IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 4, pp. 173–214. DOI:10.46586/tches.v2021.i4.173-214

Masking Kyber: First- and Higher-Order Implementations

Joppe W. Bos[1], Marc Gourjon[1,2], Joost Renes[1], Tobias Schneider[1] and Christine van Vredendaal[1]

[1] NXP Semiconductors, Eindhoven, Netherlands
[2] Hamburg University of Technology, Hamburg, Germany
joppe.bos@nxp.com, marc.gourjon@nxp.com, joost.renes@nxp.com, tobias.schneider@nxp.com, christine.cloostermans@nxp.com

Abstract. In the final phase of the post-quantum cryptography standardization effort, the focus has been extended to include the side-channel resistance of the candidates. While some schemes have been already extensively analyzed in this regard, there is no such study yet of the finalist Kyber.

In this work, we demonstrate the first completely masked implementation of Kyber which is protected against first- and higher-order attacks. To the best of our knowledge, this results in the first higher-order masked implementation of any post-quantum secure key encapsulation mechanism algorithm. This is realized by introducing two new techniques. First, we propose a higher-order algorithm for the one-bit compression operation. This is based on a masked bit-sliced binary-search that can be applied to prime moduli. Second, we propose a technique which enables one to compare uncompressed masked polynomials with compressed public polynomials. This avoids the costly masking of the ciphertext compression while being able to be instantiated at arbitrary orders.

We show performance results for first-, second- and third-order protected implementations on the Arm Cortex-M0+ and Cortex-M4F. Notably, our implementation of first-order masked Kyber decapsulation requires 3.1 million cycles on the Cortex-M4F. This is a factor 3.5 overhead compared to the unprotected optimized implementation in pqm4. We experimentally show that the first-order implementation of our new modules on the Cortex-M0+ is hardened against attacks using 100 000 traces and mechanically verify the security in a fine-grained leakage model using the verification tool scVerif.

Keywords: Post-Quantum Cryptography · Masking · Kyber

1 Introduction

Public-key cryptography is based on conjectured-to-be-hard mathematical problems. The most widely used examples are RSA, based on the integer factorization problem, and elliptic curve cryptography, based on the discrete logarithm problem. Both are vulnerable to polynomial-time attacks using a quantum computer [Sho94, PZ03].

To defend against this threat, research is directing its attention to post-quantum cryptography (PQC). To streamline this effort the USA National Institute of Standards and Technology (NIST) "has initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms" [Nat] in 2016. In total, 69

Marc Gourjon is partially funded by the VeriSec project 16KIS0601K and the Jupiter project 16ME0231K from the Federal Ministry of Education and Research (BMBF).

Received: 2021-04-15 Accepted: 2021-06-15 Published: 2021-08-11

complete and proper proposals were submitted for the first evaluation round. In October 2020, 15 candidates were announced to have made it through to a third round. It is expected that towards the end of 2021 the winners will be announced which become the NIST PQC standard.

One of the current finalists is Kyber [BDK+18, SAB+20]; this scheme belongs to the lattice-based key encapsulation mechanism (KEM) family. Among the other finalists, Saber [DKRV18, DKR+20] and NTRU [HPS98, CDH+20] also fall in this category. Kyber's hardness is based on the module learning-with-errors problem (M-LWE) in module lattices [LS15]. Unlike prime factorization and the discrete logarithm problem, the M-LWE problem is conjectured to be hard to solve even by an adversary who has access to a full-scale quantum computer.

Initially, the main evaluation criteria focused on the mathematical security and algorithmic design of the proposals. With the advance of the selection of the schemes also other important characteristics become relevant requirements: one of these is the implementation security. One important and well-known attack family is Side-Channel Attacks (SCA). First introduced by Kocher [Koc96], SCAs exploit meta-information when running the implementation to recover secret-key information. This could be obtained using, for example, timing analysis, static and dynamic power analysis, electromagnetic analysis or photoemission analysis.

Not surprisingly, works over the recent years have shown that side-channel attacks also affect post-quantum cryptography [TE15]. Timing attacks were first shown to be applicable to lattice-based cryptography by Silverman and Whyte [SW07]. Since then, it has been demonstrated that an adversary can utilize the non-constant time behavior of Gaussian samplers [BHLY16, EFGT17] as well as a generic cache-attack behavior [BBK+17]. Power analysis attacks on lattices have been shown to be able to attack even masked implementations of lattice-based cryptography by targeting the number theoretic transform [PPM17, PP19, XPRO20], message encoding [RBRC20, ACLZ20], polynomial multiplication [HCY19], error correcting codes [DTVV19], decoders [SRSW20] or CCA-transform [GJN20, RRCB20].

To mitigate the threat of side-channel attacks various types of countermeasures can be applied. This research area has grown over the past few decades for classical cryptography. Techniques to offer side-channel attack resistance for both symmetric and asymmetric primitives are readily available. Applying countermeasures to cryptographic algorithms against side-channel attacks has an impact on the run-time of those algorithms. This impact is even more significant when protecting against *higher-order attacks*: the situation where an attacker attempts to combine multiple points to overcome the protection mechanisms. NIST specifically asked the scientific community to assist in the evaluation of the final round-3 submissions from a side-channel perspective [AASA+20].

Related Work. One of the most well-known countermeasures against side-channel attacks is masking [CJRR99, PR13]. While block ciphers are typically completely protected using Boolean masking, PQC schemes often require a mixture of both arithmetic and Boolean masking in order to be implemented efficiently. Therefore, efficient and secure conversions, e.g., [CGV14, BBE+18], between these two masking types play an essential role in protecting such schemes.

The first masked implementation of a ring-LWE (R-LWE) scheme was presented at CHES'15 [RRVV15]. It uses the linear properties of polynomial arithmetic with arithmetic masking combined with a table-based masked decoder. However, the target scheme analyzed considers only a CPA decryption of R-LWE. In practice, CCA2 security is required which is typically achieved with the Fujisaki-Okamoto transformation [FO99]. Hence, it is necessary to protect most modules of the resulting CCA scheme and not only the R-LWE core. An initial first-order masking scheme of a complete KEM similar to NewHope [BCNS15, ADPS16] was presented at CHES'18 [OSPG18]: building on the concepts of [RRVV15] but presenting a new decoding algorithm without tables and, in addition, proposes to mask all other secret-dependent modules. Similar as for the KEM case, masked signature schemes have been proposed in [BBE+18, MGTF19, GR19].

The modular nature of KEMs makes it easy to focus on one aspect only which then can be re-used in multiple other schemes. To utilize this flexibility, [SPOG19] and [BPO+20] propose higher-order efficient masked implementations of a binomial sampler and a polynomial comparison as used in many schemes. Note, however, that in a recent paper by Bhasin, D'Anvers, Heinz, Pöppelman and Van Beirendonck an attack was presented on a masked implementation of R-LWE implementations [BDH+21]. The authors show that the first-order masked comparison of [OSPG18] and the higher-order version of [BPO+20] are vulnerable to side-channel attacks.

A challenge when protecting against side-channel attacks is the fact that many popular schemes, such as Kyber, use a prime modulus. As observed in both [MGTF19] and [GR19], this results in a significant performance overhead compared to power-of-two moduli, which allow more efficient bit-operations and conversions. Due to the usage of such prime moduli in PQC schemes many prior algorithms needed to be adapted to fit this specific use-case. One of the other NIST finalists Saber [DKR+20] does use a power-of-two moduli for its operations, and it has been shown how to turn this into an efficient first-order protected scheme by Beirendonck, D'Anvers, Karmakar, Balasch and Verbauwhede in [BDK+20]. An attack on this masked Saber implementation was subsequently presented by Ngo, Dubrova, Guo and Johansson [NDGJ21] who apply deep learning power analysis in combination with a lattice reductions step to recover the long-term secret key used. Note that this attack does not invalidate the first-order masking scheme of [BDK+20], but rather efficiently exploits higher-order leakages. Therefore, generic solutions to thwart it are increasing the masking order or the noise level of the implementation.

Contributions. To the best of our knowledge, a complete analysis on how to mask Kyber has not been conducted. Due to its similarity to other schemes, in particular NewHope, many masked modules can be re-used from previous works. The masking of the polynomial arithmetic with arithmetic masking, using a prime modulus q, and the masking of the symmetric components can be straightforwardly reused from prior implementations. However, there are some Kyber-specific functions for which no concrete masking scheme has been proposed yet, or previous solutions are limited or sub-optimal.

In this work, we present the first analysis to realize a complete masked Kyber. Notably, we show techniques how to construct both *first- and higher-order* masking schemes for Kyber with formal proofs in the probing model for the newly-proposed masked components. In addition, we present an implementation of our masked Kyber algorithms on a Cortex-M0+ with hardening and experimental validation of the security order for the first-order secure variants. The security of our first-order implementation is mechanically verified using the verification tool scVerif and the refined Stateful strong Non-Interference security notions [BGG+21], capturing concrete execution and device-specific leakage behavior. We also present a Cortex-M4F implementation using the pqm4 [KRSS19] framework including several assembly-optimized routines and compare performance numbers to the unprotected implementation of pqm4.

To achieve a complete first- and higher-order masking of Kyber, we propose new masked algorithms for the following two modules.

- **Masked One-Bit Compression.** Kyber requires compressing an arithmetically masked polynomial to a Boolean-masked bit-string. Prior solutions are either limited to first-order masking (cf. [OSPG18]) or compression using a modulus which is a power-of-two (cf. [BDK+20]). We propose a new approach based on a bit-sliced binary search, which overcomes both these limitations.

- **Masked Decompressed Comparison.** Kyber uses ciphertext compression. While this can be efficiently masked for power-of-two moduli (cf. [BDK+20]), it introduces a non-negligible overhead for prime moduli. We introduce a new approach which compares *uncompressed* masked polynomials with *compressed* public polynomials. This enables us to avoid having to explicitly mask the ciphertext compression.

Previous works include discussions on how one could extend the techniques to higher order but no further details are provided. To the best of our knowledge, this paper presents the first higher-order masked implementation of any PQC KEM.

Our target platforms are two of the most popular Arm Internet-of-Things processors that offer a 32-bit instruction set. Our first target platform is the most energy-efficient Arm processor available for constrained embedded applications: the Cortex-M0+. A popular Internet-of-Things processor which offers a 32-bit instruction set. Our first-order implementation together with the hardened new modules results in a slowdown of a factor 2.2 compared to an unmasked version of Kyber on this target platform. Furthermore, we present performance figures for second- and third-order masked implementation as well. The first-order unhardened implementation with optimized assembly routines for polynomial arithmetic on the second target platform, the Cortex-M4F, leads to a slowdown of a factor 3.5 compared to the optimized pqm4 version of Kyber.

We conduct experimental verification of the protection order by measuring the power consumption of the two proposed modules and assessing their leakage using Test Vector Leakage Assessment (TVLA) [GGJR+11] using 100 000 measurements. The first-order hardened implementation of our two new modules does not show any detectable leakage.

2 Background

In this section, we first introduce the PQC KEM Kyber. In the description we focus on the functions that process secret-key-dependent material and therefore will need to be masked. For the full description of Kyber, we refer to [SAB+20].

After a high-level description of in-scope Kyber concepts, we detail our SCA notation and concepts. This serves as a basis for the SCA security analysis of Kyber in Section 3.

Notation. We denote the ring of integers modulo q with $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. Centered modular reduction is denoted by $r' = r \bmod^{\pm} q$, where $-\frac{q-1}{2} < r' \leq \frac{q-1}{2}$, and other reductions by $r' = r \bmod q$ where $0 \leq r' < q$. The rings $\mathbb{Z}[X]/(X^n+1)$ and $\mathbb{Z}_q[X]/(X^n+1)$ are respectively denoted by and R and R_q. Further, note that rounding to the closest integer is denoted by $\lfloor \cdot \rceil$ (with ties rounded up) and rounding up is denoted by $\lceil \cdot \rceil$.

We denote vectors and matrices by boldfaced variables $\mathbf{b}$ and $\mathbf{A}$. As an ingredient of Kyber we need to define the centered binomial distribution CBD_η, for a positive integer η. Sampling from this distribution is achieved by sampling the 2η elements of $\{(a_i, b_i)\}_{i=0}^{\eta-1}$ uniformly from $\{0,1\}$ and outputting $\sum_{i=0}^{\eta-1}(a_i - b_i)$.

Compression, Decompression and Sampling. As a first building block of Kyber, we define a function $\mathsf{Compress}_q(x,d)$ that takes an element $x \in \mathbb{Z}_q$ and outputs an integer in $\{0,\ldots,2^d-1\}$, where $d < \lceil \log_2(q) \rceil$. We furthermore define a function $\mathsf{Decompress}_q$, such that $x' = \mathsf{Decompress}_q(\mathsf{Compress}_q(x,d),d)$ is an element close to x, more specifically $|x' - x \bmod^{\pm} q| \leq B_q := \lceil \frac{q}{2^{d+1}} \rfloor$. The functions satisfying these requirements are defined as

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

When $\mathsf{Compress}_q$ or $\mathsf{Decompress}_q$ is used with $x \in R_q$ or $\mathbf{x} \in R_q^k$, the procedure is applied to each coefficient individually.

As a second ingredient there is the sampling function CBD which converts uniformly random bytes into polynomials whose coefficients are distributed as CBD_η. This algorithm is summarized in Algorithm 4 in the Appendix.

Kyber PKE. Kyber is a module-LWE scheme [BGV12, LS15]. Given a parameter k its hardness relies on the hardness of distinguishing samples $(\mathbf{a}_i, b_i) \in R_q^k \times R_q$, where all elements are uniformly drawn from R_q, from those where the elements of $\mathbf{a}_i$ are drawn from a uniform distribution and $b_i = \mathbf{a}_i^T \mathbf{s} + e_i$, for a secret $\mathbf{s} \in \beta_\eta^k$, and $e_i \in \beta_\eta$ is refreshed for each sample.

The IND-CPA-secure Kyber public-key encryption (PKE) scheme consists of three algorithms; key generation, encryption (Algorithm 5) and decryption (Algorithm 6). Kyber.CPAPKE is parameterized by $n, k, q, \eta_1, \eta_2, d_u$ and d_v. The recommended parameter sets are listed in Table 3 where δ is the failure probability of the decryption. Since the key generation only processes the secret key once, and masking is commonly aimed at mitigating multi-trace attacks, we omit its description here.

Kyber KEM. An IND-CCA2-secure KEM Kyber.CCAKEM can be constructed from the Kyber.CPAPKE scheme by applying a version of the Fujisaki–Okamoto transform [FO99, HHK17]. The resulting scheme consists of key generation, encapsulation and decapsulation schemes. The high-level decapsulation Kyber.CCAKEM.Dec is of main interest in this work since this is the only part affected by our masking techniques: its description is given in Algorithm 7. Again, we refer to [SAB+20] for details on G, H and KDF.

Side-Channel Notation and Notions. The core concept of masking is to split the sensitive variables into multiple shares and transform the underlying circuit to process these shared variables securely. To formally argue about the security provided by these shared implementations, Ishai, Sahai and Wagner introduced the t-probing model in [ISW03], which models an adversary that can probe up to t intermediate variables. If every possible t-tuple of a given masked circuit is independent of the secret, it is considered to be secure against t-order SCA attacks.

In the following, a sensitive variable x is split into n_s secret shares and the resulting n_s-tuple is denoted as $x^{(\cdot)}$. Where applicable, we denote an arithmetic encoding of a variable $x \in \mathbb{Z}_q$ as $x^{(\cdot)_A}$ consisting of n_s arithmetic shares $x^{(i)_A} \in \mathbb{Z}_q, 0 \leq i < n_s$ such that $x^{(0)_A} + \ldots + x^{(n_s-1)_A} \equiv x \bmod q$. Where applicable, we denote a Boolean encoding of a variable $x \in \mathbb{Z}_2^k$ as $x^{(\cdot)_B}$ consisting of n_s Boolean shares $x^{(i)_B} \in \mathbb{Z}_2^k, 0 \leq i < n_s$ such that $x^{(0)_B} \oplus \ldots \oplus x^{(n_s-1)_B} = x$.

