Where Star Wars Meets Star Trek: SABER and Dilithium on the Same Polynomial Multiplier

Andrea Basso[1,2], Furkan Aydin[3,4], Daniel Dinu[3], Joseph Friel[3], Avinash Varna[3], Manoj Sastry[2] and Santosh Ghosh[2]

[1] University of Birmingham, UK
[2] Intel Labs, Intel Corporation, USA
[3] IPAS, Intel Corporation, USA
[4] North Carolina State University, USA

Abstract. Secure communication often require both encryption and digital signatures to guarantee the confidentiality of the message and the authenticity of the parties. However, post-quantum cryptographic protocols are often studied independently. In this work, we identify a powerful synergy between two finalist protocols in the NIST standardization process. In particular, we propose a technique that enables SABER and Dilithium to share the exact same polynomial multiplier. Since polynomial multiplication plays a key role in each protocol, this has a significant impact on hardware implementations that support both SABER and Dilithium. We estimate that existing Dilithium implementations can add support for SABER with only a 4% increase in LUT count. A minor trade-off of the proposed multiplier is that it can produce inexact results with some limited inputs. We thus carry out a thorough analysis of such cases, where we prove that the probability of these events occurring is near zero, and we show that this characteristic does not affect the security of the implementation.

We then implement the proposed multiplier in hardware to obtain a design that offers competitive performance/area trade-offs. Our NTT implementation achieves a latency of 519 cycles while consuming 2,012 LUTs and only 331 flip-flops when implemented on an Artix-7 FPGA. We also propose a shuffling-based method to provide side-channel protection with low overhead during polynomial multiplication. Finally, we evaluate the side-channel security of the proposed design on a Sakura-X FPGA board.

Keywords: Polynomial multiplication · SABER · Dilithium

1 Introduction

Since Peter Shor developed a quantum algorithm to factorize integers and solve discrete log problems in 1997 [Sho97], the advent of quantum computers poses a threat against the majority of widely used cryptosystems. While publicly available quantum computers are still far from being powerful enough to break RSA or Diffie-Hellman as used today, the high pace of technological development might close the gap soon. It is thus urgent to migrate current cryptosystems to quantum-resistant protocols. To bootstrap the development and the analysis of such protocols, in 2016 the American National Institute of Standards and Technology (NIST) started a post-quantum cryptography standardization process [oST]. The process is now in its third round, and the first batch of selected standards is expected to be announced at the beginning of 2022. In the current round, there are four finalist key encapsulation mechanisms (KEMs), Classic McEliece, Kyber, NTRU, and SABER, and three finalist digital signature protocols, Dilithium, Falcon, and Rainbow. These

protocols rely on different mathematical primitives and offer a diverse range of performance, communication costs, and additional properties.

The protocols submitted to the NIST competition have been studied and analyzed by the cryptographic community, both in their theoretical guarantees as well as in their implementational aspects. Most implementation studies have researched the protocols independently of each other, and they have not considered whether it is possible to optimize different schemes to work with each other. It is of particular interest to research the possible synergies between KEMs and digital signatures. This is because a wide array of applications requires implementing both. A classic example is the TLS handshake [Res18] that guarantees the security of HTTPS as well as many other internet communications. Another example is the Signal protocol [MP] that underlies the security of private messaging for billions of people. More generally, whenever secure communication is needed between authenticated parties, both encryption and signatures are used.

In this work, we aim to close this research gap and present a powerful synergy between a finalist KEM protocol and a finalist digital signature scheme. In particular, we make the following contributions:

1. We propose a technique that computes polynomial multiplication in SABER and Dilithium using exactly the same multiplier. Since polynomial multiplication makes up the majority of computations in both protocols, reusing the same multiplier can have significant benefits for hardware implementations that support both protocols.

2. The proposed multiplier has a non-zero failure rate, i.e. there are specific inputs that cause the multiplier to produce an inexact result. We thoroughly study such cases and prove that they happen with negligible probability. We also show that this does not affect the security of the system.

3. We design a hardware implementation of the polynomial multiplication based on the NTT. The resulting design achieves comparable performance with multipliers reported in the literature, while reducing the area consumption and achieving a high clock frequency.

4. We propose a constant-time Fisher-Yates shuffling-based protection technique that increase side-channel security with low overheads. We also perform a side-channel evaluation to validate the security of the design.

The choice of protocols   The reason why we propose a strong synergy between SABER and Dilithium is not casual, since we argue that these protocols are the most compatible among the round 3 finalists of the NIST standardization process. If we consider all protocols in the last round, there are four KEM schemes and three digital signature protocols. One KEM (Classic McEliece) and one signature (Rainbow) rely on mathematical primitives that do not have a correspondent in the other category, which makes them ill-fitted to share a significant part with other protocols. The remaining protocols are all lattice-based, with three (Kyber, SABER, and Dilithium) being based on the hardness of the Learning With Error (LWE) problem, and the remaining two (NTRU, Falcon) based on the NTRU problem. While further research is needed to evaluate the possibility of a unique multiplier for both NTRU and Falcon, the two protocols have significantly different polynomial multiplication characteristics. This makes them unlikely to be able to share the same multiplier. Lastly, among the LWE-based protocols, Kyber and Dilithium have incompatible multiplication strategies because both protocols prescribe the use of the NTT for polynomial multiplication, but each protocol requires different moduli and different techniques. Conversely, SABER does not require any particular multiplication algorithm, which allows it to adapt to use the Dilithium polynomial multiplier.

Related work The literature does not report any fully in-hardware implementation that supports both post-quantum encryption and digital signatures. A recent work by Fritzmann et al. [FBR$^+$] proposes a software/hardware codesign that accelerates many building blocks for a wide variety of post-quantum protocols, but each protocol uses dedicated logic within the NTT to support different prime moduli. There exist several reported implementations that support a single protocol. The first complete hardware implementations of Dilithium [RMJ$^+$21] brings high speed results at a significant area consumption cost, while a more recent implementation by Land et al. [LSG21] achieves better trade-offs. A software/hardware codesign for Dilithium has also been reported by Zhou et al. [ZHL$^+$21], but the hardware accelerator only focuses on Keccak and polynomial multiplication. For SABER, several implementations have been reported. The first hardware implementation [RB20] computes polynomial multiplication using the hardware-friendly schoolbook method. The same design was later improved by [BR20] and eventually transformed into an ASIC design [IAR$^+$21] that can achieve high clock frequencies. Another ASIC implementation by Zhu et al. [ZZY$^+$21] achieves high performance by computing polynomial multiplication with the Karatsuba algorithm. The application of the NTT in SABER is first reported by Chung et al. [CHK$^+$20], which was used for a software implementation. The only hardware implementations of SABER that rely on the NTT are two hardware/software codesigns: the already mentioned accelerator by Fritzmann et al. [FBR$^+$] and RISQ-V [FSS20], a RISC-V accelerator for post-quantum cryptography. Lastly, shuffling-based countermeasures against side-channel attacks on the NTT are discussed by Ravi et al. [RPBC20] in software and by Zijlstra et al. [ZBT19] in hardware. Recently, [AMJ$^+$21] proposed a unified hardware accelerator that supports both Dilithium and SABER. The accelerator uses the same NTT core for both protocols, but unlike this work it relies on different prime moduli for SABER and Dilithium.

Paper organization The paper introduces some preliminaries in Section 2, including a brief description of SABER and Dilithium. Then, Section 3 presents a technique to use the exact same multiplier in both protocols, while the implementation of such a multiplier is reported in Section 4. Section 5 proposes a low-overhead countermeasure against side-channel attacks based on shuffling, and the results of our implementation and side-channel analysis are reported in Section 6.

2 Preliminaries

2.1 Notation

Several operations in this work act on polynomials. We denote polynomials with lowercase letters as c, or when confusion may arise as $c(x)$. In the context of SABER, we often refer to a generic public polynomial in the matrix $\boldsymbol{A}$ as a or $a(x)$. Similarly, secret polynomials are denoted as s or $s(x)$. Vectors and matrices of polynomials are referred in bold typeface and respectively lowercase ($\boldsymbol{b}$) and uppercase letters ($\boldsymbol{A}$).

