IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2022, No. 4, pp. 553–588. DOI:10.46586/tches.v2022.i4.553-588

Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit

with Application to Lattice-Based KEMs

Olivier Bronchain and Gaëtan Cassiers

Crypto Group, ICTEAM Institute, UCLouvain, Louvain-la-Neuve, Belgium.
{olivier.bronchain,gaetan.cassiers}@uclouvain.be

Abstract. The performance of higher-order masked implementations of lattice-based based key encapsulation mechanisms (KEM) is currently limited by the costly conversions between arithmetic and Boolean masking. While bitslicing has been shown to strongly speed up masked implementations of symmetric primitives, its use in arithmetic-to-Boolean and Boolean-to-arithmetic masking conversion gadgets has never been thoroughly investigated. In this paper, we first show that bitslicing can indeed accelerate existing conversion gadgets. We then optimize these gadgets, exploiting the degrees of freedom offered by bitsliced implementations. As a result, we introduce new arbitrary-order Boolean masked addition, arithmetic-to-Boolean and Boolean-to-arithmetic masking conversion gadgets, each in two variants: modulo 2^k and modulo p (for any integers k and p). Practically, our new gadgets achieve a speedup of up to 25x over the state of the art. Turning to the KEM application, we develop the first open-source embedded (Cortex-M4) implementations of `Kyber768` and `Saber` masked at arbitrary order. The implementations based on the new bitsliced gadgets achieve a speedup of 1.8x for `Kyber` and 3x for `Saber`, compared to the implementation based on state-of-the-art gadgets. The bottleneck of the bitslice implementations is the masked `Keccak-f[1600]` permutation.

Keywords: Masking · Lattice-based KEM · Kyber · Saber · Bitslice · PINI

1 Introduction

Quantum attacks against traditional asymmetric cryptography schemes (based on RSA, discrete logarithm or elliptic curves) have been a growing concern. This led to the introduction of post-quantum (PQ) schemes for signatures and key encapsulation mechanisms (KEM), many of which are based on lattices. Their implementation raises new challenges, in particular for embedded systems that require protection against side-channel attacks (SCA) such as power or electro-magnetic analysis [KJJ99, QS01]. Such attacks are particularly powerful against many state-of-the-art PQ KEMs due to their usage of the Fujisaki-Okamoto (FO) transform [FO99]: an adversary can carefully forge ciphertexts to trigger the re-encryption of a single bit whose value depends on a secret (sub-)key. The leakage from this re-encryption depends only on this single secret bit, which is thus easily recovered and from which information on the secret key can be retrieved [RRCB20, UXT+22]. Strong protection against side-channel attacks is therefore a must for lattice-based cryptography in embedded systems deployed on-the-field [ABH+22].

The most studied countermeasure against SCA is masking, whose core idea is to randomize the intermediate computations while maintaining their correctness [CJRR99, ISW03]. When using arithmetic masking, each intermediate variable x of the original computation is replaced by a sharing $(x_0, \ldots, x_{d-1})$ such that $x = x_0 + \cdots + x_{d-1} \mod p$

Licensed under Creative Commons License CC-BY 4.0.
Received: 2022-04-15 Accepted: 2022-06-15 Published: 2022-08-31

for some integer p, where the addition degenerates to the Boolean XOR in the particular case $p = 2$, which is therefore named Boolean masking. Masked implementations are usually analyzed in the t-probing model [ISW03], which formalizes the notion of t-order security by requiring all tuples of t intermediate values in the computation to be independent of any secret value. However, security in the t-probing model is not composable: the sequential use of two t-probing secure gadgets (gadgets are algorithms computing on masked values) is not necessarily probing secure [CPRR13]. To circumvent the t-probing security analysis of a full masked cryptographic algorithm (which is impractical), composable security properties have been introduced, such as (strong-)non-interference (NI/SNI) [BBD⁺16], or probe-isolating non-interference (PINI) [CS20]. These properties are stronger than probing security and gadgets that satisfy them can be securely composed.

The protection of masking does not come for free and sometimes leads to orders of magnitudes larger costs than non-masked implementation [BGR⁺21]. A key question in the design of masked implementation is therefore the minimization of computational cost, which is particularly critical when considering embedded software PQ KEMs implementations. Indeed, unprotected implementations of PQ KEMs are already computationally expensive [KRSS], and on top of this a high masking order is needed, due to the low intrinsic noise level on commercial micro-controllers [BS20, BS21]. Masking overheads (in randomness usage and runtime) generally grow quadratically with the number of shares, except for masked linear operations modulo p, which incur only linear computational overhead (and no randomness usage).

Lattice-based KEMs use many arithmetic operations in the field of integers modulo p (e.g., $p = 3329$, 2^{10} or 2^{13}). These operations are often linear with respect to the secret values [ABD⁺19, BBMD⁺19], which leads to a very efficient implementation when using arithmetic masking modulo p [RRVV15, OSPG18]. These KEMs also use symmetric cryptography primitives to generate pseudo-randomness, which are often best implemented using Boolean masking since they contain many bit-level operations [BDPA13, GR16, BDM⁺20]. As a result, conversions between arithmetic and Boolean masking are key components of masked implementations of lattice-based KEMs.

These conversions are a bottleneck of the current state-of-the-art implementations [BGR⁺21, FBR⁺22] and they are an active field of research. Arbitrary-order arithmetic-to-Boolean masking conversions (A2B) were first introduced in [CGV14] for fields of characteristic two and a masking order equal to half of the number of shares. In a series of works [CGTV15, BBE⁺18, SPOG19], the construction was generalized to arbitrary p and optimal masking order $(d-1)$, along with optimizations to reach $\mathcal{O}\left(d^2 \log(\log p)\right)$ CPU instructions. Alternative table-based constructions have also been introduced, achieving similar properties [CGMZ21a]. Boolean-to-arithmetic conversion (B2A) has also been studied thoroughly. The original arbitrary-order B2A [CGV14] is based on A2B and benefited from its improvements, as well as being proven secure at optimal security order in [BBE⁺18]. Recently, efficient B2A algorithms for conversion of a single bit have been introduced [SPOG19, CGMZ21a], from which a B2A algorithm for an arbitrary number of bits can be derived. Finally, the *compression* modulo p is an operation which consists in a linear scaling then a rounding, and is commonly found in Lattice-based KEMs. Its masking can be performed thanks to A2B conversions and has been recently optimized in [BPO⁺20, BDH⁺21, CGMZ21b].

In parallel over the last years, the *bitslicing* technique has brought significant speed improvements to software implementations of symmetric cryptography, be it masked [GR16, BDM⁺20] or not [Bih97, AP21]. In short, bitslicing leverages the intrinsic parallelism of bitwise operations within processors. E.g., a processor that manipulates 32-bit integers can perform 32 bitwise operations with a single instruction. Therefore, bitslicing only applies to algorithms whose operations are bitwise, such as [GLSV14], but sometimes an algorithm can be re-written to use bit-level operations (while preserving efficiency) [BMP13]. In

particular, Boolean masking is very well suited to bitslicing since most Boolean masking gadgets only use bit-level operations, whereas arithmetic masking gadgets use additions and multiplications (whose equivalent bitwise circuits are large) and therefore do not benefit from bitslicing. To the best of our knowledge, despite many works on A2B and B2A, no efficient bitslice implementation of such conversion algorithm has ever been introduced.

Contributions We introduce the usage of bitslicing for the masked implementation of lattice-based cryptography, and for this purpose, we design new masked gadgets for all masking orders. Our new gadgets are A2B and B2A conversions. Additionally, we also design a new addition gadget for Boolean masking which is used in the conversion gadgets. These gadgets come in two variants: one for arithmetic modulo any integer p, and one for the particular case of arithmetic modulo 2^k, which is more efficient. All our gadgets are PINI, and are therefore easily composed.