Given a polynomial $f \in R_q$, the i-th coefficient of f is denoted as f_i. Given a bitstring $b \in \mathbb{Z}_2^k$, the i-th bit of b is denoted as b_i.

While proving probing security alone is sufficient for single functions (*gadgets* in the following), it does not easily allow arguing about compositions of multiple gadgets at higher orders (i.e., $t > 1$). Therefore, it is common to rely on the concepts of t-(Strong)-Non-Interference (t-(S)NI) as introduced in [BBD+16] to argue about the security of such constructions. We recall the t-NI and t-SNI security notions as presented in [BCZ18]. We consider a gadget taking as input a (or multiple) n_s-tuple $x^{(\cdot)}$ of shares, and outputting a (or multiple) n_s-tuple $y^{(\cdot)}$. Given a subset $I \subset [0, n_s - 1]$, we denote by $x^{(I)}$ all elements $x^{(i)}$ such that $i \in I$.

Definition 1 (t-NI security (from [BBD+15, BBD+16])). Let G be a gadget taking as input $x^{(\cdot)}$ and outputting $y^{(\cdot)}$. The gadget G is t-NI secure if for any set of $t_G \leq t$ intermediate variables, there exists a subset $I \subset [0, n_s - 1]$ of input indices with $|I| \leq t_G$, such that the t_G intermediate variables can be perfectly simulated from $x^{(I)}$.

Definition 2 (t-SNI security (from [BBD+16])). Let G be a gadget taking as input $x^{(\cdot)}$ and outputting $y^{(\cdot)}$. The gadget G is t-SNI secure if for any set of $t_G \leq t$ intermediate variables and any subset $O \subset [0, n_s - 1]$ of output indices, such that $t_G + |O| \leq t$, there exists a subset $I \subset [0, n_s - 1]$ of input indices with $|I| \leq t_G$, such that the t_G intermediate variables and the output variables $y^{(O)}$ can be perfectly simulated from $x^{(I)}$.

In Section 3, we prove our new algorithms to be t-SNI with $n_s = t + 1$ to provide resistance against t-order attacks and allow formal arguing about their composition.

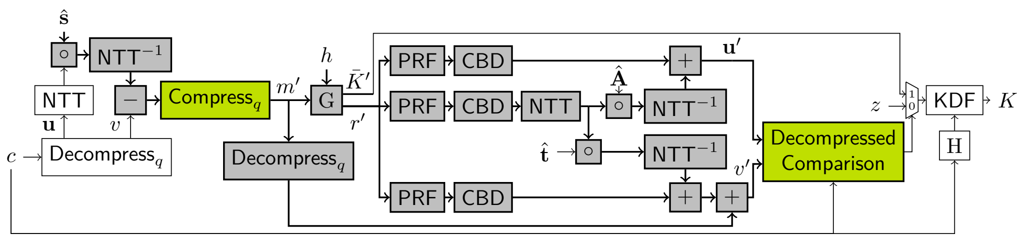


Figure 1: An overview of the various components in the Kyber CCA decapsulation. The components which need to be protected / masked are in color (gray or green) where we present new approaches for the green components.

3 Masking Kyber at Arbitrary Order

The post-quantum secure key encapsulation mechanism Kyber has a structure similar to other submissions to the NIST PQC standardization effort such as NewHope and Saber. In particular, it first applies a CPA decryption to the ciphertext in order to create a message m. This message is then re-encrypted with the CPA encryption and the resulting ciphertext is compared with the original input. Depending on the Boolean result of this comparison, a session-key K is derived either from the message if the ciphertext and the original input are the same or from a secret fixed value z otherwise.

A graphical overview of the various modules in Kyber decapsulation is given in Figure 1; the colored components are those that need to be masked. The decapsulation is deterministic and therefore all modules which process sensitive data that is derived from the long-term secret s, need to be protected against SCAs. In this section we first focus on the two green Compress$_q$ and DecompressedComparison modules. We present two new approaches for these modules: masked one-bit compression (Section 3.1) and masked comparison (Section 3.2). For each, we first provide the basic intuition about their functionality and then prove their t-SNI security in the probing model. We then put the components together to achieve a fully masked Kyber in Section 3.3.

3.1 Higher-Order One-Bit Compression

For Saber, where the used modulus is a power-of-two, the compression operation represents a shift of the sensitive value: this can be efficiently masked using look-up tables as demonstrated in [BDK+20]. For schemes that use a prime modulus, masking this step is more involved. The authors of [OSPG18] propose a first-order masked solution based on two mask conversions: one arithmetic-to-arithmetic (A2A) and one arithmetic-to-Boolean (A2B) conversion per polynomial coefficient. To improve efficiency and allow extensions to higher orders, we propose a new approach which works for *any modulus* and at *any order* that requires only *one conversion per coefficient*.

Informally, the compression to one bit in Kyber splits the domain of each polynomial coefficient into two disjunctive intervals and assigns a bit value depending on which interval the value of the coefficient is contained. In Kyber, this is done with the function Compress$_q(x,d) = \lceil (2^d/q) \cdot x \rfloor \bmod 2^d$. The compression to one bit results in the following mapping

$$\mathsf{Compress}_q(x,1) = \lceil (2/q) \cdot x \rfloor \bmod 2 = \begin{cases} 1 & \text{if } \frac{q}{4} < x < \frac{3q}{4}, \\ 0 & \text{otherwise.} \end{cases}$$

This computation is trivial without masks, but poses a challenge when q is prime and masking is required. When the modulus is a power-of-two, less-than-comparisons can be

Algorithm 1 Masked version of $\mathsf{Compress}_q(x,1) = \mathsf{Compress}^{\mathsf{s}}_q(x + \lfloor \frac{q}{4} \rfloor \bmod q)$ as used in Kyber for any order using one A2B conversion per coefficient.

```
Input: An arithmetic sharing $a^{(\cdot)_A}$ of a polynomial $a \in \mathbb{Z}_q[X]$.
Output: A Boolean sharing $m'^{(\cdot)_B}$ of the message $m' = \mathsf{Compress}_q(a,1) \in \mathbb{Z}_{2^{256}}$.
 1: for $i = 0$ to 255 do
 2:     $a_i^{(0)_A} = a_i^{(0)_A} + \lfloor \frac{q}{4} \rfloor \bmod q$
 3:     $a_i^{(\cdot)_B} = \mathsf{A2B}(a_i^{(\cdot)_A})$
 4: $x^{(\cdot)_B} = \mathsf{Bitslice}(a^{(\cdot)_B})$
 5: $m'^{(\cdot)_B} = \mathsf{SecAND}(\mathsf{SecREF}(\neg x_8^{(\cdot)_B}), x_7^{(\cdot)_B})$
 6: $m'^{(\cdot)_B} = \mathsf{SecREF}(\mathsf{SecXOR}(m'^{(\cdot)_B}, x_8^{(\cdot)_B}))$
 7: $m'^{(\cdot)_B} = \mathsf{SecAND}(m'^{(\cdot)_B}, x_9^{(\cdot)_B})$
 8: $m'^{(\cdot)_B} = \mathsf{SecAND}(m'^{(\cdot)_B}, x_{10}^{(\cdot)_B})$
 9: $m'^{(\cdot)_B} = \mathsf{SecAND}(m'^{(\cdot)_B}, \neg x_{11}^{(\cdot)_B})$
10: $m'^{(\cdot)_B} = \mathsf{SecXOR}(m'^{(\cdot)_B}, x_{11}^{(\cdot)_B})$
11: return $m'^{(\cdot)_B}$
```

computed using a B2A conversion [OSPG18]. However, for prime moduli the value space is not equally divided by specific bits. In this case masking $\mathsf{Compress}_q$ requires either the use of pre-computed tables or a dedicated masked-compression algorithm.

Let us first recall the first-order based approach from [OSPG18]. Given a masked coefficient $a^{(\cdot)_A}$, they first apply an A2A conversion to produce a masked coefficient $b^{(\cdot)_A}$ with a power-of-two modulus such that

$$\sum_{i=0}^{n_s-1} a^{(i)_A} \bmod q = \sum_{i=0}^{n_s-1} b^{(i)_A} \bmod 2^k = a_i,$$

where $2^k > q$. Next, an appropriate offset is subtracted from $b^{(\cdot)_A}$ such that the MSB of the sensitive variable denotes the value to which the coefficient should be compressed. This shared bit can be extracted from the Boolean shares after applying an A2B conversion. Hence, this technique requires one A2A and one A2B conversion per coefficient. Given that these conversions are usually quite expensive this introduces a significant overhead. Furthermore, to the best of our knowledge, there are no known results for higher-order A2A conversion for arbitrary moduli. The only other published solution in this direction is presented in [BDK+20] and applies only to power-of-two moduli.

We present a solution which can be applied in first- and higher-order settings, can be applied to the setting where a prime modulus is used and is faster by omitting one A2A conversion per coefficient compared to the state-of-the-art. In the remainder we introduce the method with focus on the Kyber application, however, it should be noted that this approach works for any modulus q. We start with adding the offset $\lfloor \frac{q}{4} \rfloor = 832$ modulo q from the arithmetic shares with a subsequent A2B conversion to create k-bit Boolean shares of the coefficient, where $k = \lceil \log_2(q) \rceil = 12$. Given these Boolean shares, it then suffices to securely compute whether the masked value is smaller than $\frac{q}{2}$. Let us denote this shifted function as $\mathsf{Compress}^{\mathsf{s}}_q(x)$ such that $\mathsf{Compress}_q(x,1) = \mathsf{Compress}^{\mathsf{s}}_q(x + \lfloor \frac{q}{4} \rfloor \bmod q)$ where

$$\mathsf{Compress}^{\mathsf{s}}_q(x) := \begin{cases} 0 & \text{if } x < \frac{q}{2}, \\ 1 & \text{otherwise.} \end{cases}$$

To compute $\mathsf{Compress}^{\mathsf{s}}_q$ in a masked fashion, we perform a masked-binary-search on the Boolean-shared bits of the coefficient starting from the MSB. For example, if the MSB is set to 1, we can ignore the values of all subsequent bits and compress the coefficient to 1 as $2^{\mathsf{MSB}} = 2^{k-1} = 2^{11} > \frac{q}{2}$. If the MSB is set to 0, the remaining bits need to be taken into account. This process is repeated until all possible coefficient values have been mapped to a single bit value. For the case of Kyber, $\left\lfloor \frac{q}{2} \right\rfloor = 1664$, bits 11 to 7 are taken into account. In this case $\mathsf{Compress}_q^{\mathsf{s}}$ is computed as

$$\mathsf{Compress}_q^{\mathsf{s}}(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7)))\,. \tag{1}$$

In a masked implementation, the $\oplus$ and $\cdot$ operations should be replaced with calls to their secure counterparts (SecXOR and SecAND). Moreover, to improve efficiency, we can first transform the Boolean shares of the polynomial to a bitsliced representation and compute the compress function for all coefficients in parallel (limited by the word size of the target platform). This complete masked algorithm for $\mathsf{Compress}_q(x, 1)$ is given in Algorithm 1. Note that the algorithm is independent of the specific masked algorithms used for the modules $\mathsf{A2B}$, $\mathsf{Bitslice}$, SecAND, SecREF, and SecXOR. Instead, we provide a short description of the computed functionality and the assumed security property for the proof. $\mathsf{A2B}$ denotes a t-SNI secure conversion of arithmetic shares with a prime modulus to Boolean shares encoding the same value. In our higher-order implementations, we use Algorithm 3 from [SPOG19]. $\mathsf{Bitslice}$ maps a Boolean-masked polynomial to its Boolean-masked bitsliced representation. This is a linear function and can, therefore, be computed on each share separately. The most efficient way to accomplish this strongly depends on the capabilities of the target platform. In our implementation, we realized it as a sequence of bitshift, bitwise OR, and bitwise AND to rearrange the bits share by share. With SecAND and SecREF, we describe t-SNI algorithms to compute the masked bitwise AND and refresh Boolean shares. There are multiple proposals that fulfill this property, e.g., [GJRS18, BCPZ16], and we use Algorithm 18 (resp. Algorithm 20) from [SPOG19] for SecAND (resp. SecREF) in our implementations. SecXOR refers to the t-NI computation of the bitwise XOR of Boolean shares. This is usually achieved by computing the XOR of the input shares separately, e.g., [CGV14, Algorithm 3, Line 3] Furthermore, we use $\neg$ to indicate the negation of only the first share of the Boolean-masked input.

Correctness. For Kyber, we use $q = 3329$ with the parameters $k = 12$, $\left\lfloor \frac{q}{4} \right\rfloor = 832$ and $\left\lfloor \frac{q}{2} \right\rfloor = 1664$. Let us provide the detailed steps to derive the equation to compute the compression operation $\mathsf{Compress}_q^{\mathsf{s}}(x)$ using only XOR, AND, and negation.

1. $2^{11} > 1664$: If $x_{11} = 1$, then x should be compressed to 1. Otherwise, we need to take less significant bits into consideration, therefore: $\mathsf{Compress}_q^{\mathsf{s}}(x) = x_{11} \oplus (\neg x_{11} \cdot (\dots))$.

2. $2^{10} < 1664$: If $x_{10} = 0 \wedge x_{11} = 0$, then x should be compressed to 0. Otherwise, we need to take less significant bits into consideration: $\mathsf{Compress}_q^{\mathsf{s}}(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot (\dots))$.

3. $2^{10} + 2^9 < 1664$: If $x_9 = 0 \wedge x_{10} = 1 \wedge x_{11} = 0$, then x should be compressed to 0. Otherwise, we need to take less significant bits into consideration: $\mathsf{Compress}_q^{\mathsf{s}}(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (\dots))$.

4. $2^{10} + 2^9 + 2^8 > 1664$: If $x_8 = 1 \wedge x_9 = 1 \wedge x_{10} = 1 \wedge x_{11} = 0$, then x should be compressed to 1. Otherwise, we need to take less significant bits into consideration: $\mathsf{Compress}_q^{\mathsf{s}}(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot (\dots))))$.

5. $2^{10} + 2^9 + 2^7 = 1664$: If $x_7 = 1 \wedge x_9 = 1 \wedge x_{10} = 1 \wedge x_{11} = 0$, then x should be compressed to 1. All remaining combinations should be compressed to 0 since they are necessarily < 1664, therefore: $\mathsf{Compress}_q^{\mathsf{s}}(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7)))$.

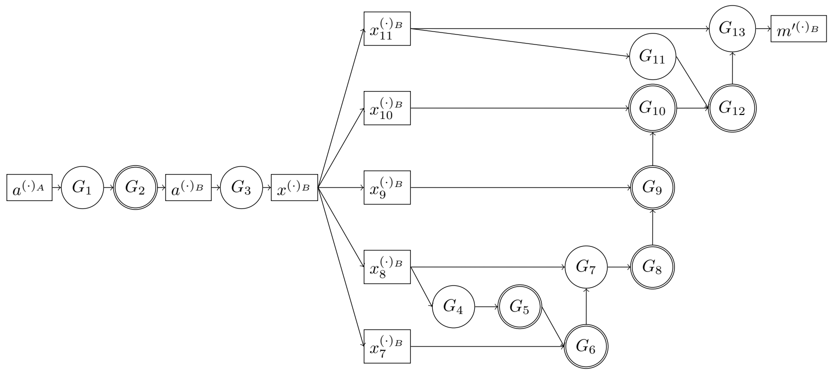


Figure 2: The gadgets considered in the proof of Theorem 1. t-NI gadgets are depicted with a single circle, t-SNI gadgets are depicted with a double circle.

