Gate-Level Masking of Streamlined NTRU Prime Decapsulation in Hardware

Georg Land[1], Adrian Marotzke[2,3], Jan Richter-Brockmann[1] and Tim Güneysu[1,4]

[1] Ruhr University Bochum, Horst Görtz Institute for IT Security, Germany
{georg.land,jan.richter-brockmann,tim.gueneysu}@rub.de
[2] Hamburg University of Technology, Germany adrian.marotzke@tuhh.de
[3] NXP Semiconductors, Hamburg, Germany adrian.marotzke@nxp.com
[4] DFKI GmbH, Cyber-Physical Systems, Bremen, Germany

Abstract. Streamlined NTRU Prime is a lattice-based Key Encapsulation Mechanism (KEM) that is, together with X25519, currently the default algorithm in OpenSSH 9. Being based on lattice assumptions, it is assumed to be secure also against attackers with access to large-scale quantum computers. While Post-Quantum Cryptography (PQC) schemes have been subject to extensive research in the recent years, challenges remain with respect to protection mechanisms against attackers that have additional side-channel information such as the power consumption of a device processing secret data. As a countermeasure to such attacks, masking has been shown to be a promising and effective approach. For public-key schemes, including any recent PQC schemes, usually a mixture of Boolean and arithmetic approaches are applied on an algorithmic level. Our generic hardware implementation of Streamlined NTRU Prime decapsulation, however, follows an idea that until now was assumed to be only applicable to symmetric cryptography: *gate-level* masking. There, a hardware design that consists of logic gates is transformed into a secure implementation by replacing each gate with a *composably secure gadget* that operates on uniform random shares of secret values. In our work, we show the feasibility of applying this approach also to PQC schemes and present the first Public-Key Cryptography (PKC) – pre- and post-quantum – implementation masked at gate level considering several trade-offs and design choices. We synthesize our implementation both for Artix-7 Field-Programmable Gate Arrays (FPGAs) and 45 nm Application-Specific Integrated Circuits (ASICs), yielding practically feasible results regarding area, randomness demand and latency. Finally, we also analyze the applicability of our concept to Kyber which will be standardized by the National Institute of Standards and Technology (NIST).

Keywords: PQC, Masking, FPGA, ASIC, Streamlined NTRU Prime, Higher-order Masking, Gate-level Masking

1 Introduction

Wide deployment of Post-Quantum Cryptography (PQC) algorithms in practical solutions is indispensable. Even though there is no guarantee that the advent of large-scale quantum computers will happen at all, the protection of future data must be ensured by deploying algorithms that are secure even in the presence of large-scale quantum computers *in the near future*.

Moreover, it is vital for deployed implementations processing sensitive data to provide also security against physical attacks. For implementations on server machines and personal computers, it usually suffices to use strictly constant time implementations by means of having an execution time independent of secret values. This does not include branching based on secret data and loading values from secret addresses.

For embedded devices, however, we additionally have to consider adversaries who can measure the power consumption or electromagnetic (EM) emanation of a device processing secret data. In this context, many practical attacks have been shown in the past on "classical", but also PQC schemes. For instance, several attacks have been published attacking Kyber [XPR+21, SPH22, HHP+21, HPP21], Saber [NDGJ21], Falcon [KA21], NTRU [AR21], or even generic on lattice-based constructions [RRCB20]. Notable here are attacks targeting *side-channel protected* implementations, such as the recent attacks on a fifth-order masked Kyber implementation [DNG22], a third-order masked Saber implementation [NWDP22] and a first-order masked Saber implementation [NDJ21].

Specifically for Streamlined NTRU Prime, two attacks have been proposed. First, Xu et *al.* show single-trace attacks on fixed weight sampling as used in Streamlined NTRU Prime and NTRU key generation as well as Dilithium signing [KAA21]. Furthermore, Ravi et *al.* present a method to recover the Streamlined NTRU Prime secret key with a side-channel assisted chosen-ciphertext attack [FBR+22]. They demonstrate the capability of a full key recovery with just 3 005 traces for the smallest parameter set and with 4 688 traces for larger parameter sets.

Contrary, dedicated countermeasures aiming at decoupling the connection between secret data and power consumption have been proposed in the past decades. The main technique for that purpose is *masking* which splits secret values into multiple uniform random shares. In this context, research has focused recently on masked PQC implementations in *software*, mostly for Kyber and Saber. A recent preprint presents a masked implementation of NTRU for embedded software [CGTZ22]. In contrast, there are far less works on hardening hardware implementations, again focusing on Kyber [JGCS21, FBR+22, KNAH22] and Saber [AMD+21, FBR+22]. To the best of our knowledge, no PQC schemes other than Kyber and Saber have a fully masked hardware implementation published, where both implementations target Field-Programmable Gate Arrays (FPGAs) and first order only.

To date, secure implementations for the Streamlined NTRU Prime scheme have not been yet proposed, neither for software nor for hardware platforms, despite of the fact Streamlined NTRU Prime is already the default choice for the widely used OpenSSH suite, starting from version 9.0. In our work, we aim to close this gap by presenting the first masked hardware implementation of Streamlined NTRU Prime decapsulation for hardware devices, synthesizable both for FPGA and Application-Specific Integrated Circuit (ASIC) and aiming at use-cases where OpenSSH-supported connections are established with external hardware devices that are potentially under exposure to physical attackers. Moreover, our implementation is masked on *gate level* rather than algorithmic level, which has the advantage of being easily configurable to provide protection for any arbitrary order. To the best of our knowledge, this is the first gate-level masked implementation of a Public-Key Cryptography (PKC) decapsulation or decryption, both for the pre- and post-quantum settings and the first masked ASIC implementation of any PQC scheme.

Contribution. Hence, we can summarize our contributions as follows:

- We present the first gate-level masked implementation of any PKC scheme.
- Our approach can be generalized to an arbitrary-order masked hardware implementation of any PQC scheme for which the masking degree can be adjusted easily.
- We implement our design both on a Xilinx Artix-7 FPGA, and as an ASIC using the 45nm Nangate open cell library[1].

[1]Available at https://si2.org/open-cell-library/

- Compared to other existing fully masked PQC FPGA implementations, our implementation has similar (in the case of Saber) or significantly lower (in the case of Kyber) resource requirements for first-order security.
- We present the first arbitrary-order masked SHA-2 hardware implementation in scientific literature.
- The side-channel resistance of our implementation is formally verified using the VERICA tool.
- Our source code will be available at https://github.com/AdrianMarotzke/Masked-SNTRUP.

2 Preliminaries

In this section, we briefly introduce our notations used throughout this work. Afterwards, we recap masking and important composability notions. Eventually, we describe Streamlined NTRU Prime and particularly the decapsulation.

2.1 Notation

Throughout this work, we denote $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^p - x - 1)$, and $\mathcal{R}_3 = \mathbb{Z}_3[x]/(x^p - x - 1)$ with p, q being primes. Furthermore, we write $x[i : j]$ for bit vectors of length $|i - j| + 1$ and also allow multiple dimensions for this, e.g., $x[i : j, k : l]$ is a vector of $|i - j| + 1$ bit vectors each of length $|k - l| + 1$. For masking, we use d as the masking degree, i.e., the number of probes an attacker has access to. It follows that we split secrets into $d + 1$ shares, referring to a single share as $x^{(i)}$ with $0 \leq i \leq d$. Moreover, we denote $x^{(0:d)}$ as a masked variable. At any occurrence of Boolean operations that involve masked variables, we assume to perform this securely, e.g., by means of a secure gadget. Finally, we stress that $\overline{x^{(0:d)}}$ denotes inverting the secret value by inverting one share rather than inverting each share (which would not invert the secret value for odd d).

2.2 Masking

Masking is an approach based on Shamir's secret sharing. It has been proven as an effective countermeasure against power or EM side-channel attacks by splitting secret values into uniform random shares. In our work, we employ only Boolean masking where a secret value x is split into $d + 1$ shares $x^{(i)}$, such that $x = \bigoplus_{i=0}^{d} x^{(i)}$. While functions that are linear or affine in the masking domain can be applied trivially to each share individually, we use specialized methods to secure non-linear functions like AND or OR operations.

In order to verify and evaluate the resistance against side-channel attacks of such special function, a range of different attacker models have been proposed in the past. In 2003, Ishai, Sahai, and Wagner [ISW03] introduced the d-probing model which is still frequently used as appropriate abstraction. However, this model neither includes glitches nor transitions or couplings and thus has been extended to a *robust d-probing model* incorporating these phenomena [BGI+18, FGP+18].

Nevertheless, the robust d-probing model is not sufficient to analyze the composability of atomic building blocks called *gadgets*. Hence, Barthe et *al.* introduced Non-Interference (NI) as the first composability notion in 2015 [BBD+15]. Although NI limits the leakage between shared intermediate results, it does not guarantee probing security of composed circuits. Therefore, Barthe et *al.* presented the notion of Strong Non-Interference (SNI) [BBD+16] which ensures composability of gadgets. Eventually, Cassiers and Standaert proposed Probe-Isolating Non-Interference (PINI) [CS20] reducing the overhead introduced by SNI gadgets. PINI ensures that all shared AND gadgets are composable and XOR as well as NOT operations can be performed share-wise without refreshing.

