_q$, the polynomial multiplication $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$ is defined as $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \boldsymbol{a}[i] \cdot \boldsymbol{b}[j] \cdot x^{i+j}$. When the multiplication is performed in a polynomial ring, a separate reduction operation with reduction polynomial $\phi(x)$ is required. When $\phi(x)$ has a special form, this reduction operation can be merged into the multiplication operation free of cost. For example, when $\phi(x)$ is $x^n + 1$, $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$ is defined as $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (-1)^{\lfloor (i+j)/n \rfloor} \cdot \boldsymbol{a}[i] \cdot \boldsymbol{b}[j] \cdot x^{i+j \pmod n}$. using $x^n \equiv -1$ [LFK+19]. Although the Schoolbook method has high computational complexity ($\mathcal{O}(n^2)$), it works with every parameter set and does not have any constraint for the polynomial ring or ring modulus.

Karatsuba polynomial multiplication: Karatsuba method follows a divide-and-conquer approach and has the complexity $\mathcal{O}(n^{\log_2 3})$ [KO62]. Let $\boldsymbol{a} = \boldsymbol{a}_h x^{n/2} + \boldsymbol{a}_l$ and $\boldsymbol{b} = \boldsymbol{b}_h x^{n/2} + \boldsymbol{b}_l$ be polynomials of size n where $\boldsymbol{a}_h$, $\boldsymbol{a}_l$, $\boldsymbol{b}_h$ and $\boldsymbol{b}_l$ are $(\frac{n}{2} - 1)$-degree polynomials. Then, $\boldsymbol{a} \times \boldsymbol{b}$ can be written as $(\boldsymbol{a}_h \times \boldsymbol{b}_h)x^n + (\boldsymbol{a}_h \times \boldsymbol{b}_l + \boldsymbol{a}_l \times \boldsymbol{b}_h)x^{n/2} + (\boldsymbol{a}_l \times \boldsymbol{b}_l)$. Karatsuba method reduces the number of $(\frac{n}{2} - 1)$-degree polynomial multiplications from

4 to 3 by using the term $(\boldsymbol{a}_h \times \boldsymbol{b}_l + \boldsymbol{a}_l \times \boldsymbol{b}_h) = (\boldsymbol{a}_h + \boldsymbol{a}_l) \times (\boldsymbol{b}_h + \boldsymbol{b}_l) - \boldsymbol{a}_h \times \boldsymbol{b}_h - \boldsymbol{a}_l \times \boldsymbol{b}_l$. Karatsuba can be applied recursively to reduce the size of the multiplication operations.

NTT-based polynomial multiplication: The NTT of an n-size polynomial $\boldsymbol{a}$ can be computed as $\overline{\boldsymbol{a}}[i] = \sum_{j=0}^{n-1} \boldsymbol{a}[j] \cdot \omega^{ij} \pmod q$ for $i \in [0, n-1]$, where $\omega \in \mathbb{Z}_q$ is a n-th root of the unity, i.e. it satisfies the conditions $\omega^n \equiv 1 \pmod q$, $\omega^i \neq 1 \pmod q$ $\forall i < n$. Such a root exists only if $q \equiv 1 \pmod n$. NTT-based polynomial multiplication has the fastest time complexity $\mathcal{O}(n \cdot \log n)$ [CT65]. When the polynomial multiplication operation is performed in a ring $\mathcal{R}_q = \mathbb{Z}_q/\phi(x)$, a separate reduction with polynomial $\phi(x)$ is required after the completion of multiplication. If $\phi(x)$ has the special form $x^n + 1$, then *negative wrapper convolution* (NWC) can be used to merge the reduction with the actual polynomial multiplication. NTT requires the modulus q of ring $\mathcal{R}_q = \mathbb{Z}_q/\langle x^n + 1\rangle$ to be prime. Therefore, cryptographic schemes that use such a modulus are NTT-friendly (e.g., [BDK+18, BDK+21a] and the ones that do not are NTT-unfriendly (e.g., [BMD+20, MKKV21]).

2.4 Lattice-based public key encryption schemes

Several lattice-based PKE/KEM schemes are based on the Learning With Errors (LWE) problem and its variants [Reg09]. For a given public matrix $\boldsymbol{A} \in \mathbb{Z}_q^{k \times l}$ and vector $\boldsymbol{B} = \boldsymbol{A} \times \boldsymbol{S} + \boldsymbol{E} \in \mathbb{Z}_q^k$, there is no known algorithm that can recover secret vector $\boldsymbol{S} \leftarrow \chi(\mathbb{Z}_q^l, \sigma)$ in polynomial time where $\boldsymbol{E} \leftarrow \chi(\mathbb{Z}_q^k, \sigma)$ is an error vector. Solving the secret $\boldsymbol{S}$ in the presence of error $\boldsymbol{E}$ is known as the LWE problem. A variant of the LWE problem is the Learning With Rounding (LWR) problem, which introduces deterministic errors by scaling the elements of $\boldsymbol{A} \times \boldsymbol{S}$ by a non-integer constant and rounding them to the closest integer. Ring-LWE/LWR (RLWE/RLWR) and Module-LWE/LWR (MLWE/MLWR) problems [LS15] are implementation-friendly variants of LWE/LWR, and both operate over polynomial rings instead of integers. In the following, we briefly describe the LPR encryption scheme [LPR10] which has been used as a framework for building modern LWE/LWR-based public-key encryption and encapsulation schemes.

LPR Encryption Scheme: In Algorithm 1, we present the high-level description of key generation, encryption, and decryption procedures of the RLWE-based LPR scheme. Although LPR-based schemes are IND-CPA secure, they are not secure against chosen ciphertext attacks (CCA) where an adversary with access to the decrypted messages can recover the secret key by using carefully-selected ciphertexts. In order to achieve IND-CCA security, LWE/LWR-based schemes are using Fujisaki-Okamoto (FO) transformation [FO99] which uses a combination of encryption/decryption procedures with hash functions to generate CCA-secure encapsulation and decapsulation procedures. With the FO transformation, the decapsulation procedure performs a re-encryption after decrypting the message and then compares the input ciphertext and the generated ciphertext by re-encryption procedure. If the comparison fails, i.e. the input ciphertext is invalid, the decapsulation mechanism outputs a random byte array as the decrypted message.

2.5 Masking lattice-based public-key schemes and gaps

In the last several years, especially during the NIST PQC standardization competition, several masking schemes for PQC algorithms have been proposed. In this section, we revisit a subset of existing masking schemes that are relevant to this paper.

The first masking of a ring-LWE-based LPR public-key encryption (Algorithm 1) was proposed in [RRVV15], based on arithmetic masking in $\mathcal{R}_q$. Concretely, to mask the decryption, it splits the secret $\boldsymbol{s} \in \mathcal{R}_q$ into two uniformly random arithmetic shares $\boldsymbol{s}_0, \boldsymbol{s}_1 \in \mathcal{R}_q$. Next, $\tilde{\boldsymbol{m}}'$ is computed as two arithmetic shares $\tilde{\boldsymbol{m}}'_0 = (\boldsymbol{v} - \boldsymbol{u} \times \boldsymbol{s}_0)$ and $\tilde{\boldsymbol{m}}_1{}' = -\boldsymbol{u} \times \boldsymbol{s}_1$. Finally, for the comparison-based decoding step, a probabilistic masked

Algorithm 1 LPR Encryption Scheme [LPR10]

Procedure PKE.KeyGen()
- $\boldsymbol{a} \xleftarrow{\$} \mathcal{R}_q$
- $\boldsymbol{s}, \boldsymbol{e} \leftarrow \chi(\mathcal{R}_q, \sigma) \in \mathcal{R}_q$
- $\boldsymbol{t} = (\boldsymbol{a} \times \boldsymbol{s} + \boldsymbol{e}) \in \mathcal{R}_q$
- **return** $\mathsf{pk} = (\boldsymbol{a}, \boldsymbol{t}), \mathsf{sk} = (\boldsymbol{s})$

Procedure PKE.Encrypt($\mathsf{pk} = (\boldsymbol{a}, \boldsymbol{t}), \mathsf{m} \in \mathcal{B}^{32}, \mathsf{r} \in \mathcal{B}^{32}$)
- $\boldsymbol{s}', \boldsymbol{e}', \boldsymbol{e}'' \leftarrow \chi(\mathcal{R}_q, \sigma, \mathsf{r})$ `/* r is used as random seed */`
- $\boldsymbol{u} = (\boldsymbol{a} \times \boldsymbol{s}' + \boldsymbol{e}') \in \mathcal{R}_q$
- $\boldsymbol{v}' = (\boldsymbol{t} \times \boldsymbol{s}' + \boldsymbol{e}'') \in \mathcal{R}_q$
- $\tilde{\boldsymbol{m}} = \mathsf{Encode}(\mathsf{m}) \in \mathcal{R}_q$ `/* Encoding msg bits to poly. coefficients */`
- $\boldsymbol{v} = (\boldsymbol{v}' + \tilde{\boldsymbol{m}}) \in \mathcal{R}_q$
- **return** $\mathsf{ct} = (\boldsymbol{u}, \boldsymbol{v})$

Procedure PKE.Decrypt($\mathsf{ct} = (\boldsymbol{u}, \boldsymbol{v}), \mathsf{sk} = (\boldsymbol{s})$)
- $\tilde{\boldsymbol{m}}' = (\boldsymbol{v} - \boldsymbol{u} \times \boldsymbol{s}) \in \mathcal{R}_q$
- $\mathsf{m}' = \mathsf{Decode}(\tilde{\boldsymbol{m}}') \in \mathcal{B}^{32}$ `/* Decoding poly. coefficients to msg bits */`
- **return** m'

decoder is used in [RRVV15]. A hardware implementation of the masked decryption is provided where the masked polynomial multiplication is implemented using the NTT method. Masking the polynomial arithmetic is simple as it incurs only $d\times$ computation overhead while d number of arithmetic shares of the secret. On the other hand, the paper shows that masked decoding is a more complex and slow operation. A more efficient masked decoding technique is presented in [OSPG18]. To perform decoding on the input arithmetic shares of a coefficient, it first shifts the distribution of the shares, then applies the arithmetic-to-Boolean (A2B) transformation, and finally computes the sign bit. Besides improving the masked decoding, the paper also presents a complete masking framework for ring-LWE-based public-key encryption with CCA security (i.e., chosen ciphertext attack resistance). Hence, the paper masks sensitive parts of the re-encryption operation including error and ephemeral secret polynomial sampling, and random number generation. With a first-order masked implementation, the NTT-based polynomial multiplication cost increases to 2× and the bit-wise building blocks for example, error sampling, and random number generation become more than 5× slower. The masked decryption for a CCA-secure variant of NewHope [ADPS16] is around 5.7× slower than its non-masked operation.