Complexity. We estimate the run-time complexity $\mathcal{T}_{A_1}(n_s, k)$, where n_s goes to infinity, of Algorithm 1. For ease of notation we write $\mathcal{T}_f(n_s, k)$ as $\mathcal{T}_f$.

$$\mathcal{T}_{A_1} = 256 \cdot (\mathcal{O}(1) + \mathcal{T}_{\mathsf{A2B}}) + \mathcal{T}_{\mathsf{Bitslice}} + \frac{256}{w} \cdot 2 \cdot (2 \cdot \mathcal{T}_{\mathsf{SecAND}} + \mathcal{T}_{\mathsf{SecREF}} + \mathcal{T}_{\neg} + \mathcal{T}_{\mathsf{SecXOR}})$$

with $k = \lceil \log_2(q) \rceil = 12$ and w denoting the word size of the target platform, i.e., how many bits can be processed in parallel. Given $\mathcal{T}_{\mathsf{A2B}} = \mathcal{O}\left(n_s^2 \cdot \log_2(k)\right)$ [SPOG19], $\mathcal{T}_{\mathsf{Bitslice}} = \mathcal{O}\left(256 \cdot n_s \cdot k\right)$ (transforming each of the 256 k-bit coefficients sharewise), $\mathcal{T}_{\mathsf{SecAND}} = \mathcal{T}_{\mathsf{SecREF}} = \mathcal{O}\left(n_s^2\right)$ [SPOG19], $\mathcal{T}_{\neg} = \mathcal{O}\left(1\right)$ (negating one of the input shares), and $\mathcal{T}_{\mathsf{SecXOR}} = \mathcal{O}\left(n_s\right)$ (computing the sharewise XOR) as the complexities for the modules, we derive the asymptotic run-time complexity for Algorithm 1 as $\mathcal{T}_{A_1} = \mathcal{O}\left(n_s^2 \cdot \log_2(k)\right)$ for a constant p. Analogously, we can derive the randomness complexity $\mathcal{R}_{A_1} = \mathcal{O}\left(n_s^2 \cdot \log_2(k)\right)$ by replacing the run-time complexity of the modules with the corresponding randomness complexities.

Security. To argue about the higher-order security of Algorithm 1, we prove it to be t-SNI with $n_s = t + 1$ shares. This provides resistance against a probing adversary with t probes and allows using the algorithm in larger compositions. The proof requires us to show how probes on intermediate (and output) variables in the algorithm can be perfectly simulated with only a limited number of the input shares. To this end, we iterate over all possible intermediate variables, starting from the output, and provide formal arguments on how they can be simulated relying on the t-(S)NI properties of the modules. In this step, it is important to ensure that the simulation of t_x probes on one intermediate variable does not require more than t_x shares of another intermediate variable. Otherwise, the simulation is not sound as it would require more than t shares of one intermediate variable for $t_x = t$. For t-SNI, it is important to further show that the simulation of the intermediate and output probes can be performed with only a subset of the input shares with cardinality equal to the number of intermediate probes.

Theorem 1 (t-SNI of Algorithm 1)**.** *Let $a^{(\cdot)_A}$ be the input and let $m'^{(\cdot)_B}$ be the output of Algorithm 1. For any set of t_{A_1} intermediate variables and any subset $O \subset [0, n_s - 1]$ with $t_{A_1} + |O| < n_s$, there exists a subset I of input indices such that the t_{A_1} intermediate variables as well as $m'^{(O)_B}$ can be perfectly simulated from $a^{(I)_A}$, with $|I| \leq t_{A_1}$.*

Proof. We model Algorithm 1 as a sequence of t-(S)NI gadgets as depicted in Figure 2. For simplicity, we model the linear operations in Lines 2, 4, 5, 9 as t-NI gadgets, which can be trivially shown as the operations process the inputs share-wise. Furthermore, as the iterations of the initial loop are independent, we consider them to be executed in parallel and summarize them into single gadgets, one for Line 2 and one for Line 3. The exact mapping of gadgets in Figure 2 to Algorithm 1 is as follows:

- G_1 (NI): Subtraction in Line 2.
- G_2 (SNI): A2B in Line 3.
- G_3 (NI): Bitslice in Line 4.
- G_4 (NI): $\neg$ in Line 5.
- G_5 (SNI): SecREF in Line 5.
- G_6 (SNI): SecAND in Line 5.
- G_7 (NI): SecXOR in Line 6.
- G_8 (SNI): SecREF in Line 6.
- G_9 (SNI): SecAND in Line 7.
- G_{10} (SNI): SecAND in Line 8.
- G_{11} (NI): $\neg$ in Line 9.
- G_{12} (SNI): SecAND in Line 9.
- G_{13} (NI): SecXOR in Line 10.

An adversary can place probes internally and on the output shares for each gadget. The number of internal (resp. output) probes for gadget G_i is denoted as t_{G_i} (resp. o_{G_i}) with

$$t_{A_1} = \sum_{i=1}^{13} t_{G_i} + \sum_{i=1}^{12} o_{G_i}, \quad |O| = o_{G_{13}}$$

where t_{A_1} and $|O|$ refer to the number of probes and output shares of the complete Algorithm 1 as used in Theorem 1. To prove Theorem 1, we show that the internal probes and output shares can be perfectly simulated with $\leq t_{A_1}$ of the input shares $a^{(\cdot)_A}$. To this end, we argue about the internal probes and output shares of each gadget relying on their t-(S)NI property. In particular, we rely on the characteristic that the simulation of a t-SNI gadget can be performed independent of the number of probed output shares. This allows stopping the propagation of probes from the output shares to the input shares. For example, to simulate the $t_{G_{13}}$ intermediate and $o_{G_{13}}$ output probes of the t-NI gadget G_{13}, we require $t_{G_{13}} + o_{G_{13}}$ shares of both inputs of G_{13} (i.e., $x_{11}^{(\cdot)_B}$ and the output of G_{12}). Throughout a larger composition, the shares required for simulation are added up. To avoid an unsound simulation, it is often required to use t-SNI gadgets to stop the propagation of probes on the output shares, e.g., the $t_{G_{12}}$ intermediate and $o_{G_{12}}$ output probes of the t-SNI gadget G_{12} can be simulated with only $t_{G_{12}}$ input shares (i.e., without $o_{G_{12}}$). As shown later, we need to insert t-SNI SecREF gadgets to ensure that our simulation is sound.

In the following, we provide details for the simulation at particular points in the algorithm. The complete explanation for each gadget is provided in Appendix A. To simulate the internal probes and output shares of gadgets G_4 to G_{13}, we need the following number of shares of $x_7^{(\cdot)_B}$ to $x_{11}^{(\cdot)_B}$:

$$\begin{aligned}
t_{x_7^{(\cdot)_B}} &= t_{G_6}, & t_{x_8^{(\cdot)_B}} &= t_{G_4} + o_{G_4} + t_{G_5} + t_{G_7} + o_{G_7} + t_{G_8}, \\
t_{x_9^{(\cdot)_B}} &= t_{G_9}, & t_{x_{10}^{(\cdot)_B}} &= t_{G_{10}}, \\
t_{x_{11}^{(\cdot)_B}} &= t_{G_{11}} + o_{G_{11}} + t_{G_{12}} + t_{G_{13}} + o_{G_{13}}.
\end{aligned}$$

To argue about Bitslice, we summarize the variables of each bit to $x^{(\cdot)_B}$. For the simulation, we add up the number of shares for each bit as $t_{x^{(\cdot)_B}} = \sum_{i=7}^{11} t_{x_i^{(\cdot)_B}}$. This simulation can only be performed if there are no duplicate entries in the sum: without the t-SNI refresh G_5, the simulation would require t_{G_6} shares of both $x_7^{(\cdot)_B}$ and $x_8^{(\cdot)_B}$. In effect, $t_{x^{(\cdot)_B}}$ would be $\geq 2 \cdot t_{G_6}$, which cannot be simulated for, e.g., $t_{G_6} = t$. Therefore, it is necessary to refresh[1] the input to G_6, and analogously to G_9. For the other SecAND gadgets, this issue does not occur and, therefore, we do not need to refresh their inputs.

Given the t-NI property of Bitslice, we can simulate the $t_{x^{(\cdot)_B}}$ shares of $x^{(\cdot)_B}$ with the corresponding number of shares of $a^{(\cdot)_B}$. Following the flow through gadgets G_2 and G_1,

[1]Refreshing may be avoided by gadgets which conform to stricter security notions, e.g., PINI [CS20].

the simulation of Algorithm 1 requires $|I| = t_{G_1} + o_{G_1} + t_{G_2}$ of the input shares $a^{(\cdot)_A}$. In particular, the t-SNI property of G_2 allows to simulate the shares of $a^{(\cdot)_B}$ with only t_{G_2} of its input, i.e., it is independent of the number of the probes on $a^{(\cdot)_B}$, which stops the propagation of $t_{x^{(\cdot)_B}}$ to $|I|$. As $|I| \leq t_{A_1}$ and independent of o_{13}, Algorithm 1 is t-SNI. $\square$

Extension $\mathsf{Compress}_q(x, d)$ for $d > 1$. We use Algorithm 1 for compression to $d = 1$ bits, but it can be adapted to create masked compression functions for $d > 1$ bits as well. To this end, it is necessary to derive the Boolean equations for each of the d output bits, analogous to Equation (1). These are then computed using instantiations of SecXOR and SecAND with independent shared inputs. A generic description of Algorithm 1 for any d would, therefore, need to refresh the input to any SecAND, which would induce a significant overhead. In this section an optimized version for $d = 1$, as used in Kyber, is provided. The creation of optimized versions for other d is straight-forward when using formal verification tools to check which of the refreshes are needed, e.g., scVerif [BGG+21] or MaskVerif [BBD+15, BBC+19]. In the following section, we develop a dedicated technique to avoid masking the ciphertext compression of Kyber.CPAPKE.Enc, i.e., extending Algorithm 1 to $d > 1$, as this would require to process all input bits.

3.2 Higher-Order Masked Comparison

The masked ciphertext comparison requires computing $c \overset{?}{=} c'$ in a masked fashion, which assumes prior ciphertext compression in Kyber.CPAPKE.Enc. More explicitly, the comparison verifies whether

$$(\mathsf{Compress}_q(\mathbf{u}', d_u), \mathsf{Compress}_q(v', d_v)) \overset{?}{=} c\,. \tag{2}$$

For the ciphertext compression, to the best of our knowledge, there is no efficient higher-order solution beyond generic approaches, e.g., masked look-up tables, when using prime moduli. In [OSPG18] a hash-based first-order comparison approach is proposed. However, this only checks for equality and is independent of the ciphertext compression. To apply this technique to Kyber, it would be necessary to perform a masked ciphertext compression as a prior step. In [BPO+20], a higher-order polynomial comparison is proposed which also checks for equality but suffers from similar drawbacks as the techniques from [OSPG18]. Note that this approach was also shown to be flawed in [BDH+21] and the proposed fix significantly reduces the performance.[2] Given that none of the prior-art solutions work without a masked ciphertext compression, we propose a new masked algorithm to perform the comparison between a *masked uncompressed ciphertext* (i.e., output of our masked re-encryption) and a *public compressed ciphertext*.

The core idea is to not perform the costly masked compression of the sensitive values but work the other way around: decompressing the public ciphertext. Since this is public information this can be done efficiently. Informally, this changes Eq. (2) to

$$(\mathbf{u}', v') \overset{?}{=} \mathsf{Decompress}_q(c). \tag{3}$$

Since the compression is lossy, one cannot simply check for equality. Instead, one has to perform a masked range check for each coefficient to verify that the uncompressed sensitive values fall into the decompressed interval. In particular, one has to first derive the interval start- and end-point for a compressed coefficient using public functions S and E. These border values are then subtracted from the compressed masked coefficients separately; which is efficient given that they are arithmetically masked. Then each of these values are transformed to Boolean masking to extract the MSB which contains the result of the

[2]Note that [BDH+21] also shows a flaw in the implementation of the masked comparison of [OSPG18], but this one can be trivially fixed without impacting the performance.

Algorithm 2 Masked DecompressedComparison as used in Kyber.

```
Input:  1. An arithmetic sharing $\mathbf{u}'^{(\cdot)_A}$ of a vector of polynomials $\mathbf{u}' \in \mathbb{Z}_q[X]^k$,
        2. An arithmetic sharing $v'^{(\cdot)_A}$ of a polynomial $v' \in \mathbb{Z}_q[X]$,
        3. A ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$,
        4. Two public functions $S$ and $E$ defined by Kyber which specify the
           start- and end-points of the intervals in compression.
Output: A Boolean sharing $b^{(\cdot)_B}$ of $b$ where $b = 1$ if and only if
        $(\mathsf{Compress}_q(\mathbf{u}', d_u), \mathsf{Compress}_q(v', d_v)) = c$, otherwise $b = 0$.
1:  function DecompressedComparison
2:      $(\mathbf{u}'', v'') = \mathsf{Decode}\,(c)$
3:      $t_{\mathbf{w}}^{(\cdot)_B}, t_{\mathbf{x}}^{(\cdot)_B} = \mathtt{PolyCompare}(\mathbf{u}'^{(\cdot)_A}, \mathbf{u}'')$
4:      $t_y^{(\cdot)_B}, t_z^{(\cdot)_B} = \mathtt{PolyCompare}(v'^{(\cdot)_A}, v'')$
5:      $b^{(\cdot)_B} = \mathsf{SecAND}(\mathsf{SecAND}(t_{\mathbf{w}}^{(\cdot)_B}, t_{\mathbf{x}}^{(\cdot)_B}), \mathsf{SecAND}(t_y^{(\cdot)_B}, t_z^{(\cdot)_B}))$
6:      for $i = \log_2 256 - 1$ to $0$ do
7:          $t_b^{(\cdot)_B} = \mathsf{LSR}(b^{(\cdot)_B},\ 2^i)$
8:          $b^{(\cdot)_B} = b^{(\cdot)_B} \bmod (2^{2^i} - 1)$
9:          $b^{(\cdot)_B} = \mathsf{SecAND}(b^{(\cdot)_B}, t_b^{(\cdot)_B})$
10:     return $b^{(\cdot)_B}$

11: function PolyCompare($u'^{(\cdot)_A}, u''$)
12:     for $i = 0$ to $255$ do
13:         $s_{u''} = S(u''_i)$
14:         $e_{u''} = E(u''_i)$
15:         $w_i^{(\cdot)_A} = x_i^{(\cdot)_A} = u_i'^{(\cdot)_A}$
16:         $w_i^{(0)_A} = (w_i^{(0)_A} + 2^{\lceil \log_2(q) \rceil - 1} - s_{u''}) \bmod q$
17:         $x_i^{(0)_A} = (x_i^{(0)_A} - e_{u''}) \bmod q$
18:         $w_i^{(\cdot)_B} = \mathsf{MSB}(\mathsf{A2B}(w_i^{(\cdot)_A}))$
19:         $x_i^{(\cdot)_B} = \mathsf{MSB}(\mathsf{A2B}(x_i^{(\cdot)_A}))$
20:     return $\mathsf{Bitslice}(w^{(\cdot)_B})$, $\mathsf{Bitslice}(x^{(\cdot)_B})$
```

coefficient interval check: the MSB can be viewed as something similar to the "sign" bit, see for a more detailed explanation the correctness paragraph below.

If the compressed coefficient is indeed inside the desired interval, the MSB of both range checks should be one. For the comparison, we need to combine the interval checks of all coefficients into one masked output bit. This is achieved using bitsliced calls to SecAND until one bit remains, which is only set to one if and only if Eq. (3) is fulfilled. The complete masked algorithm for the DecompressedComparison is given in Algorithm 2. Again we provide a short description of the new modules and their assumed security property. MSB extracts the Boolean-masked most significant bit of the given input Boolean shares, which is assumed to be t-NI as it can be applied on each share separately. LSR refers to the sharewise logical shift to the right of input Boolean shares by a given offset, which is also assumed to be t-NI.