The uniform distribution over the range $[0, x)$ or the range $[-x, x]$ (in the case of secrets) is represented as $\mathcal{U}_x(\cdot, \cdot)$. The argument of the function denotes the size of the matrix returned, where the matrix entries are polynomials with coefficients uniformly distributed. With a slight abuse of notation, when only one argument is present, the function returns a vector of polynomials. The centered binomial distribution is represented by $\beta_\mu(\cdot)$, where the argument represents the length of the vector returned, and the vector elements are polynomials with binomially distributed coefficients. A centered binomial distribution with parameter μ is a discrete distribution over the interval $[-\mu/2, \mu/2]$, with

---

Algorithm 1: SABER KeyGen

1 $seed_{\boldsymbol{A}} \leftarrow \mathcal{U}_1(1)$
2 $\boldsymbol{A} = \mathsf{gen}(seed_{\boldsymbol{A}})$
3 $\boldsymbol{s} = \beta_\mu(l)$
4 $\boldsymbol{b} = \lfloor \frac{1}{8}\boldsymbol{A}^T\boldsymbol{s} \rceil$
5 **return** $pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$, $sk = \boldsymbol{s}$

Algorithm 3: Saber Encryption

1 $\boldsymbol{A} = \mathsf{gen}(seed_{\boldsymbol{A}})$
2 $\boldsymbol{s'} = \beta_\mu(l)$
3 $\boldsymbol{b'} = \lfloor \frac{1}{8}\boldsymbol{A}\boldsymbol{s'} \rceil$
4 $v' = \boldsymbol{b}^T\boldsymbol{s'}$
5 $c_m = \mathsf{embed}(v', m)$
6 **return** $c = (c_m, \boldsymbol{b'})$

Algorithm 2: SABER Decryption

1 $v = \boldsymbol{b'}^T\boldsymbol{s}$
2 $m' = \mathsf{extract}(v, c_m)$
3 **return** m'

---

the probability density function given by

$$p(X = x) = \frac{\mu!}{(\mu/2 + x)!(\mu/2 - x)!}2^{-\mu}. \tag{1}$$

Since a centered binomial distribution represents the distribution of the sum of μ random bits, the sum of multiple centered binomial variables B_i with parameter μ_i is again a centered binomial random variable with parameter $\mu' = \sum_i \mu_i$.

For modular reduction, we use $x \bmod p$ to denote the usual modular reduction that takes in input a value x in the range $[0, (p-1)^2]$ and outputs a value $x' \in [0, p-1]$ such that $x' \equiv x \bmod p$. Centered modular reduction is denoted by $\bmod^\pm$ and outputs a value in the range $\left[-\lfloor\frac{p-1}{2}\rfloor, \lfloor\frac{p}{2}\rfloor\right]$[1]. Throughout the paper, q denotes the Dilithium prime $2^{23} - 2^{13} + 1$. Additional notation and constants used in the protocol descriptions are explained in their corresponding specifications [BMD$^+$20, BDK$^+$20], to which we refer for a more complete treatment of SABER and Dilithium.

The Number Theoretical Transform (NTT) is an adaptation of the Fast Fourier Transform in the finite field domain and it is used to achieve the asymptotically fastest polynomial multiplication algorithm. We refer to the forward NTT as $\mathrm{NTT}(\cdot)$, and we usually denote the resulting polynomial in the NTT domain with a tilde, so that $\tilde{a} = NTT(a)$. We write the inverse NTT as $\mathrm{INTT}(\cdot)$ and the coefficient-wise multiplication between $\tilde{a}$ and $\tilde{s}$ as $\tilde{a} \circ \tilde{s}$. For a more complete treatment of the NTT, we refer to [CHK$^+$20, Sec. 2], [POG15], and [LN16].

2.2 SABER

SABER [BMD$^+$20] is a key encapsulation mechanism (KEM) based on the Module Learning With Rounding (Mod-LWR) problem. Most of its operations act on polynomials. Matrices and vectors have polynomials as their elements, and all polynomials have 256 coefficients. The coefficients are integers modulo 2^{13} or 2^{10}. A key characteristic of SABER is its usage of power-of-two moduli. This greatly simplifies the rounding operation, which becomes a simple addition and bitshift, and it increases the performance of side-channel resistant implementations, but it prevents a straightforward application of the NTT to compute polynomial multiplication.

The key generation (Algorithm 1) of the public-key encryption protocol mainly consists of generating a LWR sample. A second LWR sample is computed during encryption (Algorithm 3), and the new secret is used to compute a polynomial v'. The message is then embedded into this polynomial, which makes up the ciphertext together with the generated LWR sample. Lastly, the message is decrypted (Algorithm 2) by recomputing the

---

[1]We use this interval, rather than the more common $\left[-\lfloor\frac{p}{2}\rfloor, \lfloor\frac{p-1}{2}\rfloor\right]$ to follow Dilithium's specification.

Algorithm 4: Dilithium KeyGen

1 $\boldsymbol{A} = \mathcal{U}_q(k, l)$
2 $\boldsymbol{s_1} = \mathcal{U}_\eta(l)$
3 $\boldsymbol{s_2} = \mathcal{U}_\eta(k)$
4 $\boldsymbol{t} = \boldsymbol{A}\boldsymbol{s_1} + \boldsymbol{s_2}$
5 **return** $pk = (\boldsymbol{A}, \boldsymbol{t}), sk = (\boldsymbol{s_1}, \boldsymbol{s_2})$

Algorithm 5: Dilithium Verify

1 $\boldsymbol{w_1'} = \mathsf{HighBits}(\boldsymbol{A}\boldsymbol{z} - \boldsymbol{c}\boldsymbol{t})$
2 **if** $\mathsf{Valid}_z(\boldsymbol{z})$ **and** $c == \mathsf{H}(M \parallel \boldsymbol{w_1'})$ **then**
3 | **return** 1
4 **else**
5 | **return** 0

Algorithm 6: Dilithium Sign

1 $\boldsymbol{z} = \perp$
2 **while** $\boldsymbol{z} == \perp$ **do**
3 | $\boldsymbol{y} = \mathcal{U}_{\gamma_1}(l)$
4 | $\boldsymbol{w_1} = \mathsf{HighBits}(\boldsymbol{A}\boldsymbol{y})$
5 | $c(x) = \mathsf{H}(M \parallel \boldsymbol{w_1})$
6 | $\boldsymbol{z} = \boldsymbol{y} + c\boldsymbol{s_1}$
7 | **if** *not* $\mathsf{Valid}_z(\boldsymbol{z})$ *or*
   | *not* $\mathsf{Valid}_A(\boldsymbol{A}\boldsymbol{y} - c\boldsymbol{s_2})$ **then**
8 | | $\boldsymbol{z} = \perp$
9 **return** $\sigma = (\boldsymbol{z}, c)$

polynomial v, similar to v', and extracting the message. In Algorithm 2 and Algorithm 3, the function extract is the inverse of embed, i.e. $\mathsf{extract}(v, \mathsf{embed}(v, m)) = m$. The KEM protocol is then obtained by applying the Fujisaki-Okamoto to the public-key encryption scheme.

From an implementation point of view, most of the computation time of SABER is spent on polynomial multiplication. Note that implementers are free to choose the most suitable multiplication algorithm.

2.3 Dilithium

CRYSTALS-Dilithium [BDK+20] is a digital signature protocol that bases its security on the modular learning with error (Mod-LWE) problem. This is similar to the underlying problem of SABER, but Dilithium randomly samples the error term while SABER introduces it deterministically through a rounding operation.

Similarly to SABER, all operations act on polynomials modulo the $\langle x^{256}+1\rangle$ polynomial. The main difference is the use of the prime modulus $q = 2^{23} - 2^{13} + 1$, which enables fast multiplication via the NTT. Since the matrix $\boldsymbol{A}$ is generated directly in the NTT domain, it is not possible to use alternative multiplication algorithms.