Significant reduction in the masking overhead for CCA-secure public-key encryption or KEM is demonstrated in [BDK+21b]. The integration of bit-wise building blocks with the polynomial arithmetic blocks inside a masking scheme requires expensive Boolean-to-arithmetic and arithmetic-to-Boolean conversions. The paper observes that by choosing the modulus q to be a power-of-2, these conversions can be simplified a lot compared to the case with a prime modulus. For such power-of-2 modulus, the paper heavily optimizes masked logical shifting on arithmetic shares and error sampling from a binomial distribution. On the ARM Cortex-M4 platform, masking the Saber KEM (which uses power-of-2 modulus) incurs only 2.5× computation overhead unlike 5.7× overhead with prime modulus in [OSPG18]. Polynomial multiplication is computed using a combination of Toom-Cook, Karatsuba, and low-degree schoolbook methods as the NTT method is not a natural choice with a power-of-2 modulus.

A lightweight masked hardware implementation of Saber KEM is presented in [AMD+21]. It shows an area overhead of 2.9× and the polynomial multiplication is implemented using the Schoolbook method. In [KDB+22], the authors propose a higher-order masked Saber implementation on ARM Cortex-M4. They show that the increase in performance overhead for higher-order masking of Saber is smaller compared to Kyber. Their first-order masked

implementation shows 2.7× performance overhead. Both works do not take the advantage of secret coefficients of Saber being small.

A masked HW/SW codesign implementation of several NIST PQC finalists (such as Saber, Kyber, and NTRU) is presented in [FBR+22]. It extends the instruction set of a RISC-V processor by incorporating an NTT-based polynomial multiplier. Hence, all PQC schemes irrespective of cohesion with NTT, are forced to use the NTT method for computing polynomial multiplication. Critical non-linear operations for masking are computed in the hardware. When the masking countermeasure is turned off, Kyber (uses NTT-friendly prime) is around 1.2× faster than Saber (uses NTT-unfriendly power-of-2 modulus). However, with first-order masking turned on, Kyber's decapsulation becomes around 1.53× slower than that of Saber. Such results clearly demonstrate that in applications where masking countermeasure becomes essential, using a power-of-2 modulus is more beneficial for the performance even though it may cause a minor slowdown in polynomial multiplication. Therefore, in this work, we primarily consider masking with power-of-2 modulus. We would like to remark that a power-of-2 modulus can be used in standard as well as structured lattices with both LWE and LWR foundations.

Research gap: Taking advantage of small operands for the implementation of polynomial multiplication is studied by several works. In [RB20], the authors propose an efficient schoolbook multiplier implementation where one of the operands has small coefficients. Their implementation uses an add-shift-based approach to perform the multiplication with small coefficients and eliminates large multiplier units. For NTT-based multiplication implementations in $\mathcal{R}_q$ where q is power-of-2, [AMJ+22, CHK+21, BAD+21] show how to take advantage of small coefficients and use a smaller NTT-friendly modulus Q. For example, authors in [AMJ+22] show that a 25-bit NTT-friendly prime Q would be sufficient to perform NTT-based polynomial multiplication for Saber instead of a 34-bit prime [FSS20].

A natural question is raised in [ACC+21], can the masked implementations also benefit from one of the multiplicands being small? Existing works targeting masked implementations of schemes with power-of-2 modulus (e.g., Saber, Florete, Espada, and Sable [BMD+20, MKKV21]) fail to utilize the small secret property. The size of the datapath for schoolbook or NTT-based implementations grew in this case as shown in previous works [FBR+22, AMD+21], which is not needed and is an undersell for efficient MLWR-based schemes. In this work, we show how to utilize small coefficients in masked implementations.

3 Masking NTT-based polynomial multiplication in NTT-unfriendly rings

In this section, we introduce our first method: an efficient masked NTT-based multiplication for small polynomials in NTT-unfriendly rings. This method is well-suited to platforms where an NTT accelerator is already available, for example, micro-controllers with NTT acceleration or hardware components that implement multiple lattice-based algorithms, including NTT-based ones such as Kyber and Dilithium.

In general, NTT-based polynomial multiplication requires the coefficient modulus q to be a prime, since otherwise (e.g., when q is power-of-2) the NTT does not exist in $\mathcal{R}_q$. To force NTT-based polynomial multiplication in such rings, one solution is to perform an NTT-based multiplication using a large and prime modulus Q such that the coefficients in all the intermediate values are never larger than the modulus, and therefore no actual modular reduction (mod Q) ever happens. Then, the final result is obtained with a reduction modulo q. In general, one must ensure $Q > n \cdot q^2$ to avoid unwanted modular reductions [AMJ+22, FBR+22].

The size of Q can be reduced when one polynomial operand is always of small coefficients,

for example, an error or secret polynomial in an LWE/LWR scheme. Concretely, if one of the polynomials is small (i.e. all its coefficients belong to $[-\mu, +\mu] \mod q$), a prime $Q > n{\cdot}q{\cdot}\mu$ is sufficient. Optimized but unprotected hardware [AMJ+22] and software [SR19] implementations of NTT-based polynomial multiplication in NTT-unfriendly rings benefit from this relaxation on Q to improve speed and also reduce the memory requirement and width of the datapath.

However, when the small polynomial is arithmetically masked, the shares of its coefficients are uniform in $\mathbb{Z}_q$ and are therefore not small. The natural solution is therefore to switch back to using a large prime $Q > n \cdot q^2$, resulting in performance degradation [FBR+22]. The key idea of our new technique is the change of perspective: instead of multiplying each of the secret shares in modulo q with the public polynomial (achieved by working with a large $Q > n \cdot q^2$), we consider the non-masked multiplication (hence using a smaller $Q > n \cdot q \cdot \mu$) and mask the operations of this multiplication using arithmetic masking modulo Q. Therefore, we do not care about modular reductions happening on the shares: the final unmasked result is guaranteed to be correct.

Our new method is described in Algorithm 2. Let the unprotected polynomial multiplication be $\boldsymbol{a} \times \boldsymbol{s}$ in $\mathcal{R}_q$ where the public polynomial $\boldsymbol{a}$ is uniformly random in $\mathcal{R}_q$ and the secret polynomial $\boldsymbol{s}$ is a small polynomial with coefficients in $[-\mu, +\mu]$. We take as input an arithmetic sharing $(\boldsymbol{s}_0, \boldsymbol{s}_1)$ in $\mathcal{R}_Q$ for a prime $Q > n \cdot q \cdot \mu$ (converting from $\mathcal{R}_q$ if needed), then compute two polynomial multiplications $\boldsymbol{a} \times \boldsymbol{s}_0$ and $\boldsymbol{a} \times \boldsymbol{s}_1$ independently in $\mathcal{R}_Q$ using NTTs. Finally, we perform masked modular reduction modulo q to bring the result to $\mathcal{R}_q$.

Algorithm 2 NTT-based polynomial multiplication with masking

Input: $\boldsymbol{a} \in \mathcal{R}_q$ (public polynomial), $\boldsymbol{s}_0$, $\boldsymbol{s}_1 \in \mathcal{R}_Q$ (secret shares s.t. $\boldsymbol{s}_0 + \boldsymbol{s}_1 = \boldsymbol{s} \bmod Q$)
Output: $\boldsymbol{t}_{0,q}, \boldsymbol{t}_{1,q}$ (s.t. $\boldsymbol{t}_{0,q} + \boldsymbol{t}_{1,q} = \boldsymbol{a} \times \boldsymbol{s} \bmod q$)
1: $\boldsymbol{t}_{0,Q} \leftarrow \texttt{INTT}(\texttt{NTT}(\boldsymbol{a}) \star \texttt{NTT}(\boldsymbol{s}_0)) \in \mathcal{R}_Q$
2: $\boldsymbol{t}_{1,Q} \leftarrow \texttt{INTT}(\texttt{NTT}(\boldsymbol{a}) \star \texttt{NTT}(\boldsymbol{s}_1)) \in \mathcal{R}_Q$
3: $\{\boldsymbol{t}_{0,q}, \boldsymbol{t}_{1,q}\} \leftarrow \texttt{MaskedRed}_{Q \to q}(\boldsymbol{t}_{0,Q}, \boldsymbol{t}_{1,Q})$
4: **return** $\boldsymbol{t}_{0,q}, \boldsymbol{t}_{1,q} \in \mathcal{R}_q$

As a concrete example, let us consider Saber [BMD+20]: $q = 2^{13}$ and $\mathcal{R}_q = \mathbb{Z}_q/\langle x^{256}+1\rangle$. If NTT is to be used for unprotected $\boldsymbol{a} \times \boldsymbol{s}$, an ephemeral-prime modulus Q (25-bit number [AMJ+22, CHK+21, DMG23]) is needed since the coefficients of the public $\boldsymbol{a}$ and secret $\boldsymbol{s}$ are respectively 13 and 4-bits long. Our technique (Algorithm 2) can use the same value for Q. Without using our technique, the coefficient of the shares of the secret polynomial are uniformly distributed, and Q must be 34-bit for the sake of correctness, which is commonly done in masking literature [ACC+21, FBR+22].

For one such NTT-unfriendly parameter set, the authors in [AMJ+22] propose 23, 24, and 25-bit primes in order to support NTT-based polynomial multiplication. The LUT consumption for making a 34×34-bit multiplier is twice as much as a 25×25-bit multiplier and 3× compared to 23×23-bit multiplier. The 39-bit NTT-based multiplier proposed by the authors in [FBR+22] consumes 2,454 LUTs, 1,917 FFs, 7 DSPs, 4.5 BRAMs, and requires 4,096 clock cycles. The proposed method ensures that no changes need to be made to the NTT-based polynomial arithmetic unit and can therefore, finish the NTT in just 512 clock cycles and consume only 2,257 LUTs, 1,079 FFs, 4 DSPs, and 1 BRAM (from [AMJ+22]).

Regarding modulus conversions for arithmetic masking, a generic solution is to use an arithmetic-to-Boolean (A2B) conversion followed by a Boolean-to-arithmetic (B2A) conversion [BC22]. However, some of these conversions can be avoided to reduce costs, e.g. when the input and/or output is generated/used in a Boolean-masked representation.

4 Compact polynomial arithmetic using hybrid masking

In this section, we introduce our second method for masking polynomial multiplication. Unlike our NTT-based method, this *hybrid masking* technique produces small arithmetic share coefficients if the unmasked polynomial has small coefficients. Then, the polynomial multiplication can be performed on the shares using any multiplication algorithm, with the ability to optimize for the small coefficients. We introduce the hybrid sharing representation and explain how to generate and use it. Finally, we describe how these techniques can be applied and optimized for polynomial multiplication.