Correctness. To better understand Algorithm 2, let us first go through the unmasked decompressed comparison using one coefficient as an example. We move the costly compression step to the public variable as $a \stackrel{?}{=} \mathsf{Decompress}_q(b)$, i.e., we check if the public b would be decompressed to an interval which contains a. As the compression is lossy, there are multiple values for a which can be mapped to b through $\mathsf{Compress}_q$. Therefore, a straight-forward check for this equality would only work for one specific value of a. Instead,

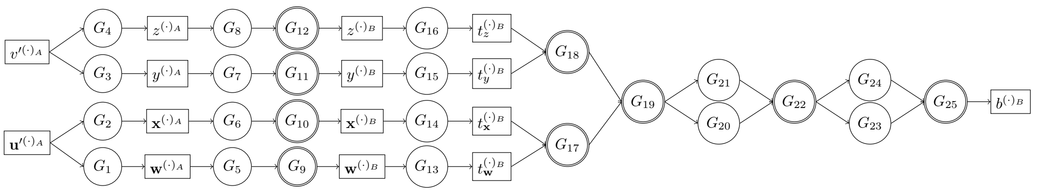


Figure 3: The gadgets considered in the proof of Theorem 2. t-NI gadgets are depicted with a single circle, t-SNI gadgets are depicted with a double circle.

we denote the lower bound $\mathsf{S}(b)$ and the upper bound $\mathsf{E}(b)$ such that for $\mathsf{S}(b) \leq a \leq \mathsf{E}(b) - 1$ one has $\mathsf{Compress}_q(a) = b$. Given these pre-computed public values $\mathsf{S}(b)$ and $\mathsf{E}(b)$, we need to decide if a given a is indeed in the interval $[\mathsf{S}(b), \mathsf{E}(b) - 1]$. While this is trivial for unmasked values, a is sensitive and, therefore, this operation needs to be masked.

Performing a generic less-than comparison check is straight-forward for power-of-two moduli, but challenging for prime moduli such as used in Kyber. The idea to achieve this is to compute $a - \mathsf{S}(b) \bmod q$ and $a - \mathsf{E}(b)$ and checking the "sign" bits. This is done in a masked fashion by performing first an A2B and subsequently extracting the MSB of the masks. If a is indeed in the interval $[\mathsf{S}(b), \mathsf{E}(b) - 1]$ then one expects $a - \mathsf{S}(b)$ to return an MSB with a 0 while $a - \mathsf{E}(b)$ should be 1. These can be combined with a SecAND by first negating the masked bit of the first range check.

In order to avoid this negation, one can shift the values in the first check appropriately (by adding $2^{\lceil \log_2(q) \rceil - 1}$) such that it becomes $a - \mathsf{S}(b) + 2^{11}$ in the setting of Kyber. Now both the resulting MSB need to be put into a SecAND to produce the masked output of the comparison for this coefficient. It should be noted that this approach requires that the size of the largest interval $[\mathsf{S}(b), \mathsf{E}(b) - 1]$ should be smaller or equal to the difference of the used modulus to the next smaller power of two; i.e, $q - 2^{\lceil \log_2(q) \rceil} - 1$. For Kyber this is indeed the case for all parameter sets; for the values $d \in \{4, 5, 10, 11\}$ defined in Kyber and used in $\mathsf{Compress}_q(x, d)$ we have an interval size of at most 209 which is well below $q - 2^{\lceil \log_2(q) \rceil} - 1 = 3329 - 2^{11} = 1281$. Note that the special case where $d = 1$ is handled in detail in Section 3.1.

Complexity. Again, we estimate the run-time complexity $\mathcal{T}_{A_2}(n_s, k)$, where n_s goes to infinity, of Algorithm 2.

$$\begin{aligned}\mathcal{T}_{A_2} &= \mathcal{O}(1) + 4 \cdot 256 \cdot (\mathcal{O}(1) + 2 \cdot (\mathcal{T}_{\mathsf{MSB}} + \mathcal{T}_{\mathsf{A2B}} + \mathcal{T}_{\mathsf{Bitslice}})) \\ &+ 3 \cdot \frac{256}{w} \cdot \mathcal{T}_{\mathsf{SecAND}} + \sum_{i=0}^{7} \frac{2^i}{w} (\mathcal{T}_{\mathsf{LSR}} + \mathcal{T}_{mod} + \mathcal{T}_{\mathsf{SecAND}})\end{aligned}$$

with $k = \lceil \log_2(q) \rceil = 12$ and w denoting the word size of the target platform. Given in addition to the previous section, $\mathcal{T}_{\mathsf{MSB}} = \mathcal{T}_{\mathsf{LSR}} = \mathcal{T}_{mod} = \mathcal{O}(n_s)$ (applying the operation sharewise), as the complexities for the modules, we derive the asymptotic run-time complexity for Algorithm 2 as $\mathcal{T}_{A_2} = \mathcal{O}\left(n_s^2 \cdot \log_2(k)\right)$ for a constant p. Again it is dominated by the A2B conversion and, as in the previous case, the randomness complexity is $\mathcal{R}_{A_2} = \mathcal{O}\left(n_s^2 \cdot \log_2(k)\right)$.

Security. As we also did for Algorithm 1, we now prove Algorithm 2 to be t-SNI with $n_s = t + 1$ shares.

Theorem 2 (t-SNI of Algorithm 2)**.** *Let $\mathbf{u}'^{(\cdot)_A}$ and $v'^{(\cdot)_A}$ be the inputs and let $b^{(\cdot)_B}$ be the output of Algorithm 2. For any set of t_{A_2} intermediate variables and any subset $O \subset [0, n_s - 1]$ with $t_{A_2} + |O| < n_s$, there exists a subset I of input indices such that the t_{A_2} intermediate variables as well as $b^{(O)_B}$ can be perfectly simulated from $\mathbf{u}'^{(I)_A}$ and $v'^{(I)_A}$, with $|I| \leq t_{A_2}$.*

Proof. Again, we model Algorithm 2 as a sequence of t-(S)NI gadgets as depicted in Figure 3 and, as was the case in Theorem 1, we model the linear operation in lines 7, 8, 15, 16, 17, 20 as t-NI gadgets. The input c and its derived variables $(\mathbf{u}'', v'')$ and $(s_{\mathbf{u}''}, s_{\mathbf{u}''}, s_{v''}, e_{v''})$ are not explicitly considered in the proof as they are public values and their simulation is therefore trivial. As before, given that the iterations of the initial loops (i.e., `PolyCompare`) are independent, we consider them to be executed in parallel and summarize them into single gadgets. In this regard, we model the sequence $\mathsf{MSB} \circ \mathsf{A2B}$ as a single t-SNI gadget, which holds if the conversion $\mathsf{A2B}$ is t-SNI and MSB is applied sharewise. We unroll the final loop into two iterations, but the presented simulation concept generalizes to any number of rounds due to the t-SNI property of the used SecAND. The exact mapping of gadgets in Figure 3 to Algorithm 2 is as follows:

- G_{1-4} (NI): Assignment in Line 15.
- G_{5-8} (NI): Linear arithmetic in Lines 16 - 17.
- G_{9-12} (SNI): $\mathsf{MSB} \circ \mathsf{A2B}$ in Lines 18 - 19.
- G_{13-16} (NI): $\mathsf{Bitslice}$ in Lines 20.
- G_{17-19} (SNI): SecAND in Line 5.
- $G_{20,23}$ (NI): Upper half extraction in Line 7.
- $G_{21,24}$ (NI): Lower half extraction in Line 8.
- $G_{22,25}$ (SNI): SecAND in Line 9.

An adversary can place probes internally and on the output shares of each gadget. The number of internal (resp. output) probes for gadget G_i is denoted as t_{G_i} (resp. o_{G_i}) with

$$t_{A_2} = \sum_{i=1}^{25} t_{G_i} + \sum_{i=1}^{24} o_{G_i}, \quad |O| = o_{G_{25}}$$

where t_{A_2} and $|O|$ refer to respectively the number of probes and output shares of the complete Algorithm 2 as used in Theorem 2. To prove Theorem 2, we show that the internal probes and output shares can be perfectly simulated with $\leq t_{A_2}$ of the input shares $\mathbf{u}'^{(\cdot)_A}$ and $v'^{(\cdot)_A}$. Again, we provide details for the simulation at particular points in the algorithm. The complete explanation for each gadget is provided in Appendix B.

Starting with G_{25}, its $t_{G_{25}}$ internal probes and $o_{G_{25}}$ output shares can be simulated with $t_{G_{25}}$ of the output shares of G_{23} and G_{24}. This leads to a problem, however, as the simulation of these output shares requires a corresponding number of shares of the output of G_{22}, i.e., $2 \cdot t_{G_{25}}$. Therefore, on a variable-level these probes cannot be perfectly simulated. To overcome this issue, we model the gadgets G_{17} to G_{25} to work on bit-level rather than on the complete variables. This requires that we build multi-bit SecAND from parallel and independent one-bit instantiations of SecAND which each are t-SNI. The t-NI gadgets which are used to extract the upper and lower halves ($G_{20,21,23,24}$) can be represented similarly by one-bit t-NI gadgets, i.e., only selected bits are passed through, while the others are discarded.

We now explain the simulation for probes on the Least Significant Bit (LSB), but the presented approach applies to probes on arbitrary bits. To simulate $t_{G_{25}}$ internal probes and $o_{G_{25}}$ output shares of the LSB of G_{25}, we need $t_{G_{25}}$ output shares of the LSB of G_{23} and G_{24}. The former can be simulated with $(t_{G_{25}} + t_{G_{23}} + o_{G_{23}})$ output shares of the LSB of the upper half of G_{22}, while the latter requires $(t_{G_{25}} + t_{G_{24}} + o_{G_{24}})$ output shares of the LSB of the lower half of G_{22}. As these halves are independent, the simulation succeeds.

The same approach can be applied to simulate gadgets G_{20-22}. They require $(t_{G_{22}} + t_{G_{20}} + o_{G_{20}})$ output shares of the LSB of the upper half and $(t_{G_{22}} + t_{G_{21}} + o_{G_{21}})$ output shares of the LSB of the lower half of G_{19}. For G_{17} (resp. G_{18}), we need $t_{G_{17}}$ (resp. $t_{G_{18}}$) output shares of the LSB of $t_{\mathbf{w}}^{(\cdot)_B}$ and $t_{\mathbf{x}}^{(\cdot)_B}$ (resp. $t_y^{(\cdot)_B}$ and $t_z^{(\cdot)_B}$). To extend the simulation to probes on arbitrary bits, it is sufficient to replace the LSB with the corresponding indices of the probed bits.

As in the proof of Theorem 1, we add up the shares required for simulation for each of the bits of the bitsliced variables $(t_{\mathbf{w}}^{(\cdot)_B}, t_{\mathbf{x}}^{(\cdot)_B}, t_y^{(\cdot)_B}, t_z^{(\cdot)_B})$ to argue about $\mathsf{Bitslice}$. This is valid as there are no duplicate entries in the sums even without refreshes due to the t-SNI property of $G_{17,18}$, e.g., $t_{t_{\mathbf{w}}^{(\cdot)_B}} = \sum_{i=0}^{255} t_{G_{17}-\mathsf{Bit}_i}$. Again relying on the t-NI property of $\mathsf{Bitslice}$, we can simulate the shares of $(t_{\mathbf{w}}^{(\cdot)_B}, t_{\mathbf{x}}^{(\cdot)_B}, t_y^{(\cdot)_B}, t_z^{(\cdot)_B})$ with the corresponding number of shares of $(\mathbf{w}^{(\cdot)_B}, \mathbf{x}^{(\cdot)_B}, y^{(\cdot)_B}, z^{(\cdot)_B})$. Following the flow through gadgets G_{1-12}, the simulation of Algorithm 2 requires $|I|$ of the input shares $\mathbf{u'}^{(\cdot)_A}$ and $v'^{(\cdot)_A}$ with

$$\begin{aligned}
|I_{\mathbf{u}'}| &= t_{G_1} + o_{G_1} + t_{G_2} + o_{G_2} + t_{G_5} + o_{G_5} + t_{G_6} + o_{G_6} + t_{G_9} + t_{G_{10}} \\
|I_{v'}| &= t_{G_3} + o_{G_3} + t_{G_4} + o_{G_4} + t_{G_7} + o_{G_7} + t_{G_8} + o_{G_8} + t_{G_{11}} + t_{G_{12}} \\
|I| &= |I_{\mathbf{u}'}| + |I_{v'}|
\end{aligned}$$

Any probe on computations involving the outputs of Gadgets G_9 to G_{12} does not propagate to the respective inputs due to the t-SNI property of these gadgets. As $|I| \leq t_{A_2}$ and independent of o_{25}, Algorithm 2 is t-SNI. □

3.3 Masked CCA Kyber Decapsulation

We now return to the Kyber decapsulation given in Figure 1 and reason on the SCA security of the complete decapsulation. Note that we omitted the encode- and decode-operations as well as the generation of the public matrix A in the figure, as they are either trivial to mask or process only public values.

The linear polynomial operations (i.e., $\circ$, NTT, $-$, $+$) in Kyber.CPAPKE.Dec and Kyber.CPAPKE.Enc are masked as in previous works by applying the operation on each share separately. For $\mathsf{Compress}_q(x, 1)$ of Kyber.CPAPKE.Dec, we rely on our new approach as presented in Section 3.1.

To mask the symmetric components G and PRF, we rely on prior art. In particular, we use the masked Keccak approach and implementation from [BBD+16] to instantiate the modules at higher order while we use the more efficient approach from [BDPVA10] for the first order. We believe there is room for performance improvement by creating dedicated and more efficient masking schemes of Keccak aiming at a specific masking order (as done for the first-order setting), but this is out of scope for the current work.

For $\mathsf{Decompress}_q$, we first convert each bit of the Boolean-shared message $m'^{(\cdot)_B}$ to arithmetic shares modulo q (e.g., using the efficient one-bit $\mathsf{B2A}$ algorithm from [SPOG19] at higher orders) which are then multiplied with a constant.

To mask the sampler CBD of Kyber.CPAPKE.Enc, we adapt the bitsliced approach from [SPOG19] to the parameters of Kyber. For example, for $\eta = 2$ we first sum the input bits using Boolean-masked bitsliced addition. To compute the subsequent subtraction $a - b$ we first convert $b = (b_1 b_0)_2$ to $\bar{b} = (8 - b) \bmod 8 = (\bar{b}_1 \bar{b}_1 \bar{b}_0)_2$ by setting $\bar{b}_1 = b_1 \oplus b_0$ and $\bar{b}_0 = b_0$. We then compute $f = (a - b) \bmod 8 = (a + \bar{b}) \bmod 8 = (f_2 f_1 f_0)_2$ as

$$\begin{aligned}
f_0 &= a_0 \oplus \bar{b}_0\,, \quad f_1 = (a_0 \cdot \bar{b}_0) \oplus (a_1 \oplus \bar{b}_1)\,, \\
f_2 &= \bar{b}_1 \oplus (a_1 \cdot \bar{b}_1) \oplus (a_0 \cdot \bar{b}_0 \cdot (a_1 \oplus \bar{b}_1))\,.
\end{aligned}$$

Then we convert f to the arithmetic domain with shift constant 4, i.e., apply the conversion to $f \oplus 4$. Note that all these operations are applied in a bitsliced manner: on a 32-bit target

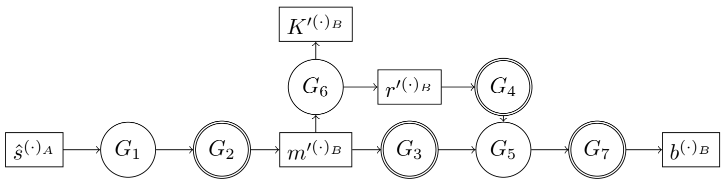


Figure 4: The gadgets considered in the proof of Theorem 3. t-NI gadgets are depicted with a single circle, t-SNI gadgets are depicted with a double circle.

platform these operations can be performed on 8 coefficients at the same time, assuming a and b are represented with 2 bits each. The subtraction of 4 after the conversion is trivial in arithmetic domain.