As a testbed for our new gadgets, we develop arbitrary-order masked Kyber and Saber implementations on the Cortex-M4 platform. First, for each of them, we build a non-bitsliced masked implementation (hereafter named respectively K1 and S1) based on state-of-the-art components: the gadgets of Coron et al. [CGMZ21a], some gadgets from [SPOG19] and some (non-masked) functions from the NIST PQ benchmarking project (PQM4) [KRSS]. To the best of our knowledge, implementations K1 and S1 are the first open-source [1] embedded masked at arbitrary order Kyber and Saber software implementations. Next, we build new bitslice implementations (named K2 and S2) that use our new gadgets and satisfy the PINI secure composition strategy. Implementation K2 achieves a speedup of up to 1.84x over K1, and up to 8.7x over the best reported performance in the state-of-the-art on an embedded platform [BGR+21]. Similarly, S2 achieves a speedup of 3x over S1. In both K2 and S2, the execution time is dominated by hashing respectively by 50% for Kyber and 72% for Saber. Eventually, we also propose implementations K3 and S3 which include assembly implementation of masked Boolean gates to avoid lower-order leakages due to transitions.[2]

Related work We note that the noise sampling proposed in [SPOG19, BDK+21] leverages bitslicing in order to perform the `CBD` with Boolean masking, but the conversion to arithmetic masking is not bitsliced. Moreover, [DHP+22] mentions a bitsliced implementation of the A2B conversion of [CGV14] but does not optimize the algorithm.[3]

Organization In Section 2, we introduce some preliminaries on masking and describe the state-of-the-art gadgets for Boolean masked addition, A2B masking conversion, as well as B2A. Next, we present our new gadgets and prove that they are PINI in Section 3, before comparing their performance to the state-of-the-art in Section 4. We then perform leakage assessment of the proposed gadgets in Section 5. Finally, we describe our `Kyber768` and `Saber` implementations and measure their performance in Section 6.

2 Background

In this Section, we first introduce our notations and the masking schemes we use, then we describe state-of-the-art gadgets that operate on masked values to perform simple operations, namely addition and conversion between masking schemes.

[1] The implementations K1/S1 are available at https://github.com/uclcrypto/pqm4_masked/files/8048895/implems.zip.

[2] The implementations K2/K3/S2/S3 are available at https://github.com/uclcrypto/pqm4_masked.

[3] Another recent work [DBV22] (which appeared online after the original submission of this paper to TCHES) implements with bitslicing the B2A algorithm of [CGV14].

Notations We denote by $[\![x, y]\!]$ the set $[x, y] \cap \mathbb{N}$ and by $[\![x, y[\![$ the set $[x, y) \cap \mathbb{N}$. For non-negative integers x and y, $x \oplus y$ is the (unsigned) integer whose binary representation is the bitwise XOR of the binary representations of x and y.

2.1 Masking and elementary gadgets

In this paper, we consider two masking schemes: arithmetic and Boolean masking. A secret variable $x \in [\![0, p[\![$ for some integer p is represented by the d-shares arithmetic sharing

$$\boldsymbol{x}^{A_p} = \left(\boldsymbol{x}_i^{A_p}\right)_{i=0,\ldots,d-1} \in [\![0, p[\![^d \text{ such that } x = \boldsymbol{x}_0^{A_p} + \boldsymbol{x}_1^{A_p} + \cdots + \boldsymbol{x}_{d-1}^{A_p} \mod p.$$

In order to achieve $d-1$-order security for x, any set of $d-1$ shares must be uniformly distributed. Similarly, the k-bit Boolean sharing of a secret $x \in [\![0, 2^k[\![$ is

$$\boldsymbol{x}^{B,k} = \left(\boldsymbol{x}_i^{B,k}\right)_{i=0,\ldots,d-1} \in [\![0, 2^k[\![^d \text{ such that } x = \boldsymbol{x}_0^{B,k} \oplus \boldsymbol{x}_1^{B,k} \oplus \ldots \oplus \boldsymbol{x}_{d-1}^{B,k}.$$

Computation on sharings is performed by algorithms named gadgets. The inputs and outputs of a d-share gadget are d-shares sharings, which allows such gadgets to be composed: the composition of multiple gadgets (which must all have the same number of shares) results in a composite gadget. The input sharings of the composing gadgets (named hereafter sub-gadgets) may be the input sharing of the composite gadget, or an output sharing of another sub-gadget.

For both arithmetic and Boolean masking, the operations that are linear with respect to the sharing operation are implemented by simple gadgets: the operation can be applied share-wise, hence the computational cost is $\mathcal{O}(d)$. In particular, for arithmetic (respectively Boolean) masking, one such operation is the addition modulo p (resp. bitwise XOR) of two shared variables. We denote these algorithms as $+^{\mathtt{A}}$ (resp. $\oplus^{\mathtt{B}}$).

The ISW multiplication gadget [ISW03], which we denote `SecAnd` allows computing bitwise AND of Boolean-shared values at a randomness and computational cost $\mathcal{O}\left(d^2\right)$. This gadget may also be used to compute the product modulo p of two arithmetically shared secrets.

A last commonly used gadget is the refresh gadget, which implements the identity function, but re-randomizes the sharing. This gadget is sometimes used to ensure the security of a computation that composes multiple simpler gadgets.

2.2 Composable probing security

In this paper, we target $(d-1)$-probing security for our d-shares implementations. That is, the statistical distribution of any $d-1$ intermediate values (named probes) in our computation should be independent of any secret. We build our masked gadgets by composing multiple smaller gadgets. However, probing security is not composable [CPRR13]: composing $(d-1)$-probing secure gadgets is not enough to ensure $(d-1)$-probing security.

As a result, we consider stronger security definitions that are composable. These definitions rely on the notion of simulatability.

Definition 1 (Simulatability [BBP+16])**.** A set of t probes in a masked gadget G can be simulated with a subset I of the input shares of G if there exists a randomized algorithm S (named the simulator) such that for any value taken by the input shares of G, the joint distribution of the probes is equal to the distribution of the output of S when the values of the shares in I are given to it as inputs.

The two following composable security definitions were introduced in [BBD+16].

Definition 2 (t-NI)**.** A gadget is t-Non-Interfering (t-NI) if every set of t probes can be simulated by using at most t shares of each input sharing.

Definition 3 (t-SNI)**.** A gadget with one output sharing is t-Strong-Non-Interfering (t-SNI) if every set of t_1 probes on the internal values and t_2 probes on the output shares, with $t_1 + t_2 \leq t$, can be simulated by using at most t_1 shares of each input sharing.

The $+^{\mathtt{A}}$ and $\oplus^{\mathtt{B}}$ gadgets are $(d-1)$-NI while the ISW multiplication is $(d-1)$-SNI. Furthermore, the refresh gadget obtained by setting one input sharing of the ISW multiplication to $(1, 0, \ldots, 0)$ is also SNI, and this set of gadgets enables to securely mask any computation [BBD+16].

Composition based on the NI and SNI definitions requires the usage of refresh gadgets, which may significantly increase the computational and randomness cost. More recently, Cassiers and Standaert [CS20] introduced a new definition that allows to remove those refresh gadgets.

Definition 4 (t-PINI)**.** A gadget is t-Probe-Isolating-Non-Interfering (t-PINI) if, for every set P of t_1 probes on the internal values and set $A \subset [\![0, d[\![$ with $t_1 + |A| \leq t$, there exists a set $B \subset [\![0, d[\![$ with $|B| \leq t_1$ such that the probes in P and the output shares whose index (i.e., the position of the share in the sharing) belongs to A can be simulated by using the input shares whose share index belongs to $A \cup B$.

Following [CGZ20], we say in the following that a gadget with d shares is PINI if it is $(d-1)$-PINI, since this implies that it is t-PINI for any t. The $+^{\mathtt{A}}$ and $\oplus^{\mathtt{B}}$ are *share-isolating*: all the computation on the input and output shares with a given share index is isolated from computations for any other share index. All share-isolating gadgets are PINI [CS20], but the ISW multiplication is not PINI. There however exists a PINI `SecAnd` gadget [CS20, Algorithm 2] with a cost similar to the ISW multiplication: same amount of randomness and roughly double the number of arithmetic operations. Finally, PINI gadgets are trivially composable: the composition of t-PINI gadgets is t-PINI [CS20], which enables composition without the use of refresh gadgets.