The key generation of Dilithium (Algorithm 4) consists of generating a Mod-LWE sample. When signing (Algorithm 6), Dilithium generates a random vector $\boldsymbol{y}$ and derives a polynomial c based on the message and the product $\boldsymbol{w_1} = \boldsymbol{A}\boldsymbol{y}$. The signature then consists of the polynomial c and the vector $\boldsymbol{z}$, obtained as $\boldsymbol{z} = \boldsymbol{y} + c\boldsymbol{s_1}$, if these values pass some tests that guarantee correctness and security. If not, the procedure is repeated with a different random vector $\boldsymbol{y}$ until the final checks succeed. On average, Dilithium requires 5.1 repetitions until a valid signature is computed at security level 3. Lastly, verification (Algorithm 5) recomputes $\boldsymbol{w_1}$ and considers a signature valid only if the vector z satisfies the signing checks and c can be recomputed from the message and $\boldsymbol{w_1}$. Note that the protocol depicted in Algs. 4, 5, 6 is simplified for clarity and does not contain message compression. Similarly to SABER, polynomial multiplication takes up the majority of computation times, since it is required in most operations that are not sampling or hashing.

3 Polynomial multiplier strategy

Polynomial multiplication plays a key role in both SABER and Dilithium. The main similarities and differences in how multiplication is computed in the two protocols are

represented in Table 1. We see that both protocols use polynomials with 256 coefficients and compute multiplication modulo $\langle x^{256} + 1 \rangle$. However, the similarities end here. The two protocols use different moduli (10/13-bit long vs 23) and SABER relies on power-of-two moduli, rather than prime ones, which prevents a straightforward application of the NTT. Lastly, Dilithium generates the public matrix $\boldsymbol{A}$ in the NTT form to reduce computation times. This implies that any implementation of Dilithium is required to use NTT modulo q to compute polynomial multiplication. In SABER, however, implementers are free to choose the most suitable multiplication algorithm.

Table 1: Comparison of polynomial multiplication characteristics in SABER and Dilitihium.

	SABER	**Dilithium**
Polynomial modulus	$\langle x^{256} + 1 \rangle$	$\langle x^{256} + 1 \rangle$
Coefficient modulus	$2^{10}, 2^{13}$ (power-of-two)	$2^{23} - 2^{13} + 1$ (prime)
Secret coeff. range	$[-3, 3], [-4, 4], [-5, 5]$	$[-2, 2], [-4, 4]$
Multiplication algorithm	–	NTT

In the next section, we present a technique to compute polynomial multiplication in both SABER and Dilitihium with the same algorithm. Since it is not possible to deviate from the NTT modulo q technique in Dilitihium, we present a solution that exploits the flexibility of SABER to compute multiplication in SABER by reusing the polynomial multiplier of Dilithium.

3.1   NTT-based multiplication in SABER

The application of the NTT to compute polynomial multiplication modulo $\langle x^{256} + 1 \rangle$ requires that the coefficient ring contains a 512-th root of unity, which implies that the coefficient modulus needs to be prime. This prevents a straightforward application of the NTT in SABER, which uses power-of-two moduli. However, it is possible to avoid this issue by switching to a larger modulus.

In the context of SABER, Chung et al. [CHK$^+$20] proposed such a technique, where they relied on an incomplete NTT with a 25-bit modulus. Another work by Fritzmann et al. [FBR$^+$] has also reported an implementation of SABER with NTT-based multiplication, where larger moduli are used (39-bit long). The technique used in these works involves lifting the coefficients of the polynomials in SABER from $\mathbb{Z}/2^{13}\mathbb{Z}$ to $\mathbb{Z}/q'\mathbb{Z}$, where q' is a larger prime modulus such that $\mathbb{Z}/q'\mathbb{Z}$ contains a 512-th root of unity. It is then possible to compute the NTT-based multiplication and finally reduce the result modulo 2^{13} to obtain the final result.

These techniques use large moduli to avoid precision errors introduced by the modular reduction. In other words, one needs the new modulus to be larger than the maximum possible coefficient in the product polynomial. If that is not the case, coefficients larger than the modulus would be reduced, and the final product would be incorrect due to the incompatibility between different moduli.

Let us now consider what modulus q' should be used in SABER. All polynomials have 256 coefficients, and the public polynomial coefficients lie in the interval $[0, 2^{13} - 1]$, while the secret coefficients range between -4 and 4 at security level 3. A straightforward approach would represent the negative secret coefficients as modulo 2^{13}, so that -1 is represented as $2^{13} - 1$. In this case, the largest product polynomial coefficient is obtained when the public polynomial consists of all coefficients set to $2^{13} - 1$ and all the secret coefficients are equal to -1, so that the product polynomial has a coefficient equal to

$256 \times (2^{13} - 1) \times (2^{13} - 1) = 17{,}175{,}675{,}136$. The NTT modulus q' then needs to be greater than this value, thus at least 34-bit long. More simply, if the coefficients are 13-bit long and there are 2^8 coefficients, the modulus q' needs to be at least $13 + 13 + 8 = 34$-bit long.

It is possible to do better by considering that one operand, the secret polynomial, has small coefficients between -4 and 4. To exploit such smallness, however, signed operations are needed to represent elements such as -4 with 3 bits (plus sign), rather than with 13 bits as $q' - 4$. Thus, we can replace the modular reduction operation with the centered modular reduction, denoted by $\mathrm{mod}^{\pm}$, that maps integer to the interval $[-(q' - 1)/2, +(q'/2)]$, where q' is the modulus. The public polynomial coefficients are thus represented as integers in the interval $[-2^{12} + 1, 2^{12}]$.

This allows us to reduce the size of the prime q', since the maximum absolute value of a coefficient in the product of two polynomials is then $256 \times 4 \times 2^{12}$. To avoid modular reductions, we require that the prime q' satisfies $q'/2 > 256 \times 4 \times 2^{12}$, where $q'/2$ is due to the centered modular reduction. We would thus need a modulus q' that is strictly larger than 2^{23}, hence 24-bit long.

Algorithm 7: NTT-based multiplication for SABER

Data: Polynomials $a(x)$ with $a_i \in [0, 2^{13})$ and $s(x)$ with $s_i \in [-4, 4]$
Result: $a(x)s(x) \bmod \langle x^{256} + 1 \rangle$, with coefficients in $[0, 2^{13})$
1 $a^{\pm}(x) \leftarrow a(x) \bmod^{\pm} 2^{13}$;
2 $\tilde{a}(x) \leftarrow \mathrm{NTT}(a^{\pm})$;
3 $\tilde{s}(x) \leftarrow \mathrm{NTT}(s)$;
4 $\tilde{c}(x) \leftarrow \tilde{a} \circ \tilde{s}$;
5 $c^{\pm}(x) \leftarrow \mathrm{INTT}(\tilde{c})$;
6 $c(x) \leftarrow c^{\pm}(x) \bmod 2^{13}$;
7 **return** $c(x)$

Our proposal We have established that computing polynomial multiplication in SABER via the NTT requires a prime modulus q' such that $q'/2 > 256 \times 4 \times 2^{12}$. The Dilithium prime $q = 2^{23} - 2^{13} + 1$ *almost* satisfies the condition. Indeed, we have $q/2 \approx (256 \times 4 \times 2^{12}) - 2^{12}$. This means that if polynomial multiplication is computed in SABER with the Dilithium modulus, there exist polynomials for which the procedure returns the wrong result. However, such polynomials are extremely rare and, if we accept a negligible failure probability, we can use the exact same Dilithium polynomial multiplier to compute multiplication in SABER without any adaptation.

We are thus proposing a method to compute polynomial multiplication in SABER and Dilithium using exactly the same multiplier, without the need to adapt it for either protocol. The overall computations are represented in Algorithm 7. An important component that enables the unified multiplier is the fact that the NTT (lines 2 and 3), coefficient-wise multiplication (line 4), and inverse NTT (line 5) are computed with signed arithmetic (in two's complement) and a centered modular reduction. Signed arithmetic and the centered modular reduction are used to reduce the size of the modulus that SABER can work with, whereas the usage of two's complement allows to implement lines 1 and 6 practically "for free". Indeed, they do not correspond to any additional computations because the modulus is a power-of-two and that allows to freely move between centered and positive representations. That is because the bitstring representation of a number $n \bmod 2^x$ is the same as that of the two's complement representation of the centered reduction $n \bmod^{\pm} 2^x$. Lastly, note that lines 2 to 5 represent how polynomial multiplication is computed within Dilithium, which shows that the same algorithm can be used unchanged for both SABER and Dilithium.

3.2   Error analysis

We now quantify the error probability when polynomial multiplication in SABER is computed following Algorithm 7. From the analysis in the previous section, we have seen that the multiplication fails when the product polynomial has a coefficient whose absolute value is greater than $q/2 = 2^{22} - 2^{12} + 1/2$.