As depicted in Figure 1, our approach does not explicitly mask the ciphertext compression of Kyber.CPAPKE.Enc. Instead, we instantiate the comparison as presented in Section 3.2 which can process masked uncompressed polynomials. Furthermore, as in line with the findings of [BDH+21], we collapse the result of the comparison to a single masked bit before unmasking it for the selection of the KDF input.

We follow the approach and reasoning of [BDK+20] and do not mask the KDF. Instead, if the comparison outputs true (i.e., the ciphertext is valid), we unmask K' and perform an unmasked KDF. For a valid ciphertext this leaks only ephemeral secret information and not the long-term secret. Should this short-term secret also be protected, other countermeasures besides masking can be applied to mitigate against single-trace attacks. Note that it is important to not unmask K' if the comparison fails, because this could be used to attack the long-term secret. If the comparison does fail, we apply an unmasked KDF to the secret value z. This value is independent of the secret key, but leaking it allows an adversary to detect ciphertext rejection explicitly. This does not impact the IND-CCA security claims of Kyber [HHK17, Figure 1] as Kyber is γ-spread for sufficiently large γ.

To argue about the probing security of masked Kyber.CCAKEM.Dec, we analyze a reduced composition (denoted as G_{Dec}) excluding the unmasked components. The structure of G_{Dec} is depicted in Figure 4.

Theorem 3 (t-SNI of G_{Dec})**.** *Let $\hat{\mathbf{s}}^{(\cdot)_A}$ be the input and let $\bar{K}'^{(\cdot)_B}$ and $\mathrm{b}^{(\cdot)_B}$ be the output of G_{Dec}. For any set of $t_{G_{\mathsf{Dec}}}$ intermediate variables and any subset $O \subset [0, n_s - 1]$ with $t_{G_{\mathsf{Dec}}} + |O| < n_s$, there exists a subset I of input indices such that the $t_{G_{\mathsf{Dec}}}$ intermediate variables as well as $\mathrm{b}^{(O)_B}$ and $\bar{K}'^{(O)_B}$ can be perfectly simulated from $\hat{\mathbf{s}}^{(I)_A}$, with $|I| \le t_{G_{\mathsf{Dec}}}$.*

Proof. We model the linear operations of the decryption and encryption as t-NI gadgets G_1 and G_5. The new t-SNI $\mathsf{Compress}_q(x, 1)$ and comparison algorithms are included as G_2 and G_7. The symmetric components are modeled as a t-NI gadget G_6. As shown in [SPOG19], the sampling algorithm is t-SNI and their proof is independent of the concrete instantiation parameters. Therefore, we model it as t-SNI gadget G_4. For $\mathsf{Decompress}_q$, we assume a t-SNI gadget G_3 which relates to the t-SNI B2A conversion. The subsequent linear multiplication is included in G_5.

An adversary can place probes internally and on the output shares of each gadget. The number of internal (resp. output) probes for gadget G_i is denoted as t_{G_i} (resp. o_{G_i}) with

$$t_{G_{\mathsf{Dec}}} = \sum_{i=1}^{7} t_{G_i} + \sum_{i=1}^{5} o_{G_i}, \quad |O| = o_{G_6} + o_{G_7}$$

where $t_{G_{\mathsf{Dec}}}$ and $|O|$ refer to respectively the number of probes and output shares of the complete gadget G_{Dec} as used in Theorem 3. To prove Theorem 3, we show that the internal probes and output shares can be perfectly simulated with $\le t_{G_{\mathsf{Dec}}}$ of the input shares $\hat{\mathbf{s}}^{(\cdot)_A}$. Again, we provide details for the simulation at particular points in the algorithm. The complete explanation for each gadget is provided in Appendix C.

To simulate the internal probes and output shares of gadgets G_3 to G_7, we need $t_{G_3} + t_{G_6} + o_{G_6} + t_{G_4}$ shares of $m'^{(\cdot)_B}$. Following the flow through gadgets $G_{1,2}$, the simulation of G_{Dec} requires $|I| = t_{G_1} + o_{G_1} + t_{G_2}$ of the input shares $\hat{\mathbf{s}}^{(I)_A}$. As $|I| \leq t_{G_{\mathsf{Dec}}}$ and independent of o_{G_6} and o_{G_7}, gadget G_{Dec} is t-SNI. $\square$

4 Implementation and Evaluation

We present performance and practical security results of the new masked algorithms presented in Section 3. We target Kyber768 since this is the recommended parameter set targeting the NIST security level 3. We select two platforms: firstly, benchmarks (using the SysTick timer) and measurements are performed on an NXP Freedom Development Board for Kinetis Ultra-Low-Power KL82 MCUs (FRDM-KL82Z [NXP16]). The Cortex-M0+ was chosen because it is the most energy-efficient Arm processor available for constrained embedded applications. The processor comes equipped with a 2-stage pipeline, the Armv6-M architecture and the Thumb/Thumb-2 subset of instruction support. This allows the Cortex-M0+ to be a perfect candidate to harden cryptographic primitives since the hardened assembly code for the Cortex-M0+ can run on more advanced Arm instruction sets while vice-versa this is not necessarily true. Therefore, this hardened Cortex-M0+ implementation can serve as a helpful starting point to create secure hardened implementations for other Cortex target processors. Moreover, the side-channel behavior of the Cortex-M0+ is well understood, allowing to mitigate device-specific leakage behavior with fine-grained hardening strategies [BGG+21], whereas the unclear side-channel characteristics of the Cortex-M4 forces inserting dummy operations in many places, purely based on assumptions, resulting in additional performance penalties which are independent of the masked algorithms. The difference in understanding has been used to save up to 72% of dummy operations and can lead to optimized implementations which are twice as fast [BGG+21].

Secondly, although we do not perform measurements or any hardening, we do performance benchmarks on the STM32F407G-DISC1 board that comes equipped with a Cortex-M4F (previously known as STM32F4DISCOVERY). This is the platform used by the embedded crypto benchmark platform pqm4 [KRSS19] and recent masked implementations of Saber [BDK+20], so allows us to compare to existing work more easily. We make use of the standard measurement framework of pqm4, with minor modifications to measure the run-time of subroutines.

In this section, a component-wise performance comparison for various orders and implementation choices is given. Our masked Kyber implementation is generally written in C and based on the C-reference code from the Round 3 Kyber submission. For the Cortex-M4 processor we included the optimized assembly routines from pqm4, but the used assembly is incompatible with the much simpler Cortex-M0+. On the other hand, for our first-order Cortex-M0+ implementation we provide low-level formal verification and physical leakage assessments based on power measurements. For this purpose we target our own components $\mathsf{Compress}_q(., 1)$ and $\mathsf{DecompressedComparison}$. These hardened components (and any components that they rely on) are therefore written in assembly. Although hardening involves adding dummy operations that would decrease efficiency, our hand-written hardened assembly still performs better than the compiler-generated versions from the (unhardened, masked) plain-C implementation. Following the same approach as [BDK+20], we use an already existing masked implementation of Keccak in our masked Kyber implementation. More specifically, we re-used the first-order masked implementation from [BDPVA10] and for higher orders use the more generic higher-order secure implementation of Keccak from maskComp [BBD+16].

```
void A2B(boolean_share_t x, arith_share_t a) {
  uint16_t R, a0;
  rng(&R, KYBER_Q_BITSIZE);
  a0 = csubq(a[0] + KYBER_Q − $r_a$);
  a0 = csubq(a0 + a[1]);
  x[0] = $\mathbb{L}$[a0] ^ R;
  x[1] = $r_b$ ^ R;
}
```

Figure 5: LUT-based arithmetic to Boolean version based on [Deb12].

Randomness Generation. During the execution of decapsulation, fresh randomness is needed for the masked operations. For example, the first-order masked implementation on the FRDM-KL82Z uses 11 665 uniformly randomly sampled bytes for the decapsulation operation (see Table 2). As we would like the power measurements to be reproducible, the numbers for the FRDM-KL82Z reported in Table 2 assume that the random bytes can be readily read off from a table, which is filled before execution of the Kyber functions. Therefore, the cost of randomness generation is not included in our performance numbers for this platform. On the other hand, the STM32F407G-DISC1 board comes equipped with a TRNG. For fair comparison to existing work we do include the randomness generation in the cycle counts on this platform.

4.1 Performance Comparison

The main goal of this section is to demonstrate the feasibility of the new techniques to realize a (higher-order) masked Kyber implementation. For the FRDM-KL82Z we present the results of plain-C implementations and do not optimize on assembly level except for hardening some components. That being said, our first-order masked implementation does aim to be efficient from an algorithmic point of view to fairly represent the performance impact. For the STM32F407G we include the optimized assembly routines from pqm4. All implementations were compiled with `arm-none-eabi-gcc` version 8.3.1 with optimization level O3. The higher-order implementations (i.e., the second and third order results in Table 2) are not as aggressively optimized and therefore have more room left for improvement, in particular because the existing higher-order masked Keccak implementations are not heavily optimized.

First-Order Masking. Recall that we use Algorithm 1 for $\mathsf{Compress}_q(.,1)$, Algorithm 2 for DecompressedComparison and [SPOG19, Algorithm 3] for the conversion from arithmetic to Boolean shares (A2B). However, for first-order masking the algorithm from [SPOG19] is not the most efficient. Instead, we use a Look-Up-Table (LUT) based approach; more specifically, the improved [Deb12] version of the Coron–Tchulkine method [CT03]. This algorithm was designed for power-of-two moduli so cannot be used directly for our prime q. To overcome this we simply use larger tables to avoid dealing with any carry propagation. Moreover, we also refresh the output with fresh randomness as the input and output masks in our case are from different domains, and to achieve the assumed t-SNI property.

Let us give a concrete example of the approach for the implementation of the first-order A2B conversion. The table $\mathbb{L}$ satisfies $\mathbb{L}\,(a) = (a + r_a \bmod q) \oplus r_b$, where a is a secret value in $[0, q-1]$ which is arithmetically masked with randomness $r_a \in [0, q-1]$ on the input side and a Boolean mask with the random value $r_b \in [0, 2^{\lceil \log_2(q) \rceil} - 1]$ is applied on the output side. Then the arithmetic to Boolean conversion is implemented as in Figure 5. Here $\texttt{rng}(x, y)$ stores y uniformly random sampled bits in x, and `csubq` performs a conditional subtraction by the modulus q. More explicitly, the constant-time equivalent of the C-expression “`c = ((c >= q)? c-q : c)`”.

Table 1: The different LUT settings used in our first-order (FO) and higher-order (HO) implementations on the FRDM-KL82Z and their corresponding cycle counts rounded up to nearest 10^3 cycles. For first order a LUT is used for A2B, for higher orders it is not.

Setting		**Approach**		**#Cycles**			
		$\mathsf{Comp}_q(.,1)$	Dec.Comp.	Init	$\mathsf{Comp}_q(.,1)$	Dec.Comp.	**Total**
HO	0	Alg. 1	Alg. 2	–	–	–	–
FO	1	LUT	LUT	2 032	65	1 407	3 504
	2	LUT	Alg. 2	766	66	1 232	2 064
	3	Alg. 1	Alg. 2	181	145	1 255	1 581

The A2B of [Deb12] outperforms the method of [SPOG19] at first order, and therefore we use it in our first-order implementation. It is however not directly clear whether a similar approach can be used for $\mathsf{Compress}_q(.,1)$ and DecompressedComparison. Indeed, a completely analogous masked LUT can be created for these functions. For example, one can replace DecompressedComparison by implementing $\mathsf{Compress}_q(.,d)$ with a LUT, followed by a hashed comparison as done in previous work [OSPG18, BDK+20]. We compare the performance of the various options in Table 1. Concretely, we use the following four settings in our masked implementations. The first setting (denoted setting 0) uses no LUTs at all; this is the default for the higher-order (> 1) masked-implementations. The first-order implementations do use the LUT-approach for A2B and the respective possible settings for the FRDM-KL82Z for our modules can be found in Table 1. The LUTs are generated fresh for every Kyber invocation and this run-time is included in the overall reported performance results (Init). Since Init, $\mathsf{Compress}_q(.,1)$ and DecompressedComparison are only called once per decapsulation, the total cost is computed as the sum of the separate functions. It is clear that using the new algorithms introduced in this work is favorable compared to using the LUT-approach even in the first-order setting. This is due to the fact that the LUT-initialization takes a non-negligible amount of time, which is significant because the functions are only used once (as opposed to A2B). Therefore, our first-order implementations use setting 3.

Performance Discussion. We present a complete overview of our performance results on both the FRDM-KL82Z and the STM32F407G in Table 2. As a baseline unmasked implementation for the FRDM-KL82Z we take the Kyber768 implementation from the PQClean [KRS+19] software library. This is purely written in C and therefore a comparison to our plain-C implementation is fair. Of course some of our modules contain assembly modifications to harden them against power analysis, but this leads to only minor differences in cycle counts. For the STM32F407G we take the currently best optimized implementation from pqm4.

We see that the overall slowdown factor for `crypto_kem_dec` masked at first order on the FRDM-KL82Z is 2.2x. A large part of this can be attributed to the masked encryption step, which uses the masked PRF as part of the binomial sampler while also about doubling the cost of the polynomial arithmetic. The $\mathsf{Compress}_q(.,1)$ and DecompressedComparison (denoted `comparison` in the table) also introduce large slowdowns, but this was to be expected as their cost in the unmasked version was almost negligible. It should be noted that the slowdown factor is relatively small due to the lack of assembly optimizations: since the cost of polynomial arithmetic is still significant, while it has a small slowdown factor, the overall slowdown compared to the reference implementation is brought down.

On the other hand, the slowdown factor for first-order masked decapsulation on the STM32F407G is 3.5x. Although comparing subroutines directly is difficult due to interleaving in the pqm4 reference, the overall slowdown is larger than on the FRDM-KL82Z as the cost of polynomial is relatively lower due to assembly optimization. More precisely, the cost of masking is dominated by Keccak operations rather than polynomial arithmetic, which are more expensive to mask. Our slowdown factor is better than the (tentative) factor 4.2x reported recently by Heinz et al. for a first-order masked implementation [Dan21] at the PQC Standardization Conference organized by NIST, though their version is hardened for the Cortex-M4 while ours is unhardened. The overall slowdown is larger than the 2.52x factor reported in [BDK+20, Table 5] for masking Saber on the same platform. This is mainly caused by the high cost of the Keccak-based binomial sampler: constructing the four error polynomials for Kyber requires the use of the binomial sampler which uses rejection sampling modulo 3329 to convert the shares from arithmetic to Boolean (256 per polynomial). A similar operation is also performed in Saber and Kyber for the generation of 3 secret polynomials. However, Saber uses rounding as opposed to random errors and therefore does not need to generate these errors vectors.

Unsurprisingly, the number of random bytes used by Kyber is larger than for Saber. Whereas [BDK+20] makes 1262 calls to a 32-bit TRNG, using 5048 bytes in total, we sample a total of 12 072 random bytes. This is firstly caused by the generation of additional error polynomials, as mentioned above. Secondly, the $\mathsf{Compress}_q(.,1)$ and DecompressedComparison components require more randomness compared to their counterparts in Saber and use 704 and 4396 random bytes respectively.

The performance impact for the higher order implementations is much larger. In particular, the relative cost of DecompressedComparison compared to the whole decapsulation increases. This is mainly due to the poor performance of the A2B component for higher orders [SPOG19, Algorithm 3], as opposed to the LUT-based version for first order, which is both slow and requires most of the random bytes in decapsulation. We expect many optimizations are still possible in the higher-order A2B.