2.3 Modular addition in Boolean masking

We first consider the addition modulo 2^k of two k-bit Boolean shared operands, and denote this gadget as `SecAdd`. It can be implemented by taking the Boolean circuit of a k-bit binary adder, rewriting it to only use AND and XOR gates, and finally implementing this circuit with 1-bit `SecAnd` and $\oplus^{\mathtt{B}}$ gadgets. The 1-bit inputs of this circuit are obtained by selecting single bit sharings in the k-bit input sharings. Using a chain of full-adders, this technique yields a complexity of $\mathcal{O}\left(kd^2\right)$ operations (each on single-bit words).

This technique has been refined in [CGTV15] by using the Kogge-Stone (KS) adder for the 2-shares case. This circuit allows to perform some Boolean operations in parallel, that is, with multiple-bit `SecAnd` and $\oplus^{\mathtt{B}}$ gadgets. This gives a complexity of $\mathcal{O}\left(\log(k)d^2\right)$ operations (on up-to k-bit words). Barthe et al. then generalized the gadget to arbitrary masking order and, by inserting refresh gadgets, proved it $(d-1)$-NI (Algorithm 9 of [BBE+18]).

Next, we consider the `SecAddModp` gadget which performs the addition modulo p. The construction of Algorithm 2 (from [BBE+18]) is based on the `SecAdd` gadget. Namely, it first computes the sum s of the inputs x and y on $k+1$ (to avoid overflow and thus modulo 2^k reduction), then adds $2^k - p$ to obtain s'. The most significant bit of s' indicates whether $x + y \geq p$. Based on this bit, either s or s' is selected as the output, using a MUX implemented with `SecAnd` and $\oplus^{\mathtt{B}}$ gadgets. Finally, the most significant bit is dropped to get the result on k bits. The complexity is still $\mathcal{O}\left(\log(k)d^2\right)$ operations on up-to k-bit words.

Algorithm 1 $\texttt{BitCopyMask}_k^d$ (share-isolating)

Input: Boolean sharing $\boldsymbol{x}^{B,1}$ and integer $p < 2^k$.
Output: Boolean sharing $\boldsymbol{y}^{B,k}$.

1: **for** $i = 0, \ldots, k-1$ **do**
2: **if** $\lfloor (p \mod 2^i)/2^i \rfloor = 1$ **then** ▷ Test if i-th bit of p is set.
3: $\boldsymbol{y}^{B,k}[i] \leftarrow \boldsymbol{x}^{B,1}$
4: **else**
5: $\boldsymbol{y}^{B,k}[i] \leftarrow (0, \ldots, 0)$

Algorithm 2 $\texttt{SecAddModp}_k^d$ from [BBE+18] (NI)

Input: Boolean sharings $\boldsymbol{x}^{B,k}$ and $\boldsymbol{y}^{B,k}$, integer p such that $p < 2^k$ and $x, y \in [\![0, p[\![$.
Output: Boolean sharing $\boldsymbol{z}^{B,k}$ such that $z = x + y \mod p$.

1: $\boldsymbol{p}^{B,k+1} \leftarrow (2^k - p, 0, \ldots, 0)$
2: $\boldsymbol{s}^{B,k+1} \leftarrow \texttt{SecAdd}_{k+1}^d(\boldsymbol{x}^{B,k}, \boldsymbol{y}^{B,k})$ ▷ Algorithm 9 of [BBE+18].
3: $\boldsymbol{s'}^{B,k+1} \leftarrow \texttt{SecAdd}_{k+1}^d(\boldsymbol{s}^{B,k+1}, \boldsymbol{p}^{B,k+1})$
4: $\boldsymbol{b}^{B,1} \leftarrow \boldsymbol{s'}^{B,k+1}[k]$
5: $\boldsymbol{c}^{B,1} \leftarrow \texttt{RefreshSNI}_1^d\left(\boldsymbol{b}^{B,1}\right)$
6: $\boldsymbol{c'}^{B,1} \leftarrow \neg\texttt{RefreshSNI}_1^d\left(\boldsymbol{b}^{B,1}\right)$
7: $\boldsymbol{c}^{B,k} \leftarrow \texttt{BitCopyMask}_k^d(\boldsymbol{c}^{B,1}, 2^k - 1)$ ▷ Copy input sharing where bitmask $(2^k - 1)$ is set.
8: $\boldsymbol{c'}^{B,k} \leftarrow \texttt{BitCopyMask}_k^d\left(\boldsymbol{c'}^{B,1}, 2^k - 1\right)$
9: $\boldsymbol{z}^{B,k} \leftarrow \texttt{SecAnd}_k^d(\boldsymbol{s}^{B,k+1}[[\![0, k[\![], \boldsymbol{c}^{B,k}) \oplus^{\mathsf{B}} \texttt{SecAnd}_k^d\left(\boldsymbol{s'}^{B,k+1}[[\![0, k[\![], \boldsymbol{c'}^{B,1}\right)$ ▷ MUX

2.4 Arithmetic-to-Boolean masking conversion

Coron et al. [CGV14] introduced a simple way to convert from arithmetic to Boolean masking (`SecA2BModp`). This technique first masks each arithmetic share into a d-shares Boolean sharing and then computes the addition modulo p of these Boolean shared values. This removes the arithmetic masking, its result is therefore a Boolean masking of the original value.

This can be optimized by remarking that the addition of d' arithmetic shares can be securely masked using d'-shares Boolean masking instead of d. Therefore, the optimized technique (Algorithm 3 from [SPOG19]) proceeds recursively: it splits the arithmetic sharing into two groups of $d/2$ arithmetic shares, converts each group separately into a $d/2$-shares Boolean sharing, re-masks each Boolean sharing to d shares, computes their sum. This algorithm has a complexity of $\mathcal{O}\left(\log(k)d^2\right)$ on up-to k-bit words. As an alternative, a table-based `SecA2BModp` implementation with the same complexity was recently introduced in [CGMZ21a].

Algorithm 3 $\texttt{SecA2BModp}_k^d$ from [SPOG19] (SNI)

Input: d shares arithmetic sharing $\boldsymbol{x}^{A_p}$, integer p such that $p < 2^k$ and $x \in [\![0, p[\![$.
Output: d shares Boolean sharing $\boldsymbol{z}^{B,k}$ such that $z = x$.

1: **if** $d = 1$ **then**
2: $\quad \boldsymbol{z}^{B,k} \leftarrow \boldsymbol{x}^{A_p}$
3: **else**
4: $\quad \boldsymbol{y}^{B,k} \leftarrow \texttt{SecA2BModp}_k^{\lfloor d/2 \rfloor}\left(\boldsymbol{x}^{A_k}_{[\![0,\lfloor d/2\rfloor[\![}\right)$
5: $\quad \boldsymbol{y'}^{B,k} \leftarrow \texttt{SecA2BModp}_k^{d-\lfloor d/2 \rfloor}\left(\boldsymbol{x}^{A_k}_{[\![\lfloor d/2\rfloor,d[\![}\right)$
6: $\quad \boldsymbol{y}^{B,k} \leftarrow \texttt{RefreshSNI}_k^d\left(\left(\boldsymbol{y}_0^{B,k}, \boldsymbol{y}_1^{B,k}, \ldots, \boldsymbol{y}_{\lfloor d/2\rfloor-1}^{B,k}, 0, \ldots, 0\right)\right)$ ▷ Expand to d shares.
7: $\quad \boldsymbol{y'}^{B,k} \leftarrow \texttt{RefreshSNI}_k^d\left(\left(0, \ldots, 0, \boldsymbol{y}_{\lfloor d/2\rfloor}^{B,k}, \ldots, \boldsymbol{y}_{d-1}^{B,k}\right)\right)$ ▷ Expand to d shares.
8: $\quad \boldsymbol{z}^{B,k} \leftarrow \texttt{SecAddModp}_k^d\left(\boldsymbol{y}^{B,k}, \boldsymbol{y'}^{B,k}\right)$