Bringing this concept to concrete instantiations of SecAND gadgets in hardware, Cassiers et *al.* proposed Hardware Private Circuits (HPCs) [CGLS21]. HPC allows to instantiate an arbitrary-order masked SecAND gate with two clock cycles latency for one input and one clock cycle for the other input denoted as HPC1. Moreover, they optimized this gadget for less randomness demand denoted as HPC2 gadget. Following this, Knichel et *al.* proposed Generic Hardware Private Circuits (GHPCs) to build more complex gadgets that are PINI [KSM22]. Finally, in a recent work, Knichel and Moradi presented HPC3 achieving a lower latency by using more fresh randomness [KM22].

2.3 Streamlined NTRU Prime

Streamlined NTRU Prime is a lattice-based Key Encapsulation Mechanism (KEM) which is resistant against both classical and quantum adversaries [BCLv17,BBC+20]. It has been designed carefully using structured lattices while firmly avoiding potentially exploitable attack surfaces. In particular, it eliminates decryption failures and employs large Galois groups instead of cyclotomics.

Streamlined NTRU Prime defines Short as the set of polynomials in $\mathcal{R}_q$ with exactly w non-zero coefficients from $\{-1, 1\}$ [BCLv17]. Furthermore, we also use the notation of an underline indicating that the respective value is encoded.

As a KEM, it uses the Fujisaki-Okamoto transform to achieve indistinguishability under chosen-ciphertext attacks (IND-CCA) and builds upon a public-key encryption scheme that fulfills one-wayness against passive attacks. In the following, we describe the three procedures of the KEM: key generation, encapsulation, and decapsulation.

Key Generation. First, a uniform random polynomial g in $\mathcal{R}_3$ is generated. This step is repeated until g is invertible in $\mathcal{R}_3$. Then, the inverse polynomial of g is computed. Furthermore, f is sampled to be a polynomial from Short. The secret key consists of f and g^{-1} as well as a random bit string ρ which is used for implicit rejection during decapsulation. Finally, the public key is computed as $h \in \mathcal{R}_q = g/(3f)$.

Encapsulation. The first step is to sample a uniformly random polynomial r from Short, which is then multiplied with the public key polynomial h. In the resulting polynomial, each coefficient is rounded to the nearest multiple of three. The output of this operations is denoted as the polynomial c. Subsequently, the encoded r and the encoded public key are hashed to create the ciphertext confirmation hash. The confirmation hash together with the encoded c is the ciphertext. The session key is computed by hashing the encoded r and the ciphertext.

Decapsulation. The decapsulation is shown in Algorithm 1 in detail. The basic idea is to remove the denominator of the public key from the ciphertext by multiplying $3f$ in $\mathcal{R}_q$. The subsequent application of modulo 3 to each coefficient removes the rounding error which is succeeded by the multiplication with $1/g \in \mathcal{R}_3$ to also remove the numerator of the public key and to obtain the plaintext. This plaintext is checked to be in the correct space Short. Furthermore, to ensure that no chosen-ciphertext attack is carried out, the obtained plaintext is re-encrypted and the result is compared to the original ciphertext. If everything matches, the correct session key is reconstructed, else an implicit rejection is performed by using ρ. Note that this final rejection step is strictly required to be performed in constant time.

Algorithm 1 Streamlined NTRU Prime Decapsulation [BBC+20]

```
Require: ciphertext $C = (\underline{c}, \gamma)$, secret key ($\underline{k} = \mathsf{Encode}(f, g^{-1})$, $\underline{K} = \mathsf{Encode}(h)$, $\rho$, $\mathsf{hash}_4(\underline{K})$)
1: $c \in \mathcal{R}_3 := \mathsf{Decode}(\underline{c})$
2: $(f, v) \in \mathcal{R}_3 \times \mathcal{R}_3 := \mathsf{Decode}(\underline{k})$
3: $h \in \mathcal{R}_q := \mathsf{Decode}(\underline{K})$
4: $e \in \mathcal{R}_3 := ((3fc) \in \mathcal{R}_q) \bmod 3$
5: $r' \in \mathcal{R}_3 := ev$
6: if $r'$ does NOT have weight $w$ then
7:     $r' := (1, 1, \dots, 1, 0, 0, \dots, 0)$        ▷ The first $w$ elements are 1, the rest 0
8: end if
9: $c' \in \mathcal{R}_q := \mathsf{Round}(hr')$        ▷ re-encrypt with $h, r'$, compute new ciphertext $c'$
10: $\underline{c}' := \mathsf{Encode}(c')$
11: $\underline{r}' := \mathsf{Encode}(r')$
12: $\gamma' := \mathsf{hash}_2(\mathsf{hash}_3(\underline{r}'), \mathsf{hash}_4(\underline{K}))$        ▷ re-compute the ciphertext confirmation hash
13: $C' = (\underline{c}', \gamma')$
14: if $C' = C$ then
15:     return $\mathsf{hash}_1(\mathsf{hash}_3(\underline{r}'), C)$
16: else
17:     return $\mathsf{hash}_0(\mathsf{hash}_3(\rho), C)$
18: end if
```

3 Conceptual Considerations

To implement the decapsulation as shown in Algorithm 1, we essentially need six major modules:

1. Polynomial multiplication with operands in $(\mathcal{R}_q, \mathcal{R}_3)$ and return values in $\mathcal{R}_q$,
2. Polynomial multiplication with operands in $(\mathcal{R}_3, \mathcal{R}_3)$ and result in $\mathcal{R}_3$,
3. Reduction component modulo 3,
4. Weight check component,
5. Rounding module, and
6. SHA-512.

Standard Approach. Usually, to mask polynomial multiplication modules, additive masking would be applied, with either multiple polynomial multipliers being instantiated in parallel, or one polynomial multiplier being instantiated that processes the shares consecutively. Moreover, two of the three multiplications have one public and one secret input which can be realized very efficiently by applying additive masking as it only requires $d + 1$ polynomial multiplications and no re-sharing. The other multiplication, however, has two secret input polynomials. In order to perform a secure polynomial multiplication in additive domain, this requires sampling $\frac{d^2+d}{2}$ fresh random polynomials, $2(d^2 + d)$ polynomial additions, and $d^2 + d$ polynomial multiplications.

In contrast, masking the reduction, weight check, and rounding is non-trivial in arithmetic domain and would be solved in Boolean domain. Finally, SHA-512 uses 64 bit additions, which is efficient in additive domain and feasible, but less efficient in Boolean domain, as well as non-linear Boolean operations that strictly require Boolean masking.

In summary, this traditional approach is expected to yield a relatively efficient implementation at the cost of converting multiple times between additive and Boolean masking domain. Moreover, often this type of implementation is very specific in terms of masking degree, i.e., not being parametrizable. Besides, the big variety of techniques applied produces a larger attack surface, as shown in recent attacks on masking conversions [NWDP22].

Applicability of Gate-Level Masking. To overcome these downsides, we follow a recent line of research from the field of masking symmetric cryptographic schemes: *gate-level* masking. For schemes in symmetric cryptography, we usually can find a Boolean description which enables masking them at gate-level. This is not the case for public-key and post-quantum cryptography as these schemes typically employ arithmetic operations on number-theoretic structures such as multiplications in polynomial fields. Polynomial multiplications, however, consists of modular multiplications and additions in some finite number field. While the modular additions can be masked easily in Boolean domain by means of a secure adder, the modular multiplications are vastly more complex and are deemed to be infeasible to be masked in Boolean domain.

However, for Streamlined NTRU Prime, we observe that the three polynomial multiplications each have at least one factor in $\mathcal{R}_3$. As a consequence, if we employ schoolbook multiplication, the underlying coefficient multiplication-accumulation has an input from $\mathbb{Z}_q$ being multiplied either with 1, 0, or -1, and then accumulated to another value in $\mathbb{Z}_q$. We immediately observe that no complex modular multiplication must be carried out in this case. Instead, we can securely multiplex between the input coefficient from $\mathbb{Z}_q$, its precomputed additive inverse, and zero. The result then is added securely to the accumulation value. As indicated before, all other operations are already feasible in Boolean domain, enabling the first fully Boolean masked implementation of a public key and post-quantum secure scheme.

In the following, we describe our design considerations for each module in Boolean domain. Note that in contrast to conventional hardware development, where it is desirable to have as many NAND gates as possible as they are the cheapest gates, the design goal in our case is to have as few as possible SecAND gates, as they require fresh randomness. Throughout our design, we use the HPC2 SecAND gadget.

3.1 Polynomial Multiplication

Polynomial multiplications are the most expensive operations in the decapsulation. Thus, research usually focuses on improving their performance [Mar20, PMT+22, CHK+21, ACC+21]. Instead, we focus on achieving a *secure* implementation. During decapsulation, two types of multiplications are required:

1. Multiplication in $\mathcal{R}_q$ with one operand from $\mathcal{R}_3$ (Lines 4 and 9 in Algorithm 1) and
2. Multiplication in $\mathcal{R}_3$ (Line 5 in Algorithm 1).