4.1 Hybrid masking

In a nutshell, hybrid masking represents a small value s in a large group $\mathbb{Z}_q$ by starting from an arithmetic sharing (s_0, s_1) in a smaller group $\mathbb{Z}_K$ (i.e., $s = s_0 + s_1 \bmod K$). It converts this modular sharing into a non-modular one (i.e., in $\mathbb{Z}$) by explicitly computing the overflow bit c: $s = s_0 + s_1 - Kc$, which is then also a correct (non-uniform) arithmetic sharing in $\mathbb{Z}_q$ for any q. Finally, since that overflow bit c may leak information on the secret, we encode it using Boolean masking. These two sharings (arithmetic and Boolean) form together a hybrid sharing.

Definition 1 (First-order hybrid sharing)**.** Let $s \in \{0, \ldots, K-1\}$[1], $s_0, s_1 \in \{0, \ldots, K-1\}$ and $c_0, c_1 \in \{0, 1\}$. The tuple (s_0, s_1, c_0, c_1) is a first-order hybrid sharing of s if

$$s = s_0 + s_1 - K(c_0 \oplus c_1).$$

The hybrid sharing is *uniform* if (s_0, s_1) is a uniform arithmetic sharing of s in $\mathbb{Z}_K$ and if the Boolean sharing (c_0, c_1) is uniform and independent of (s_0, s_1), conditioned on the overflow bit $c = c_0 \oplus c_1$.

Example. For example, let $K = 8$, and $s = 5$. If the arithmetic part of the sharing is $(s_0, s_1) = (2, 3)$, then there is no overflow and $c = 0$ (hence (c_0, c_1) must be $(0, 0)$ or $(1, 1)$). However if we take $s_0 = 6$, then $s_1 = 7$ (the hybrid sharing definition implies $s = s_0 + s_1 \bmod K$), and there is an overflow, giving the carry bit $c = 1$.

Let us now show why it is important to mask c.

Proposition 1. *In a uniform hybrid sharing (s_0, s_1, c_0, c_1) of s modulo K, the distribution of the overflow bit $c = c_0 \oplus c_1$ depends on s.*

Proof. As noted above, $s_0 + s_1 = s \mod K$. Therefore, if $s_0 \leq s$, then $s_1 = s - s_0$ and $c = 0$. Conversely, if $s_0 > s$, $s_1 = K + s - s_0$ and $c = 1$. Moreover, since (s_0, s_1) is a uniform sharing of s in $\mathbb{Z}_K$, s_0 is uniform in $\mathbb{Z}_K$ and independent of s. As a result $\Pr[c = 1] = \Pr[s_0 > s] = (K - 1 - s)/K$. □

Proposition 2. *In a uniform hybrid sharing (s_0, s_1, c_0, c_1) of s modulo K, for any $i, j \in \{0, 1\}$, the hybrid share (s_i, c_j) tuple is uniformly distributed over $\mathbb{Z}_K \times \{0, 1\}$ and independent of s.*

Proof. For all $i, j \in \{0, 1\}$, we marginalize with respect to c_{1-j} the conditional independence assumption for a uniform hybrid sharing to observe that c_j is independent of (s_0, s_1), since c_j is independent of c. This implies that c_j is independent of (s_i, s). Next, since (s_0, s_1) are uniform arithmetic shares of s, s_i is independent of s. We conclude by remarking that c_j and s_i are uniform. □

[1] If s belongs to another integer interval (e.g., $s \in \{-\mu, \ldots, \mu\}$), we can share a shifted version of s (e.g., $s + \mu \in \{0, \ldots, 2\mu\}$) and handle the subtraction of that offset at a later stage.

Algorithm 3 `AOverflowBit` (`Reg`$(\cdot)$ indicates the need for a register in a hardware implementation to ensure security against glitches.)

Input: Shares $s_0, s_1 \in \mathbb{Z}_K$, with $K = 2^k$.
Output: Shares $c_0, c_1 \in \mathbb{Z}_2$ such that $c_0 \oplus c_1 = (s_0 + s_1) \gg k$.

1: $\gamma, c_1 \xleftarrow{\$} \mathbb{Z}_2$, $r \xleftarrow{\$} \mathbb{Z}_K$
2: Initialize the table C_A using Eq. (1)
3: $T \leftarrow \texttt{Reg}\left((s_0 - r - K\gamma) \bmod 2K\right)$
4: $T_1 \leftarrow \texttt{Reg}\left((T + Kc_1 + s_1) \bmod 2K\right)$
5: $T_l \leftarrow T_1 \bmod K$
6: $T_h \leftarrow T_1 \gg k$
7: $c_0 \leftarrow (T_h + C_A[T_l]) \bmod 2$
8: **return** c_0, c_1

An interesting property of hybrid masking is that it can be viewed as (non-uniform) three shares arithmetic sharing $(s_0, s_1, -Kc)$ of s in any integer group, with the twist that the bit c is encoded with Boolean sharing. We now discuss how to generate hybrid shares from arithmetic shares modulo K. This will be followed by an introduction to the gadget that exploits this property to compute any linear operation on a hybrid masked value, and outputs arithmetically masked shares.

4.2 Hybrid shares generation

Let us first consider the problem of generating uniform hybrid shares from arithmetic shares (s_0, s_1) modulo K. We can directly use s_0 and s_1 as the arithmetic part of the hybrid shares and only have to compute fresh Boolean shares of $c = \lfloor \frac{s_0+s_1}{K} \rfloor$ without leaking any information about $s = s_0 + s_1 \mod K$. In other words, we want Boolean shares of the overflow bit of the summation $s_0 + s_1$.

In this section, we propose two solutions for this problem. The first one is inspired by the table-based Arithmetic-to-Boolean (A2B) technique of [CT03] that can evaluate any function on arithmetic shares in $\mathbb{Z}_K$ and provides its result as an arithmetic sharing in $\mathbb{Z}_{K'}$. The second one is based on the masked modular addition of [BC22] in which the carry is computed in order to perform the modular reduction. Although the second approach is slower compared to the first one, it is more compact.

Table-based technique. We develop a hardware version of the A2B algorithm of [CT03] (using $n = 1$ nibble, hence the fix of [Deb12] is not needed) that generates only the overflow bit. In the following, $K = 2^k$. Our `AOverflowBit` (Algorithm 3) takes two arithmetic shares, $(s_0, s_1) \in \mathbb{Z}_K^2$ and outputs $(c_0, c_1) \in \mathbb{Z}_2^2$ such that $c_0 \oplus c_1 = (s_0 + s_1) \gg k$. The gadget uses three uniform random values, $r \in \mathbb{Z}_K$ and $\gamma, c_1 \in \mathbb{Z}_2$, to hide the sum of the two shares and their carry. Firstly, a table C_A with K entries is generated (it is implemented as a RAM). For a ranging from zero to $K - 1$, each entry is computed as

$$C_A[a] = \begin{cases} \gamma, & \text{if } a < K - r, \\ \gamma \oplus 1, & \text{otherwise.} \end{cases} \tag{1}$$

Finally, the share c_0 is computed from the result of the table lookup and the overflow bit of the index computation.

The idea behind this gadget is that since we cannot compute $s_0 + s_1$ explicitly and compare it to K, we sample randomness r, compute $(s_0 - r) + s_1$ and compare it to $K - r$. Since the result of the comparison must be masked, we take the random bit γ as a Boolean mask for the result of the comparison, which is implemented as a table lookup.

Proposition 3. *`AOverflowBit` (Algorithm 3) is correct.*

Proof. Let us remark that $T_l = (s_0 + s_1 - r) \bmod K$ and that $T_h = \lfloor((s_0 + s_1 - r) \bmod 2K)/K\rfloor \oplus \gamma \oplus c_1$. Since $-K < s_0 + s_1 - r < 2K$, we identify three cases. In the first case, $-K < s_0 + s_1 - r < 0$, which implies that $s_0 + s_1 < K$, and $T_l = s_0 + s_1 - r + K$, therefore $C_A[T_l] = \gamma \oplus 1$, while $T_h = 1 \oplus \gamma \oplus c_1$, and as a result $c_0 = c_1$, which is correct. In the second case $K \leq s_0 + s_1 - r < 2K$, we have $s_0 + s_1 \geq K$ and $T_l = s_0 + s_1 - r - K$, and the proof is similar to the first case. The third case $0 \leq s_0 + s_1 - r < K$ has to be split in two sub-cases $s_0 + s_1 < K$ and $s_0 + s_1 \geq K$, which are analyzed similarly to the other cases. □

Proposition 4. *`AOverflowBit` is first-order glitch-robust probing secure. Its output shares are uniformly distributed and independent of the input shares, conditioned on the unmasked output $c = c_0 \oplus c_1$.*

Proof. Let us assume that (s_0, s_1) are uniform arithmetic shares of s modulo K. s_0 and s_1 are the arithmetic shares of s.

We will show that any glitch-extended probe on an intermediate value is independent of s. First, the pre-computation of the table C_A does not depend on any input share. Then, the computation of T depends on only one input share, hence it is independent of s. Additionally, $r + K \cdot \gamma$ is uniformly distributed in $\{0, \ldots, 2K - 1\}$, therefore T is independent of s_0. Next, in the computation of T_1, T_l and T_h, s_0 does not appear (except through T, but it is independent of s_0 since r is uniform[1]), hence all these intermediate values are independent of s. In the table lookup, we assume that the RAM does not glitch (i.e., does not read multiple cells in a single cycle), therefore this only leaks the address T_l and the read value (γ or $\gamma \oplus 1$). However, T_l is independent of γ (due to the selection of the LSBs in the reduction modulo K), hence leaking both values at once is still independent of s. Finally, in the computation of c_0, T_h is independent of γ thanks to the addition of c_1 in its computation, and therefore it is independent of $C_A[T_l]$, ensuring that the computation is independent of s.

Lastly, it is clear that (c_0, c_1) are uniform shares due to the independent sampling of c_1. This also ensures the independence of (s_0, s_1), conditioned on c. □

This proof relies crucially on glitch-free memory access, which has been empirically validated by the experiments shown in Section 6.

Adder-based technique. We now sketch an adder-based algorithm of `AOverflowBit`. This variant does not use a table, hence it does not require the instantiation of a RAM (it can be fully implemented using registers and combinatorial logic), and moreover its size is $\mathcal{O}(k)$ instead of $\mathcal{O}(2^k)$ of the table-based gadget. However, the adder-based gadget has a minimum latency of $k + 1$ clock cycles, in contrast with the 3 cycles for Algorithm 3 (assuming one cycle latency for RAM access).

When $K = 2^k$, we remark that $(s_0, 0)$ and $(0, s_1)$ are two k-bit (non-uniform) Boolean shares of s_0 and s_1, respectively. The masked overflow bit can then be obtained with the Boolean-masked k-bit adder described in Algorithm 4. We use the ripple-carry adder of [BC22], and adapt it to the hardware design context by unrolling the loops and using the HPC2 masked hardware AND gadget [CGLS21, CS21], instead of the software PINI1 [CS20].