4.2 Verification

In addition to the hand-written proofs of Non-Interference we employ the verification tool scVerif to mechanically verify the side-channel resilience of the introduced algorithms on assembly level in realistic leakage models [BGG+21]. The disassembled object files of our implementation are verified to be Stateful (Strong) Non-Interferent in a fine-grained leakage model to ensure resistance against both the Cortex-M0+ device-specific leakage behavior and the residual state in concrete execution. Stateful NI differs from NI in that it mandates the state (e.g., registers and stack) after execution to be independent of secrets and random values, except for specified locations, and thereby facilitates secure composition of masked assembly components.

scVerif performs verification in fine-grained leakage models by augmenting an internal representation of the assembly code with user-supplied explicit leakages (denoted as "leakage model") in the form of **leak** statements which are considered as internal probes t_G in the proof of (Stateful) NI (Definition 1). Verification with scVerif is split into two phases, (1) partial evaluation to lower the assembly representation into a simpler language amenable to (2) verification in the subsequent stage.

In the following, we detail the changes required to adapt scVerif to the Kyber implementation. The leakage model presented in [BGG+21] is extended for arithmetic and shift instructions as well as branching instructions which are modeled without leakage. Our leakage model serves as design aid, containing known and assumed leakage behavior but comes without profound physical validation, as in e.g. [BGG+21, MPW21]. Instead, we augment the model whenever a discrepancy between the model and the observed physical leakage is encountered by constructing tests for the concerned instruction as in [PV17]. This allows us to prove the absence of vulnerabilities arising from known leakage behavior. During our hardening phase we adopt the leakage of multiple single-operand instructions, which differs from instructions with multiple operands. The extended formal leakage model

Table 2: Performance benchmarks on the FRDM-KL82Z (Cortex-M0+) and STM32F407G (Cortex-M4F) platforms of the masked implementation of the various parts of KYBER768. The cycle counts are reported in thousands and rounded up to the nearest 10^3 cycles. The FRDM-KL82Z results do not include randomness generation while the STM32F407G results do. The slowdown factor of the 1st order implementation compared to PQClean and pqm4 is included in brackets. The high-level pqm4 functions are not subdivided as they are implemented in an interleaved fashion to reduce the memory use. The ciphertext `comparison` is in compressed form for PQClean and pqm4, and decompressed form for the masked components. Results marked * are hardened.

Operation	FRDM-KL82Z				STM32F407G			
	PQClean[a]	**New**			**pqm4**[b]	**New**		
	(unmasked)	*1st*	*2nd*	*3rd*	*(unmasked)*	*1st*	*2nd*	*3rd*
`crypto_kem_dec`	5 530	12 208 (2.2x)	107 352	231 632	882	3 116 (3.5x)	44 347	115 481
`LUT_create`	0	241 (∞x)	0	0	–	37	47	47
`indcpa_dec`	703	1 096 (1.6x)	6 886	18 166	–	174	2 916	9 288
`hashg (SHA3-512)`	61	361 (5.9x)	4 457	6 507	–	118	1 543	2 659
`indcpa_enc`	4 160	8 708 (2.1x)	52 132	75 864	–	2 196	17 743	30 838
`comparison`	5	*1 206 (241.0x)	43 270	130 489	–	462	22 017	72 568
`hashh (SHA3-256)`	530	535 (1.0x)	540	540	–	115	115	115
`kdf`	65	65 (1.0x)	66	66	–	14	14	14
#randombytes	–	11 665	901 880	2 408 880	–	12 072	902 126	2 434 170
`indcpa_enc`	4 160	8 708 (2.1x)	52 132	75 864	676	2 196 (3.3x)	17 743	30 838
`decompression`	21	287 (13.7x)	644	994	–	113	267	490
`gen_at`	1 755	1 723 (1.0x)	1 771	1 748	–	398	398	398
`poly_getnoise`	494	3 729 (7.5x)	44 227	66 112	–	1 384	16 625	29 347
`poly_arith`	1 683	2 968 (1.8x)	5 490	7 010	–	301	452	603
#randombytes	–	6556	277 304	537 684	–	7 030	277 550	562 974
`indcpa_dec`	703	1 096 (1.6x)	6 886	18 166	64	174 (2.7x)	2 916	9 288
`unpack`	53	68 (1.3x)	86	102	–	23	30	36
`poly_arith`	638	885 (1.4x)	1 388	1 710	–	89	119	149
`compress`	22	*143 (6.5x)	5 411	16 354	–	61	2 767	9 102
#randombytes	–	704	66 432	201 984	–	640	66 432	201 984

[a] `https://github.com/PQClean/PQClean/tree/master/crypto_kem/kyber768/clean` commit `c00cb2d`

[b] `https://github.com/mupq/pqm4/tree/master/crypto_kem/kyber768/m4` commit `157e271`

Algorithm 3 Simplified scVerif code to represent table lookups for formal verification.

```
LUT(Rd, Rn, Rm)
1: val ← (Rn + Rm − baseaddress$_{\mathbb{L}}$ + r$_a$) ⊕ r$_b$;
2: leak lutOperand (opA, opB, Rn, Rm);
3: leak lutMemOperand (opR, val);
4: leak lutTransition (Rd, val);
5: opR ← val; opA ← Rn; opB ← Rd; Rd ← val;
```

developed might be of independent interest and is provided in a Listing 1 in Appendix E.

Kyber makes use of constants (e.g., the modulus q) which are stored alongside the program code. We extend scVerif to allow program counter (`pc`) relative memory accesses to be partially evaluated for the subsequent verification phase. In doing so, we extend the front-end of scVerif to process the assembly `.word` directive which introduces a constant at some fixed address: pairs of addresses and constants are placed in the memory view ρ for the state $\langle p, c, \mu, \rho, ec\rangle$ of the partial evaluator presented in [BGG+21].

Verification of code containing table lookups, e.g., as used in the table-base A2B conversion, poses problems as it cannot be partially evaluated since the secret value (share) used as address or offset in the memory access is a symbol which does not resolve to a concrete table index. We resolve this by patching the code during verification and substituting the respective lookup instruction (e.g. `ldr`) with a virtual instruction (**LUT**) defined in scVerif intermediate language which exposes the same leakage behavior but expresses the semantic of the lookup in functional form.

Let us give an example of this approach for the implementation of the first-order A2B as explained in Section 4.1. In assembly the lookup in $\mathbb{L}$ is implemented as `ldr Rd, [Rn, Rm]`, where `Rd` is the destination register, while `Rn` and `Rm` contain the base address of $\mathbb{L}$ and an offset. The accessed table index defined by `Rn` and `Rm` cannot be partially evaluated since it is secret dependent. To allow partial evaluation the instruction is substituted by the virtual instruction **LUT** implemented in the scVerif intermediate language depicted in Algorithm 3. The global variables `r`$_a$ and `r`$_b$ are annotated to contain uniformly distributed random masks, specific to the masked table. Line 1 is a leak-free assignment which allows one to convert and re-mask the sensitive value, satisfying the table's functionality $\mathbb{L}(a) = (a + r_a \bmod q) \oplus r_b$. The explicit leaks ensure that the side-channel behavior of the substituted `ldr` is equivalently modeled. Using this approach we verify our table-based A2B conversion to be Stateful Strong Non-Interferent.

In the subsequent verification of our $\mathsf{Compress}_q(.,1)$ and DecompressedComparison implementations we replace calls (branches) to A2B and random number generators by simplified variants implemented in the scVerif intermediate language, exposing a worst-case leakage assumption that leaks a combination of all registers. This allows us to harden our implementation with respect to different implementations of random number generators and A2B implementations.

Given these pre-requisites, the security of $\mathsf{Compress}_q(.,1)$ and DecompressedComparison is verified. The large size of our DecompressedComparison implementation forces us reduce its parameters (i.e., $k = 1$ and $n = 64$) for verification, while $\mathsf{Compress}_q(.,1)$ can be verified for the Kyber768 parameters. Both components take nine minutes each to verify successfully; stating that both implementations are Stateful Strong Non-Interferent in our fine-grained leakage model.

4.3 Leakage Assessment

Finally, we evaluate the practical side-channel resilience of our hardened first-order implementation by performing statistical leakage detection on physical side-channel measurements. We use the KL82Z development board with capacitors `C31`, `C39`, `C43`, `C45`, `C46`, `C59` and `C61` de-soldered and an inductive current clamp connected on Jumper `J15`. A PicoScope 6404C oscilloscope samples the power consumption at 312.5 MS/s, a bandwidth of 500 MHz and 8 bit quantification. The micro-controller is clocked at 12 MHz, resulting in slightly more than 26 samples per clock cycle.

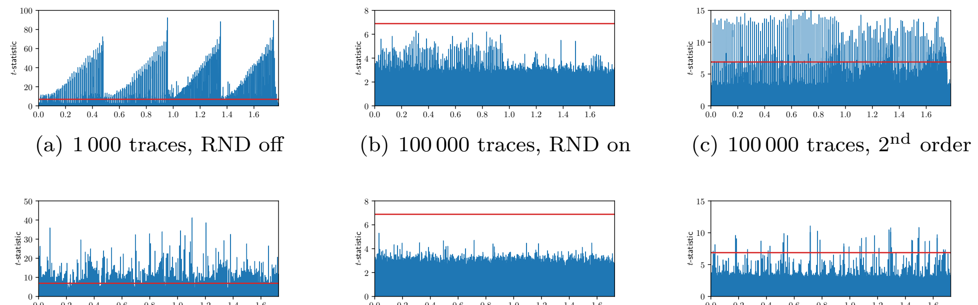
For leakage detection, we rely on the widely-used t-test-based Test Vector Leakage Assessment (TVLA) [GGJR+11] comparing fixed with random inputs. In particular, a Welch t-test comparing the fixed and random measurements is computed, and the resulting t-value is compared to a set threshold of ± 4.5, representing $\alpha = 0.0001$. Informally, if the threshold is exceeded, it is assumed to be possible to distinguish between fixed and random inputs, which indicates the existence of exploitable leakage. We refer the reader to [CDG+13, SM16] for more details on TVLA. However, Ding et al. have shown in [DZD+17], that this threshold needs to be adapted for very long traces to avoid false positives during leakage detection. As our measurements indeed consist of numerous sample points, we adapt their approach to set the threshold for our leakage assessments to avoid erroneous results.

We measure the power consumption of the Cortex-M0+ processor executing 50 000 invocations of the algorithms on a fixed secret value which is freshly masked for each execution, and another set of 50 000 invocations on uniformly distributed secret values. In both cases, the implementation is provided with fresh, pre-sampled randomness stored in a table. In line with [OSPG18] and [BDK+20], we instantiate the measured module $\mathsf{DecompressedComparison}$ with reduced parameter sets, i.e., $k = 1$ and $n = 64$ to mitigate the large size, while ensuring that the entire function can be assessed. The parameters are chosen such that loops are executed for at least two iterations. We choose the public compressed values in such a way that the invocations with the fixed value compare correctly to all but the last compressed coefficient, whereas the invocations on random (uncompressed) coefficients result in an invalid comparison to the fixed compressed coefficients with high chance. Only by comparing uncompressed to compressed coefficients which match in the fixed invocation, we can assess the secrecy of all intermediate comparisons and the handling of the resulting flag.

The $\mathsf{Compress}_q(.,1)$ is assessed without reducing parameters (i.e., $n = 256$). For $\mathsf{DecompressedComparison}$, the measurements consist of 1,782,438 sample points for which we set the threshold to 6.89 as described in [DZD+17]. For $\mathsf{Compress}_q(.,1)$, we need to process 1,726,452 sample points, and therefore set the threshold to 6.88.

For a first-order secure implementation, the assessment is expected to show no significant leakage at first-order, while exceeding the threshold at second order. To validate our setup, we first run the test when the randomness source turned off. In this case, the thresholds are exceeded for just 1000 traces, as depicted in Figures 6(a) and 6(d). The visible sawtooth pattern in Figure 6(a) corresponds to the bitsliced comparison of 32 coefficients in parallel.

In normal operation (i.e., randomness source turned on) our hardened algorithms do not exhibit significant first-order leakage at 100 000 measurements as can be seen in Figures 6(b) and 6(e). On the other hand, significant univariate second-order leakage is detectable, as depicted in Figures 6(c) and 6(f), indicating that second-order attacks are likely to succeed. To increase the SCA resilience beyond the first-order resilience which is provided by our first-order masked implementation, additional countermeasures are required, e.g., increasing the masking order. Our presented higher-order masked algorithms enable to implement Kyber at arbitrary orders, allowing to protect against higher-order SCA attacks.



(a) 1 000 traces, RND off (b) 100 000 traces, RND on (c) 100 000 traces, 2^{nd} order

(d) 1 000 traces, RND off (e) 100 000 traces, RND on (f) 100 000 traces, 2^{nd} order

Figure 6: Results of TVLA assessment: the top row shows decompressed comparison for (a) 1^{st} order without randomness, (b) 1^{st} order with randomness, and (c) 2^{nd} order with randomness, while the bottom row shows 1-bit compression for (d) 1^{st} order without randomness, (e) 1^{st} order with randomness, and (f) 2^{nd} order with randomness. The x axis represents sample point index $\times 10^6$.

5 Conclusion

In this work, we presented the first masking scheme for a complete Kyber decapsulation, at both first and higher orders. This is achieved by combining known techniques with two new approaches to respectively mask a one-bit compression and decompressed comparison. We prove both algorithms to be t-SNI and show how to compose them to create a masked Kyber.CCAKEM.Dec. We implement our proposed masking scheme on an Arm Cortex-M0+ and Cortex-M4F at orders one to three. For first-order masked Kyber, this resulted in an overhead factor of 3.5, 3.3 and 2.7 compared to unmasked for Kyber.CCAKEM.Dec, Kyber.CPAPKE.Enc, and Kyber.CPAPKE.Dec respectively on the Cortex-M4F. We explicitly hardened the first-order implementations of our new algorithms on the Cortex-M0+. Their leakage behavior was both formally and practically verified using scVerif and TVLA with 100 000 measurements. Both approaches do not detect leakage in our hardened modules of $\mathsf{Compress}_q(.,1)$ and DecompressedComparison.

References

[AASA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the second round of the NIST post-quantum cryptography standardization process. Technical Report NISTIR 8309, National Institute of Standards and Technology, 2020. https://doi.org/10.6028/NIST.IR.8309.

[ACLZ20] Dorian Amiet, Andreas Curiger, Lukas Leuenberger, and Paul Zbinden. Defeating NewHope with a single trace. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 189–205. Springer, Heidelberg, 2020.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016.

[BBC+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskVerif: Automated verification of higher-order masking in presence of physical defaults. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *ESORICS 2019, Part I*, volume 11735 of *LNCS*, pages 300–318. Springer, Heidelberg, September 2019.

[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.

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 354–384. Springer, Heidelberg, April / May 2018.

[BBK+17] Nina Bindel, Johannes Buchmann, Juliane Krämer, Heiko Mantel, Johannes Schickel, and Alexandra Weber. Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics. In Abdessamad Imine, José M. Fernandez, Jean-Yves Marion, Luigi Logrippo, and Joaquín García-Alfaro, editors, *Foundations and Practice of Security - 10th International Symposium, FPS 2017, Nancy, France, October 23-25, 2017, Revised Selected Papers*, volume 10723 of *Lecture Notes in Computer Science*, pages 225–241. Springer, 2017.

[BCNS15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy – SP*, pages 553–570. IEEE Computer Society, 2015.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 23–39. Springer, Heidelberg, August 2016.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. Improved high-order conversion from Boolean to arithmetic masking. *IACR TCHES*, 2018(2):22–45, 2018. https://tches.iacr.org/index.php/TCHES/article/view/873.

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

[BDK+18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy – Euro S&P*, pages 353–367. IEEE, 2018.

[BDK+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel resistant implementation of SABER. Cryptology ePrint Archive, Report 2020/733, 2020. https://eprint.iacr.org/2020/733.