Firstly, if the secret polynomial has two or more coefficients different than 4 or -4, then there is no public polynomial that can cause any multiplication failure. That is because even if $s(x)$ has 254 coefficients equal to 4 and the two equal to 3, and the public polynomial is maximal (all coefficients equal to 2^{12}), the largest coefficient in the product polynomial is $2^{12} \times (4 \times 254 + 3 \times 2) = 4{,}186{,}112$, which is less than $q/2$. Similarly, we obtain that if the secret coefficient has 255 coefficients equal to 4 and the remaining coefficient equal to 3, the only public polynomial that causes a failure is the one with all coefficients equal to 2^{12}. Lastly, when the secret polynomial consists of only coefficients equal to 4, more public polynomials can lead to failure cases. Indeed, all the public polynomials with all the coefficients greater or equal to $2^{12} - 4$ do so. Moreover, note that generally if polynomials $a(x)$ and $s(x)$ lead to a multiplication failure, so do $-a(x)$ and $-s(x)$, but also $-a(x)$ and $s(x)$, and $a(x)$ and $-s(x)$. This is because the failure cases are determined by the largest absolute value in the product polynomial, thus the sign does not have a significant effect. There is one main exception though, because the asymmetry in the range of the centered modular reduction. Since the coefficient range is $[-(2^{12}-1), 2^{12}]$, we have that $-2^{12} \equiv 2^{12}$ and thus there is no additional failure case.

Given a complete description of the failure cases, it is now possible to compute the probability of these events happening. Let us consider the joint probability distribution over both polynomials. The public polynomial is uniformly distributed, hence the probability that all its coefficients are in the set $\mathcal{S} := \{\pm(2^{12}-4), \pm(2^{12}-3), \pm(2^{12}-2), \pm(2^{12}-1), 2^{12}\}$ is given by

$$p_{pub} = \left(\frac{\#\mathcal{S}}{2^{13}}\right)^{256} \approx 2^{-2516}.$$

The coefficients of the secret polynomial are binomially distributed, which means coefficients such as 4 and -4 are not very common. To simplify the analysis, we only compute the probability of secret polynomials which have at least 255 coefficients equal to 4 or -4, with remaining coefficient equal to 3 or -3. Such a probability can be computed as

$$p_{sec} = \binom{256}{255}(2 \times 2^{-8})^{255} \times (2 \times 8 \times 2^{-8}) + (2 \times 2^{-8})^{256} \approx 2^{-1782}, \tag{2}$$

where 2×2^{-8} is the probability of a coefficient being 4 or -4 (see Eq. 1), $2 \times 8 \times 2^{-8}$ is the probability of a coefficient being 3 or -3, and $\binom{256}{255}$ account for any coefficient permutation.

We can thus conclude that the probability p_{tot} that our proposed multiplication technique fails is the product of the two probabilities, hence

$$p_{tot} = p_{pub} \times p_{sec} = 2^{-2516} \times 2^{-1782} = 2^{-4298}.$$

Such a probability is extremely minimal and can be considered as virtually zero. Note that these computations only provide a upper bound to the failure probability, since not all combinations of secret polynomials and public polynomials considered lead to a failure case.

Lastly, we consider the probability for different security levels. At NIST security level 5, FireSABER uses secret polynomials with coefficients in the range $[-3, 3]$. Thus, the polynomial multiplier is always exact and never fails. At security level 1, however, LightSABER has secret coefficients in the interval $[-5, 5]$, which does lead to a higher failure probability. When the secret polynomial has all coefficients equal to 5, experimental results show that the public polynomials that lead to failures are those with coefficients

whose absolute value is greater than $2^{12} - 823$. This means that $\#\mathcal{S} = 2 \times 823 + 1$ and the probability of obtaining such a polynomial is $p_{pub} = 2^{-592}$. To compute p_{pub}, we only consider the case where the public polynomial is maximal. While this slightly underestimate the failure probability, the additional terms do not affect the final result much and this approach greatly simplifies the analysis. In this case, the largest coefficient in the product polynomial is given by $|\sum a_i s_i| = 2^{12}|\sum s_i|$, where s_i and a_i represent the i-th coefficient of the secret and the public polynomial. Since the multiplier fails when the largest product coefficient is greater than $q/2 = 2^{22} - 2^{12}$, this implies that multiplication failures only occur when the sum of the secret coefficients has absolute value greater than $2^{10} - 1$. The sum of the secret coefficients is distributed according to a central binomial distribution with $\mu' = 256 \times \mu$ (see Section 2.1), where μ is the parameter of the coefficient binomial distribution. This thus gives the probability of a secret polynomial possibly leading to a failure as $p_{sec} = 2^{-1360}$, with the total failure probability equal to $p_{tot} = 2^{-1952}$. Hence, at all security levels the proposed multiplier only fails with near-zero probabilities.

Security considerations   It is important to consider the security implications of using a polynomial multiplier that has a negligible but non-zero failure probability.

Firstly, we note that an attacker can learn some information about the secret polynomial only if the uniformly random public polynomial can lead to a multiplication failure, since otherwise the multiplication does not fail and no information on the secret polynomial is leaked. Thus the probability of an attacker recovering any information is negligible. However, even if the public polynomial has large coefficients and could possibly lead to a multiplication failure, the failure only happens if the secret coefficients has nearly all coefficients with maximal absolute value. Thus an attacker would only learn with overwhelming probability that the secret polynomial does not have 255 or more coefficients equal to 4 or -4. While this does reduce the search space, the reduction is so minimal that it does not lead to any feasible attack.

Moreover, the computed probability considers the distribution over both polynomials. A natural question is whether an attacker may choose a malicious public polynomial to boost the failure probability and possibly gain information on the secret polynomial. This is not possible, because the public polynomial is obtained by expanding an initial seed through a cryptographically secure hash function. This means that it is computationally hard to find a seed such that it expands to a desired value. However, an attacker may construct maliciously formed ciphertexts that are then multiplied with secret polynomials during decryption. Note though that the ciphertext polynomials have 10-bit coefficients, thus the proposed multiplier algorithm would never fail. Furthermore, even if an attacker may influence the public polynomial through fault attacks or other methods, they would not be able to gain significant secret information. The optimal attack would force the public polynomial to be maximal, i.e. one with all coefficients set to 2^{12}, which would increase the multiplier failure probability to p_{sec}. Thus any resulting attack would have a complexity much higher than the claimed security of SABER.

Hence, the negligible probabilities of failure-leading public and secret polynomials prevent any efficient attack and shows that the proposed implementation is secure. The near-zero probability p_{pub} prevents recovering any secret information, while the near-zero probability p_{sec} shows that an attacker may not recover any meaningful information even if they could affect the public polynomial.

3.3   Applications and extendability

The proposed technique allows to compute polynomial multiplication within SABER and Dilithium with the exact same algorithm. This has an important effect on every implementation that needs to support both protocols.

Firstly, both software and hardware implementation can benefit from our proposed technique because it reduces the code base, speeds up development and debugging, and simplifies the development of formally verified implementations. However, a single polynomial multiplier is particularly useful in hardware implementations. Any implementation that supports both protocols would significantly reduce their area consumption. Indeed, in SABER implementations, polynomial multiplication takes up as much as 50% of the overall area, with another 30% being taken up by the Keccak core (see, for instance, [RB20]). Since the Keccak core is already a shared block between the two protocols, with our proposed multiplier a Dilithium+SABER implementation would have only a modest area consumption increase over a Dilithium only implementation. If we consider the most compact hardware implementation of Dilithium [LSG21], we see that adding support for SABER level 3 using the additional building blocks from [RB20] would only require a 4% increase in LUT count and a 15% increase in flip-flop count. The same benefits would also be seen in hardware/software codesign solutions, where polynomial multiplication is often the first block to be hardware-accelerated. Similarly, a Dilithium accelerator that exposes only high-level APIs, such as NTT or entire polynomial multiplication instructions, would be able to accelerate SABER without any additional modification.