Algorithm 4 instantiates k carry-generating units (i.e., full-adders that compute only the carry-out) whose inputs are the shares $a = (s_0[i], 0)$, $b = (0, s_1[i])$ and $c = (x_0[i], x_1[i])$. The carry-out shares are computed as $a \oplus ((a \oplus b)\,\&(a \oplus c))$. By using the AND gate in this expression with a `AND-HPC2` gadget, we achieve a latency of one cycle w.r.t. the carry-in bit c (and two cycles with respect to the input a), leading to a total adder latency of $k + 1$

[1] Let us remark that, beyond the probing model, we should avoid horizontal attacks that recover r, that is, re-generate the table C_A often enough that an adversary is not able to recover r.

Algorithm 4 `AOverflowBit-Adder`

Input: Shares $s_0, s_1 \in \mathbb{Z}_K$, with $K = 2^k$.
Output: Shares $c_0, c_1 \in \mathbb{Z}_2$ such that $c_0 \oplus c_1 = (s_0 + s_1) \gg k$.
1: $(x_0[0], x_1[0]) \leftarrow (0, 0)$
2: **for** $i = 0, \ldots, k-1$ **do**
3: $\quad (y_0, y_1) \leftarrow (s_0[i], s_1[i])$
4: $\quad (z_0, z_1) \leftarrow (s_0[i] \oplus x_0[i], x_1[i])$
5: $\quad (t_0, t_1) \leftarrow$ `AND-HPC2`$((y_0, y_1), (z_0, z_1))$
6: $\quad (x_0[i+1], x_1[i+1]) \leftarrow (t_0 \oplus s_0[i], t_1)$
7: $(c_0, c_1) \leftarrow (x_0[k], x_1[k])$
8: **return** c_0, c_1

Algorithm 5 `MskMux` (multiplexer with the masked select bit.)

Input: Boolean sharing $(\boldsymbol{c}_0, \boldsymbol{c}_1)$ with $\boldsymbol{c}_0, \boldsymbol{c}_1 \in \mathbb{Z}_2^l$ representing $\boldsymbol{c} = \boldsymbol{c}_0 \oplus \boldsymbol{c}_1$, $\boldsymbol{r}_1, \boldsymbol{r}_2 \in \mathbb{Z}_q^l$.
Output: $\boldsymbol{z}[i] = \boldsymbol{c}[i] \,?\, \boldsymbol{r}_2[i] : \boldsymbol{r}_1[i]$ for $i = 0, \ldots, l-1$.
1: **for** $i = 0, \ldots, l-1$ **do**
2: $\quad \boldsymbol{u}_0[i] \leftarrow \mathtt{Reg}\,(\boldsymbol{c}_1[i] \,?\, \boldsymbol{r}_2[i] : \boldsymbol{r}_1[i])$
3: $\quad \boldsymbol{u}_1[i] \leftarrow \mathtt{Reg}\,(\boldsymbol{c}_1[i] \,?\, \boldsymbol{r}_1[i] : \boldsymbol{r}_2[i])$
4: $\quad \boldsymbol{z}[i] \leftarrow \mathtt{Reg}\,(\boldsymbol{c}_0[i] \,\&\, \boldsymbol{u}_1[i]) \mid \mathtt{Reg}\,(\,!\boldsymbol{c}_0[i] \,\&\, \boldsymbol{u}_0[i])$
5: **return** $\boldsymbol{z}$

cycles. Since Algorithm 4 is a straight port to the hardware of the masked adder of [BC22], its correctness proof trivially carries over, and our choice of AND gadget ensures that the security proof also remains valid (and covers glitches), since HPC2 satisfies the PINI security property, like PINI1 (see [CGLS21] for details). Finally, `AOverflowBit-Adder` can be generalized to non power-or-two K by using the modular adder of [CGLS21].

4.3 Hybrid masking of scalar linear functions

The `HybridLin` gadget (Algorithm 6) computes $L(x)$ where L is a linear function (e.g., multiplication by a constant value) that maps $\{0, \ldots, K-1\}$ to the group $\mathbb{Z}_q$. The gadget takes as input hybrid shares and outputs uniform arithmetic shares in $\mathbb{Z}_q$. The core idea of its construction is a masked Mux gadget (Algorithm 5) that takes as input the Boolean shares $(\boldsymbol{c}_0, \boldsymbol{c}_1)$ of length-l vectors and outputs uniform arithmetic shares $(\boldsymbol{v}_0, \boldsymbol{v}_1)$ of $K \cdot L(1)$ if $\boldsymbol{c} = 1$, or 0 if $\boldsymbol{c} = 0$.[1] Note that operations presented in Algorithm 5 are generic for vectors of length-l. For polynomials, $l = n$, and for scalars, $l = 1$. Then, L can be applied to the arithmetic input shares s_0 and s_1, leading to the output sharing $(L(s_0) + v_0, L(s_1) + v_1)$.

Proposition 5. *If $(s_0, s_1, c_0, c_1) \in \mathbb{Z}_K^2 \times \mathbb{Z}_2^2$ is a hybrid sharing of $x \in \mathbb{Z}_K$ and the function $L : \mathbb{Z}_K \to \mathbb{Z}_q$ is linear, then the output (y_0, y_1) of* `HybridLin` *(Algorithm 6) satisfies $y_0 + y_1 = L(x) \mod q$.*

Proof. We observe that v_0 is equal to $-r_1$ if $c_0 \oplus c_1 = 0$, and to $-r_2$ otherwise. Therefore, since $L(0) = 0$, the output of the masked mux (v_0, v_1) is an arithmetic sharing modulo q of $-K \cdot L(c_0 \oplus c_1)$, (y_0, y_1) is an arithmetic sharing of $L(s_0 + s_1 - K(c_0 \oplus c_1)) = L(x)$. □

Let us remark that Algorithm 5 includes registers ($\mathtt{Reg}\,()$), that are needed to ensure that hardware glitches do not reduce the security order of the gadget, as discussed in the next proof. Additional registers may be used in the implementation (e.g., to allow

[1] Here, we select between two public values, but the masked multiplexer can be extended to select from more arithmetic shares by simply duplicating the logic: select v_0 as either a_0 or b_0 depending on (c_0, c_1), and duplicate that circuit to select v_1 as a_1 or b_1.

Algorithm 6 `HybridLin` (linear operation on hybrid shares.)

Input: Hybrid sharing (s_0, s_1, c_0, c_1) with $s_0, s_1 \in \mathbb{Z}_K$, $c_0, c_1 \in \mathbb{Z}_2$ representing the secret $s \in \mathbb{Z}_K$.
Input: Linear function $L : \mathbb{Z}_K \to \mathbb{Z}_q$.
Output: Arithmetic sharing (y_0, y_1) of $y = L(s)$ with $y_0, y_1 \in \mathbb{Z}_q$.

1: $r_1 \xleftarrow{\$} \mathbb{Z}_q$
2: $r_2 \leftarrow (K \cdot L(1)) + r_1 \mod q$
3: $v_0 \leftarrow -\texttt{MskMux}((c_0, c_1), r_1, r_2)$
4: $v_1 \leftarrow r_1$
5: $y_0 \leftarrow L(s_0) + v_0 \mod q$
6: $y_1 \leftarrow L(s_1) + v_1 \mod q$
7: **return** (y_0, y_1)

pipelining). Besides, the output of `HybridLin` (Algorithm 6) is a uniform sharing in the (typically larger) group $\mathbb{Z}_q$. This is because $L(x)$ is typically not small anymore, hence having a hybrid sharing as the output of the gadget would not be efficient. Next, we prove the security of Algorithm 6 in the glitch-robust probing model [FGP+18].

Proposition 6. *`HybridLin` is first-order glitch-robust probing secure.*[1] *Furthermore, its output is a uniform arithmetic sharing in $\mathbb{Z}_q$.*

Proof. We will prove that any glitch-extended probe is independent of the secret s if the input hybrid sharing is uniform. First, for every variable in the computation of $\boldsymbol{u}_0[i]$ and $\boldsymbol{u}_1[i]$ in Algorithm 5, the only share involved is $\boldsymbol{c}_1[i]$, which is independent of s, as a consequence of Proposition 2. Next, in the computation of v_0 in Algorithm 6, we remark that each $\boldsymbol{u}_0[i]/\boldsymbol{u}_1[i]$ in Algorithm 5 is uniformly distributed in $\mathbb{Z}_K$ and independent of $\boldsymbol{c}_1[i]$. Therefore, $\boldsymbol{c}_0[i]\,\&\,\boldsymbol{u}_1[i]$ and $!\,\boldsymbol{c}_0[i]\,\&\,\boldsymbol{u}_0[i]$ depend on the input share $\boldsymbol{c}_0[i]$ but each of these values is independent of $\boldsymbol{c}_1[i]$. Since only one of $\boldsymbol{c}_0[i]$ and $!\boldsymbol{c}_0[i]$ is non-null, a glitch-extended probe on v_0 depends on $\boldsymbol{c}_0[i]$ and observes either $\boldsymbol{u}_0[i]$ or $\boldsymbol{u}_1[i]$ but not both, therefore it is independent of $\boldsymbol{c}_1[i]$. Finally, a probe on y_0 depends on s_0 and $\boldsymbol{c}_0[i]$ (that is independent of s, by Proposition 2), and similarly, y_1 only depends on s_1, which is independent of s.

To prove the uniformity of the output sharing, it suffices to remark that r_1 is a fresh uniform random value in $\mathbb{Z}_q$, and $(y_0, y_1) = (L(s_0) - K \cdot L(c_0 \oplus c_1) - r_1, L(s_1) + r_1)$. □

The security against transitions [FGP+18] of `HybridLin` depends on its actual implementation, hence cannot be proven at an algorithmic level. Let us simply note that security against transition is trivially guaranteed if the implementation satisfies the pipeline definition of [CS21] (i.e., every logic gate is used once per execution). In more complex implementations, composition techniques of [CS21] may be used. Another solution is the use of formal verification tools such as maskVerif [BBC+19], SILVER [KSM20] or Coco [HB21].

4.4 Hybrid masking of vector linear functions

The `HybridLin` gadget operates on a single scalar, hence it is fairly limited. We now generalize it to the `HybrindLinV` (Algorithm 7) gadget that computes any vector linear function $L : \mathbb{Z}_K^l \to \mathbb{Z}_q^{l'}$ and takes as input a length-l vector of hybrid sharings. The core idea of this gadget is similar to `HybridLin`: compute $L(\boldsymbol{s}_0)$, $L(\boldsymbol{s}_1)$ and use `MskMux` to handle the computation of $K \cdot L(\boldsymbol{c}_0 \oplus \boldsymbol{c}_1)$. We exploit the linearity of L to write the latter expression in the canonical basis: $L(\boldsymbol{c}_0 \oplus \boldsymbol{c}_1) = \sum_{i=0}^{l-1} L(\boldsymbol{e}_i) \cdot (\boldsymbol{c}_0[i] \oplus \boldsymbol{c}_1[i])$ (see Algorithm 7

[1] We actually prove that the gadget is a glitch-robust probe isolating non-interference (PINI) [CS20], which allows for easy composition of this gadget with other gadgets.