3.1.1 Multiplication in $\mathcal{R}_q$

We observe that if we employ a standard schoolbook multiplication approach for both occasions of this multiplication, no coefficient multiplier is necessary. Rather, we use a secure adder and a secure three-way multiplexer. It is important to note that for both multiplications in $\mathcal{R}_q$, the input polynomial from $\mathcal{R}_q$ is public while the other factor from $\mathcal{R}_3$ is secret. Thus, the idea is to compute the additive inverse of the input coefficient from $\mathcal{R}_q$ which is unmasked. Then, we multiplex securely – with masked select signal – between both values and zero, and finally accumulate the result securely to the (intermediate) result coefficient.

Secure Multiplexing. Furthermore, we need a secure three-way multiplexer. The three *public* input signals are $z = 0, a_p = a, a_n = q - a \in \mathbb{Z}_q$. However, here we view them as boolean values in $\mathbb{F}_2^{13}$. The secret select signal is $(f[1], f[0]) \in \{(0,0), (0,1), (1,1)\}$. We perform two consecutive secure 2-input multiplexing operations:

$$\begin{aligned} x[0]^{(0:d)} &= a_p \wedge f[1]^{(0:d)} \oplus a_n \wedge \overline{f[1]^{(0:d)}} = a_p \wedge f[1]^{(0:d)} \oplus a_n \wedge (f[1]^{(0:d)} \oplus 1) \\ &= a_p \wedge f[1]^{(0:d)} \oplus a_n \wedge f[1]^{(0:d)} \oplus a_n \\ &= ((a_p \oplus a_n) \wedge f[1]^{(0:d)}) \oplus a_n \qquad (1) \\ x[1]^{(0:d)} &= x[0]^{(0:d)} \wedge f[0] \oplus z \wedge \overline{f[0]^{(0:d)}} = ((x[0]^{(0:d)} \oplus z) \wedge f[0]^{(0:d)}) \oplus z \\ &= x[0]^{(0:d)} \wedge f[0]^{(0:d)} \qquad (2) \end{aligned}$$

Note that the public inputs can be set as first shares and all other shares are just zeros. This is the reason why we can simply omit z in Equation 2. The SecAND gadget generates a uniformly random output also for the case that $(f_1, f_0) = (0, 0)$.

Secure Addition. Efficient addition in hardware can be achieved by parallel prefix adders. These concepts also have been adapted to the Boolean masked domain first in [SMG15]. This was followed by a broader examination with more recent techniques like threshold implementation and gate-level masking [BG22], which we also deploy for our work.

3.1.2 Multiplication in $\mathcal{R}_3$

The idea for this case is that there are only nine possible input combinations with three output combinations. Thus, we develop a direct Boolean masking utilizing the fact that the single inputs have a limited range. Multiplying two signed two-bit coefficients $e[1:0] = e[0] - 2e[1]$ and $v[1:0] = v[0] - 2v[1]$ to a signed two-bit value $r[1:0] = r[0] - 2r[1]$ can be done as follows:

$$r[0]^{(0:d)} = e[0]^{(0:d)} \wedge v[0]^{(0:d)} \qquad (3)$$

$$r[1]^{(0:d)} = e[0]^{(0:d)} \wedge v[0]^{(0:d)} \wedge (e[1]^{(0:d)} \oplus v[1]^{(0:d)}) \qquad (4)$$

Then, we add $r[1:0]^{(0:d)}$ to the accumulation value $a[1:0]^{(0:d)}$ and map the result back to the signed $a'[1:0]^{(0:d)} \in \{-1, 0, 1\}$ which can be done with the following formulas that take into account that only $00_2, 01_2, 11_2$ are valid inputs:

$$a'[0]^{(0:d)} = \left(r[0]^{(0:d)} \oplus a[0]^{(0:d)}\right) \vee \left(r[0]^{(0:d)} \wedge \left(\overline{r[1]^{(0:d)} \oplus a[1]^{(0:d)}}\right)\right) \qquad (5)$$

$$a'[1]^{(0:d)} = \left(r[1]^{(0:d)} \wedge \overline{a[0]^{(0:d)}}\right) \oplus \left(\overline{r[1]^{(0:d)}} \wedge \left(r[0]^{(0:d)} \oplus a[1]^{(0:d)}\right)\right) \qquad (6)$$

3.1.3 Schoolbook Polynomial Multiplication

Generally, there are three approaches for this: Either we rotate one of the input polynomials or the output polynomial. For our two “big” multiplications in $\mathcal{R}_q$, we have a small secret input represented by $2(d+1)$ bit, a big public input represented by $\lceil \log_2 q \rceil$ bit, and a big secret output represented by $(d+1)\lceil \log_2 q \rceil$ bit. Since shifting many data is expensive in terms of routing, Flip-Flop (FF) demand, and dynamic power consumption, the natural choice is to rotate either of the input polynomials.

3.1.4 Polynomial Reduction modulo $x^p - x - 1$

For the schoolbook multiplication, we can directly perform the polynomial reduction. We observe that $x^p \equiv x + 1 \bmod x^p - x - 1$, which indicates that the uppermost coefficient

(x^p) during rotation must be additionally added to the before lowermost coefficient. As we indicated before, we want to rotate either of the input polynomials. Applying this strategy to the $\mathcal{R}_3$ polynomial, would result in an increased coefficient range of $[-2, 2]$ due to the extra addition during polynomial reduction. We would require a 5-way multiplexer instead of a 3-way multiplexer, increasing both area and randomness demand. Thus, we choose to rotate the public $\mathcal{R}_q$ input polynomial and perform polynomial reduction in the same domain.

3.2 Modular Reductions

For Streamlined NTRU Prime decapsulation, we require two different modular reductions.

3.2.1 Reduction Modulo q

This reduction is only applied for the accumulation within the $\mathcal{R}_q$ polynomial multiplications. We decide to employ usage of the non-negative modular representation in the interval $[0, q)$ only since in the centered representation we would need to check both for underflows and overflows. Therefore, the value to reduce only grows by a maximum of one bit and can only provoke an overflow. Thus, a conditional subtraction by q suffices resulting in two possible approaches:

1. Assuming the accumulation produces a carry-out bit which we use to conditionally subtract q from the result. By this, our value always remains correctly in $\mathbb{Z}_q$, but not necessarily in the interval $[0, q)$. Therefore, a final pass over the polynomial is required to reduce it to the minimal interval.
2. Subtract q from all accumulation results and obtain the carry bit from that operations. If this is 1, we know an underflow occurred. Thus, we can use the carry bit to multiplex securely between the original accumulation value and the subtracted value. This keeps all intermediate values in the minimal interval $[0, q)$.

3.2.2 Reduction Modulo 3

For the modulo 3 reduction, we have given an input from $\mathbb{Z}_q$ and want to reduce it to $\{-1, 0, 1\}$. We start with an unsigned 13-bit number $z[12:0]$ and repeatedly exploit the relation $2 \equiv -1 \bmod 3$. Note that all operations here also carried out in masked domain, but we omit the masking notation when dealing with arithmetic modulo 3.

$$\begin{aligned} z[12:0] &\equiv 2z[12:1] + z[0] \equiv -z[12:1] + z[0] \bmod 3 \\ &\equiv -2z[12:2] - z[1] + z[0] \equiv z[12:2] - z[1] + z[0] \equiv \ldots \bmod 3 \\ &\equiv \sum_{i=0}^{6} z[2i] - \sum_{i=0}^{5} z[2i+1] \bmod 3 \end{aligned} \tag{7}$$

The result of this computation ranges from -6 to 7 and is represented by a signed 4-bit integer $y[3:0] = -2^3 y[3] + y[2:0]$. We again exploit the above relation:

$$\begin{aligned} -2^3 y[3] + y[2:0] &\equiv y[3] + y[2:0] \equiv y[3] + 2y[2:1] + y[0] \bmod 3 \\ &\equiv y[3] - y[2:1] + y[0] \equiv y[3] - 2y[2] - y[1] + y[0] \bmod 3 \\ &\equiv y[3] + y[2] - y[1] + y[0] \bmod 3 \end{aligned} \tag{8}$$

This results in a value ranging from -1 to 3, represented by a signed 3-bit integer $x[2:0] = -4x[2] + x[1:0] \equiv x[1:0] - x[2] \bmod 3$. This value can already be mapped to a value $w[1:0] \in \{-1,0,1\}$ efficiently:

$$w[0]^{(0:d)} = x[0]^{(0:d)} \oplus x[1]^{(0:d)} \oplus x[2]^{(0:d)} \tag{9}$$

$$w[1]^{(0:d)} = \left(x[0]^{(0:d)} \land x[1]^{(0:d)}\right) \oplus x[1]^{(0:d)} \oplus x[2]^{(0:d)} \tag{10}$$

One additional point to consider is that this modulo 3 calculation assumes an unsigned 13-bit number. However, in the NTRU Prime specification, the modulo 3 operation is used on signed 13-bit numbers, in the interval $[-q/2, q/2]$ [BBC+20]. This means that numbers in the interval $[q/2, q)$ must be treated slightly differently, as these were originally negative. However, the solution is simple: since $q = 4591$, and $4591 = 1 \bmod 3$, we simply have to add 1 to the final result if the original number was in the interval $[q/2, q)$. This addition can be in a similar way to the multiplication in $\mathcal{R}_3$ (see section 3.1.2).