The proposed polynomial multiplier approach can thus bring several advantages and significantly improve the performance of hardware implementations. It would be beneficial to extend this to more protocols. However, the proposed technique relies on both the specific design of SABER and Dilithium as well as their parameter choices. It thus does not seem possible to extend this approach to more protocols. Kyber, which shares many similarities with both SABER and Dilithium, seems to be a good candidate. Nonetheless, Kyber requires the usage of an incomplete NTT modulo 3329, which is incompatible with the NTT required by Dilithium. A variant of Kyber that doen not generates the public matrix A in the NTT domain could easily be supported by the proposed unified multiplier.

4   Hardware design

We designed a hardware architecture that implements the proposed unified polynomial multiplier. As described in the previous section, the multiplier is based on the NTT multiplication algorithm and it uses signed arithmetic with two's complement representation. The proposed architecture is designed for ASIC implementation first, although applications to FPGAs are also considered. The design goals strike a balance between performance and area consumption, and they result in an architecture that achieves comparable latency with existing designs while also reducing the overall area consumption.

4.1   Modular reduction

We start by optimizing the modular reduction operation through hardware-friendly computations. The impact of these optimizations is significant because modular reduction is a low-level fundamental computation whose performance affects the overall performance of the NTT computation. By reducing the critical path of the modular reduction, it is thus possible to increase the clock speed for the entire NTT block.

In Dilithium, the modulus is $q = 2^{23} - 2^{13} + 1$, which is a Solinas prime. The input a is the result of the multiplication between two 23-bit values, and it is thus 46-bit long. The value a is split into five blocks as

$$a = a_4 2^{43} + a_3 2^{33} + a_2 2^{23} + a_1 2^{13} + a_0,$$

since mod q we have that $2^{23} \equiv 2^{13} - 1$, which means $2^{33} \equiv 2^{23} - 2^{10} \equiv 2^{13} - 2^{10} - 1$, which also implies $2^{43} \equiv 2^{23} - 2^{20} - 2^{10} \equiv -2^{20} + 2^{13} - 2^{10} - 1$. We can thus compute

modular reduction by simple bitshift operations and additions. The resulting algorithm is represented in Algorithm 8. Adders and subtractors implemented signed arithmetic. The value A in line 6 is in the range $[-q, 3q]$. Firstly, we reduce it to $[-q, q]$ in line 7 and 8. Note that explicit comparison can be avoided here by replacing it with the carry output of the $A - q$ and $A - 2q$. We then explicitly compare the result with $q/2$ and add or subtract q to obtain a result in the range $[-(q-1)/2, q/2]$

Algorithm 8: Shift-and-add modular reduction for Dilithium

Result: $A \equiv a \bmod q$, with $A \in [0, q)$

1 $\{a_4 \parallel a_3 \parallel a_2 \parallel a_1 \parallel a_0\} \leftarrow a$;   ▷ $|a_1| = |a_2| = |a_3| = 10$,
2 $A_0 \leftarrow a_2 + a_3 + a_4$;   ▷ $|a_0| = 13$, $|a_4| = 3$
3 $A_1^* \leftarrow a_1 + A_0$;
4 $A_1 \leftarrow A_1^* 2^3 - (a_3 + a_4)$;
5 $A_2 \leftarrow -a_4$;
6 $A \leftarrow A_2 2^{20} + A_1 2^{10} - A_0 + a_0$;   ▷ $A_2 2^{20} + a_0 = \{A_2, 7'b0, a0\}$
7 **if** $A > q$ **then** $A \leftarrow A - q$;
8 **if** $A > q$ **then** $A \leftarrow A - q$;
9 **if** $A > q/2$ **then** $A \leftarrow A - q$;
10 **if** $A < q/2$ **then** $A \leftarrow A + q$;
11 **return** A

4.2 Butterfly unit

We show how we designed and optimized the butterfly computation unit. Such a unit sits at the core of the NTT algorithm, and is used to update the polynomial coefficients. There exist two types of butterfly units, the Cooley-Tukey (CT) butterfly and the Gentleman-Sande (GS) butterfly, which are shown in Figure 1 and Figure 2, respectively. They are both composed of one multiplier, one adder and one subtractor, but the order varies.
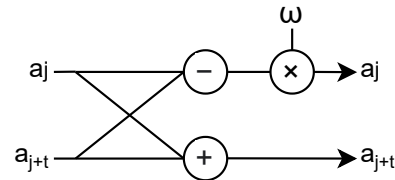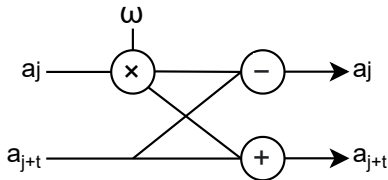

Figure 1: Cooley-Tukey (CT) butterfly.   Figure 2: Gentleman-Sande (GS) butterfly.

Using the same butterfly type for both the forward NTT and the inverse NTT produces results in bit-reverse order, but bit-reverse reordering operations in hardware are simple, since they just consist of rewiring, whereas supporting both types of butterfly requires additional logic. However, this is only possible when the NTT does not implicitly compute any polynomial reduction, i.e. it outputs a 512 coefficient polynomial. This would then require dedicated logic to compute the reduction modulo $\langle x^{256} + 1 \rangle$. It is possible to implicitly compute such reduction within the NTT, as shown in [POG15], if there exists a 512-th root of unity modulo the coefficient modulus. While a 256-th root of unity is always needed to compute the general NTT, the additional 512-th root of unity is needed because its 256-th power is -1, which is then needed in the modular reduction since $x^{256} \equiv -1$ mod $\langle x^{256} + 1 \rangle$. When such an approach is chosen, using both the Cooley-Tukey and the Gentleman-Sande butterfly greatly simplifies computations, as shown in [POG15], and it is thus preferable over a single butterfly type.

To support both types while minimizing area consumption, we designed a single unit that can supports both modes of operation. The corresponding algorithm is shown in Algorithm 9 and its implementation is depicted in Figure 3. NTT-based multiplication also requires to compute coefficient-wise multiplication, and the computation of matrix-vector multiplication also involves summing several coefficient-wise multiplications before computing the inverse NTT. To reduce the area consumption and reuse the butterfly multipliers, the butterfly units also support a third mode of operation (not depicted) where the butterfly receives three values a_{acc}, a_0, and a_1 and computes the multiply-and-accumulate operation that returns $a_{acc} + a_0 \times a_1$. This is needed to compute coefficient-wise multiplication in parallel with summing the result with that of previous coefficient-wise multiplications. No additional data port is needed for this mode since the omega port can be reused to carry the third input value.

Algorithm 9: Unified butterfly algorithm.

Data: Coefficients a_0, a_1,
        Mode selector CT: 1 for CT, 0 for GS
Result: Updated coefficients a_0', a_1'
1   $m_0 \leftarrow$ CT ? $a_1 : a_0$;
2   $m_1 \leftarrow$ CT ? $0 : a_1$;
3   $P \leftarrow (m_0 - m_1) \times \omega$;
4   $s \leftarrow$ CT ? $P : a_1$;
5   $a_0' \leftarrow a_0 + s$;
6   $a_1' \leftarrow$ CT ? $a_0 - P : P$;
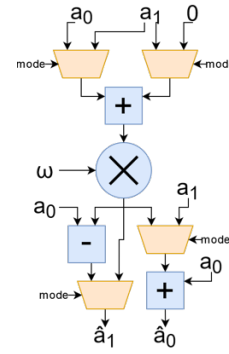7   **return** (a_0', a_1')

Figure 3: Unified butterfly design.
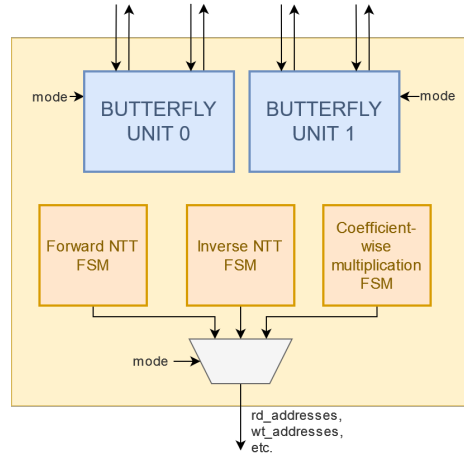
4.3 High-level design

Figure 4: High-level design of the NTT core.