Algorithm 7 `HybridLinV` (vector linear operation on hybrid shares.)

Input: Hybrid sharing $(\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{c}_0, \boldsymbol{c}_1)$ with $\boldsymbol{s}_0, \boldsymbol{s}_1 \in \mathbb{Z}_K^l$, $\boldsymbol{c}_0, \boldsymbol{c}_1 \in \mathbb{Z}_2^l$ representing the secret $\boldsymbol{s} \in \mathbb{Z}_K^l$.
Input: Linear function $L : \mathbb{Z}_K^l \to \mathbb{Z}_q^{l'}$.
Output: Arithmetic sharing $(\boldsymbol{y}_0, \boldsymbol{y}_1)$ of $\boldsymbol{y} = L(\boldsymbol{s})$ with $\boldsymbol{y}_0, \boldsymbol{y}_1 \in \mathbb{Z}_q^{l'}$.

for $i = 0, \ldots, l-1$ **do**
  $r \xleftarrow{\$} \mathbb{Z}_q$
  $\boldsymbol{r}_1 \leftarrow (K \cdot L(\boldsymbol{e}_i)) + \{r\}^{l'} \mod q$ $\quad \{\boldsymbol{e}_i[i] = 1$ and $\boldsymbol{e}_i[j] = 0$ for all $j \neq i\}$
  $\boldsymbol{V}_0[i] \leftarrow -\texttt{MskMux}((\{\boldsymbol{c}_0[i]\}^{l'}, \{\boldsymbol{c}_1[i]\}^{l'}), \{r\}^{l'}, \boldsymbol{r}_1)$
  $\boldsymbol{V}_1[i] \leftarrow \{r\}^{l'}$
$\boldsymbol{v}_0 \leftarrow \sum_{i=0}^{l-1} \boldsymbol{V}_0[i] \mod q$
$\boldsymbol{v}_1 \leftarrow \sum_{i=0}^{l-1} \boldsymbol{V}_1[i] \mod q$
$\boldsymbol{y}_0 \leftarrow \texttt{Reg}\,(\boldsymbol{v}_0 + L(\boldsymbol{s}_0) \bmod q) + L(\boldsymbol{s}_1) \bmod q$
$\boldsymbol{y}_1 \leftarrow \boldsymbol{v}_1$
return $(\boldsymbol{y}_0, \boldsymbol{y}_1)$

for the definition of $\boldsymbol{e}_i$). This expression can in turn be evaluated using l `MskMux` gadgets, each being l'-element wide.

On top of this base construction, we introduce an optimization to reduce randomness usage and save some computation. Indeed, the shares of output coordinates of `HybridLinV` do not have to be independent if no computation involves these together (this condition is often satisfied in lattice-based cryptography). Therefore, we can use the same randomness for all coordinates. For simplicity, we keep a vector accumulator $\boldsymbol{V}_1$, however during implementation we use a scalar accumulator instead of a vector accumulator. Finally, since $\boldsymbol{v}_0$ is freshly randomized, we can compute the output sharing as $(\texttt{Reg}\,(\boldsymbol{v}_0 + L(\boldsymbol{s}_0)) + L(\boldsymbol{s}_1), \boldsymbol{v}_1)$, resulting in a common share for all output coordinates, which spares some logic.

Proposition 7. *If $(\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathbb{Z}_K^{2l} \times \mathbb{Z}_2^{2l}$ is a hybrid sharing of $\boldsymbol{x} \in \mathbb{Z}_K^l$ and the function $L : \mathbb{Z}_K^l \to \mathbb{Z}_q^{l'}$ is linear, then the output $(\boldsymbol{y}_0, \boldsymbol{y}_1)$ of `HybridLinV` satisfies $\boldsymbol{y}_0 + \boldsymbol{y}_1 = L(\boldsymbol{x}) \mod q$.*

Proof. We observe that $\boldsymbol{V}_0[i]$ is equal to $\{-r\}^{l'}$ if $(\boldsymbol{c}_0[i] \oplus \boldsymbol{c}_1[i]) = 0$, and to $-\boldsymbol{r}_1$ otherwise. Therefore, thanks to the linearity of L, $\boldsymbol{v}_0 = -K \cdot L(\boldsymbol{c}_0 \oplus \boldsymbol{c}_1) - \boldsymbol{v}_1 \mod q$. As a result, $(\boldsymbol{v}_0, \boldsymbol{v}_1)$ is an arithmetic sharing modulo q of $-K \cdot L(\boldsymbol{c}_0 \oplus \boldsymbol{c}_1)$ (taking the vector $\boldsymbol{v}_1$ as the second share of all the elements). The final part of the proof is identical to the proof of Proposition 5. □

Let us now prove the security of `HybridLinV`.

Proposition 8. *`HybridLinV` is first-order glitch-robust probing secure.*[1] *In addition, each of its output coordinates is a uniform arithmetic sharing in $\mathbb{Z}_q$.*

Proof. The proof is almost identical to the proof of Proposition 6. The first difference is that a glitch-extended probe on $\boldsymbol{v}_0$ may reveal a coordinate of multiple $\boldsymbol{V}_0[i]$. This is not a problem since the computation of every $\boldsymbol{V}_0[i]$ uses fresh randomness, hence the arguments in the proof of Proposition 6 apply. The other difference lies in the output computation: every coordinate of $\boldsymbol{v}_0$ is a coordinate of $-K \cdot L(\boldsymbol{c}_0 \oplus \boldsymbol{c}_1)$ masked with the uniform random r. Therefore, each output coordinate is uniform sharing, and each coordinate of $\texttt{Reg}\,(\boldsymbol{v}_0 + L(\boldsymbol{s}_0) \bmod q)$ is uniform, hence $\texttt{Reg}\,(\boldsymbol{v}_0 + L(\boldsymbol{s}_0) \bmod q) + L(\boldsymbol{s}_1) \bmod q$ can be simulated knowing only $\boldsymbol{s}_1$. □

[1] As for `Hybridlin`, `HybridLinV` is actually glitch-robust probe isolating PINI.

Algorithm 8 `HybridPolyMul` (hybrid shared polynomial multiplication with a public polynomial.)

Input: Hybrid sharing $(\boldsymbol{s_0}, \boldsymbol{s_1}, \boldsymbol{c_0}, \boldsymbol{c_1})$ with $\boldsymbol{s_0}, \boldsymbol{s_1} \in \mathcal{R}_K$, $\boldsymbol{c_0}, \boldsymbol{c_1} \in \mathcal{R}_2$ representing the secret polynomial $\boldsymbol{s} \in \mathcal{R}_q$.
Input: Public polynomial $\boldsymbol{a} \in \mathcal{R}_q$.
Output: Arithmetic sharing $(\boldsymbol{y_0}, \boldsymbol{y_1})$ of $\boldsymbol{y} = \boldsymbol{a} \times \boldsymbol{s}$ with $\boldsymbol{y_0}, \boldsymbol{y_1} \in \mathcal{R}_q$.

for $i = 0, \ldots, n-1$ **do**
  $r \xleftarrow{\$} \mathbb{Z}_q$
  for $j = 0, \ldots, n-1$ **do**
    $\boldsymbol{r}_1[j] \leftarrow (-1)^{\lfloor (i+j)/N \rfloor} (K \cdot \boldsymbol{a}[i]) + r \mod q$
  $\boldsymbol{V}_0[i] \leftarrow -\texttt{MskMux}((\{\boldsymbol{c}_0[i]\}^n, \{\boldsymbol{c}_1[i]\}^n), \{r\}^n, \boldsymbol{r}_1)$
  $\boldsymbol{V}_1[i] \leftarrow \{r\}^n$
$\boldsymbol{v}_0 \leftarrow \sum_{i=0}^{n-1} \boldsymbol{V}_0[i] \mod q$
$\boldsymbol{v}_1 \leftarrow \sum_{i=0}^{n-1} \boldsymbol{V}_1[i] \mod q$
$\boldsymbol{y}_0 \leftarrow \texttt{Reg}\,(\boldsymbol{v}_0 + (\boldsymbol{a} \times \boldsymbol{s}_0) \bmod q) + (\boldsymbol{a} \times \boldsymbol{s}_1) \bmod q$
$\boldsymbol{y}_1 \leftarrow \boldsymbol{v}_1 \bmod q$
return $(\boldsymbol{y}_0, \boldsymbol{y}_1)$

4.5 Polynomial multiplication with hybrid masking

In this section, we will use hybrid masking to compute the product $\boldsymbol{a} \times \boldsymbol{s}$ where $\boldsymbol{s}$ is a small secret polynomial and $\boldsymbol{a}$ is a large public polynomial.

Let the small coefficients of the secret polynomial $\boldsymbol{s}$ be in $\mathbb{Z}_K$. Starting from arithmetic sharing of $\boldsymbol{s}$ in $\mathbb{Z}_K$, we transform it to a hybrid sharing using `AOverflowBit` (Algorithm 3). We then use the `HybridLinV` gadget (Algorithm 7) with multiplication by $\boldsymbol{a}$ as the linear function L. This algorithm uses two multiplications ($\boldsymbol{a} \times \boldsymbol{s_0}$ and $\boldsymbol{a} \times \boldsymbol{s_1}$) that can be computed using any polynomial multiplication algorithm (these can take advantage of the smallness of the coefficients of $\boldsymbol{s_0}$ and $\boldsymbol{s_1}$ to achieve better performance or area or both). Then, we have to perform the computation on Boolean shares: `MskMux` and additions. In the remaining part of this section, we introduce a small tweak (similar to [RB20]) to `HybridLinV` that exploits specific properties of polynomial multiplication to ease the hardware implementation and security proof.

Our tweak, implemented in the `HybridPolyMul` gadget (Algorithm 8), is based on the following observation: let $L(\boldsymbol{s}) = \boldsymbol{a} \times \boldsymbol{s}$, `HybridLinV` computes $\sum_{i=0}^{n-1} \boldsymbol{a} \times (x^i \cdot (\boldsymbol{c_0}[i] \oplus \boldsymbol{c_1}[i]))$. Assuming that one term of this sum is computed and accumulated per clock cycle, this requires having the full $\boldsymbol{a}$ available at all clock cycles, as well as one $(\boldsymbol{c_0}[i], \boldsymbol{c_1}[i])$ single-bit sharing. If $\boldsymbol{a}$ is stored in a RAM and $\boldsymbol{c_0}, \boldsymbol{c_1}$ in registers (which is the case in the implementation that we use as the starting point for our implementation [RB20]), then it is better to perform the computation as $\sum_{i=0}^{n-1} \boldsymbol{a}[i] \cdot (x^i \times (\boldsymbol{c_0} \oplus \boldsymbol{c_1}))$. Another advantage of this approach is that it manipulates in parallel all the shares instead of a single one, which typically leads to better side-channel security. In such an implementation, the computation of $x^i \times (\boldsymbol{c_0} \oplus \boldsymbol{c_1})$ can be performed with a cyclic shift register (for negacyclic ring, with a bit to encode positive or negative sign).