3.3 Weight Check

Let $r'[0:1, 0:p-1]^{(0:d)}$ be an array of p shared two-bit numbers. Valid values are $(0,0)$, $(0,1)$, $(1,1)$ if the signed representation is used, and $(0,0)$, $(0,1)$, $(1,0)$ for the unsigned representation. We wish to check if exactly w array elements are non-zero. Thus, the basic idea depends on the chosen representation.

Signed. We accumulate all $r'[0,:]^{(0:d)}$ values together, with a secure $\lceil \log_2 w \rceil$-bit adder.

Unsigned. We compute $r'[0,:]^{(0:d)} \lor r'[1,:]^{(0:d)}$ and accumulate the resulting shared bit vector with a $\lceil \log_2 w \rceil$-bit adder.

It follows that the signed representation demands less non-linear Boolean operations. For the secure adder, the same adder as used for the polynomial multiplications is applied.

Following this, we then bit-wise XOR the shared adder output with the public target weight w, and then OR all bits of the result together to a single shared result bit.

The overwriting of r' can be performed with a secure 2-way multiplexer deciding between the secret r' and the fixed public vector $(1,1,\dots,1,0,0,\dots,0)$.

3.4 Rounding

For rounding, we first perform a reduction of the coefficient modulo 3 and then subtract the result from the original coefficient. As a result of the modulo operation, we obtain two masked bits $a[1:0]^{(0:d)} \in \{(0,0),(0,1),(1,1)\}$. With this, we want to

1. add 1 for $a[1:0]^{(0:d)} = (1,1)$
2. add $q-1$, which is analogue to subtracting 1, for $a[1:0]^{(0:d)} = (0,1)$, and
3. add zero for $a[1:0]^{(0:d)} = (0,0)$

One way to achieve this is by multiplexing securely between $q-1$, 1 and zero depending on $a[1:0]^{(0:d)}$, which in turn would include more non-linearity. To avoid this, we can construct the value $a[0]^{(0:d)} \cdot (q-1) - a[1]^{(0:d)} \cdot q$ and add that to the initial coefficient. In other words, this value consists of $a[0]^{(0:d)}$ in all binary positions where $q-1$ is 1, except the least significant bit, where it consists of $a[0]^{(0:d)} \oplus a[1]^{(0:d)}$. For the addition, we can re-use the addition-reduction procedure as used for polynomial multiplication.

3.5 SHA-512

SHA-512 employs a Merkle-Damgård construction processing a 512 bit state divided into eight 64 bit words A, B, C, D, E, F, G, H. In order to update the state, SHA-512 implements seven adders (modulo 2^{64}), the two functions Σ_0 and Σ_1, and the functions SHA-Ch and

SHA-Ma. The former two functions Σ_0 and Σ_1 consist of simple shift operations by three different values for each function processing A and E, respectively. The outputs of the shifts are added together by XOR operations. SHA-Ch and SHA-Ma are both non-linear function processing E, F, G and A, B, C, respectively.

For our masked hardware implementation, we can secure the seven adders by applying the concept of the masked adder introduced in Section 3.1. We instantiate a complete 64-bit adder to realize the correct addition. Masking Σ_0 and Σ_1 can be accomplished in a straightforward way since the shift operations do not introduce additional implementation overhead in hardware and all XOR gates can simply be replaced by secure XOR gates.

Finally, SHA-Ch and SHA-Ma are bit-wise operations, that can be implemented in parallel to match the width of the adder to be used. Hence, we can modify the formulas for both to reduce the number of non-linear gates in order to minimize the amount of required randomness and the area overhead leading to

$$\begin{aligned}\mathsf{SHA\text{-}Ch}(E,F,G) &= (E \wedge F) \oplus (\overline{E} \wedge G) = (E \wedge F) \oplus ((E \oplus 1) \wedge G)\\ &= (E \wedge (F \oplus G)) \oplus G \qquad (11)\\ \mathsf{SHA\text{-}Ma}(A,B,C) &= (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)\\ &= (A \wedge (B \oplus C)) \oplus (B \wedge C). \qquad (12)\end{aligned}$$

3.6 Encoding, Decoding & Comparison

Streamlined NTRU Prime defines multiple en- and decoding algorithms for transforming polynomials in $\mathcal{R}_3$ and $\mathcal{R}_q$ to and from byte arrays [BBC+20]. Decoding the ciphertext and public key can be done unmasked as they are both public. We use the decoder described in [PMT+22]. For decoding the secret polynomials f and g^{-1}, we also use the decoder from [PMT+22], and apply masking afterwards. However, to compute the confirmation hash and session key, we need to securely encode r' into a byte array. For this, we apply masking to the $\mathcal{R}_3$ encoder from [PMT+22]. As the encoder only consists of a shift register and a 2-bit adder, this is straightforward.

In the original algorithm specification, the recomputed ciphertext polynomial c' is encoded (line 10 in Algorithm 1) before the ciphertext comparison (line 14), using an $\mathcal{R}_q$ encoder. However, the $\mathcal{R}_q$ encoder requires a 16-bit multiplication which would be prohibitively expensive to implement securely. We instead compare the ciphertext polynomial coefficients directly, after which we compare the confirmation hashes. This allows us to not have to implement the masked $\mathcal{R}_q$ encoder. The masked ciphertext comparison is straightforward: We do a bit-wise secure XOR of the two ciphertext coefficients, and then repeatedly OR the output together.

4 Implementation

After introducing the theoretical background of masking all required operations, we now discuss the implementations of each building block. Afterwards, we briefly discuss the generation of fresh randomness that is necessary to achieve a side-channel protected implementation.

4.1 Building Blocks

In order to implement the operations described in Section 3, we define the following modules:

Add13 pipelined 13 bit Sklansky adder with carry-in for usage in polynomial multiplication, weight check and rounding

CSubQ	pipelined 13 bit Sklansky adder with one operand being fixed to the two's complement of q, with subsequent multiplexer
Mod3	pipelined reduction from 13 bit modulo 3
Mul3	$\mathbb{Z}_3$ multiplier
Mux3	3-way mux with public input and secret select signal
Mux2	2-way mux with secret input and secret select signal
SHA-Ch	64 bit wide SHA-Ch step
SHA-Ma	64 bit wide SHA-Ma step
Add64	pipelined 64 bit Sklansky adder

4.1.1 Add13 and Add64

In their work [BG22], Bache and Güneysu compare the Brent-Kung, Kogge-Stone, and Sklansky adder architectures in the context of Boolean masking. For gate-level masking, the Sklansky adder turns out to be the optimal choice of of these, having the same low latency like Kogge-Stone but less randomness demand, while having a lower latency than Brent-Kung at the cost of slightly more randomness.

The 13-bit Sklansky adder with carry-in deployed in our implementation is shown in Figure 1a. For input bits $a[i]^{(0:d)}, b[i]^{(0:d)}$ where $i \in \{0, \dots, 12\}$, we compute in each circle:

$$g[i]^{(0:d)} = a[i]^{(0:d)} \wedge b[i]^{(0:d)} \tag{13}$$

$$p[i]^{(0:d)} = a[i]^{(0:d)} \oplus b[i]^{(0:d)} \tag{14}$$

Note that the dotted circle indicates the uppermost p, g from a previous addition which can be used to realize cascaded additions. Each square node has four inputs, the two "left" inputs $g_l^{(0:d)}, p_l^{(0:d)}$ and the two "right" inputs $g_r^{(0:d)}, p_r^{(0:d)}$, and computes the following outputs:

$$g^{(0:d)} = g_l^{(0:d)} \oplus \left(p_l^{(0:d)} \wedge g_r^{(0:d)}\right) \tag{15}$$

$$p^{(0:d)} = p_l^{(0:d)} \wedge p_r^{(0:d)} \tag{16}$$

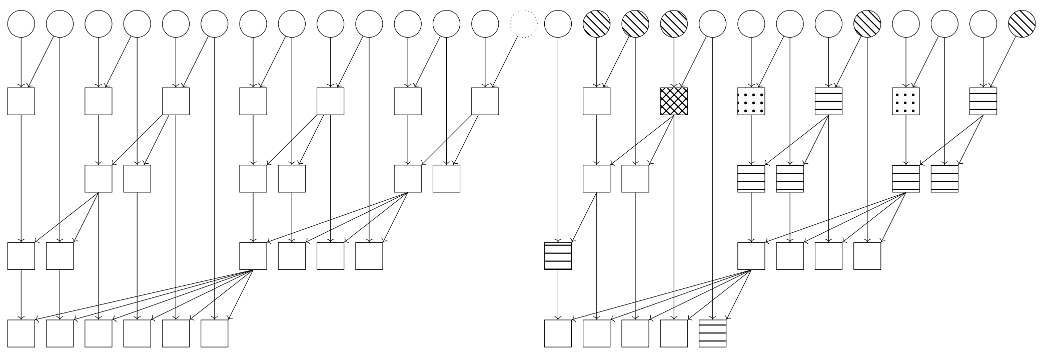
(a) 13 bit Adder with Carry In (b) CSubQ with optimizations for $q = 4591$

Figure 1: Sklansky Adder Constructions

Finally, note that all leaf nodes do not need to compute $p^{(0:d)}$, as only the final $g^{(0:d)}$ values are needed. The only exception is the uppermost $p^{(0:d)}$, which might be needed for a cascaded addition.