We now present the high-level design of the NTT core, which is depicted in Figure 4. The NTT core is used to compute all three stages of polynomial multiplication: the forward NTT, coefficient-wise multiplication, and the inverse NTT. As described in Section 4.2, the butterfly units can support all three functionalities. The NTT core is then equipped with three finite-state machines, one for each functionality, that determine the reading and writing addresses for the coefficients and root-of-unity constants. To simplify the design,

Memory A			Memory B	
0	128		64	192
1	129		65	193
2	130		66	194
⋮	⋮		⋮	⋮
63	191		127	255

(a) Before the NTT and after INTT.

Memory A			Memory B	
0	1		254	255
4	5		⋮	⋮
8	9		10	11
⋮	⋮		6	7
252	253		2	3

(b) After the NTT and before INTT.

Figure 5: Memory order. For simplicity, only the index of each coefficient is shown.

the finite state-machine for the forward and inverse NTT contains 512 states that specify the addresses as well as the memory writing pattern. In this way, our design remains simple and lightweight while supporting all three functionalities.

Our design uses two butterflies in parallel to achieve a trade-off between performance and area consumption. In this way, an NTT computation requires 512 cycles. Since each memory block can only be read and written once per cycle, four memory blocks would be needed to support two butterflies, one for each coefficient that is read and written each turn. However, we can reduce the number of memory blocks down to two by storing two coefficients per memory word. This reduces the memory consumption in both FPGA and ASIC implementations, and it simplifies the memory handling logic. To do achive this reduction, we use a technique similar to the one presented in [CMV+15]. At the beginning of the forward NTT computation, the memory blocks contain coefficients as represented in Figure 5a. Each cycle, two memory words are read, the four coefficients are written back to memory in a mixed order. For instance, after the first cycle the first word of memory A contains the coefficients in position 0 and 64, while the first word of memory B contains the coefficients in position 128 and 192. Furthermore, at some iterations the coefficients are swapped before being processed by the butterfly cores, while at others the results are fully swapped, i.e. the two coefficients that were originally saved in memory A are stored in memory B and vice versa. By carefully scheduling these operations, it is possible to guarantee that the four coefficients that are to be processed by the two butterfly units are always fully stored in two words. After completing the forward NTT computation, the memory blocks contain coefficients as represented in Figure 5b. The inverse NTT does the opposite: it expects the input to be stored as in Figure 5b and produces the output as in Figure 5a. The coefficient-wise multiplication reads and writes the coefficients in the same order and thus does not affect the memory allocation.

5   Lightweight side-channel protection

There exists several side-channel attacks against implementations of the NTT, such as those reported in [KLH+20] and [PP19]. Both propose single trace attacks against the NTT in Dilithium. One efficient countermeasure is shuffling, which consists of randomly reordering operations that do not depend on each other, so that the resulting power or EM traces change at each iteration of the protocol. While this does not guarantee the same protection as more computationally demanding techniques, such as masking, shuffling can increase the side-channel security while requiring only low overheads.

The NTT algorithm is composed of stages of 128 operations that are independent of each other. Since our implementation uses two butterflies in parallel, our NTT takes 512 cycle to complete and consists of eight stages of 64 cycles, where each operation in a stage is independent of each other. We thus propose to shuffle those 64 operations in a random order. This leads to 64! (approximately 2^{296}) different configurations, hence 64! different

power/EM traces. Since our implementation of the forward and inverse NTT relies on two 512-state finite-state machines, shuffling can be easily achieved by simply changing the way that the state is updated within the finite-state machines.

One thing to note is that shuffling may cause memory collisions. Since the order of execution is randomized, one operation may write at the same memory address that the next operation needs to read from. This does not happen between operations in the same 64-cycle stage, but can take place–for instance–between the 64th and the 65th cycle of execution. Memory collisions leads to errors because there is a one-cycle delay between the writing operation and when that value can be read from memory. There are two possible solutions to this error: either the random shuffle is generated in a specific way to avoid such collisions, or an artificial 1-cycle delay is introduced between each stage. To keep the implementation simple and have a robust implementation that does not rely on the correctness of the shuffle generation, we opted for the second solution. Note that this increases the latency by only 7 cycles, or about a 1% increase in cycle count.

The random order can either be generated in software and passed to the hardware accelerator as input or generated directly in hardware. Software implementations can use, for example, the inside-out version of the Fisher-Yates algorithm [Knu97] to generate the shuffling order. The inside-out version of the Fisher-Yates algorithm may also be used for in-hardware generation, but it requires 64 random numbers in a varying interval between 1 and 64, i.e. 64 random numbers are required, where the i-th number is uniformly random in the interval $[0, i-1]$. As SABER and Dilithium already require a Keccak core, the random numbers can be obtained by hashing an input seed using a Keccak-based hash. However, the results are uniformly random in a power-of-two range, which creates an issue for hardware implementations. This is because uniformly random with non power-of-two moduli requires more involved solutions, such as rejection sampling, which significantly increase the complexity of the design.

We thus propose that in-hardware generation use a simple modular reduction operation (which becomes an even simpler subtraction) to reduce the range of the values to the correct one. This leads to a biased distribution, but a careful analysis shows that the entropy of the randomization is about 2^{293} bits, i.e. only 3 bits lower than the ideal one. With such a trade-off, the shuffle generation requires less randomness, uses a simpler algorithm (which requires less logic and less development time), and guarantees a constant-time execution at the cost of only three bits of entropy.

If the random shuffle is generated in software and passed as input to the multiplier, the latency overhead is only 7 cycles. If the Fisher-Yates algorithm is implemented in hardware, the random shuffle can be stored in memory (BRAM in FPGAs), thus the area overhead is also minimal because it only consists of a single subtractor and memory read/write logic. The performance overhead is somewhat more relevant since it requires 128 cycles, two for each coefficients due to the memory limitations. While this is one quarter of the entire NTT execution, if the larger context of SABER and Dilithium is considered, the Fisher-Yates computation can also be parallelized with other operations to minimize its latency effects.

We propose to use the same randomization for all eight stages in a single NTT, but different trade-offs between security and performance can be achieved by using different randomizations for each stage, or conversely using the same randomization for multiple NTTs. We also note there are other fine-grained techniques that can be applied to further increase the noise and thus the security of the implementation, such as randomize which butterfly unit operates on which set of data.

5.1   Extending the protection beyond the NTT

While this work focuses only on polynomial multiplication, the multiplier is designed to be used within a larger implementation of SABER and Dilithium. If other accessory

functions are also implemented in hardware, it is possible to extend the side-channel protection to them without any additional logic or computation. We start by noting that the unit that computes polynomial multiplication via NTT, inverse NTT and coefficient-wise multiplication is constantly writing and reading from memory. In the last 64 cycles, the coefficients that are written to memory are the final result and will not be further modified. Since all accessory computations in SABER and Dilithium take place on the result of the inverse NTT and are computed coefficient-wise, it is possible to compute them right after the inverse NTT block and before those values are saved to memory.

For instance, consider the Power2Round function used during the Dilithium key generation. A straightforward approach consists of computing the inverse NTT and storing the result in memory. Then, the values $\boldsymbol{As_1}$ and $\boldsymbol{s_2}$ are read from memory and $\boldsymbol{t} = \boldsymbol{As_1} + \boldsymbol{s_2}$ is written back to memory. Eventually, all the coefficients of the polynomials in $\boldsymbol{t}$ are read again to compute Power2Round($\boldsymbol{t}, d$) (here d is a constant), and eventually $\boldsymbol{t_0}$ and $\boldsymbol{t_1}$ are written back to memory. In our proposed design, while computing the inverse NTT, we also load $\boldsymbol{s_2}$ and compute Power2Round($\boldsymbol{As_1} + \boldsymbol{s_2}$) after the final coefficients of $\boldsymbol{As_1}$ are computed by the INTT and before they are stored to memory. In this way, we write directly the final value and thus reduce the latency as well as the area consumption. The same applies to all the other accessory computation blocks. Other examples of accessory function that can be implemented with this approach are the rounding operations in SABER, the message embedding and extracting in SABER, but also the HighBits and LowBits functions in Dilithium, the MakeHint instruction in Dilithium, etc.