For conciseness, we avoid repeating the proofs of correctness and security for `HybridPolyMul`, since they are identical to the ones of `HybridLinV`, except for the functional change explained above, which does not have any impact on the security proof.

Performance and Area Advantage To bring forth the advantage of this method, we will now present a comparison with the naive way of using small coefficient multipliers. We provided a hybrid gadget, but instead, a naive way of using a small coefficient multiplier is to split one of the coefficients into smaller parts. This technique is used in applications like homomorphic encryption. If we assume there is no hybrid gadget, then naive masking would

split small secret $\boldsymbol{s} \in \mathcal{R}_K$ into two large shares $\boldsymbol{s}_0, \boldsymbol{s}_1 \in \mathcal{R}_q$. Now, if we want to multiply these with the public polynomial $\boldsymbol{a} \in \mathcal{R}_q$ using the compact multipliers, one option would be to split $\boldsymbol{s}_0$ and $\boldsymbol{s}_1$ base K (e.g., $\boldsymbol{s}_0 = \boldsymbol{s}_{0,r} \cdot K^r + \boldsymbol{s}_{0,r-1} \cdot K^{r-1} + \cdots + \boldsymbol{s}_{0,1} \cdot K + s_{0,0}$). Now, this requires r multiplications, where r is the number of words when a coefficient in $\mathbb{Z}_q$ is sliced into smaller words in base K ($r = \lceil(\log_2(q)/\log_2(K))\rceil$). When $K = 2^4$ and $q = 2^{13}$, r will be 4. Thus, we will require four multiplications per share, that is a total of eight multiplications for a first-order masked implementation. By using the proposed hybrid gadget, this is reduced to three multiplications only, where one of the multiplications is just an accumulation and does not require a multiplier. Instead, if a user decides to pay the cost in terms of area and save the runtime, then $\log_2(q) \times \log_2(q)$ multipliers would be required. These consume 4.5× more look-up tables compared to $\log_2(q) \times \log_2(K)$ multiplier and will require two multiplications. Thus, using hybrid shares gives us a significant advantage over state-of-the-art methods.

Higher-order generalization. We remark that the hybrid masking technique can be generalized to higher-order masking, with the following differences: with d shares $s_i \in \mathbb{Z}_K$, the overflow c has to be represented using multiple bits, and each of the d Boolean shares c_i is a $\lceil\log_2(d)\rceil$-bit word (i.e., $c_i \in \{0, \ldots, d-1\}$), such that

$$s = \sum_{i=0}^{d-1} s_i - K \cdot \bigoplus_{i=0}^{d-1} c_i.$$

The definition of uniform shares is unchanged: $(s_i)_{i=0,\ldots,d-1}$ must be a uniform sharing of s in $\mathbb{Z}_K$ and $(c_i)_{i=0,\ldots,d-1}$ must be a uniform Boolean sharing of c, independent of $(s_i)_{i=0,\ldots,d-1}$, conditioned on c. Note that the multiplication with the Boolean shares of the hybrid sharing (Algorithm 5) is expected to have $O(d^2)$ complexity. Hence, for asymptotically high masking orders, this algorithm will have worse performance than directly operating in $\mathbb{Z}_q$ ($O(d)$ cost). This also serves as an interesting future scope for the proposed technique.

In this paper, we focus on optimizing the first-order masking and leave the implementation of the higher-order as future work (note that `AOverflowBit-Adder` already works at arbitrary order). We use the Table-based algorithm `AOverflowBit` since it has the lowest latency. Next, we will introduce an efficient hardware instantiation of `HybridPolyMul`.

5 Implementation of masked polynomial multiplication

In Section 4, we presented an algorithmic optimization technique for masked and efficient implementation of polynomial multiplication in lattice-based public-key algorithms. Experimental evaluation is needed to validate the efficiency of the proposed algorithms. In this section, we present a first-order DPA-protected hardware implementation of polynomial multiplication using the hybrid masking scheme.[1] This proof-of-concept architecture works in $\mathcal{R}_q = \mathbb{Z}_{2^{13}}/(x^{256} + 1)$ and the unprotected secret polynomial's coefficients are in $[-4, 4]$.

The first step of the multiplication is loading the secret polynomial $\boldsymbol{s}$, which has its coefficients in $[-4, 4]$. We use a shifted representation where $\mu = 4$ is added to each coefficient, resulting in unsigned coefficients in the range $[0, 8]$ (this shift will be compensated later to preserve the correctness of the result). For the sake of simplicity, our

[1] Note that the masking technique (non-hybrid) in Section 3 for NTT-based polynomial multiplication in NTT-unfriendly rings does not require a separate implementation and evaluation, as it does not lead to architectural challenges (we already give the implementation cost in Section 3 by adapting a well-known architecture). Regarding security, the NTT routine is evaluated twice in isolation on the two secret-share polynomials without requiring any interaction between the shares, therefore no order reduction can happen if the state is properly cleared between the two multiplications.

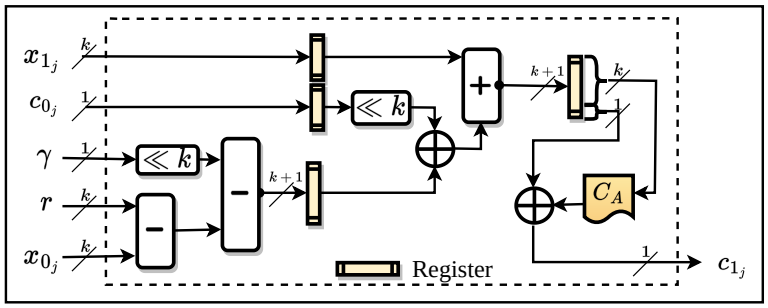

Figure 1: Architecture diagram illustrating the `AOverflowBit` unit

implementation receives as input the uniform arithmetic shares $(\boldsymbol{s_0}, \boldsymbol{s_1})$ modulo K of the shifted $\boldsymbol{s}$, where $K = 2^4$ and $n = 256$. The initial sharing of this long-term secret is out of the scope of our implementation. As a second step, we convert the input arithmetic sharing into hybrid sharing by computing the Boolean shares $\boldsymbol{c_0}, \boldsymbol{c_1}$, using a pipelined implementation of Algorithm 3, as illustrated in Fig. 1. Then, we implement the core part of the multiplication, as described in Algorithm 8.

Our implementation is based on the open-source hardware library provided by [RB20]. The polynomial multiplier in this library uses a heavily-parallel schoolbook algorithm by instantiating 256 small-area multiply-and-accumulate (MAC) units in parallel in the hardware for the polynomial ring $\mathcal{R}_q = \mathbb{Z}_{2^{13}}/(x^{256}+1)$. Moreover, it relies on the property that one operand polynomial is always small (e.g., with coefficients in $[-4, 4]$), and then it optimizes the MAC units to reduce the area consumption. This multiplier is, therefore, a natural starting point for our implementation since our hybrid-shares polynomial multiplication algorithm is based on small-coefficient polynomial multiplications.

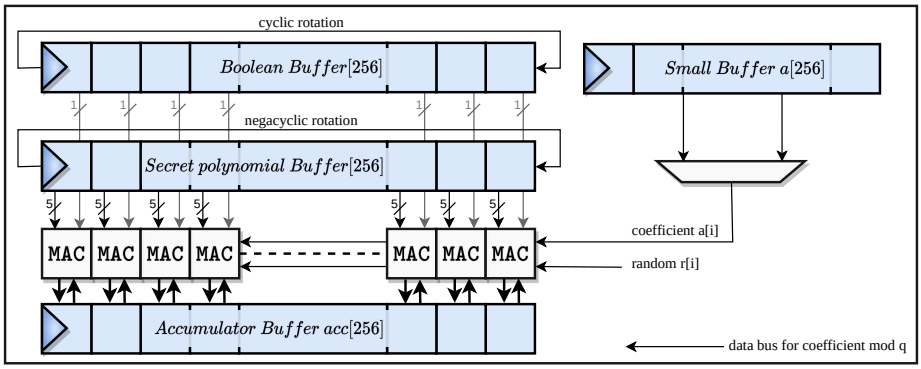

Figure 2: Polynomial multiplier architecture for the hybrid masking scheme.

Fig. 2 shows the high-level architecture diagram of our schoolbook polynomial multiplier. The two main differences with the schoolbook multiplier of [RB20] are the changes in the MAC unit to support the Boolean-masked multiplication $\boldsymbol{a} \times (\boldsymbol{c_0} \oplus \boldsymbol{c_1})$ on top of the normal multiplication (used for $\boldsymbol{a} \times \boldsymbol{s_0}$ and $\boldsymbol{a} \times \boldsymbol{s_1}$), and the addition of the Boolean secret polynomial buffer to store $\boldsymbol{c_1}$ ($\boldsymbol{c_0}$ is stored in the other secret polynomial buffer). The multiplier operates in three steps, following the requirements of Algorithm 8: (1) computation of $\boldsymbol{a} \times (\boldsymbol{c_0} \oplus \boldsymbol{c_1})$ (first share in the accumulator buffer, and the second share $\boldsymbol{v_1}$ in a separate buffer not shown on Fig. 2), (2) accumulation of $\boldsymbol{a} \times \boldsymbol{s_0}$ in the accumulator buffer, and (3) accumulation of $\boldsymbol{a} \times \boldsymbol{s_1}$ in the accumulator buffer.

We start with the description of steps (2) and (3), which work identically to the unpro-

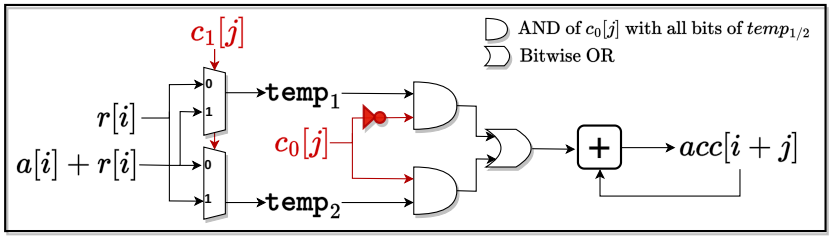

Figure 3: Secure multiply-and-accumulate unit for computing $acc[i+j] = acc[i+j]+c[j]\cdot a[i]$ using Boolean shares $c0[j], c1[j]$ of $c[j]$. Registers for glitches are not shown.