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

[BGG+21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR TCHES*, 2021(2):189–228, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8792.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 323–345. Springer, Heidelberg, August 2016.

[BPO+20] Florian Bache, Clara Paglialonga, Tobias Oder, Tobias Schneider, and Tim Güneysu. High-speed masking for polynomial comparison in lattice-based kems. *IACR TCHES*, 2020(3):483–507, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8598.

[CDG+13] Jeremy Cooper, Elke Demulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, Pankaj Rohatgi, et al. Test vector leakage assessment (tvla) methodology in practice. In *International Cryptographic Module Conference*, volume 20, 2013.

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between Boolean and arithmetic masking of any order. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 188–205. Springer, Heidelberg, September 2014.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, August 1999.

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

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. A new algorithm for switching from arithmetic to Boolean masking. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, pages 89–97. Springer, Heidelberg, September 2003.

[Dan21] Daniel Heinz and Matthias J. Kannwischer and Georg Land and Thomas Pöppelmann and Peter Schwabe and Daan Sprenkels. First-Order Masked Kyber on ARM Cortex-M4. https://csrc.nist.gov/CSRC/media/Presentations/first-order-masked-kyber-on-arm-cortex-m4/images-media/session-4-heinz-first-order-masked-kyber.pdf, 2021.

[Deb12] Blandine Debraize. Efficient and provably secure methods for switching from arithmetic to Boolean masking. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 107–121. Springer, Heidelberg, September 2012.

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

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 18*, volume 10831 of *LNCS*, pages 282–305. Springer, Heidelberg, May 2018.

[DTVV19] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. Timing attacks on error correcting codes in post-quantum schemes. In Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen, editors, *Proceedings of ACM Workshop on Theory of Implementation Security Workshop, TIS@CCS 2019, London, UK, November 11, 2019*, pages 2–9. ACM, 2019.

[DZD+17] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, volume 10728 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2017.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongSwan and electromagnetic emanations in microcontrollers. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1857–1874. ACM Press, October / November 2017.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Heidelberg, August 1999.

[GGJR+11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 359–386. Springer, Heidelberg, August 2020.

[GJRS18] Dahmun Goudarzi, Anthony Journault, Matthieu Rivain, and François-Xavier Standaert. Secure multiplication for bitslice higher-order masking: Optimisation and comparison. In Junfeng Fan and Benedikt Gierlichs, editors, *COSADE 2018*, volume 10815 of *LNCS*, pages 3–22. Springer, Heidelberg, April 2018.

[GR19] François Gérard and Mélissa Rossi. An efficient and provable masked implementation of qtesla. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*, volume 11833 of *Lecture Notes in Computer Science*, pages 74–91. Springer, 2019.

[HCY19] Wei-Lun Huang, Jiun-Peng Chen, and Bo-Yin Yang. Power analysis on NTRU prime. *IACR TCHES*, 2020(1):123–151, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8395.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Heidelberg, November 2017.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe Buhler, editor, *Algorithmic Number Theory – ANTS-III*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998.

[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.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, Heidelberg, August 1996.

[KRS$^+$19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, Douglas Stebila, and Thom Wiggers. The PQClean project, November 2019. https://github.com/PQClean/PQClean.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. Workshop Record of the Second PQC Standardization Conference, 2019. https://cryptojedi.org/papers/#pqm4.

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

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium - efficient implementation and side-channel evaluation. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 344–362. Springer, Heidelberg, June 2019.

[MPW21] Ben Marshall, Dan Page, and James Webb. Miracle: Micro-architectural leakage evaluation. Cryptology ePrint Archive, Report 2021/261, 2021. https://eprint.iacr.org/2021/261.

[Nat] National Institute of Standards and Technology. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[NDGJ21] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A side-channel attack on a masked IND-CCA secure saber KEM. *IACR Cryptol. ePrint Arch.*, 2021:79, 2021.

[NXP16] NXP Semiconductors. FRDM-KL82Z User's Guide. https://www.nxp.com/docs/en/user-guide/FRDMKL82ZUG.pdf, 2016.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure masked Ring-LWE implementations. *IACR TCHES*, 2018(1):142–174, 2018. https://tches.iacr.org/index.php/TCHES/article/view/836.

[PP19] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In Peter Schwabe and Nicolas Thériault, editors, *LATINCRYPT 2019*, volume 11774 of *LNCS*, pages 130–149. Springer, Heidelberg, 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, *CHES 2017*, volume 10529 of *LNCS*, pages 513–533. Springer, Heidelberg, September 2017.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, Heidelberg, May 2013.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In Sylvain Guilley, editor, *COSADE 2017*, volume 10348 of *LNCS*, pages 282–297. Springer, Heidelberg, April 2017.

[PZ03] J. Proos and C. Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Inf. Comput.*, 3:317—344, 2003.

[RBRC20] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. Drop by drop you break the rock - exploiting generic vulnerabilities in lattice-based PKE/KEMs using EM-based physical attacks. Cryptology ePrint Archive, Report 2020/549, 2020. https://eprint.iacr.org/2020/549.

[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.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-LWE implementation. In Tim Güneysu and Helena Handschuh, editors, *CHES 2015*, volume 9293 of *LNCS*, pages 683–702. Springer, Heidelberg, September 2015.

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

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[SM16] Tobias Schneider and Amir Moradi. Leakage assessment methodology - extended version. *Journal of Cryptographic Engineering*, 6(2):85–99, June 2016.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 534–564. Springer, Heidelberg, April 2019.

[SRSW20] Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. A power side-channel attack on the cca2-secure HQC KEM. 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 119–134. Springer, 2020.

[SW07] Joseph H. Silverman and William Whyte. Timing attacks on NTRUEncrypt via variation in the number of hash calls. In Masayuki Abe, editor, *CT-RSA 2007*, volume 4377 of *LNCS*, pages 208–224. Springer, Heidelberg, February 2007.

[TE15] Mostafa Taha and Thomas Eisenbarth. Implementation attacks on post-quantum cryptographic schemes. Cryptology ePrint Archive, Report 2015/1083, 2015. `https://eprint.iacr.org/2015/1083`.

[XPRO20] Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, and David Oswald. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. Cryptology ePrint Archive, Report 2020/912, 2020. `https://eprint.iacr.org/2020/912`.

A Supporting Material: Proof Theorem 1

- G_{13} (NI): The $t_{G_{13}}$ internal probes and $o_{G_{13}}$ output shares of the gadget can be simulated with $t_{G_{13}} + o_{G_{13}}$ shares of $x_{11}^{(\cdot)_B}$ and of the output of G_{12}.
- G_{12} (SNI): The $t_{G_{12}}$ internal probes and $o_{G_{12}}$ output shares of the gadget can be simulated with $t_{G_{12}}$ shares of the output G_{10} and G_{11}.
- G_{11} (NI): The $t_{G_{11}}$ internal probes and $o_{G_{11}}$ output shares of the gadget can be simulated with $t_{G_{11}} + o_{G_{11}}$ shares of $x_{11}^{(\cdot)_B}$.
- G_{10} (SNI): The $t_{G_{10}}$ internal probes and $o_{G_{10}}$ output shares of the gadget can be simulated with $t_{G_{10}}$ shares of $x_{10}^{(\cdot)_B}$ and of the output of G_9.
- G_9 (SNI): The t_{G_9} internal probes and o_{G_9} output shares of the gadget can be simulated with t_{G_9} shares of $x_9^{(\cdot)_B}$ and of the output of G_8.
- G_8 (SNI): The t_{G_8} internal probes and o_{G_8} output shares of the gadget can be simulated with t_{G_8} shares of the output of G_7.
- G_7 (NI): The t_{G_7} internal probes and o_{G_7} output shares of the gadget can be simulated with $t_{G_7} + o_{G_7}$ shares of $x_8^{(\cdot)_B}$ and of the output of G_6.
- G_6 (SNI): The t_{G_6} internal probes and o_{G_6} output shares of the gadget can be simulated with t_{G_6} shares of $x_7^{(\cdot)_B}$ and of the output of G_5.
- G_5 (SNI): The t_{G_5} internal probes and o_{G_5} output shares of the gadget can be simulated with t_{G_5} shares of the output of G_4.
- G_4 (NI): The t_{G_4} internal probes and o_{G_4} output shares of the gadget can be simulated with $t_{G_4} + o_{G_4}$ shares of $x_8^{(\cdot)_B}$.
- G_3 (NI): The t_{G_3} internal probes and o_{G_3} output shares of the gadget can be simulated with $t_{G_3} + o_{G_3}$ shares of $a^{(\cdot)_B}$.
- G_2 (SNI): The t_{G_2} internal probes and o_{G_2} output shares of the gadget can be simulated with t_{G_2} shares of the output of G_1.
- G_1 (NI): The t_{G_1} internal probes and o_{G_1} output shares of the gadget can be simulated with $t_{G_1} + o_{G_1}$ shares of the input $a^{(\cdot)_A}$.

B Supporting Material: Proof Theorem 2

- G_{25} (SNI): The $t_{G_{25}}$ internal probes and $o_{G_{25}}$ output shares of the gadget can be simulated with $t_{G_{25}}$ shares of the output of G_{23} and G_{24}.
- G_{24} (NI): The $t_{G_{24}}$ internal probes and $o_{G_{24}}$ output shares of the gadget can be simulated with $t_{G_{24}} + o_{G_{24}}$ shares of the output of G_{22}.
- G_{23} (NI): The $t_{G_{23}}$ internal probes and $o_{G_{23}}$ output shares of the gadget can be simulated with $t_{G_{23}} + o_{G_{23}}$ shares of the output of G_{22}.
- G_{22} (SNI): The $t_{G_{22}}$ internal probes and $o_{G_{22}}$ output shares of the gadget can be simulated with $t_{G_{22}}$ shares of the output of G_{20} and G_{21}.
- G_{21} (NI): The $t_{G_{21}}$ internal probes and $o_{G_{21}}$ output shares of the gadget can be simulated with $t_{G_{21}} + o_{G_{21}}$ shares of the output of G_{19}.

- G_{20} (NI): The $t_{G_{20}}$ internal probes and $o_{G_{20}}$ output shares of the gadget can be simulated with $t_{G_{20}} + o_{G_{20}}$ shares of the output of G_{19}.
- G_{19} (SNI): The $t_{G_{19}}$ internal probes and $o_{G_{19}}$ output shares of the gadget can be simulated with $t_{G_{19}}$ shares of the output of G_{17} and G_{18}.
- G_{18} (SNI): The $t_{G_{18}}$ internal probes and $o_{G_{18}}$ output shares of the gadget can be simulated with $t_{G_{18}}$ shares of the output of G_{15} and G_{16}.
- G_{17} (SNI): The $t_{G_{17}}$ internal probes and $o_{G_{17}}$ output shares of the gadget can be simulated with $t_{G_{17}}$ shares of the output of G_{13} and G_{14}.
- G_{16} (NI): The $t_{G_{16}}$ internal probes and $o_{G_{16}}$ output shares of the gadget can be simulated with $t_{G_{16}} + o_{G_{16}}$ shares of the output of G_{12}.
- G_{15} (NI): The $t_{G_{15}}$ internal probes and $o_{G_{15}}$ output shares of the gadget can be simulated with $t_{G_{15}} + o_{G_{15}}$ shares of the output of G_{11}.
- G_{14} (NI): The $t_{G_{14}}$ internal probes and $o_{G_{14}}$ output shares of the gadget can be simulated with $t_{G_{14}} + o_{G_{14}}$ shares of the output of G_{10}.
- G_{13} (NI): The $t_{G_{13}}$ internal probes and $o_{G_{13}}$ output shares of the gadget can be simulated with $t_{G_{13}} + o_{G_{13}}$ shares of the output of G_9.
- G_{12} (SNI): The $t_{G_{12}}$ internal probes and $o_{G_{12}}$ output shares of the gadget can be simulated with $t_{G_{12}}$ shares of the output of G_8.
- G_{11} (SNI): The $t_{G_{11}}$ internal probes and $o_{G_{11}}$ output shares of the gadget can be simulated with $t_{G_{11}}$ shares of the output of G_7.
- G_{10} (SNI): The $t_{G_{10}}$ internal probes and $o_{G_{10}}$ output shares of the gadget can be simulated with $t_{G_{10}}$ shares of the output of G_6.
- G_9 (SNI): The t_{G_9} internal probes and o_{G_9} output shares of the gadget can be simulated with t_{G_9} shares of the output of G_5.
- G_8 (NI): The t_{G_8} internal probes and o_{G_8} output shares of the gadget can be simulated with $t_{G_8} + o_{G_8}$ shares of the output of G_4.
- G_7 (NI): The t_{G_7} internal probes and o_{G_7} output shares of the gadget can be simulated with $t_{G_7} + o_{G_7}$ shares of the output of G_3.
- G_6 (NI): The t_{G_6} internal probes and o_{G_6} output shares of the gadget can be simulated with $t_{G_6} + o_{G_6}$ shares of the output of G_2.
- G_5 (NI): The t_{G_5} internal probes and o_{G_5} output shares of the gadget can be simulated with $t_{G_5} + o_{G_5}$ shares of the output of G_1.
- G_4 (NI): The t_{G_4} internal probes and o_{G_4} output shares of the gadget can be simulated with $t_{G_4} + o_{G_4}$ shares of the input $v'^{(\cdot)_A}$.
- G_3 (NI): The t_{G_3} internal probes and o_{G_3} output shares of the gadget can be simulated with $t_{G_3} + o_{G_3}$ shares of the input $v'^{(\cdot)_A}$.
- G_2 (NI): The t_{G_2} internal probes and o_{G_2} output shares of the gadget can be simulated with $t_{G_2} + o_{G_2}$ shares of the input $\mathbf{u}'^{(\cdot)_A}$.
- G_1 (NI): The t_{G_1} internal probes and o_{G_1} output shares of the gadget can be simulated with $t_{G_1} + o_{G_1}$ shares of the input $\mathbf{u}'^{(\cdot)_A}$.

Table 3: Recommended parameter sets for KYBER.

	n	k	q	η_1	η_2	(d_u, d_v)	δ
KYBER512	256	2	3329	3	2	(10,4)	2^{-139}
KYBER768	256	3	3329	2	2	(10,4)	2^{-164}
KYBER1024	256	4	3329	2	2	(11,5)	2^{-174}

Algorithm 4 Sampling from $\mathsf{CBD}_\eta \colon \mathcal{B}^{64\eta} \to R_{3329}$ as used in Kyber.

```
Input: Byte array $B = (b_0, b_1, \dots, b_{64\eta-1}) \in \mathcal{B}^{64\eta}$
Output: Polynomial $f \in R_{3329}$
  $(\beta_0, \dots, \beta_{512\eta-1}) \coloneqq \mathsf{BytesToBits}(B)$
  for $i$ from 0 to 255 do
      $a \coloneqq \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
      $b \coloneqq \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$
      $f_i \coloneqq a - b$
  return $f_0 + f_1 X + f_2 X^2 + \cdots + f_{255} X^{255}$
```

C Supporting Material: Proof Theorem 3

- G_7 (SNI): The t_{G_7} internal probes and o_{G_7} output shares of the gadget can be simulated with t_{G_7} shares of the output of G_5.
- G_6 (NI): The t_{G_6} internal probes and o_{G_6} output shares of the gadget can be simulated with $t_{G_6} + o_{G_6}$ shares of the output of G_2.
- G_5 (NI): The t_{G_5} internal probes and o_{G_5} output shares of the gadget can be simulated with $t_{G_5} + o_{G_5}$ shares of the output of G_3 and G_4.
- G_4 (SNI): The t_{G_4} internal probes and o_{G_4} output shares of the gadget can be simulated with t_{G_4} shares of the output of G_6.
- G_3 (SNI): The t_{G_3} internal probes and o_{G_3} output shares of the gadget can be simulated with t_{G_3} shares of the output of G_2.
- G_2 (SNI): The t_{G_2} internal probes and o_{G_2} output shares of the gadget can be simulated with t_{G_2} shares of the output of G_1.
- G_1 (NI): The t_{G_1} internal probes and o_{G_1} output shares of the gadget can be simulated with $t_{G_1} + o_{G_1}$ shares of the input $\hat{\mathbf{s}}^{(\cdot)_A}$.