While this approach leads to longer critical paths and possibly higher memory bandwith, it also significantly reduce the latency of the accessory functions and may also affect the side-channel security. With this method, we can extend the shuffling protection to all the accessory functions without any additional logic, since the computational units receive the coefficients in random order. This leads to randomized traces that contribute to the side-channel resistance. Moreover, in this way two functions are computed in parallel (since the INTT has two butterfly units and produces four coefficients per cycle) and in parallel with the INTT computation, thus increasing the overall noise. However, further evaluation is needed to experimentally validate the security of this method.

6   Results

6.1   NTT-based polynomial multiplication

The proposed multiplier was described in SystemVerilog and was compiled for different platforms. We used Synopsys Design Compiler and 10 nm Fin Field-Effect Transistor (FinFET) technology library to compile the design for ASIC, whereas we relied on Xilinx Vivado to target the Artix-7 XC7A200T-2-FBG676 FPGA and on Quartus Prime to target the Stratix-10 1SD21BPT FPGA. The implemented design supports the forward NTT, coefficient-wise multiplication and inverse NTT. The design can also compute operations in random order, based on an input shuffle. The shuffle generation itself is not included in the design to have a fairer comparison with existing designs.

Table 2 shows the results of our polynomial multiplier implementation, and it also reports recent hardware implementations of the NTT, mostly for Dilithium, and polynomial multipliers for SABER. Our design offers the lowest consumption of flip-flops of all implementations reported in Table 2. This follows from the short critical path of the design that allows to compute one entire butterfly operation per cycle. The design thus does not use any pipelining buffer. The majority of the flip-flops reported is used to store the random shuffle. Our design also offers high clock frequencies on FPGAs and it is indeed the second fastest after the much larger implementation of [RMJ+21]. When synthesized on 10 nm technology, the clock frequency increase very significantly up to almost 4 GHz.

Table 2: Comparison of implementations results of NNT-based polynomial multipliers. The FPGA column denotes the FPGA family or the ASIC technology (A7 – Artix-7; V7 U+ – Virtex-7 Ultrascale+; Z700 – Zynq 7000; S10 – Stratix-10, and 10nm – 10 nm FinFET process technology). NR – values not reported. For 10 nm, the LUT and FF column report the number of combinatorial and sequential gates. † – NTT only; ‡ – INTT only.

Impl.	FPGA	log p	LUTs	FFs	DSPs	Cycles	Freq. (MHz)
[FBR⁺]		39	2,454	1,917	7	3,584	153
[ZHL⁺21]	A7	23	2,044	NR	16	512	216
[RMJ⁺21]†	V7 U+	23	2,236	2,532	48	512	637
[RMJ⁺21]‡	V7 U+	23	3,309	3,389	84	512	637
[LSG21]	Z7000	23	531	426	17	533/536	142
[this work]	A7	23	2,012	331	6	519	333
[this work]	S10	23	2,312 ALMs	412	4	519	386
[this work]	10nm	23	12,807 (com)	537 (seq)	–	519	3,953

The implementation by Fritzman et al. [FBR⁺] uses a loosely-coupled NTT to support a wide range of post-quantum protocols, and thus it uses a larger 39-bit long prime. While the LUT count is comparable to our proposed multiplier, the flip-flop count is six times higher and the cycle count is five times higher. However, the loosely-coupled nature makes it a non-ideal comparison. The remaining implementations all focus on Dilithium. While the performance is comparable across all implementations, including ours, the resource consumption varies considerably. Note that our design has a latency of 519, instead of 512, to avoid memory collisions when the order of operation is shuffled. A non-shuffled implementation takes 512 cycles. [ZHL⁺21] reports a comparable LUT count, but it achieves a lower clock frequency while also consuming significantly more DSPs. [RMJ⁺21] instead reports the highest clock frequencies, but at the cost of a very high resource consumption. LUT and flip-flop count is significantly higher, especially considering that forward and inverse NTT are implemented separately, and the implementation also requires many times over the number of flip flops. Lastly, [LSG21] reports a very low LUT consumption, but that is due to the high number of DSPs. Indeed, the authors report implementing nearly all of the arithmetic with DSPs. We expect that LUT-only implementation would favor our design. Moreover, the clock only reaches less than half the frequency of our design.

When comparing with non-NTT based multiplier, our proposed multiplier also offers performance improvements. In [BR20], the first high-speed polynomial multiplier is reported to consume 10,844 LUTs and 5,150 flip-flops when implemented with area-optimizing strategies. A single multiplication requires 256 cycles without considering memory reading and writing operations, or 336 cycles in total. In the current design, a NTT-based multiplication requires $2 \times 519 + 128 + 519 = 1,685$ cycles, since forward and inverse NTT take up 519 cycles and coefficient-wise multiplication requires 128 cycles. This means that our multiplier takes about $5.0\times$ the computation time of the multiplier in [BR20], but consumes $5.4\times$ fewer LUTs and $15.6\times$ fewer flip-flops. However, note that the performance balance is tilted more favorably towards our proposed design, because some INTT operations are avoided when vector-vector multiplication is computed. For instance, when vectors contain three polynomials, vector-vector multiplication takes 4,017 cycles since a single INTT is needed. The multiplier in [BR20] takes instead 880 cycles for vector-vector multiplication, which means it is $4.5\times$ faster. The difference is even smaller because when the same polynomial is multiplied more than once, like the secret polynomials during encryption, the NTT operation only needs to be computed once. Note,
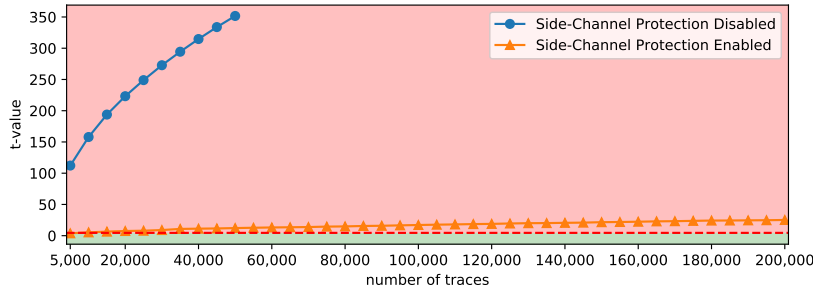
Figure 6: Leakage detection using TVLA.

however, that the multiplier in [BR20] requires fewer memory blocks and no DSPs.

Overall, our design offers a good balance between cycle count and area consumption. When both are considered, our implementation outperforms previous implementations reported in the literature. Moreover, our design offers high clock frequencies on FPGAs and very high frequencies in ASIC.

6.2   Lightweight side-channel protection

We evaluate the robustness of the lightweight side-channel protection described in Section 5 where the same randomization is used for all eight stages of the NTT using two leakage detection techniques, namely Test Vector Leakage Assessment (TVLA) and Normalized Inter-Class Variance (NICV). These techniques are applied to power traces sampled at 150 MS/s from a Sakura-X board clocked at 3 MHz. An unprotected implementation of the NTT serves as a baseline for comparison. For all experiments, we focus on the first stage of 128 operations and use Dilithium polynomials with coefficients in $[-2, 2]$ to exhibit the most favorable attack settings, where an attacker aims to recover one out of five possible values for each coefficient.

TVLA   Based on Welch's t-test, the Test Vector Leakage Assessment (TVLA) methodology is commonly used to assess the side-channel leakage of cryptographic implementations [BCDM$^+$13]. The t-value of a non-specific fixed-vs.-random t-test applied to the samples of the eleventh iteration of the NTT implementation with and without the countermeasure enabled are shown in Figure 6 for an increasing number of traces. The two curves shown in Figure 6 have different characteristics. First, the leakage of the protected implementation grows extremely slowly with the number of traces compared to the leakage of the unprotected implementation. Second, all the t-values for the countermeasure enabled for up to 200,000 traces are lower than the minimum t-value (i.e., 112 for 5,000 traces) with the countermeasure disabled. Therefore, the lightweight side-channel protection considerably reduces the leakage of the implementation. For a better understanding of the remaining leakage, we continue the analysis with another leakage detection technique.