2.5 Boolean-to-arithmetic masking conversion

Similarly to arithmetic-to-Boolean conversions, there are multiple efficient techniques for Boolean-to-arithmetic conversion. First, one may generate $d-1$ random arithmetic shares, generate a d-share Boolean masking of the opposite of their sum (using `SecA2BModp`), add this to the input sharing (with `SecAddModp`), and finally unmask (that is, XOR the shares together) the result to get the last arithmetic share. This idea, originally introduced in [CGTV15], has been adapted to the modulo p setting in [BBE+18] (see Algorithm 4). This gadget is $(d-1)$-SNI.[4]

Second, Schneider et al. [SPOG19] introduced a conversion based on the observation that if $x, y \in [\![0,1]\!]$, $x \oplus y = x + y - 2xy$. The gist of the conversion algorithm is to start from a 1-bit Boolean sharing $\boldsymbol{x}^{B,1}$, then arithmetically mask each share, and finally use the previous equation to compute the XOR of these arithmetic sharings. This single-bit conversion algorithm may then be applied to each of a multi-bit input, and the results can be recombined sharewise (with sums and multiplications by 2). Thanks to various optimizations of the algorithm [SPOG19], the complexity of this technique is $\mathcal{O}\left(kd^2\right)$ operations on k-bit words.

[4]The proof that `SecB2AModp` is SNI is not given explicitly, in [BBE+18], but it can be deduced from the proof of Lemma 5, if `SecA2BModp` is SNI.

Algorithm 4 $\texttt{SecB2AModp}_k^d$ from [BBE+18] (SNI)

Input: d shares Boolean sharing $\boldsymbol{x}^{B,k}$, integer p such that $p < 2^k$ and $x \in [\![0, p[\![$.
Output: d shares arithmetic sharing $\boldsymbol{z}^{A_p}$ such that $z = x$.

1: **for** $i = 0$ to $d - 2$ **do**
2: $\quad \boldsymbol{z}_i^{A_k} \xleftarrow{\$} \mathbb{Z}_p$
3: $\quad \boldsymbol{z'}_i^{A_k} \leftarrow p - \boldsymbol{z}_i^{A_k}$
4: $\boldsymbol{z'}_{d-1}^{A_k} \leftarrow 0$
5: $\boldsymbol{a}^{B,k} \leftarrow \texttt{SecA2BModp}_k^d \left(\boldsymbol{z'}^{A_p}\right)$
6: $\boldsymbol{b}^{B,k} \leftarrow \texttt{SecAddModp}_k^d \left(\boldsymbol{a}^{B,k}, \boldsymbol{x}^{B,k}\right)$
7: $\boldsymbol{z}_{d-1}^{A_k} \leftarrow \texttt{UnMask}_k^{d-1} \left(\texttt{FullRefresh}_k^{d-1} \left(\boldsymbol{b}^{B,k}\right)\right)$

Finally, Coron et al. [CGMZ21a] introduced recently another conversion algorithm. This algorithm also performs k single-bit conversions, but the single-bit conversion is a table-based gadget.

2.6 Bitslicing

When an algorithm computes a Boolean circuit (i.e., it operates on single-bit variables), it can be bitsliced. That is, it can be implemented to perform w evaluations parallel on a processor with w-bit words (e.g., $w = 32$) by using bitwise operations. While the bitslicing technique can bring a large performance increase, it has some drawbacks. Since it does work only on Boolean circuits, bitslicing a computation requires writing it as a Boolean circuit. Moreover, it requires the availability of a significant amount of parallelism in the operations to perform, otherwise it loses it performance benefits. Finally, bitslicing requires representation changes: the data processed is often used in a *canonical* form in which all the bits for one circuit evaluations are stored contiguously in memory words (we model the memory as a sequence of w-bit words). However, bitslicing works with a *bitslice* representation: each parallel evaluation contributes a single-bit to each word.

Let us take the example of computations on a k-bit variable a_i: the Boolean circuit takes k input bits (the bits of a_i), and outputs the k bits of $b_i = f(a_i)$. Moreover, let us assume that there are N computations to perform: $i = 0, \dots, N-1$ (and, for simplicity, we assume that N is a multiple of w). Let $a_i[k-1] \cdots a_i[0]$ be the bit representation of a_i, the canonical[5] representation would be (assuming that $w \geq k$)

$$\begin{pmatrix} a_0[0] & \cdots & a_0[k-1] & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ a_{N-1}[0] & \cdots & a_{N-1}[k-1] & 0 & \cdots & 0 \end{pmatrix}$$

where each row contains w bits and represents a word of the memory. In the same case, and using the same notation, a bitslice representation would be

$$\begin{pmatrix} a_0[0] & \cdots & a_{w-1}[0] \\ \vdots & & \vdots \\ a_{N/w-k}[0] & \cdots & a_{N/w-1}[0] \\ a_0[1] & \cdots & a_{w-1}[1] \\ \vdots & & \vdots \\ a_{N/w-k}[k-1] & \cdots & a_{N/w-1}[k-1]) \end{pmatrix} .$$

[5]The order of the words in memory usually does not matter much, compared to the way the bits are grouped into words.

Therefore, the inputs bits have to be mapped from canonical to bitslice with `CToBs` before the bitslice computation, and the result bits have to be mapped again to canonical with `BsToC` after it. Since the changes of representation can be expensive it is important to implement these efficiently (and to minimize their number, by avoiding unnecessary `CToBs` / `BsToC`). A naive implementation of representation changes requires a number of CPU instructions proportional to the number of bits manipulated.

However, in some cases, this can be made more efficiently, such as when $k = w$. Then, the change of representation can be grouped in N/w parts, each handling the words $a_{wj}, \ldots, a_{w(j+1)-1}$ for $0 \leq j < N/w$, and both `CToBs` and `BsToC` can be represented as the transposition of the following square matrix

$$\begin{pmatrix} a_{wj}[0] & \cdots & a_{wj}[w-1] \\ \vdots & & \vdots \\ a_{w(j+1)-1}[0] & \cdots & a_{w(j+1)-1}[w-1] \end{pmatrix}$$

where each row represents a memory word. This transposition can be computed more efficiently than the naive algorithm: $\mathcal{O}(w \log w)$ instead of $\mathcal{O}(w^2)$ [Jr.13].[6]

Furthermore, the technique can be adapted to $k < w$. For example, let us assume that $w/4 < k \leq w/2$ (this matches our implementation for `Kyber768`: $k = 12$ and we work on a $w = 32$-bit processor). In that case, a_{2i} and a_{2i+1} are typically stored in a single processor word (to save memory), hence the canonical form can be represented as

$$\begin{pmatrix} a_{2wj}[0] & \cdots & a_{2wj}[k-1] & 0 \cdots 0 & a_{2wj+1}[0] & \cdots & a_{2wj+1}[k-1] & 0 \cdots 0 \\ a_{2wj+2}[0] & \cdots & a_{2wj+2}[k-1] & 0 \cdots 0 & a_{2wj+3}[0] & \cdots & a_{2wj+3}[k-1] & 0 \cdots 0 \\ \vdots & & \vdots & \vdots \quad \vdots & \vdots & & \vdots & \vdots \quad \vdots \\ a_{2wj+2(w-1)}[0] & \cdots & a_{2wj+2(w-1)}[k-1] & 0 \cdots 0 & a_{2wj+2(w-1)+1}[0] & \cdots & a_{2wj+2(w-1)+1}[k-1] & 0 \cdots 0 \end{pmatrix}$$

where both chunks of "0" columns are equally large. This matrix can then be transposed, and the resulting "0" lines can be removed (by copying only the useful rows), giving the bitslice representation:

$$\begin{pmatrix} a_{2wj}[0] & a_{2wj+2}[0] & \cdots & a_{2wj+2(w-1)}[0] \\ \vdots & \vdots & \vdots & \vdots \\ a_{2wj}[k-1] & a_{2wj+2}[k-1] & \cdots & a_{2wj+2(w-1)}[k-1] \\ a_{2wj+1}[0] & a_{2wj+3}[0] & \cdots & a_{2wj+2(w-1)+1}[0] \\ \vdots & \vdots & \vdots & \vdots \\ a_{2wj+1}[k-1] & a_{2wj+3}[k-1] & \cdots & a_{2wj+2(w-1)+1}[k-1] \end{pmatrix}$$

Regarding security, the use of the `CToBs` and `BsToC` algorithms has no impact on the t-probing security since they only copy bits and therefore to not give new choices of probes to the adversary. Practically for masking, the changes of representation can be implemented as a masked `CToBs` or `BsToC` share-isolating gadget.