The 64-bit adder works equivalently, though with a total of six levels. In this case, we do not need a carry-in or carry out.

4.1.2 CSubQ

For the conditional subtraction with q, we take a similar approach. We instantiate another Sklansky adder with one public operand fixed to the two's complement of q. Then, after each addition (let us denote the result here as $x^{(0:d)}$), we perform this subtraction by q and obtain $(q-x)^{(0:d)}$ as well as the shared carry-out bit. Using this, we multiplex securely between $x^{(0:d)}$ and $(q-x)^{(0:d)}$ selecting the former if the carry-out is one (indicating an underflow has occurred) and else the latter one.

The fixed input already enables vast optimizations by the synthesizer. Further improvements could be possible by optimizing the adder architecture itself for a fixed operand. Since we know the positions of the zeros, we could simplify our adder as depicted in Figure 1b. However, note that we did not implement these optimizations, and have left them for future work.

For the computation of all p values below the first row nothing changes. However, we can completely omit computing the first row of p, g as described in Equation 13 and Equation 14. Instead, we know, given an input $a[12:0]^{(0:d)}$, for each circle in Figure 1b that

$$g[i]^{(0:d)} = \begin{cases} a[i]^{(0:d)} & \text{if } (-q)[i] = 1 \\ 0 & \text{else} \end{cases} \tag{17}$$

$$p[i]^{(0:d)} = \begin{cases} \overline{a[i]^{(0:d)}} & \text{if } (-q)[i] = 1 \\ a[i]^{(0:d)} & \text{else} \end{cases} . \tag{18}$$

In Figure 1b, the circles filled with the diagonal line pattern indicate that the fixed input bit of the two's complement of q is one. For the squares, we have four different cases now:

Non-filled	Computed as before.
Grid	$g^{(0:d)} = g_l^{(0:d)} \oplus \left(p_l^{(0:d)} \wedge g_r^{(0:d)}\right) = g_l^{(0:d)} \oplus \left(p_l^{(0:d)} \wedge 0\right) = g_l^{(0:d)}$
Dotted	$g^{(0:d)} = g_l^{(0:d)} \oplus \left(p_l^{(0:d)} \wedge g_r^{(0:d)}\right) = 0 \oplus \left(p_l^{(0:d)} \wedge 0\right) = 0$
Horizontal lines	$g^{(0:d)} = g_l^{(0:d)} \oplus \left(p_l^{(0:d)} \wedge g_r^{(0:d)}\right) = 0 \oplus \left(p_l^{(0:d)} \wedge g_r^{(0:d)}\right) = p_l^{(0:d)} \wedge g_r^{(0:d)}$

4.1.3 Mod3

The architecture to compute this is depicted in Figure 2. For the secure additions and subtractions, we employ simple ripple-carry adders as parallel prefix adders have no advantage for these small bit widths.

4.1.4 Mux3 and Mux2

Mux3 can be implemented with three pipeline stages as the HPC2-SecAND gadget has a delay of two cycles for one input and 1 clock cycle for the other one. We instantiate 13 of these two-bit MUXes in parallel in order to feed Add13 without idling.

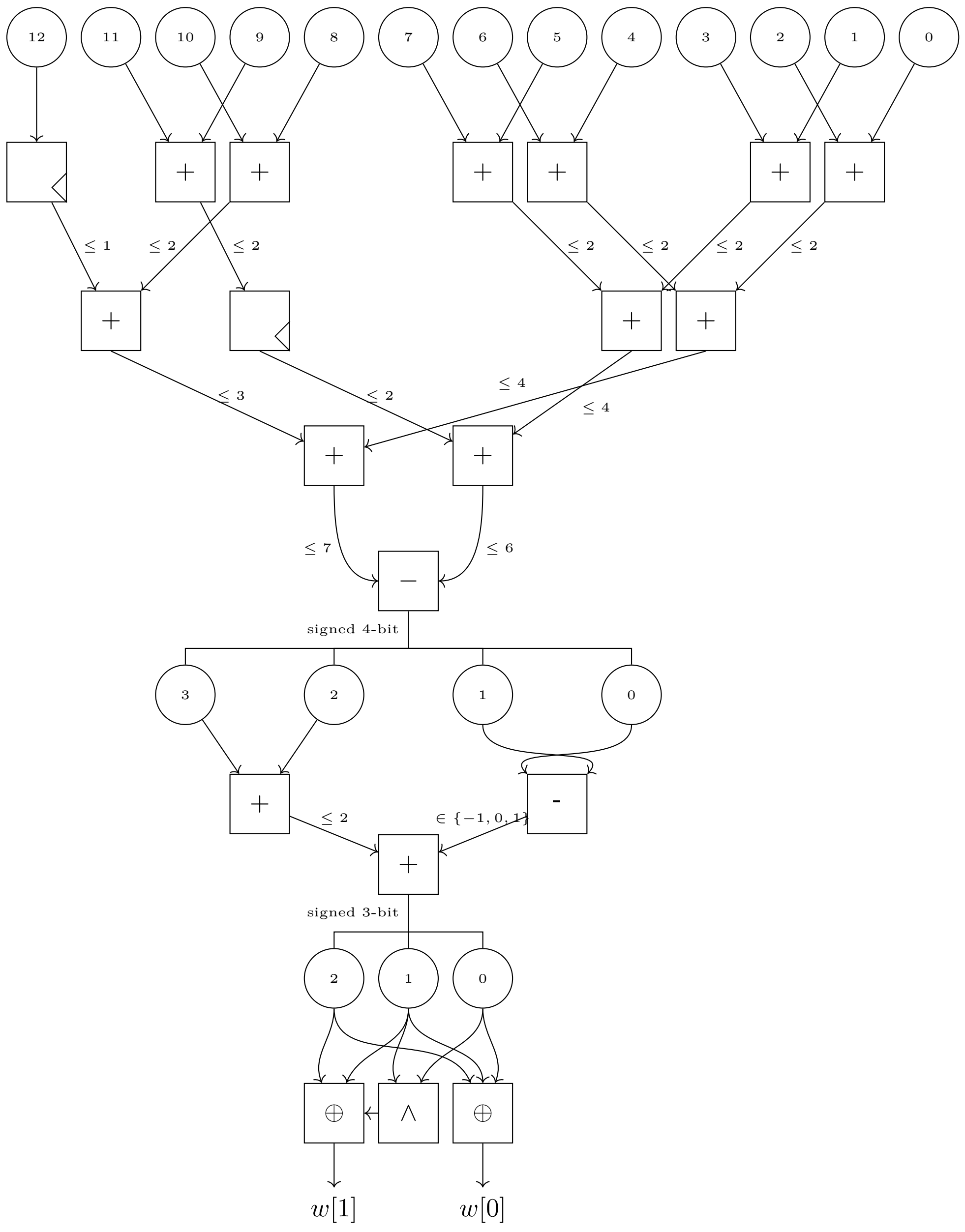

Figure 2: Mod3 module

For Mux2, which has two secret data input and a secret select input, we have a delay of two cycles. We instantiate 13 of these MUXes in the $\mathcal{R}_q$ multiplier to select between the CSubQ output and the non subtracted value. We also instantiate two multiplexer during the weight check calculation, to select between the original r' and the fixed vector. Finally, we use eight multiplexer to select between the encoded r' and ρ after the ciphertext comparison.

4.2 Randomness Generation

The generation of a large amount of randomness is comparatively easy to solve on an FPGA, but there are still different approaches:

1. Pre-generated randomness that is stored in Block-RAMs (BRAMs). This approach consumes no LUT, at the cost of a large amount of BRAMs, and is only valid for testing.
2. Pseudorandom Number Generators (PRNGs) using Linear Feedback Shift Registers (LFSRs). This approach is very lightweight, but does not create cryptographically secure randomness. However, for masking statistical randomness is sufficient, as shown, e.g., in [WDMM20]. There, just three LUTs were needed to create one random bit per cycle.
3. True Random Number Generators (TRNGs). As an example, the jitter of a freely looping ring oscillator or PLLs can be sampled. Generating secure instances is non-trivial [BBA+12] and often provides only limited throughput.
4. A hash based Extendible Output Function (XOF). This is the most expensive but also highest quality Random Number Generator (RNG). A Keccak-based XOF would likely be suitable.

5 Evaluation

After introducing our implementation concept, we present in this section the corresponding implementation results. Furthermore, we formally verify our building blocks in order to demonstrate their protection against side-channel attacks. Eventually, we compare our hardware implementation of Streamlined NTRU Prime to a hardware design of Saber.

5.1 Implementation Results

We implement our design on a Xilinx Artix-7 device, using Vivado v2021.2 (64-bit), for the sntrup761 parameter set. We also synthesize our design for an ASIC, using the 45nm Nangate open cell library. Table 1 shows the latency, frequency, and peak randomness demand per module and masking degree. As can be seen there, the cycle count is dominated by the three polynomial multiplications which take 93 % of all total cycles. At the same time, the peak randomness is always set by the 64-bit adder in the SHA-512 module. While the total cycle count is independent of the masking order, the maximum clock frequency varies: On an FPGA and at masking order 1 and 3, the design reaches 200 MHz, but the maximum frequency is lower for masking order 2, 4 and 5. For all three, the critical path lies in the SHA-512 module. For the ASIC, the design reaches a higher maximum clock frequency than the FPGA at first order, with 207 MHz. However, as the masking order increases, the maximum frequency drops off faster, reaching just 75 MHz at fifth order, and 100 MHz at sixth order. Here, the critical path also lies in the SHA-512 module.