NICV   Normalized-Inter Class Variance (NICV) is used to detect leakage samples in side-channel traces and evaluate the efficiency of a side-channel countermeasure [BDGN13]. The results of applying NICV to the traces measured from the unprotected and protected NTT implementation are shown in Figure 7a and Figure 7b, respectively. The peak shown in Figure 7a stems from the eleventh iteration of the unprotected NTT implementation and spans two clock cycles. In the first cycle, the combinational circuit computes the results of the two butterflies, which are written to memory in the next cycle. Although the memory update leaks more than the combinational circuit, the leakage in each of

(a) Side-Channel Protection Disabled.
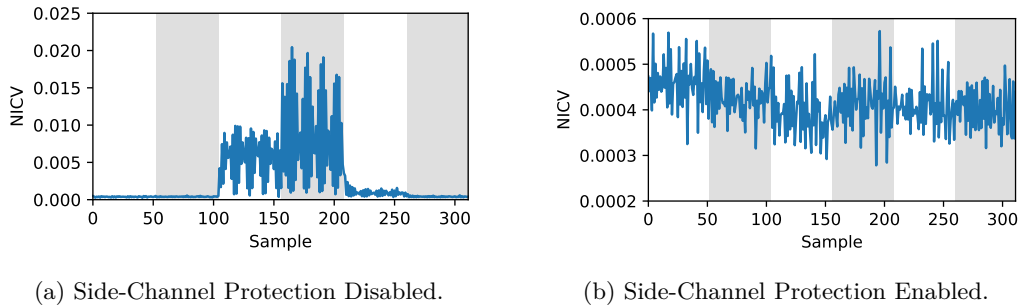

(b) Side-Channel Protection Enabled.

Figure 7: Leakage detection using NICV on 100,000 traces. The side-channel protection reduces the leakage by two orders of magnitude.

the two cycles is clearly distinguishable from the other cycles. On the other hand, no clear peak can be observed in Figure 7b for the same clock cycles of the protected NTT implementation. Hence, the shuffling countermeasure breaks the relationship between the intermediate values and the power consumption. When comparing the NICV values from the two figures, the lightweight side-channel protection reduces the leakage of the NTT implementation by two orders of magnitude.

Discussion   Shuffling the order of the butterfly operations in each of the eight stages of the NTT, makes multi-trace attacks significantly more difficult because the same operations occur at different sample points in each trace. Averaging of traces, which may improve the signal-to-noise ratio, is not straightforward for the same reason. This leaves an attacker with the option of single-trace attacks.

Single-trace attacks pose a major threat to the implementation of both unprotected and masked software implementations of the NTT as demonstrated on the 32-bit ARM Cortex-M4 microcontroller [PPM17, PP19, KLH+20]. To the best of our knowledge, there are no similar attacks on hardware implementations of the NTT in the literature. Moreover, it is not clear whether such attacks can be extended to larger word widths or hardware implementations. However, we adopt a conservative approach and implement a shuffling countermeasure, which effectively thwarts this type of attacks [PPM17, PP19].

We conducted a quantitative evaluation of the leakage of our implementation with and without the lightweight side-channel protection. When enabled, the countermeasure significantly reduces the leakage of the implementation. Hence, the Fisher-Yates shuffling is a cost-effective side-channel protection for hardware implementations of the NTT.

7   Conclusion

In this work, we proposed a technique that enables SABER and Dilithium to share the same polynomial multiplier. Given the importance of polynomial multiplication in both protocols, this has a significant impact on hardware implementations that support both protocols. We estimated that existing Dilithium implementations can add support for SABER with only a 4% increase in LUT count. A minor trade-off of the proposed multiplier is that it can produce inexact results with some specific inputs. We thus performed a thorough analysis of such cases, where we proved that the probability of these events occurring is negligible, and we showed that this does not affect the security of the implementation. We then implemented the proposed multiplier in hardware to obtain a design that offers competitive performance/area trade-offs. We also proposed a low-overhead method to provide side-channel protection during polynomial multiplication.

This work provides an example of powerful synergy between a post-quantum encryption protocol and a digital signature scheme. It is thus of interest to study other post-quantum protocols and research whether similar synergies can be found. We also leave for future work the design and implementation of a complete hardware implementation that supports both SABER and Dilithium followed by a thorough evaluation of our proposed lightweight side-channel countermeasure.

References

[AMJ+21]   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. Cryptology ePrint Archive, Report 2021/1461, 2021. https://ia.cr/2021/1461.

[BCDM+13] G Becker, J Cooper, E De Mulder, G Goodwill, J Jaffe, G Kenworthy, et al. Test vector leakage assessment (TVLA) derived test requirements (DTR) with aes. In *International Cryptographic Module Conference*, 2013.

[BDGN13]   Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. NICV: normalized inter-class variance for detection of side-channel leakage. *IACR Cryptol. ePrint Arch.*, page 717, 2013.

[BDK+20]   Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium (Round 3 Submission), 2020.

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

[BR20]   Andrea Basso and Sujoy Sinha Roy. Optimized Polynomial Multiplier Architectures for Post-Quantum KEM Saber, 2020. http://ia.cr/2020/1482.

[CHK+20]   Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings, 2020. http://ia.cr/2020/1397.

[CMV+15]   Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, Ray C. C. Cheung, Derek Pao, and Ingrid Verbauwhede. High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 62(1):157–166, January 2015.

[FBR+]   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.

[FSS20]   Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 239–280, August 2020.

[IAR+21]   Malik Imran, Felipe Almeida, Jaan Raik, Andrea Basso, Sujoy Sinha Roy, and Samuel Pagliarini. Design Space Exploration of SABER in 65nm ASIC, 2021. http://ia.cr/2021/1202.

[KLH+20]   Il-Ju Kim, Tae-Ho Lee, Jaeseung Han, Bo-Yeon Sim, and Dong-Guk Han. Novel Single-Trace ML Profiling Attacks on NIST 3 Round candidate Dilithium, 2020. http://ia.cr/2020/1383.

[Knu97]   Donald Ervin Knuth. *The Art of Computer Programming*. Addison-Wesley, Reading, Mass, 3rd ed edition, 1997.

[LN16]   Patrick Longa and Michael Naehrig. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography, 2016. http://ia.cr/2016/504.

[LSG21]   Georg Land, Pascal Sasdrich, and Tim Güneysu. A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware, 2021. http://ia.cr/2021/355.

[MP]   Moxie Marlinspike and Trevor Perrin. The X3DH Key Agreement Protocol.

[oST]   National Institute of Standards and Technology. Post-quantum cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization. Accessed October 2021.

[POG15]   Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-Performance Ideal Lattice-Based Cryptography on 8-bit ATxmega Microcontrollers, 2015. http://ia.cr/2015/382.

[PP19]   Peter Pessl and Robert Primas. More Practical Single-Trace Attacks on the Number Theoretic Transform. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, volume 11774, pages 130–149. Springer International Publishing, Cham, 2019.

[PPM17]   Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017*, volume 10529, pages 513–533. Springer, 2017.

[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*, pages 443–466, August 2020.

[Res18]   Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Request for Comments RFC 8446, Internet Engineering Task Force, 2018.

[RMJ+21]   Sara Ricci, Lukas Malina, Petr Jedlicka, David Smekal, Jan Hajny, Petr Cibik, and Patrik Dobias. Implementing CRYSTALS-Dilithium Signature Scheme on FPGAs, 2021. http://ia.cr/2021/108.

[RPBC20]   Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. On Configurable SCA Countermeasures Against Single Trace Attacks for the NTT: A Performance Evaluation Study over Kyber and Dilithium on the ARM Cortex-M4. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering*, volume 12586, pages 123–146. Springer International Publishing, Cham, 2020.

[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, October 1997.

[ZBT19]   Timo Zijlstra, Karim Bigou, and Arnaud Tisserand. FPGA Implementation and Comparison of Protections Against SCAs for RLWE. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology – INDOCRYPT 2019*, volume 11898, pages 535–555. Springer International Publishing, Cham, 2019.

[ZHL$^+$21] Zhen Zhou, Debiao He, Zhe Liu, Min Luo, and Kim-Kwang Raymond Choo. A Software/Hardware Co-Design of Crystals-Dilithium Signature Scheme. *ACM Transactions on Reconfigurable Technology and Systems*, 14(2):1–21, June 2021.

[ZZY$^+$21] Yihong Zhu, Min Zhu, Bohan Yang, Wenping Zhu, Chenchen Deng, Chen Chen, Shaojun Wei, and Leibo Liu. LWRpro: An Energy-Efficient Configurable Crypto-Processor for Module-LWR. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(3):1146–1159, March 2021.