We next introduce our new gadgets, which are all (except `SecB2AModp`) Boolean circuits, hence are trivially implemented using the bitslice technique (fully working in bitslice representation, with no `CToBs` or `BsToC` needed). We describe them as Boolean circuits and give their complexity in Boolean operations. This complexity should be divided by w to obtain the complexity in CPU instructions for bitslice implementations.

3 New gadgets

As we already mentioned in the introduction, our starting point is the observation that high-level cryptographic algorithms such as Kyber have large data parallelism, hence

[6] While this algorithm is well-known, and used in at least one bitsliced cryptographic implementation (https://github.com/Ko-/aes-armcortexm/blob/public/aes128ctrbs/aes_128_ctr_bs.s, from [SS16]), we have not found any discussion of its use in the bitslicing literature.

they may benefit from bitsliced implementations for the Boolean sharings (while staying non-bitsliced for the arithmetic sharings). We therefore introduce algorithms that represent Boolean circuits, and which are therefore well-suited to bitslicing. As main elementary gadgets, we use $\oplus^{\mathsf{B}}$ and PINI `SecAnd` from [CS20], where the `SecAnd` is more expensive than $\oplus^{\mathsf{B}}$ ($\mathcal{O}\left(d^2\right)$ vs. $\mathcal{O}\left(d\right)$).

3.1 `SecAdd`: Bitslice Boolean masked addition modulo 2^k

Our first algorithm is a new `SecAdd` implementation (Algorithm 6). Thanks to bitslicing, we do not have any structure constraint and simply aim to minimize the number of `SecAnd`. Therefore, we use a simple chain of full-adders, where the addition of x, y and z computes $a := x \oplus y$, then outputs $(a \oplus z, x \oplus a \cdot (x \oplus z))$. This requires only one `SecAnd` per full-adder, hence $k-1$ in total (since the carry-out does not have to be computed for the addition of the most significant bits), which is the minimum achievable (we prove this in Appendix A). The total complexity of Algorithm 6 is $\mathcal{O}\left(kd^2\right)$ bit operations. We finally prove the security of this gadget.

Proposition 1. *Algorithm 6 and Algorithm 5 are PINI.*

Proof. These two gadgets are the composition of PINI gadgets, therefore they are PINI. □

Algorithm 5 $\mathtt{SecFullAdder}^d$ New (PINI)

Input: Boolean sharings $\boldsymbol{x}^{B,1}$, $\boldsymbol{y}^{B,1}$ and $\boldsymbol{z}^{B,1}$.
Output: Boolean sharing $\boldsymbol{w}^{B,2}$ such that $w = x + y + z$.

1: $\boldsymbol{a}^{B,1} \leftarrow \boldsymbol{x}^{B,1} \oplus^{\mathsf{B}} \boldsymbol{y}^{B,1}$
2: $\boldsymbol{w}^{B,2}[0] \leftarrow \boldsymbol{z}^{B,1} \oplus^{\mathsf{B}} \boldsymbol{a}^{B,1}$
3: $\boldsymbol{w}^{B,2}[1] \leftarrow \boldsymbol{x}^{B,1} \oplus^{\mathsf{B}} \mathtt{SecAnd}_1^d\left(\boldsymbol{a}^{B,1}, \boldsymbol{x}^{B,1} \oplus^{\mathsf{B}} \boldsymbol{z}^{B,1}\right)$ ▷ PINI `SecAnd`

Algorithm 6 $\mathtt{SecAdd}_k^d$ New (PINI)

Input: Boolean sharings $\boldsymbol{x}^{B,k}$ and $\boldsymbol{y}^{B,k}$, such that $x, y \in [\![0, 2^k[\![$.
Output: Boolean sharing $\boldsymbol{z}^{B,k}$ such that $z = x + y \mod 2^k$.

1: $\boldsymbol{c}^{B,1} \leftarrow (0, 0, \ldots, 0)$
2: **for** $i = 0$ to $k-2$ **do**
3: $\quad \boldsymbol{t}^{B,2} \leftarrow \mathtt{SecFullAdder}^d\left(\boldsymbol{x}^{B,k}[i], \boldsymbol{y}^{B,k}[i], \boldsymbol{c}^{B,1}\right)$ ▷ Algorithm 5
4: $\quad \left(\boldsymbol{z}^{B,k}[i], \boldsymbol{c}^{B,1}\right) \leftarrow \left(\boldsymbol{t}^{B,2}[0], \boldsymbol{t}^{B,2}[1]\right)$
5: $\boldsymbol{z}^{B,k}[k-1] \leftarrow \boldsymbol{x}^{B,k}[k-1] \oplus^{\mathsf{B}} \boldsymbol{y}^{B,k}[k-1] \oplus^{\mathsf{B}} \boldsymbol{c}^{B,1}$

3.2 `SecAddModp`: Bitslice Boolean masked addition modulo p

Next, we consider addition modulo p. A simple approach is to adapt Algorithm 2 to use Algorithm 6 as `SecAdd`. On top of this adaptation, we remark that the MUX in Algorithm 2 costs $2k$ 1-bit `SecAnd` gadgets, and that we can replace it with the computation of $s' + p \cdot b \mod 2^k$, which costs one $\mathtt{SecAdd}_k^d$ (i.e., $k-1$ single-bit `SecAnd`). This replacement is correct: if $b = 0$, the result is s', and if $b = 1$ the result is $s' + p \mod 2^k = s$. Overall, our new addition modulo p requires two $k+1$-bit adders and one k-bit adder, totaling to $3k-1$ 1-bit PINI `SecAnd`, hence $\mathcal{O}\left(kd^2\right)$ bit operations and randomness.

Proposition 2. *Algorithm 7 is PINI.*

Proof. All the sub-gadgets are PINI (`BitCopyMask` only replicates a sharing, hence it is share-isolating, which implies that it is PINI). □

Algorithm 7 $\texttt{SecAddModp}_k^d$ New (PINI)

Input: Boolean sharings $\boldsymbol{x}^{B,k}$ and $\boldsymbol{y}^{B,k}$, integer p such that $p < 2^k$ and $x, y \in [\![0, p[\![$.
Output: Boolean sharing $\boldsymbol{z}^{B,k}$ such that $z = x + y \mod p$.

1: $\boldsymbol{p}^{B,k+1} \leftarrow \left(2^{k+1} - p, 0, \ldots, 0\right)$
2: $\boldsymbol{s}^{B,k+1} \leftarrow \texttt{SecAdd}_{k+1}^d\left(\boldsymbol{x}^{B,k}, \boldsymbol{y}^{B,k}\right)$ ▷ Use Algorithm 6.
3: $\boldsymbol{s'}^{B,k+1} \leftarrow \texttt{SecAdd}_{k+1}^d\left(\boldsymbol{s}^{B,k+1}, \boldsymbol{p}^{B,k+1}\right)$ ▷ Use Algorithm 6.
4: $\boldsymbol{b}^{B,1} \leftarrow \boldsymbol{s'}^{B,k+1}[k]$
5: $\boldsymbol{a}^{B,k} \leftarrow \texttt{BitCopyMask}_k^d\left(\boldsymbol{b}^{B,1}, p\right)$ ▷ Copy sharing b where bitmask p is set (computes $a = p \cdot b$).
6: $\boldsymbol{z}^{B,k} \leftarrow \texttt{SecAdd}_k^d\left(\boldsymbol{a}^{B,k}, \boldsymbol{s'}^{B,k+1}\right)$ ▷ Use Algorithm 6.