In Table 2, the area demand per module and masking degree is shown for Artix-7 FPGA. As expected, the area increases vastly with increasing masking degree. Interestingly, for all masking orders, the SHA-512 dominates the resource cost consuming roughly 61 % of all

Table 1: Latency, frequency, and randomness results after Place and Route (PnR). Note that the cycle count for SHA-512 is for a single 1024-bit block. We did not perform PnR for orders 6 and 7 for an FPGA, as they no longer fit into an Artix-7 FPGA.

Module	Cycle Count	Maximum Randomness (bits per cycle)						
		Masking Degree						
		1	2	3	4	5	6	7
Decap	1 870 049	52	82	156	252	370	510	672
Encode $\mathcal{R}_3$	765	4	12	24	40	60	84	112
C' comp.	4 050	14	42	84	140	210	294	392
Decrypt	1 171 270	96	288	576	960	1 440	2 016	2 688
mod 3	29	46	138	276	460	690	966	1 288
Mul. $\mathcal{R}_3$	581 409	6	18	36	60	90	126	168
Weight calc.	9 145	42	126	252	420	630	882	1 176
Re-Encrypt	581 501	123	369	738	1 230	1 845	2 583	3 444
Rounding	812	123	369	738	1 230	1 845	2 583	3 444
Mul. $\mathcal{R}_q$	580 646	103	309	618	1 030	1 545	2 163	2 884
Adder 13-bit	10	32	96	192	320	480	672	896
13 Mux2	2	13	39	78	130	195	273	364
13 Mux3	3	26	78	156	260	390	546	728
SHA-512	7 845	310	930	1 860	3 100	4 650	6 510	8 680
SHA-Ma	2	128	384	768	1 280	1 920	2 688	3 584
SHA-Ch	2	64	192	384	640	960	1 344	1 792
Adder 64-bit	14	310	930	1 860	3 100	4 650	6 510	8 680
Total	1 870 049	310	930	1 860	3 100	4 650	6 510	8 680
FPGA	**$f_{\max}$ (MHz)**	200	182	200	169	179	–	–
	Latency (ms)	9.35	10.3	9.35	11.1	11.4	–	–
ASIC	**$f_{\max}$ (MHz)**	207	165	148	91	75	100	–
	Latency (ms)	9.03	11.3	12.6	20.5	24.9	18.7	–

LUT and FF. The next most expensive operation is the rounding during the re-encryption, followed by the $\mathcal{R}_q$ polynomial multiplication. When comparing the ratios of cycle counts and the resources consumed, it is apparent that the current SHA-512 implementation is sub-optimal: it is too expensive when considering the whole design. In particular, the full 64-bit adder is oversized. For a better ratio of cycles and resources consumed, using a smaller, e.g., 16 bit adder multiple times for each 64 bit addition, would be more efficient, while only adding a comparatively minor number of cycles. Doing so would also allow the SHA-Ch and SHA-Ma gadgets to have a smaller widths, saving further resources. Finally, this would reduce the maximum of random bits used per cycle.

In Table 3, we list the gate equivalent area demand per module and masking degree for a ASIC. As we did not have access to a memory macro, we list the memory footprint separately. We see similar behavior to the FPGA area demand, with the SHA-512 dominating the area demand, followed by the rounding during the re-encryption. The total GE also grows significantly as the masking order increases, while the SRAM usage grows more slowly.

Different Masking Degrees for Decrypt and Re-Encrypt. In [ABH+22], the authors reason that re-encryption must be protected at a higher level than decryption during decapsulation. Our design, and all building blocks can be easily adapted to any masking order allowing a flexible configuration. However, doing so would decrease the modules that can be re-used across the design, e.g., the $\mathcal{R}_q$ multiplier which is used both during decryption and re-encryption.

Table 2: FPGA area results after PnR. Note that this does not include the area needed for randomness generation. Not listed is the Digital Signal Processor (DSP) usage: 4 DSPs are needed as multipliers in the decoder, regardless of the masking order.

Module	Masking Order											
	1			2			3			4		
	LUT	FF	BR	LUT	FF	BR	LUT	FF	BR	LUT	FF	BR
Decap	2270	1180	4.5	2493	1575	6	3088	2256	6	3766	2980	8
Encode $\mathcal{R}_3$	61	52	0	77	80	0	104	115	0	131	157	0
C' comp.	278	263	0	503	530	0	855	895	0	1273	1358	0
Decrypt	1743	1602	0	2680	3225	1.5	4847	5451	1.5	7276	8282	1.5
mod 3	542	719	0	1197	1528	0	2274	2638	0	3474	4049	0
Mul. $\mathcal{R}_3$	470	208	0	329	319	1	489	476	1	665	675	1
Weight calc.	528	612	0	1066	1286	0	1947	2194	0	2941	3350	0
Re-Encrypt	2017	2450	0.5	4138	5180	1	7755	8936	1	11696	13714	1
Rounding	1888	2387	0	4080	5108	0	7695	8851	0	11636	13616	0
Mul. $\mathcal{R}_q$	1846	2148	1.5	3693	4419	2	6686	7554	2	9885	11553	2.5
Adder 13-bit	627	715	0	1352	1545	0	2523	2690	0	3729	4150	0
13 Mux2	182	221	0	390	468	0	676	806	0	1040	1235	0
13 Mux3	211	286	0	463	625	0	848	1107	0	1314	1723	0
SHA-512	11684	12035	2	22493	23880	3	38370	39406	8	56207	59097	9
SHA-Ma	1528	1664	0	3439	3840	0	6624	6912	0	10197	10880	0
SHA-Ch	896	1088	0	1920	2304	0	3584	3968	0	5440	6080	0
Adder 64-bit	5996	5663	0	12352	23162	0	22506	21770	0	32702	33740	0
Total	19923	19725	8.5	36340	39209	13.5	62498	65463	18.5	91731	98726	22
Total w/o SHA	8239	7690	6.5	13847	15329	10.5	24128	26057	10.5	35524	39629	13
SHA Pct.	58.4	61.0	23.5	61.9	60.9	22.2	61.4	60.2	43.2	61.3	59.9	40.9

Table 3: ASIC area results in gate equivalents (GE), using the 45nm Nangate open cell library. The area does not include SRAM cells, which are listed separately. Note that this does not include the area needed for randomness generation. The area for the Encode $\mathcal{R}_3$ entity is not available for masking orders one through three, as it was merged with its parent entity.

Module	Masking Order					
	1	2	3	4	5	6
Decap	14 703	18 520	23 632	29 561	37 176	46 078
Encode $\mathcal{R}_3$	n/a	n/a	n/a	1 130	1 447	1 799
C' comp.	2 052	4 103	6 943	10 571	14 981	20 208
Decrypt	14 727	28 021	46 047	68 449	95 889	128 306
mod 3	5 744	12 216	21 101	32 386	46 085	62 295
Mul. $\mathcal{R}_3$	2 452	3 436	4 756	6 065	8 025	10 412
Weight calc.	5 688	11 202	18 584	27 825	38 949	51 986
Re-Encrypt	29 615	56 009	90 348	127 595	176 907	234 225
Rounding	29 057	55 375	89 636	126 818	176 050	233 295
Mul. $\mathcal{R}_q$	25 244	45 906	73 131	103 820	143 784	190 601
Adder 13-bit	7 115	14 482	24 375	36 840	51 817	69 367
13 Mux2	1 607	3 510	6 144	9 511	13 611	18 442
13 Mux3	2 015	4 468	7 910	12 356	17 811	24 366
SHA-512	114 570	218 453	354 242	519 545	719 019	950 021
SHA-Ma	12 416	29 440	53 674	85 120	123 776	169 642
SHA-Ch	7 914	17 280	30 250	46 826	67 008	90 794
Adder 64-bit	55 503	114 820	195 205	296 601	419 131	563 160
Total	201 112	373 349	600 100	870 124	1 204 839	1 594 022
Total w/o SHA	86 542	154 896	245 858	350 579	485 820	644 001
SHA Pct.	57.0	58.5	59.0	59.7	59.7	59.6
SRAM (bits)	189 440	246 272	294 912	343 296	393 216	443 392

Table 4: Verification results of the protected submodules using VERICA. We report for each design the number of combinational gates, memory gates and the verification time. The verification of the expected security order is indicated by green check marks.

Design	**First Order**				**Second Order**				**Third Order**			
	comb.	mem.	sec.	time	comb.	mem.	sec.	time	comb.	mem.	sec.	time
Mux2	16	17	1✓	0.383 s	39	36	2✓	0.402 s	72	62	3✓	20.609 s
Mux3	28	31	1✓	0.385 s	72	69	2✓	0.521 s	136	122	3✓	4.985 h
Mod3	586	774	1✓	1.125 s	1464	1581	2✓	90.82 min	2742	2668	–	∞
Mul3	89	94	1✓	0.412 s	221	204	2✓	23.591 s	413	356	–	∞
SHA-Ch	16	17	1✓	0.404 s	39	36	2✓	0.420 s	72	62	3✓	26.355 s
SHA-Ma	28	26	1✓	0.386 s	72	60	2✓	0.928 s	136	108	3✓	11.5 h

Table 5: Comparison with previous work. All implementations are synthesized for Artix-7, except for Kyber, which is synthesized for Virtex-7. Note that for ASIC, we are the first to report a fully masked implementation of any PQC scheme.