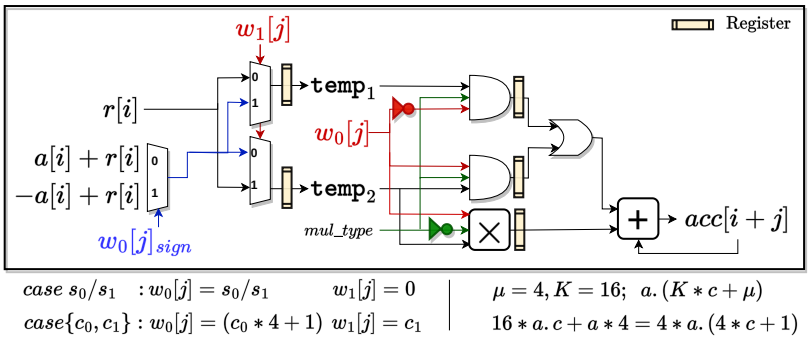

Figure 4: MAC (multiply and accumulate) unit architecture diagram. It computes multiplication for both Boolean as well as arithmetic shares.

tected multiplication in [RB20]. Before starting a multiplication, the entire polynomial $\boldsymbol{s_i}$ is loaded into the secret buffer of Fig. 2. It is a shift register that can do a negacyclic rotation, computing $\boldsymbol{s_i} \leftarrow x \times \boldsymbol{s_i} \bmod (x^{256}+1)$; the coefficients are stored in the buffer in the signed magnitude representation. During the actual polynomial multiplication, in every cycle, one coefficient of $\boldsymbol{a}$ (say $\boldsymbol{a}[j]$) is multiplied by all coefficients of $\boldsymbol{s_i}$, and then the results are accumulated (addition or subtraction depending on the sign of $\boldsymbol{s_i}[j]$) in the accumulator buffer. In the next cycle, the secret polynomial buffer is shifted, and the process is repeated with the new coefficient $\boldsymbol{a}[j+1]$. For a 256-coefficient polynomial multiplication, 256 cycles are required, excluding the overhead of data loading. The accumulator buffer is *not* reset before or after the $\boldsymbol{s_i}$ multiplication, and the '`Boolean buffer`' is not used.

For the first step, we compute a sharing of $-\boldsymbol{a} \times (K \cdot \boldsymbol{c} + \{\mu\})$ where $\{\mu\}$ is a polynomial with all coefficients set to μ. It compensates for the original shift and proceeds as follows. The polynomial $(K \cdot \boldsymbol{c_0} + \{\mu\})$ is loaded in the regular secret polynomial buffer (the multiplication and addition is just a bit concatenation), and the Boolean polynomial $\boldsymbol{c_1}$ is loaded in the 256-bit buffer '`Boolean buffer`' which is also a circular shift register. To process the Boolean coefficients, the normal MAC unit of [RB20] is augmented with the secure Boolean unit of Fig. 3, implementing the `MskMux` logic of Algorithm 8. This unit is integrated into the original MAC unit, with proper sign handling and registers to prevent glitches, which gives Fig. 4. In this area-optimized combined unit, a signal `mul_type` selects between unmasked operation (for steps (2) and (3)) and masked operation.

Let us finally discuss the security of our implementation. We rely on the security arguments of Section 4 for `HybridPolyMul` and remark that our implementation optimizations do not break the security of this gadget. Indeed, the set of glitch-extended probes for the extended MAC unit is the same as for the basic one, and using the same register for $\boldsymbol{s_0}$ and $\boldsymbol{c_0}$ is not an issue since even jointly, these shares are still independent of the secret.

Table 1: Implementation results of the design on Xilinx Artix-7 xc7a100tftg256-3 for the parameters $n = 256, K = 2^4$, and $q = 2^{13}$.

Masking	Algorithm	Area (LUT/FF/DSP/BRAM)	Frequency (MHz)	Poly. Mul. Latency (clock cycles)
Proposed	`AOverflowBit`	590/1,014/0/1	130	92
Proposed	`HybridPolyMul`	25,214/13,280/0/1	130	1020
Naive	`PolyMul`	62,163/8,801/0/1	125	752
Unmasked	`PolyMul`	17,429/5,083/0/1	250	336

Moreover, this unit is pipelined, and hence, the transitions happen between independently masked variables (except for the adder in front of the accumulator, but this only adds old values of the accumulator into the extended probes, which is not an issue, as it contains only one of the two shares). Next, we discuss the results of the implementation.

5.1 Implementation results

We implemented the proposed architecture design on Xilinx Artix-7 xc7a100tftg256-3 FPGA using Xilinx Vivado 2019.1. The synthesis and implementation strategies were kept to Vivado defaults. The `KEEP_HIERARCHY` flag was set in the security-critical components of the modules to avoid unwanted optimizations. The complete design achieves a clock frequency of $\approx$ 130 MHz. The implementation of Algorithm 3 (`AOverflowBit`) illustrated in Fig. 1, which generates the Boolean shares, consumes 590 LUTs, 1,014 FFs, 0 DSPs, and 1 BRAM. The C_A table (Eq. (1)) is pre-generated and stored before processing each polynomial, however, it can be refreshed more often. Its latency is 92 clock cycles for processing one set of polynomials. The complete core polynomial multiplication unit shown in Fig. 2 consumes 25,214 LUTs and 13,280 FFs. It can process one polynomial multiplication in 340 clock cycles and consumes 1020 clock cycles to multiply the hybrid shares of a secret polynomial with a public polynomial. From Table 1, it is evident that the area-time increase of the proposed technique with respect to the unmasked implementation is much less compared to the naive masking (the secret is split into two shares and the multiplication requires big multipliers).

Our masked polynomial multiplication implementation consumes 1.7× more area than the unmasked implementation in [RB20] (17,429 LUTs and 5,083 FFs). This is because we have instantiated $n = 256$ `MskMux` combined with 256 compact multipliers. Therefore, adding the registers to every MAC unit for glitch prevention further contributed to the area consumption. However, this is an implementation choice, and if one `MskMux` is instantiated for the multiplication with the Boolean shares, there will be no area overhead. Regarding time overhead, one unmasked polynomial multiplication translates to two multiplications and one accumulation in the masked domain. As discussed earlier (Section 4.5), the total area-time overhead of this design is less than the naive masked implementations.

6 Side-channel evaluation

In this section, we discuss the side-channel evaluation setup and results. Our setup is based on a ChipWhisperer NAE-CW305-04-7A100-0.10-X[1] board which features an Artix-7 FPGA in the FTG256 Package, and an Atmel ATSAM2U32-AU microcontroller that acts as a USB to control FPGA, as well as a trigger for starting the computations. We generate the bitsream with Vivado 2019.1. The FPGA uses a 10 MHz clock signal generated by the onboard PLL and crystal. The power consumption is measured with a 5 Ω shunt resistor,

[1] https://media.newae.com/datasheets/NAE-CW305_datasheet.pdf

amplified on-board with a 20 dB low-noise amplifier, and measured with the Picoscope 6000. The oscilloscope is synchronized with the FPGA clock and acquires 312.5 MS/s.

As mentioned in the previous section, our implementation is based on the Module-KEM design provided in [RB20]. It implements Saber, and the implementation can multiply several polynomials at once and accumulate them, depending on the polynomial vector dimension. For example, a public polynomial vector $\boldsymbol{A}$, of dimension three will have three polynomials $\boldsymbol{a}[0], \boldsymbol{a}[1], \boldsymbol{a}[2]$. When this is multiplied with the secret polynomials vector $\boldsymbol{S}$ of dimension three, the multiplication result of $\boldsymbol{A} \times \boldsymbol{S}$ is $\boldsymbol{a}[0] \times \boldsymbol{s}[0] + \boldsymbol{a}[1] \times \boldsymbol{s}[1] + \boldsymbol{a}[2] \times \boldsymbol{s}[2]$. Since we have an accumulator, we can keep accumulating the results on the fly for all three multiplications without having to do this as a separate operation, which avoids additional time overhead.

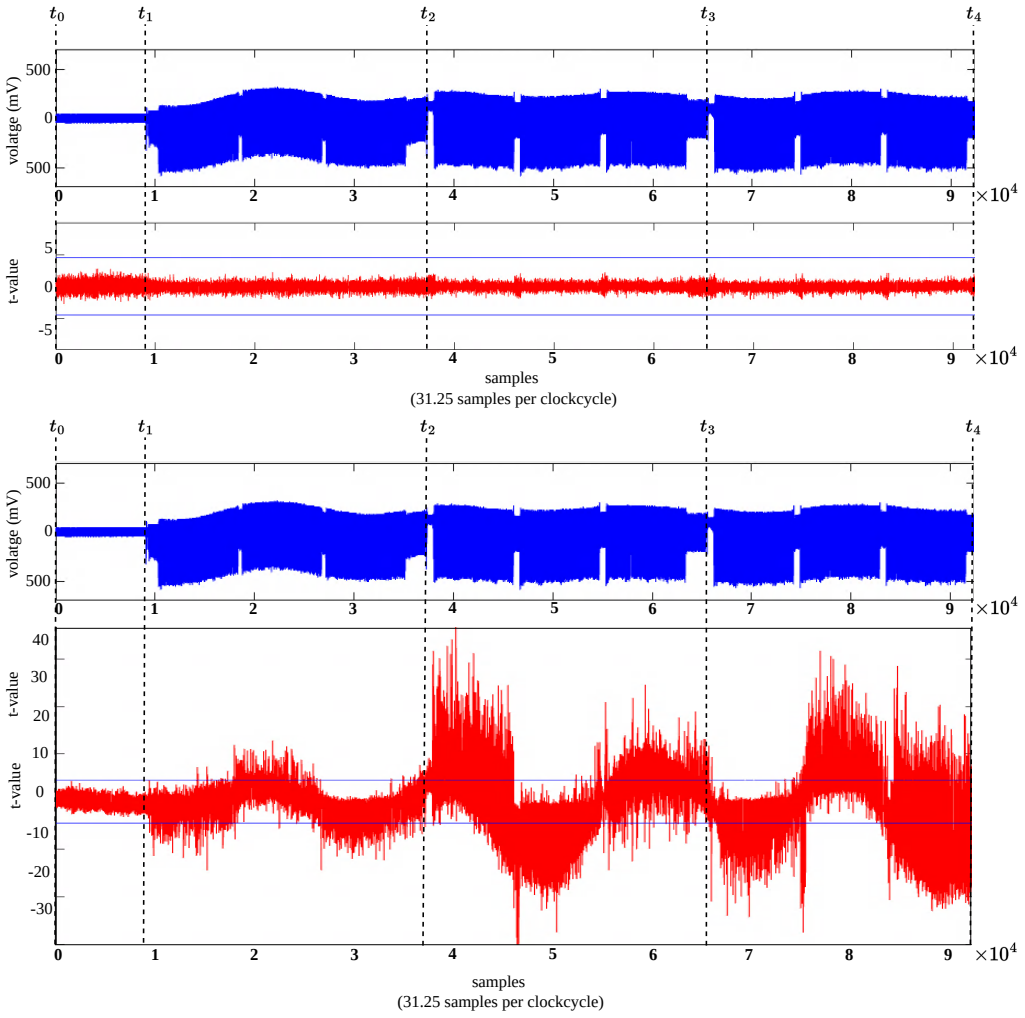

Figure 5: Raw measurement data of `AOverflowBit` and polynomial multiplication of the hybrid shares. **Top:** t-test figures with mask ON for 100,000 traces; **Bottom:** t-test figures with mask OFF for 200 traces.