3.3 SecA2B: Bitslice arithmetic-to-Boolean conversion modulo 2^k

For arithmetic modulo 2^k to Boolean conversion (`SecA2B`), we take inspiration from the conversion algorithm of [SPOG19] (Algorithm 3). Namely, we also use a recursive structure where two halves of the arithmetic sharing are first converted to Boolean, then the two resulting sharing are added together. We use our new `SecAdd` (Algorithm 6) for this purpose, which, thanks to PINI composition, allows us to remove the refresh gadget, giving Algorithm 8 whose complexity is $\mathcal{O}\left(kd^2\right)$ random bits and single-bit operations.

Algorithm 8 $\texttt{SecA2B}_k^d$ New (PINI)

Input: d shares arithmetic sharing $\boldsymbol{x}^{A_{2^k}}$, such that $x \in [\![0, 2^k[\![$.
Output: d shares Boolean sharing $\boldsymbol{z}^{B,k}$ such that $z = x$.

1: **if** $d = 1$ **then**
2: $\quad \boldsymbol{z}^{B,k} \leftarrow \boldsymbol{x}^{A_{2^k}}$
3: **else**
4: $\quad \boldsymbol{y}^{B,k} \leftarrow \texttt{SecA2B}_k^{\lfloor d/2 \rfloor}\left(\boldsymbol{x}^{A_{2^k}}[[\![0, \lfloor d/2 \rfloor[\![]\right)$ ▷ $\lfloor d/2 \rfloor$ sharing.
5: $\quad \boldsymbol{y'}^{B,k} \leftarrow \texttt{SecA2B}_k^{d-\lfloor d/2 \rfloor}\left(\boldsymbol{x}^{A_{2^k}}[[\![\lfloor d/2 \rfloor, d[\![]\right)$ ▷ $d - \lfloor d/2 \rfloor$ sharing.
6: $\quad \boldsymbol{s}^{B,k} \leftarrow \left(\boldsymbol{y}_0^{B,k}, \boldsymbol{y}_1^{B,k}, \ldots, \boldsymbol{y}_{\lfloor d/2 \rfloor - 1}^{B,k}, 0, \ldots, 0\right)$ ▷ Expand to d shares.
7: $\quad \boldsymbol{s'}^{B,k} \leftarrow \left(0, \ldots, 0, \boldsymbol{y'}_{\lfloor d/2 \rfloor}^{B,k}, \ldots, \boldsymbol{y'}_{d-1}^{B,k}, \right)$ ▷ Expand to d shares.
8: $\quad \boldsymbol{z}^{B,k} \leftarrow \texttt{SecAdd}_k^d\left(\boldsymbol{s}^{B,k}, \boldsymbol{s'}^{B,k}\right)$ ▷ Use Algorithm 6.

To prove that Algorithm 8 is PINI, we will use the PINI composition theorem from [CS20], and introduce a new technique to deal with the composition of PINI gadget with various numbers of shares. The core idea is to embed gadgets that use a lower number of shares into "virtual gadgets" that use more shares, with a mapping from the share indexes of the embedded gadgets to the indexes of the embedding gadgets. The embedding gadget discards the input shares that are not used, and sets to 0 the output shares that are not generated by the embedded gadgets, as illustrated in Figure 1.

Definition 5 (Gadget embedding)**.** Let G be a d'-share gadget with n (resp. n') input (resp. output) sharings, and let $m \in [\![0, d[\![^{d'}$ (with $d \geq d'$) have unique components

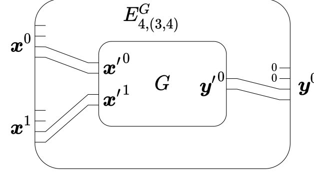


Figure 1: Example of 2-share to 4-share gadget embedding.

Algorithm 9 $E^G_{d,m}$: embedding of the d'-shares gadget G to d shares with mapping m with $d' \leq d$.

Input: n d-shares input sharings $\boldsymbol{x}^0, \ldots \boldsymbol{x}^{n-1}$;
Output: n' d-shares output sharings $\boldsymbol{y}^0, \ldots, \boldsymbol{y}^{n'-1}$

1: **for** $j = 0, \ldots, n-1$ **do**
2: **for** $i = 0, \ldots, d'-1$ **do**
3: $\boldsymbol{x'}^j_i \leftarrow \boldsymbol{x}^j_{m_i}$
4: $\left(\boldsymbol{y'}^0, \ldots, \boldsymbol{y'}^{n'}\right) \leftarrow G\left(\boldsymbol{x'}^0, \ldots, \boldsymbol{x'}^n\right)$
5: **for** $j = 0, \ldots, n'-1$ **do**
6: **for** $i = 0, \ldots, d-1$ **do**
7: $\boldsymbol{y}^j_i \leftarrow 0$ ▷ Initialize all shares to 0.
8: **for** $i = 0, \ldots, d'-1$ **do**
9: $\boldsymbol{y}^j_{m_i} \leftarrow \boldsymbol{y'}^j_i$ ▷ Override some output shares with outputs of G.

($m_i \neq m_j$ for all i, j). The d-share embedding of G with mapping m is the d-share gadget denoted $E^G_{d,m}$ described in Algorithm 9.

Lemma 1 (PINI embedding)**.** *If G is a PINI gadget, its embedding $E^G_{d,m}$ is PINI for any d and m.*

Proof. We describe the $(d-1)$-PINI simulator for $E^G_{d,m}$ that has to simulate a set of internal probes P and the output shares with index in B. First, P can be partitioned in a set P_G of probes in G and a set P_i of probes on the input shares. Next, B is partitioned as B_0 (the elements of B that appear in m), and B_1 (the remaining elements).

Let $B'_0 = \{i \in [\![0, d'[\![\text{ s.t. } m_i \in B_0\}$, we have $|B'_0| = |B_0|$. We use the PINI simulator of G to simulate the probes P_G and its output shares with index in B'_0 (which are the outputs of $E^G_{d,m}$ with index in B_0). This simulator requires knowledge of its input shares with index in $A' \cup B'$, for some A'_0 such that $|A'_0| \leq |P_G|$. Let us define $A_0 = \{m_i \text{ for all } i \in A'_0\}$, such that knowing the input shares of $E^G_{d,m}$ with index in $A_0 \cup B_0$ allows sending the inputs required to the simulator of G, that simulates the probes P_G and the output shares with index in B_0.

Finally, the probes in P_i can be simulated with the input shares with index in A_1, for some A_1 such that $|A_1| \leq |P_i|$, and all the output shares with index in B_1 can be trivially simulated (their value is always 0). As a result, all the required values can be simulated with the input shares of $E^G_{d,m}$ with index in $(A_0 \cup A_1) \cup B$, and $|A_0 \cup A_1| \leq |P|$. □

Proposition 3. *Algorithm 8 is PINI.*

Proof. In the case $d = 1$, this is trivial. In the other cases, we decompose the gadget in three parts, which are then embedded: wires carrying the constant "0" value are added such that all sharings have d shares (this has no impact on the security). This gives a decomposition of the gadget into three sub-gadgets: $E^{\texttt{SecA2B}^{\lfloor d/2 \rfloor}_k}_{d,(0,\ldots,\lfloor d/2 \rfloor -1)}$ (which computes

$\boldsymbol{s}^{B,k}$ from $\boldsymbol{x}^{A_{2^k}}$), $E_{d,(\lfloor d/2\rfloor,\ldots,d-1)}^{\mathtt{SecA2B}_k^{d-\lfloor d/2\rfloor}}$ (which computes $\boldsymbol{s'}^{B,k}$ from $\boldsymbol{x}^{A_{2^k}}$) and $\mathtt{SecAdd}_k^d$ (which computes $\boldsymbol{z}^{B,k}$ from $\boldsymbol{s}^{B,k}$ and $\boldsymbol{s'}^{B,k}$). Since $\mathtt{SecA2B}_k^{\lfloor d/2\rfloor}$ and $\mathtt{SecA2B}_k^{d-\lfloor d/2\rfloor}$ are PINI (by induction on d), their embeddings are PINI (by Lemma 1). Furthermore, $\mathtt{SecAdd}_k^d$ is PINI (Proposition 1). Therefore, Algorithm 8 is a composition of PINI gadgets. □

3.4 SecA2BModp: Bitslice arithmetic-to-Boolean conversion modulo p

A simple way to implement arithmetic modulo p to Boolean masking conversion is to adapt Algorithm 8 (SecA2B) to use addition modulo p (SecAddModp, Algorithm 7) instead of addition modulo 2^k (SecAdd, Algorithm 6).[7] On top of this adaptation, we can perform a small optimization inspired by the first-order A2B conversion from [FBR+22]: the first operation of our addition modulo p (Algorithm 7) is to subtract p from one of the two operands which can be done before double the number of shares in the A2B algorithm. This has no impact on the final result, but the cost of this subtraction is divided by about 4 (since this operation is in $\mathcal{O}\left(kd^2\right)$).