Scheme	Area				Cycle cnt.	f_{max}	max rand.	d	Ref.
	LUT	FF	BRAM	DSP		Mhz	bits / cycle		
sNTRUp-761	36 789	22 700	3.5	9	10 989	137	0	0	[PMT+22]
sNTRUp-761	6 279	3 086	3.0	7	85 628	131	0	0	[PMT+22]
sNTRUp-761	19 923	19 725	8.5	4	1 870 049	200	310	1	**this**
sNTRUp-761	36 340	39 209	13.5	4	1 870 049	182	930	2	**this**
sNTRUp-761	62 498	65 463	18.5	4	1 870 049	200	1 860	3	**this**
sNTRUp-761	91 731	98 726	22.0	4	1 870 049	169	3 100	4	**this**
Saber	19 299	21 977	0.0	64	72 005	125	DNR	1	[AMD+21]
Kyber-512	152 860	DNR	489.5	76	137 738	100	DNR	1	[KNAH22]

5.2 Side-Channel Evaluation

In order to evaluate the protection against side-channel attacks, we rely on formal verification of each of our submodules. Please note, evaluating the entire decapsulation by practical measurements is out of scope for typical setups due to the huge amount of required clock cycles. To this end, we formally verify the security of each module by using the recently presented verification tool VERICA [RFSG22]. VERICA is constructed based on the verification concepts developed in the side-channel analysis tool SILVER [KSM20] and the fault-injection analysis tool FIVER [RBSS+21]. The formal verification of a target design is performed based on its (Verilog) gate-level netlist which is transformed into a Direct Acyclic Graph (DAG) serving as circuit model. Each node in the DAG is associated with a Binary Decision Diagram (BDD) representing the Boolean function of the corresponding gate. This data structure allows efficient applications of statistical checks verifying side-channel security in the glitch-extended probing model and composability notions. To this end, we analyze our modules in the glitch-extended d-probing model for different security orders. The corresponding results are shown in Table 4. Note that all modules pass first- and second-order verification, while third-order verification is too complex for Mod3 and Mul3. For the Add13 and Add64 modules, we use the implementation by Bache and Güneysu [BG22] which is verified to be secure practically.

5.3 Comparison

In Table 5, we compare our implementation against an unmasked implementation of Streamlined NTRU Prime as well as two first-order masked FPGA implementations. To the best of our knowledge, we are the first to report a higher-order full FPGA implementation of any PQC scheme, and the first to report a masked ASIC PQC implementation, thus we cannot compare to other higher-order implementations. As expected, the two unmasked implementations (one "high speed" and one "low area") both are smaller and faster. Additionally, the masked Saber implementation has a comparable LUT and FF footprint to our first-order implementation and uses no BRAM, but uses significantly more DSP. However, it is about an order of magnitude faster. In contrast to this, the masked Kyber-512 implementation is vastly bigger even than our fourth-order implementation, but only faster by a factor of 6.8 compared to our first-order implementation.

Moreover, both the Saber and the Kyber-512 implementations only support first order, while our design can easily be instantiated at an arbitrary level, allowing protections against more advanced attacks. Finally, our masked gadgets have been verified to be secure and we do not need any masking conversion which may be used in future attacks.

6 Discussion

In this section, we address and discuss potential improvements and the huge overhead introduced by masking the symmetric core in Streamlined NTRU Prime. Additionally, we briefly discuss the application of our concepts and approaches to Kyber.

6.1 Gate-level Masking

There are several advantages in a gate-level masked implementation. First, it is very easy to adapt to an arbitrary masking order. This obviously reduces the time required for development. Moreover, no masking conversion can be attacked, since there is none. The masking conversion were the targets in the attacks against a first-order and third-order masked Saber implementation [NDJ21, NWDP22]. Additionally, it is usually easy to exchange the underlying gadgets by others with the same latency properties. For example, it could be possible to achieve a fault-secure implementation easily by deploying the work from [FRBSG22].

6.2 Potential Improvements

We leave several potential improvements as future work and address them here. The most expensive operation from the latency view point is polynomial multiplication. The two $\mathcal{R}_q$ multiplications take 62 % of the decapsulation cycle counts and the multiplication in $\mathcal{R}_3$ takes another 31 %. To speed this up, it is possible to instantiate more adders in parallel at the cost of slightly more area and a potentially higher amount of randomness per clock cycle, depending on the grade of deployed parallelism. Thus, halving the latency of both multipliers results in an 47 % speed-up at the cost of approximately 8 % more gate equivalents for the first-order ASIC implementation.

Moreover, a potential area reduction can be achieved by optimizing the CSubQ module. This would likely also have an impact to the randomness demand.

Additionally, we want to stress that the specified encoding procedure for polynomials $\mathcal{R}_q$ is suboptimal for hardware implementations, as it includes multiplications. This accounts for the four DSP slices required in the FPGA implementation and about 7.3 kGE in the ASIC implementation. However, alternatives would increase transmission sizes and would obviously require a change of specification.

6.3 Symmetric Core

As discussed in Section 5.1, masking the symmetric core (i.e., SHA-512) in Streamlined NTRU Prime consumes a considerable large part of the entire implementation's footprint, and has the highest per cycle randomness consumption. Nevertheless, secure and hardened SHA-512 implementations are widely deployed in industry, and can for example be found in smartcards and secure elements [NXP22]. Thus, one could assume that a secure SHA-512 is already available, and does not need to be implemented. If we exclude the SHA-512 from the area consumption (cf. Table 2 and Table 3), then the design is not only surprising small at first order, but the area overhead is much more moderate with an increasing masking order.

Another possibility would be to replace the 64-bit Sklansky adder that is deployed in the SHA-512 module by a smaller one, trading area for latency. Moreover, it is possible to deploy no additional adder for the SHA-512 module at all by re-using the secure adder from the polynomial multiplication module. In this case, five consecutive 13-bit additions would yield the 64-bit addition. This would require cleverly scheduling the additions required by SHA-512 such that the 13-bit adder pipeline is maximally occupied. As can be seen from Table 2 and Table 3, the 64-bit Sklansky adder occupies about half of the area of the SHA-512 module, and about a quarter of the overall area.

Additionally, in order to reduce the total area overhead introduced by the masked symmetric core in Streamlined NTRU Prime, the SHA-512 could be replaced by an implementation based on Keccak [BDPA13]. As Keccak does not use an adder internally, it is significantly easier and cheaper to mask. Most notably, it can be implemented with a very low amount of fresh randomness [BDN+13]. In addition, as the critical path lies in the SHA-512 module for both FPGAs and ASICs, using Keccak would likely increase the maximum achievable clock frequency. However, this would deviate from the Streamlined NTRU Prime specification and would not be interoperable with other Streamlined NTRU Prime implementations.

6.4 Applicability to Kyber

The efficiency of our gate-level masking is built upon the fact that the three polynomial multiplications that are carried out each include a secret polynomial with ternary coefficients, where the other one is either small and secret as well, or has a big coefficient modulus and is public. This enables us to perform schoolbook multiplication in Boolean domain. Notably, Kyber has a similar property: Here, *all* polynomial multiplications have one public input polynomial with "big" coefficients modulo $q = 3\,329$.

Moreover, the polynomial degree is far smaller with 256 compared to 677 for Streamlined NTRU Prime, enabling a faster multiplication. For Kyber, $256^2 = 65\,536$ coefficient additions are to be performed per polynomial multiplication, whereas Streamlined NTRU Prime with $p = 761$ requires $p^2 + p = 579\,882$ coefficient additions. However, Kyber requires more multiplications to be performed: for $k \in \{2, 3, 4\}$, it requires $k^2 + 2k$ polynomial multiplications, as well as $k^2 + 4k - 1$ polynomial additions, whereas Streamlined NTRU Prime constantly requires three polynomial multiplications.

We compare the cost in terms of estimated number of coefficient additions in Table 6. As can be seen there, Kyber constantly requires less coefficient additions than Streamlined NTRU Prime in the regarding security categories.

Another advantage for Kyber is that during key generation it features no operations that are infeasible to mask in Boolean domain, which is in contrast to Streamlined NTRU Prime, where this is not possible. The most complex remaining operations in Kyber both for key generation and decapsulation are (de-)compression and sampling for a centered binomial distribution using a Keccak output stream, both of which are feasible in Boolean domain.