T-test evaluation. We measure first the `AOverflowBit` which generates shares $\boldsymbol{C_0}$ and $\boldsymbol{C_1}$ for polynomial vectors $\boldsymbol{S_0}$ and $\boldsymbol{S_1}$, where both $\boldsymbol{S_0}$ and $\boldsymbol{S_1}$ are secret vectors consisting of three polynomials. Then, this is followed by multiplication of $\boldsymbol{C_0}, \boldsymbol{C_1}$ with public

polynomial vector $\boldsymbol{A}$ of dimension three. Next, we multiply $\boldsymbol{A}$ with $\boldsymbol{S_0}$ and finally with $\boldsymbol{S_1}$. A hardware implementation of the stream cipher Trivium is used as a source of randomness.

We employ the Test Vector Leakage Assessment (TVLA) method introduced in [GJJR11] to validate the security of our implementation. Namely, we run a first-order fixed-versus-random T-test with 100,000 traces for the secret shared variable $\boldsymbol{S}$ ($\boldsymbol{A}$ is kept fixed). The results are reported in Fig. 5: no first-order leakage is detected for the commonly-used significance threshold of ± 4.5 for the T-statistic. As a sanity check, we also ran the first-order t-test with 200 traces and randomness masking disabled (Fig. 5), which shows clear first-order leakage. In Fig. 5, the timestamps $t_0 - t_4$ show the different operations evaluated by the test. The interval t_0–t_1 is the generation of shares $\boldsymbol{C_0}$ and $\boldsymbol{C_1}$ for secret vectors $\boldsymbol{S_0}$ and $\boldsymbol{S_1}$. Then, the multiplication of $\boldsymbol{A}$ with $\boldsymbol{C_0}, \boldsymbol{C_1}$ is performed during the interval t_1–t_2. This is followed by multiplication of $\boldsymbol{A}$ with $\boldsymbol{S_0}$ and $\boldsymbol{S_1}$, during the t_2–t_3 and t_3–t_4 intervals, respectively.

7 Conclusion and future work

In this work, we proposed an efficient masking method for polynomial multiplication of NTT-unfriendly schemes that can benefit from one of the operands being small. Not only does it improve the masking of the current NTT-unfriendly schemes, but it also promotes the design of new schemes. The proposed hybrid gadget helps retain the small secret property and saves significant area in masked implementations. Most of the lattice-based schemes have small secrets (ternary or binomial or Gaussian distributed). They can benefit from hybrid masking unless the scheme makes NTT representation an integral part of the scheme. Our contributions are therefore directly applicable to implementations of algorithms such as SABER [BMD+20], Scabbard [MKKV21], NTRU [HPS06] or LAC [LLZ+18]. In the long term, our results should aid future lattice-based scheme design choices (in particular the power-of-two versus NTT-friendly trade-off).

This work raises further research questions. A first direction that would be interesting to explore is the use of RSA/ECC coprocessors for masking lattice-based PQC. Since these are widely deployed, they are an attractive target for implementing PQC (especially when retrofitting PQC into existing systems). Non-masked implementations for Saber and Kyber have been proposed in [GMR21], using the Kronecker substitution method, and the most efficient variants exploit the property that one of the polynomials involved in the product always has small coefficients, both for NTT-friendly and unfriendly rings. Our hybrid gadgets (e.g. `HybridLinV` or `HybridPolMul`) could therefore be used, and they can simply use the algorithms of [GMR21] for the polynomial multiplication. For the `MskMux` (Algorithm 5) part of these gadgets, many possible software implementations depend on the available instructions: the most natural solution is the use of a conditional move instruction, but arithmetic solutions are also possible (using 0/1 multiplications, then additions), or even using the bitwise logical operations (emulating our hardware implementation).

Secondly, we used a schoolbook multiplier for hybrid masking implementation (Section 5). However, we emphasize that this is not the only type of multiplication that can be used with this masking technique: other multiplication types, such as the Karatsuba or Toom-Cook multipliers, can also be used. The integration of these multipliers with the Boolean masking part of the hybrid masking, as we do with our integrated MAC unit, is a future direction for optimization work.

Acknowledgement

This work was supported in part by Semiconductor Research Corporation (SRC) and the State Government of Styria, Austria – Department Zukunftsfonds Steiermark. We thank the anonymous reviewers for their useful suggestions and comments. We also thank Fan Zhang for shepherding the paper.

References

[AAB+19] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, and many more. Quantum supremacy using a programmable superconducting processor. Nature, 2019. https://doi.org/10.1038/s41586-019-1666-5.

[ACC+21] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, Nov. 2021.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum Key Exchange - A New Hope. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 327–343. USENIX Association, 2016.

[AMD+21] Abubakr Abdulgadir, Kamyar Mohajerani, Viet Ba Dang, Jens-Peter Kaps, and Kris Gaj. A lightweight implementation of saber resistant against side-channel attacks. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *Progress in Cryptology – INDOCRYPT 2021*, pages 224–245, Cham, 2021. Springer International Publishing.

[AMJ+22] Aikata, Ahmet Can Mert, David Jacquemin, Amitabh Das, Donald Matthews, Santosh Ghosh, and Sujoy Sinha Roy. A unified cryptoprocessor for lattice-based signature and key-exchange. *IEEE Transactions on Computers*, pages 1–13, 2022.

[BAD+21] Andrea Basso, Furkan Aydin, Daniel Dinu, Joseph Friel, Avinash Varna, Manoj Sastry, and Santosh Ghosh. Where Star Wars Meets Star Trek: SABER and Dilithium on the Same Polynomial Multiplier. Cryptology ePrint Archive, Paper 2021/1697, 2021. https://eprint.iacr.org/2021/1697.

[BBC+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *ESORICS (1)*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):553–588, 2022.

[BDK+18] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 353–367, 2018.

[BDK+21a] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium. Proposal to NIST PQC Standardization, Round3, 2021. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[BDK+21b] Michiel Van Beirendonck, Jan-Pieter D'anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel-resistant implementation of saber. *J. Emerg. Technol. Comput. Syst.*, 17(2), apr 2021.

[BMD+20] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Sujoy Sinha Roy Angshuman Karmakar, Michiel Van Beirendonck, and Frederik Vercauteren. SABER: Mod-LWR based KEM (Round 3 Submission to NIST PQC), 2020.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[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 new speed records for saber and NTRU on cortex-m4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):159–188, 2021.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.

[CS21] Gaëtan Cassiers and François-Xavier Standaert. Provably secure hardware masking in the transition- and glitch-robust probing model: Better safe than sorry. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):136–158, 2021.

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

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. A new algorithm for switching from arithmetic to boolean masking. In *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 89–97. Springer, 2003.

[Deb12] Blandine Debraize. Efficient and provably secure methods for switching from arithmetic to boolean masking. In *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2012.

[DMG23] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. High-speed hardware architectures and FPGA benchmarking of crystals-kyber, ntru, and saber. *IEEE Trans. Computers*, 72(2):306–320, 2023.

[DPPvW22] Léo Ducas, Eamonn W. Postlethwaite, Ludo N. Pulles, and Wessel P. J. van Woerden. Hawk: Module LIP makes lattice signatures fast, compact and simple. *IACR Cryptol. ePrint Arch.*, page 1155, 2022.

[FBR+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):414–460, 2022.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual international cryptology conference*, pages 537–554. Springer, 1999.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(4):239–280, 2020.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Joshua Jaffe, and Pankaj Rohatgi. A testing methodology for side channel resistance. 2011.

[GMR21] Aurélien Greuet, Simon Montoya, and Guénaël Renault. On using RSA/ECC coprocessor for ideal lattice-based key exchange. In Shivam Bhasin and Fabrizio De Santis, editors, *Constructive Side-Channel Analysis and Secure Design - 12th International Workshop, COSADE 2021, Lugano, Switzerland, October 25-27, 2021, Proceedings*, volume 12910 of *Lecture Notes in Computer Science*, pages 205–227. Springer, 2021.

[HB21] Vedad Hadzic and Roderick Bloem. COCOALMA: A versatile masking verifier. In *FMCAD*, pages 1–10. IEEE, 2021.

[HPS06] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. Ntru: A ring-based public key cryptosystem. In *Algorithmic Number Theory: Third International Symposiun, ANTS-III Portland, Oregon, USA, June 21–25, 1998 Proceedings*, pages 267–288. Springer, 2006.

[IBM] IBM. Expanding the IBM Quantum roadmap to anticipate the future of quantum-centric supercomputing. https://research.ibm.com/blog/ibm-quantum-roadmap-2025.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[KDB+22] Suparna Kundu, Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. Higher-order masked saber. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings*, volume 13409 of *Lecture Notes in Computer Science*, pages 93–116. Springer, 2022.

[KO62] A Karatsuba and Yu Ofman. Multiplication of many-digital numbers by automatic computers. *Dokl. Akad. Nauk SSSR*, 145(2):293–294, 1962.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In *ASIACRYPT (1)*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[LFK+19] Weiqiang Liu, Sailong Fan, Ayesha Khalid, Ciara Rafferty, and Máire O'Neill. Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(10):2459–2463, 2019.

[LLZ+18] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, and Bao Li. LAC: practical ring-lwe based public-key encryption with byte-level modulus. *IACR Cryptol. ePrint Arch.*, page 1009, 2018.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin Heidelberg, 2010.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[MKKV21] Jose Maria Bermudo Mera, Angshuman Karmakar, Suparna Kundu, and Ingrid Verbauwhede. Scabbard: a suite of efficient learning with rounding key-encapsulation mechanisms. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):474–509, 2021.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MP21] Michele Mosca and Marco Piani. 2021 quantum threat timeline report: Global risk institute. Technical report, evolutionQ, 2021. available at https://globalriskinstitute.org/publication/2021-quantum-threat-timeline-report-global-risk-institute-global-risk-institute/.

[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*, 2018(1):142–174, Feb. 2018.

[RB20] Sujoy Sinha Roy and Andrea Basso. High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):443–466, Aug. 2020.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-lwe implementation. In *CHES*, pages 683–702. Springer, 2015.

[Sho97] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5):1484–1509, Oct 1997.

[SR19] Sujoy Sinha Roy. Saberx4: High-throughput software implementation of saber key encapsulation mechanism. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 321–324, 2019.