These changes do not impact the asymptotic complexity of the algorithm, which is still $\mathcal{O}\left(kd^2\right)$ random bits and single-bit operations.

Algorithm 10 $\mathtt{SecA2BModp}_k^d$ New (PINI)

Input: d shares arithmetic sharing $\boldsymbol{x}^{A_p}$, integer p such that $p < 2^k$ and $x \in [\![0,p[\![$.
Output: d shares Boolean sharing $\boldsymbol{z}^{B,k}$ such that $z = x$.

1: **if** $d = 1$ **then**
2: $\quad\boldsymbol{z}^{B,k} \leftarrow \boldsymbol{x}^{A_p}$
3: **else**
4: $\quad\boldsymbol{y}^{B,k} \leftarrow \mathtt{SecA2BModp}_k^{\lfloor d/2\rfloor}\left(\boldsymbol{x}^{A_p}[[\![0,\lfloor d/2\rfloor[\![]\right)$ ▷ $\lfloor d/2\rfloor$ sharing.
5: $\quad\boldsymbol{y'}^{B,k} \leftarrow \mathtt{SecA2BModp}_k^{d-\lfloor d/2\rfloor}\left(\boldsymbol{x}^{A_p}[[\![\lfloor d/2\rfloor,d[\![]\right)$ ▷ $d-\lfloor d/2\rfloor$ sharing.
6: $\quad\boldsymbol{p}^{B,k+1} \leftarrow \left(2^k - p, 0, \ldots, 0\right)$ ▷ $\lfloor d/2\rfloor$ sharing.
7: $\quad\boldsymbol{s}^{B,k+1} \leftarrow \mathtt{SecAdd}_{k+1}^{\lfloor d/2\rfloor}\left(\boldsymbol{p}^{B,k+1}, \boldsymbol{y}^{B,k}\right)$ ▷ Use Algorithm 6.
8: $\quad\boldsymbol{s}^{B,k+1} \leftarrow \left(\boldsymbol{y}_0^{B,k+1}, \boldsymbol{y}_1^{B,k+1}, \ldots, \boldsymbol{y}_{\lfloor d/2\rfloor-1}^{B,k+1}, 0, \ldots, 0\right)$ ▷ Expand to d shares.
9: $\quad\boldsymbol{s'}^{B,k} \leftarrow \left(0, \ldots, 0, \boldsymbol{y'}_{\lfloor d/2\rfloor}^{B,k}, \ldots, \boldsymbol{y'}_{d-1}^{B,k},\right)$ ▷ Expand to d shares.
10: $\quad\boldsymbol{u}^{B,k+1} \leftarrow \mathtt{SecAdd}_{k+1}^d\left(\boldsymbol{s}^{B,k+1}, \boldsymbol{s'}^{B,k}\right)$ ▷ Use Algorithm 6.
11: $\quad\boldsymbol{b}^{B,1} \leftarrow \boldsymbol{u}^{B,k+1}[k]$
12: $\quad\boldsymbol{a}^{B,k} \leftarrow \mathtt{BitCopyMask}_k^d\left(\boldsymbol{b}^{B,1}, p\right)$ ▷ Copy sharing b where bitmask p is set ($a := p \cdot b$).
13: $\quad\boldsymbol{z}^{B,k} \leftarrow \mathtt{SecAdd}_k^d\left(\boldsymbol{a}^{B,k}, \boldsymbol{u}^{B,k+1}\right)$ ▷ Use Algorithm 6.

Proposition 4. *Algorithm 10 is PINI.*

Proof. The proof is almost identical to the proof of Algorithm 10. The case $d = 1$ is trivial, and in the other cases, we exhibit a decomposition into PINI sub-gadgets. We first consider the d-share embedding of the $\lfloor d/2\rfloor$-share composite gadget whose input is $\boldsymbol{x}^{A_p}[[\![0,\lfloor d/2\rfloor[\![]$ and whose output is $\boldsymbol{s}^{B,k+1}$. This gadget is the composition of two PINI gadgets ($\mathtt{SecA2BModp}_k^{\lfloor d/2\rfloor}$ and $\mathtt{SecAdd}_{k+1}^{\lfloor d/2\rfloor}$), hence it is PINI, and the embedding is PINI. Next, the d-share embedding of $\mathtt{SecA2BModp}_k^{d-\lfloor d/2\rfloor}$ is PINI, as well as the other d-share sub-gadgets (SecAdd, BitCopyMask). □

[7] Another solution would be to use the compression algorithm (HOCompress) from [CGMZ21b] which it has a worse asymptotic complexity of $\mathcal{O}\left(kd^2\log(d)\right)$, but which might be an interesting alternative if we care only about small enough d.

3.5 SecB2AModp: Bitslice Boolean-to-arithmetic conversion modulo p

We now adapt in Algorithm 11 the SecB2AModp from [BBE+18] (Algorithm 4) to use our new SecA2BModp and SecAddModp algorithms. Furthermore, we replace the refresh gadget to reduce its cost (from $\mathcal{O}(d^2)$ to $\mathcal{O}(d \log d)$). The new refresh gadget is the input-output separative (IOS) refresh gadget from [GPRV21]. We generalize this gadget to any value of d in Algorithm 18 (Appendix B), since only the power of 2 cases were handled in [GPRV21].

Algorithm 11 combines arithmetic operations (lines 1 to 4) which are best implemented using a canonical representation (see Subsection 2.6) and bit-level operations (starting at line 5), which are best implemented bitsliced, hence with a bitslice representation. As a result, Algorithm 11 takes as input a Boolean sharing in canonical representation, applies CToBs to the sharing $\boldsymbol{z'}^{A_p}$ before its conversion to Boolean masking, and finally applies BsToC the share $\boldsymbol{z}_{d-1}^{A_p}$ to output a canonical representation of the sharing.

Algorithm 11 $\texttt{SecB2AModp}_k^d$ New (PINI)

Input: d shares Boolean sharing $\boldsymbol{x}^{B,k}$, integer p such that $p < 2^k$ and $x \in [\![0, p[\![$.
Output: d shares arithmetic sharing $\boldsymbol{z}^{A_p}$ such that $z = x$.