D Kyber Round 3 Tables & Algorithms

Algorithm 5 $\textsc{Kyber}.\mathsf{CPAPKE.Enc}(pk, m, r)$: encryption

```
Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Input: Message $m \in \mathcal{B}^{32}$
Input: Random coins $r \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
 1: $N := 0$
 2: $\hat{\mathbf{t}} := \mathsf{Decode}_{12}(pk)$
 3: $\rho := pk + 12 \cdot k \cdot n/8$
 4: for $i$ from 0 to $k-1$ do          ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in $\mathsf{NTT}$ domain
 5:     for $j$ from 0 to $k-1$ do
 6:         $\hat{\mathbf{A}}^T[i][j] := \mathsf{Parse}(\mathsf{XOF}(\rho, i, j))$
 7: for $i$ from 0 to $k-1$ do          ▷ Sample $\mathbf{r} \in R_q^k$ from $B_{\eta_1}$
 8:     $\mathbf{r}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(r, N))$
 9:     $N := N + 1$
10: for $i$ from 0 to $k-1$ do          ▷ Sample $\mathbf{e}_1 \in R_q^k$ from $B_{\eta_2}$
11:     $\mathbf{e}_1[i] := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$
12:     $N := N + 1$
13: $e_2 := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$          ▷ Sample $e_2 \in R_q$ from $B_{\eta_2}$
14: $\hat{\mathbf{r}} := \mathsf{NTT}(\mathbf{r})$
15: $\mathbf{u} := \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$          ▷ $\mathbf{u} := \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
16: $v := \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$          ▷
    $v := \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decompress}_q(m, 1)$
17: $c_1 := \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\mathbf{u}, d_u))$
18: $c_2 := \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
19: return $c = (c_1 \| c_2)$          ▷ $c := (\mathsf{Compress}_q(\mathbf{u}, d_u), \mathsf{Compress}_q(v, d_v))$
```

Algorithm 6 $\textsc{Kyber}.\mathsf{CPAPKE.Dec}(sk, c)$: decryption

```
Input: Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Message $m \in \mathcal{B}^{32}$
 1: $\mathbf{u} := \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c), d_u)$
 2: $v := \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
 3: $\hat{\mathbf{s}} := \mathsf{Decode}_{12}(sk)$
 4: $m := \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1))$ ▷ $m := \mathsf{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1))$
 5: return $m$
```

Algorithm 7 $\textsc{Kyber}.\mathsf{CCAKEM.Dec}(c, sk)$

```
Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Input: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8 + 96}$
Output: Shared key $K \in \mathcal{B}^*$
 1: $pk \coloneqq sk + 12 \cdot k \cdot n/8$
 2: $h \coloneqq sk + 24 \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
 3: $z \coloneqq sk + 24 \cdot k \cdot n/8 + 64$
 4: $m' \coloneqq \textsc{Kyber}.\mathsf{CPAPKE.Dec}(\mathbf{s}, (\mathbf{u}, v))$
 5: $(\bar{K}', r') \coloneqq \mathrm{G}(m' \| h)$
 6: $c' \coloneqq \textsc{Kyber}.\mathsf{CPAPKE.Enc}(pk, m', r')$
 7: if $c = c'$ then
 8:     return $K \coloneqq \mathsf{KDF}(\bar{K}' \| \mathrm{H}(c))$
 9: else
10:     return $K \coloneqq \mathsf{KDF}(z \| \mathrm{H}(c))$
11: return $K$
```

E Leakage Models used during verification with scVerif

We provide the formal leakage model which was used to verify the security of our masked assembly implementations of Algorithms 1 and 2 for the Cortex-M0+ processor. Our leakage model was based on the model presented in [BGG+21] but merely served as design aid. It was extended for assumed leakage behavior as well as observed effects but comes without profound physical validation.

Every macro defines the leakage behavior of an instruction in the domain specific language "IL" (refer to [BGG+21] for a detailed description), e.g., `ands2_leak` models the leakage of the Arm assembly instruction `ands` with two operands of 32 bits. The semantics of the instructions are provided in the supplementary material of [BGG+21]. The model includes the virtual `lut` instruction which specifies the leakage model and semantic of the table for A2B conversion (Section 4.2).

Listing 1: Fine-grained side-channel leakage model used during verification of concrete assembly implementations.

```
w32 opA; // global leakage state to model leakage behavior which depends on past
    ↪ instructions
w32 opB;
w32 opR;
w32 opW;

macro ands2_leak (w32 op1, w32 op2) {
  leak andsCompResult (op1 &w32 op2);
  leak andsTransition (op1, op1 &w32 op2);
  leak andsOperand (opA, op1, opB, op2);
  leak andsOperandA (opA, op1);
  leak andsOperandB (opB, op2);

  opA <- op1;
  opB <- op2;
}

macro eors2_leak (w32 op1, w32 op2) {
  leak eorsCompResult (op1 ^w32 op2);
  leak eorsTransition (op1, op1 ^w32 op2);
  leak eorsOperand (opA, op1, opB, op2);
  leak eorsOperandA (opA, op1);
  leak eorsOperandB (opB, op2);

  opA <- op1;
  opB <- op2;
}

macro ldr3_leak (w32 dst, w32 adr, w32 ofs)
  w32 val
{
  val <- [w32 mem (int) (adr +w32 ofs)];

  leak ldrOperand1 (opA, adr, opB, ofs);

  leak ldrOperand2A (opA, adr);
  leak ldrOperand2B (opB, dst);
  leak ldrMemOperand (opR, val);
  leak ldrTransition (dst, val);

  opA <- adr;
  opB <- dst;
  opR <- val;
}

// virtual instruction as substitute for the ldr instruction in the table lookup of A2B with equivalent leakage behavior, including the functional semantic of the table
macro lut3 (w32 dst, w32 adr, w32 ofs)
  w32 val, w32 ra, w32 rb, w32 baseaddr
{
  // in our modified implementation pcptr points to the static table containing the base-addres for the random masks r_a, r_b and the lookup table
  baseaddr <- [w32 mem (int) pcptr];
  ra <- [w32 mem (int) (baseaddr +w32 (w32) 0)];
  rb <- [w32 mem (int) (baseaddr +w32 (w32) 4)];

  val <- adr +w32 ofs -w32 baseaddr +w32 ra ^w32 rb;
  leak lutOperand (opA, opB, adr, ofs);
  leak lutMemOperand (opR, val);
  leak lutTransition (dst, val);

  opA <- adr;
  opB <- dst;
  opR <- val;

  dst <- val; // semantic action
}

macro str3_leak (w32 val, w32 adr, w32 ofs) {
  leak strOperand1 (opA, adr, opB, ofs);

  leak strOperand2A (opA, adr);
  leak strOperand2B (opB, val);
  leak strMemOperand (opW, val);

  opA <- adr;
  opB <- val;
  opW <- val;
}

macro mov2_leak (w32 dst, w32 src) {
  leak movCompResult (src);
  leak movOperand (opA, dst, opB, src);
  leak movTransition (dst, src);

  opA <- src; // assumption here was "opA <- dst" which is wrong
// opB <- src; // assumption here was wrong, opB is not cleared but propagated
}

macro adds3_leak (w32 dst, w32 op1, w32 op2) {
  leak addsCompResult (op1 +w32 op2);
  leak addsTransition (dst, op1 +w32 op2);
  leak addsOperand (opA, op1, opB, op2);

  opA <- opA &w32 opB &w32 op1; // worst case assumption
  opB <- opA &w32 opB &w32 op2; // worst case assumption
```

```
}

macro add3_leak (w32 dst, w32 op1, w32 op2) {
  leak addCompResult (op1 +w32 op2);
  leak addTransition (dst, op1 +w32 op2);
  leak addOperand (opA, op1, opB, op2);

  opA <- opA &w32 opB &w32 op1; // worst case assumption
  opB <- opA &w32 opB &w32 op2; // worst case assumption
}

macro mvns2_leak (w32 dst, w32 src) {
  leak movCompResult (!w32 src);
  leak movOperand (opA, dst, opB, src);
  leak movTransition (dst, !w32 src);

  opA <- opA &w32 src;
}

macro adcs2_leak (w32 dst, w32 op) {
  leak sbcsCompResult (op -w32 dst);
  leak sbcsOperand (opA, dst, opB, op);
  leak sbcsTransition (dst, op -w32 dst);

  opA <- dst; // assumption
  opB <- op;
}

macro sbcs2_leak (w32 dst, w32 op) {
  leak sbcsCompResult (op -w32 dst);
  leak sbcsOperand (opA, dst, opB, op);
  leak sbcsTransition (dst, op -w32 dst);

  opA <- opA &w32 op;
}

macro subs3_leak (w32 dst, w32 op1, w32 op2) {
  leak subsCompResult (op1 +w32 op2);
  leak subsTransition (dst, op1 +w32 op2);
  leak subsOperand (opA, op1, opB, op2);

  opA <- op1;
  opB <- op2;
}

macro sub3_leak (w32 dst, w32 op1, w32 op2) {
  leak subCompResult (op1 +w32 op2);
  leak subTransition (dst, op1 +w32 op2);
  leak subOperand (opA, op1, opB, op2);

  opA <- op1;
  opB <- op2;
}

macro ldrb3_leak (w32 dst, w32 adr, w32 ofs) {
  ldr3_leak(dst, adr, ofs);
}
```

```

macro ldrh3_leak (w32 dst, w32 adr, w32 ofs) {
  ldr3_leak(dst, adr, ofs);
}

macro strb3_leak (w32 op, w32 adr, w32 ofs) {
  str3_leak(op, adr, ofs);
}

macro strh3_leak (w32 op, w32 adr, w32 ofs) {
  str3_leak(op, adr, ofs);
}

macro sxtb2_leak (w32 dst, w32 src) {
  leak sxtbCompResult (src);
  leak sxtbOperand (opA, dst, opB, src);
  leak sxtbTransition (dst, src);

  opA <- opA &w32 src;
}

macro sxth2_leak (w32 dst, w32 src) {
  leak sxthCompResult (src);
  leak sxthOperand (opA, dst, opB, src);
  leak sxthTransition (dst, src);

  opA <- opA &w32 src;
}

macro lsls3_leak (w32 dst, w32 op1, w32 shift) {
  leak lslsCompResult (op1);
  leak lslsOperand (opA, dst, opB, op1);
  leak lslsTransition (dst, op1);

  opA <- op1; // assumption
// opB <- op1;
}

macro lsls2_leak (w32 op1, w32 op2) {
  lsls3_leak(op1, op1, op2);
}

macro lsrs3_leak (w32 dst, w32 op1, w32 shift) {
  leak lsrsCompResult (op1);
  leak lsrsOperand (opA, dst, opB, op1);
  leak lsrsTransition (dst, op1);

  opA <- op1; // assumption
// opB <- op1;
}

macro lsrs2_leak (w32 op1, w32 op2) {
  lsrs3_leak(op1, op1, op2);
}

macro asrs3_leak (w32 dst, w32 op1, w32 shift) {
  leak asrsCompResult (op1);
  leak asrsOperand (opA, dst, opB, op1);
  leak asrsTransition (dst, op1);

  opA <- op1; // assumption
// opB <- op1;
}

macro asrs2_leak (w32 dst, w32 op1) {
  asrs3_leak(dst, dst, op1);
}

macro muls3_leak (w32 dst, w32 op1, w32 op2) {
  leak mulsCompResult (op1 *w32 op2);
  leak mulsTransition (dst, op1 *w32 op2);
  leak mulsOperand (opA, op1, opB, op2);

  opA <- op1;
  opB <- op2;
}

macro muls2_leak (w32 op1, w32 op2) {
  muls3_leak(op1, op1, op2);
}

macro orrs2_leak (w32 op1, w32 op2) {
  leak orrsCompResult (op1 |w32 op2);
  leak orrsTransition (op1, op1 |w32 op2);
  leak orrsOperand (opA, op1, opB, op2);
  leak orrsOperandA (opA, op1);
  leak orrsOperandB (opB, op2);

  opA <- op1;
  opB <- op2;
}

macro rsbs2_leak (w32 op1, w32 op2) {
  leak orrsCompResult (op2 -w32 op1);
  leak orrsTransition (op1, op2 -w32 op1);
  leak orrsOperand (opA, op1, opB, op2);

  opA <- opA &w32 op2;
}

macro negs2_leak (w32 op1, w32 op2) {
  rsbs2_leak(op1, op2);
}

macro uxtb2_leak (w32 dst, w32 src) {
  leak uxtbCompResult (src);
  leak uxtbOperand (opA, dst, opB, src);
  leak uxtbTransition (dst, src);

  opA <- opA &w32 src;
}

macro uxth2_leak (w32 dst, w32 src) {
  uxtb2_leak(dst, src);
```

```
}

macro cmp2_leak (w32 op1, w32 op2) {
  subs3_leak(op1, op1, op2);
}

macro tst2_leak (w32 op1, w32 op2) {
  leak tstCompResult (op1 &w32 op2);
  leak tstOperand (opA, op1, opB, op2);
  leak tstOperandA (opA, op1);
  leak tstOperandB (opB, op2);

  opA <- opA &w32 op1;
  opB <- opB &w32 op2;
}

// the semantic of pop instructions is build from the load, therefore the values
    ↪ on the stack leak as well
macro pop1_leak (w32 op1) {
  leak pop (op1, opR);
}

macro pop2_leak (w32 op1, w32 op2) {
  leak pop (op1, op2, opR);
}

macro pop3_leak (w32 op1, w32 op2, w32 op3) {
  leak pop (op1, op2, op3, opR);
}

macro pop4_leak (w32 op1, w32 op2, w32 op3, w32 op4) {
  leak pop (op1, op2, op3, op4, opR);
}

macro pop5_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5) {
  leak pop (op1, op2, op3, op4, op5, opR);
}

macro pop6_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5, w32 op6) {
  leak pop (op1, op2, op3, op4, op5, op6, opR);
}

macro pop7_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5, w32 op6, w32 op7) {
  leak pop (op1, op2, op3, op4, op5, op6, op7, opR);
}

macro pop8_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5, w32 op6, w32 op7,
    ↪ w32 op8) {
  leak pop (op1, op2, op3, op4, op5, op6, op7, op8, opR);
}

macro pop9_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5, w32 op6, w32 op7,
    ↪ w32 op8, w32 op9) {
  leak pop (op1, op2, op3, op4, op5, op6, op7, op8, op9, opR);
}

macro push1_leak (w32 op1) {
  leak push (op1, opW);
}

macro push2_leak (w32 op1, w32 op2) {
  leak push (op1, op2, opW);
}

macro push3_leak (w32 op1, w32 op2, w32 op3) {
  leak push (op1, op2, op3, opW);
}

macro push4_leak (w32 op1, w32 op2, w32 op3, w32 op4) {
  leak push (op1, op2, op3, op4, opW);
}

macro push5_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5) {
  leak push (op1, op2, op3, op4, op5, opW);
}

macro push6_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5, w32 op6) {
  leak push (op1, op2, op3, op4, op5, op6, opW);
}

macro push7_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5, w32 op6, w32 op7) {
  leak push (op1, op2, op3, op4, op5, op6, op7, opW);
}

macro push8_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5, w32 op6, w32 op7,
    ↪ w32 op8) {
  leak push (op1, op2, op3, op4, op5, op6, op7, op8, opW);
}

macro push9_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5, w32 op6, w32 op7,
    ↪ w32 op8, w32 op9) {
  leak push (op1, op2, op3, op4, op5, op6, op7, op8, op9, opW);
}
```