Still, we want to stress that a gate-level masked Kyber implementation would require an Number-Theoretic Transform (NTT) core and would have another big downside: Kyber requires to extend a seed into a public matrix of polynomials, which are assumed to be in NTT domain. Since the implementation would not perform multiplication in NTT domain,

Table 6: Comparison to Kyber

NIST Category	Scheme	Polynomial size	Module size	Number of coefficient additions
I	Kyber-512	256	$k = 2$	527 104
	sNTRUp	653	—	1 281 186
II	sNTRUp	761	—	1 739 646
III	Kyber-768	256	$k = 3$	988 160
	sNTRUp	857	—	2 205 918
IV	sNTRUp	953	—	2 727 486
IV	sNTRUp	1013	—	3 081 546
V	Kyber-1024	256	$k = 4$	1 580 800
	sNTRUp	1277	—	4 896 018

an inverse transform of each polynomial in the matrix would be required, resulting in k^2 inverse NTTs during decapsulation. Finally, it is noteworthy that the fact that Kyber uses the same polynomial ring for all security levels is no advantage for a gate-level-masked implementation, since schoolbook multiplication is used for Streamlined NTRU Prime anyways, which also allows for easy parametrization. On the other hand, Streamlined NTRU Prime changes the coefficient modulus over the parameter sets, which might require manual adjustments.

Overall, we leave this as an interesting open idea for future work.

7 Conclusion

In our work, we have presented the first gate-level masked implementation of any PKC scheme. Notably, it is competitive regarding area demand to other protected PQC implementations while still offering a reasonable latency. The main advantage is the ability to adapt the implementation easily to arbitrary masking order. For the first-order secure instance of the implementation, 19 923 LUTs, 19 725 FFs, and 8.5 BRAMs are utilized, reaching a frequency of 200 MHz. Implemented as an ASIC, the first-oder secure instance consumes 201k GE and 189 kbit SRAM, reaching a frequency of 207 MHz. This results in a latency of only 9.35 ms on an FPGA and 9.03 ms as an ASIC, with a peak demand of fresh randomness of 310 bit per clock cycle. While for higher masking degrees, the latency only increases slightly due to a lower frequency, the randomness demand increases to 3 100 bit per clock cycle for $d = 4$. Nevertheless, further optimization of the hashing module could significantly reduce the area and randomness consumption. Finally, we also analyzed the applicability of our concept to the designated NIST standard algorithm Kyber, finding that gate-level masking could be efficient Kyber as well.

Acknowledgments

We would like to thank Joppe Bos, Daniel J. Bernstein, Bo-Yuan Peng and Bo-Yin Yang for their help. This work was labelled by the EUREKA cluster PENTA and funded by German authorities under grant agreement PENTA-2018e-17004-SunRISE. This work was supported by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany (grants 16KIS1572K, SASVI and 16KISK038, 6GEM). The work described in this paper has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA -

390781972, and by the European Commission under the grant agreement number 101070374 (CONVOLVE).

References

[ABH+22] Melissa Azouaoui, Olivier Bronchain, Clément Hoffmann, Yulia Kuzovkova, Tobias Schneider, and François-Xavier Standaert. Systematic Study of Decryption and Re-encryption Leakage: The Case of Kyber. In Josep Balasch and Colin O'Flynn, editors, *Constructive Side-Channel Analysis and Secure Design - 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*, volume 13211 of *Lecture Notes in Computer Science*, pages 236–256. Springer, 2022.

[ACC+21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial multiplication in NTRU prime. *IACR TCHES*, 2021(1):217–238, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8733.

[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 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 224–245. Springer, 2021.

[AR21] Amund Askeland and Sondre Rønjom. A side-channel assisted attack on NTRU. Cryptology ePrint Archive, Report 2021/790, 2021. https://eprint.iacr.org/2021/790.

[BBA+12] Pierre Bayon, Lilian Bossuet, Alain Aubert, Viktor Fischer, François Poucheret, Bruno Robisson, and Philippe Maurine. Contactless electromagnetic active attack on ring oscillator based true random number generator. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 151–166. Springer, 2012.

[BBC+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 457–485. Springer, Heidelberg, April 2015.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 116–129. ACM Press, October 2016.

[BCLv17] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: Reducing attack surface at low cost. In Carlisle Adams and Jan Camenisch, editors, *SAC 2017*, volume 10719 of *LNCS*, pages 235–260. Springer, Heidelberg, August 2017.

[BDN$^+$13] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2013.

[BDPA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013.

[BG22] Florian Bache and Tim Güneysu. Boolean Masking for Arithmetic Additions at Arbitrary Order in Hardware. *Applied Sciences*, 12(5):2274, 2022.

[BGI$^+$18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 321–353. Springer, Heidelberg, April / May 2018.

[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.

[CGTZ22] Jean-Sebastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. High-order masking of NTRU. Cryptology ePrint Archive, Report 2022/1188, 2022. https://eprint.iacr.org/2022/1188.

[CHK$^+$21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings. *IACR TCHES*, 2021(2):159–188, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8791.

[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.

[DNG22] Elena Dubrova, Kalle Ngo, and Joel Gärtner. Breaking a fifth-order masked implementation of crystals-kyber by copy-paste. *Cryptology ePrint Archive*, 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 TCHES*, 2018(3):89–120, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7270.

[FRBSG22] Jakob Feldtkeller, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. CINI MINIS: Domain isolation for fault and combined security. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1023–1036. ACM Press, November 2022.

[HHP+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked CCA2 secure kyber. *IACR TCHES*, 2021(4):88–113, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9061.

[HPP21] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. Fault-Enabled Chosen-Ciphertext Attacks on Kyber. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 311–334. Springer, 2021.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Heidelberg, August 2003.

[JGCS21] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. A configurable crystals-kyber hardware implementation with side-channel protection. Cryptology ePrint Archive, Report 2021/1189, 2021. https://eprint.iacr.org/2021/1189.

[KA21] Emre Karabulut and Aydin Aysu. FALCON Down: Breaking FALCON Post-Quantum Signature Scheme through Side-Channel Attacks. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 691–696. IEEE, 2021.

[KAA21] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-Trace Side-Channel Attacks on ω-Small Polynomial Sampling: With Applications to NTRU, NTRU Prime, and CRYSTALS-DILITHIUM. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*, pages 35–45. IEEE, 2021.

[KM22] David Knichel and Amir Moradi. Low-latency hardware private circuits. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1799–1812. ACM Press, November 2022.

[KNAH22] Tendayi Kamucheka, Alexander Nelson, David Andrews, and Miaoqing Huang. A masked pure-hardware implementation of kyber cryptographic algorithm. In *International Conference on Field-Programmable Technology, (IC)FPT 2022, Hong Kong, December 5-9, 2022*, page 1. IEEE, 2022.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 787–816. Springer, Heidelberg, December 2020.

[KSM22] David Knichel, Pascal Sasdrich, and Amir Moradi. Generic Hardware Private Circuits Towards Automated Generation of Composable Secure Gadgets. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):323–344, 2022.

[Mar20] Adrian Marotzke. A Constant Time Full Hardware Implementation of Streamlined NTRU Prime. In Pierre-Yvan Liardet and Nele Mentens, editors, *Smart Card Research and Advanced Applications - 19th International Conference, CARDIS 2020, Virtual Event, November 18-19, 2020, Revised Selected Papers*, volume 12609 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2020.

[NDGJ21] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A side-channel attack on a masked IND-CCA secure saber KEM implementation. *IACR TCHES*, 2021(4):676–707, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9079.

[NDJ21] Kalle Ngo, Elena Dubrova, and Thomas Johansson. Breaking masked and shuffled CCA secure saber KEM by power analysis. Cryptology ePrint Archive, Report 2021/902, 2021. https://eprint.iacr.org/2021/902.

[NWDP22] Kalle Ngo, Ruize Wang, Elena Dubrova, and Nils Paulsrud. Side-channel attacks on lattice-based KEMs are not prevented by higher-order masking. Cryptology ePrint Archive, Report 2022/919, 2022. https://eprint.iacr.org/2022/919.

[NXP22] NXP Semiconductors. EdgeLock® SE050: Plug & Trust Secure Element Family – Enhanced IoT security with high flexibility. https://www.nxp.com/se050, 2022. [Online; accessed 13-10-2022].

[PMT$^+$22] Bo-Yuan Peng, Adrian Marotzke, Ming-Han Tsai, Bo-Yin Yang, and Ho-Lin Chen. Streamlined NTRU Prime on FPGA. *Journal of Cryptographic Engineering*, pages 1–20, 2022.

[RBSS$^+$21] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. FIVER - robust verification of countermeasures against fault injections. *IACR TCHES*, 2021(4):447–473, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9072.

[RFSG22] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. VERICA - Verification of Combined Attacks Automated formal verification of security against simultaneous information leakage and tampering. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):255–284, 2022.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR TCHES*, 2020(3):307–335, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8592.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic addition over Boolean masking - towards first- and second-order resistance in hardware. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15*, volume 9092 of *LNCS*, pages 559–578. Springer, Heidelberg, June 2015.

[SPH22] Bo-Yeon Sim, Aesun Park, and Dong-Guk Han. Chosen-ciphertext clustering attack on CRYSTALS-KYBER using the side-channel leakage of barrett reduction. *IEEE Internet Things J.*, 9(21):21382–21397, 2022.

[WDMM20] Felix Wegener, Lauren De Meyer, and Amir Moradi. Spin me right round rotational symmetry for FPGA-specific AES: Extended version. *Journal of Cryptology*, 33(3):1114–1155, 2020.

[XPR$^+$21] Zhuang Xu, Owen Michael Pemberton, Sujoy Sinha Roy, David Oswald, Wang Yao, and Zhiming Zheng. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. *IEEE Transactions on Computers*, 2021.