1: **for** $i = 0$ to $d-2$ **do**
2: $\quad \boldsymbol{z}_i^{A_p} \xleftarrow{\$} \mathbb{Z}_p$
3: $\quad \boldsymbol{z'}_i^{A_p} \leftarrow p - \boldsymbol{z}_i^{A_p}$
4: $\boldsymbol{z'}_{d-1}^{A_p} \leftarrow 0$
5: $\boldsymbol{a}^{B,k} \leftarrow \texttt{SecA2BModp}_k^d\left(\boldsymbol{z'}^{A_p}\right)$ ▷ Applies CToBs to $\boldsymbol{z'}^{A_p}$ and use Algorithm 10.
6: $\boldsymbol{b}^{B,k} \leftarrow \texttt{SecAddModp}_k^d\left(\boldsymbol{a}^{B,k}, \boldsymbol{x}^{B,k}\right)$ ▷ Use Algorithm 7.
7: $\boldsymbol{c}^{B,k} \leftarrow \texttt{RefreshIOS}_k^d\left(\boldsymbol{b}^{B,k}\right)$ ▷ Use algorithm 1 of [GPRV21], generalized in Algorithm 18.
8: $\boldsymbol{z}_{d-1}^{A_p} \leftarrow \texttt{UnMask}_k^d\left(\boldsymbol{c}^{B,k}\right)$ ▷ XOR all shares together, and applies BsToC $\boldsymbol{z}_{d-1}^{A_p}$.

Let us introduce two definitions relating to the properties of the IOS refresh gadget before proving the security of Algorithm 11.

Definition 6 (Uniformity ([GPRV21], adapted))**.** A refresh gadget G is uniform if its output is a uniformly distributed sharing of x for any fixed input sharing $\boldsymbol{x}$.[8]

Definition 7 (IOS ([GPRV21], adapted))**.** A refresh gadget G is t-IOS if it is uniform and if for every pair of sharings $(\boldsymbol{x}, \boldsymbol{y})$ that represent the same value (i.e., such that $x = y$) and for every set of probes P with $|P| \leq t$, there exists a simulator that can perfectly simulate the probes (i.e., output values with the same distribution) by knowing only $|P|$ input shares and $|P|$ output shares. A refresh gadget with d shares is said to be IOS if it is $(d-1)$-IOS.

Proposition 5. *Algorithm 11 is PINI.*

Proof. We build a PINI simulator: given a set of probes P and share indexes B. We distinguish two cases: either (i) $d-1 \in B$ or there is a probe of P in the UnMask gadget, or (ii) there is no such probe.

In case (ii), we remark that the gadgets SecA2BModp and SecAddModp are PINI, as well as RefreshIOS (it is sharewise after application of the random-zero transform of [Cor18]). The probes in these gadgets can thus be simulated by knowing at most $|P|$ shares of $\boldsymbol{x}^{B,k}$

[8] This is not the same notion as the uniformity used in threshold implementations [NRR06], where the sharing $\boldsymbol{x}$ is assumed to be uniform. Here, the distribution of the output sharing $\boldsymbol{y}$ must be independent of $\boldsymbol{x}$, conditioned on x.

and some $\boldsymbol{z}_i^{A_p}$ for $i \in [\![0, d-2]\!]$. Such $\boldsymbol{z}_i^{A_p}$, which also are the possible output shares to simulate, can be perfectly simulated since they are randomly generated by the gadget.

In case (i), we consider the $(d-1)$-PINI simulator that has to simulate the output shares with index in B and the internal probes P. Let (P_0, P_r, P_u) be a partition of P such that the probes of P_0 are in `SecA2BModp` and `SecAddModp`, the ones of P_r are in `RefreshIOS`, and the ones of P_u are in `UnMask`. We first describe the simulator, then prove that it is correct.

The PINI simulator for `SecB2AModp` first selects randomly $\boldsymbol{z}_{d-1}^{A_p}$, then it generates a uniformly random sharing $\boldsymbol{c}^{B,k}$ of $\boldsymbol{z}_{d-1}^{A_p}$, from which it can simulate any probe in P_u. Next, using the IOS simulator, it determines the set of share indexes B_r of $\boldsymbol{b}^{B,k}$ required to simulate P_r, with $|B_r| \leq |P_r|$ (some shares from $\boldsymbol{c}^{B,k}$ are also needed for this simulation, but they are already simulated). We then consider the PINI simulation of the composition of `SecA2BModp` and `SecAddModp` (since these two gadgets are PINI): the shares of $\boldsymbol{b}^{B,k}$ with index in B_r and the probes P_0 can be simulated with the shares of $\boldsymbol{x}^{B,k}$ and $\boldsymbol{z'}^{A_p}$ whose index belongs to $B_r \cup B_0$, for some B_0 such that $|B_0| \leq |P_0|$. Finally, the simulator completes the simulation by requesting the shares of $\boldsymbol{x}^{B,k}$ with index in $B_r \cup B_0$ and draws randomly all shares $\boldsymbol{z}_i^{A_p}$ with $i \in (B_r \cup B_0 \cup B) \setminus \{d-1\}$, which enables the simulation of the required $\boldsymbol{z'}_i^{A_p}$.

Let us first observe that the number of inputs required for the simulation is admissible: $|B_r \cup B_0| \leq |P|$. Further, let us denote by $B^* \subset [\![0, d-2]\!]$ the set of i such that $\boldsymbol{z}_i^{A_p}$ is used in the simulation (we exclude $\boldsymbol{z}_{d-1}^{A_p}$ for now). We remark $B^* = B_r \cup B_0 \cup (B \setminus \{d-1\})$, and therefore that $|B^*| \leq |P_r \cup P_0| + |B \setminus \{d-1\}| \leq d-2$ where the latter inequality comes from the hypothesis that either $|P_u| \geq 1$ (hence $|P_0 \cup P_r| + |B| \leq d-2$), or $d-1 \in B$ (hence $|P| + |B \setminus \{d-1\}| \leq d-2$). As a result $|[\![0, d-2]\!] \setminus B^*| \geq 1$, and, taking $i^* \in [\![0, d-2]\!] \setminus B^*$, we observe that $\boldsymbol{z}_{i^*}^{A_p}$ is never used in the simulation.

We now show that the simulation is correct: for each value that is simulated, we show that its distribution matches the true distribution, and furthermore we prove that the simulation is consistent with (i.e., the simulated joint distribution is equal to the true distribution) the simulation of the values for which we already proved the correctness. First, the simulated shares $\boldsymbol{z}_i^{A_p}$ (except $\boldsymbol{z}_{d-1}^{A_p}$) and $\boldsymbol{z'}_i^{A_p}$ follow the same distribution as in Algorithm 11. Next, since $\boldsymbol{z}_{d-1}^{A_p} = z - \sum_{i=0}^{d-2} \boldsymbol{z}_i^{A_p} \mod p$ and since one of the terms of the sum ($\boldsymbol{z}_{i^*}^{A_p}$) is not used in the simulation and is uniformly distributed, $\boldsymbol{z}_{d-1}^{A_p}$ appears to the adversary as a fresh uniform value, and its simulation is correct. We continue with the correct simulation of the probes in P_0 and the shares $\boldsymbol{b}_i^{B,k}$: it follows from the PINI simulators of `SecA2BModp` and `SecAddModp`. Since `RefreshIOS` is uniform, its output sharing $\boldsymbol{c}^{B,k}$ is a uniform sharing of $\boldsymbol{z}_{d-1}^{A_p}$ which is independent of $\boldsymbol{b}^{B,k}$. The simulation of the probes in P_r by the `RefreshIOS` simulator ensures that the simulation of these probes and of $\boldsymbol{c}^{B,k}$ are correct. Finally, the simulation of the probes P_u is trivially correct. □

We finally remark that the conversion modulo 2^k $\mathtt{SecB2A}_k^d$ can be implemented by following Algorithm 11, using the new `SecA2B` and `SecAdd` instead of `SecA2BModp` and `SecAddModp`. The security proof is not changed.

4 Gadgets performance

In this section, we compare the performance of each of our new gadgets to the state-of-the-art gadgets implementing the same feature (ignoring the differences in security property). We first describe the benchmark setup and the general implementation strategy, then we report the performance of state-of-the-art gadgets compared to the new gadgets.

4.1 Benchmarking setup

We implemented all the gadgets of Section 3 in the C programming language[9] and measured their performance on a ARM Cortex-M4 32-bit micro-controller. The recursive gadgets were naively implemented, only forcing inlining at a few places where the control flow overhead was identified as a bottleneck. The benchmarks were run on the NUCLEO-L4R5ZI development board, which is used by the `PQM4` benchmarking project [KRSS].[10] We used the default clock configuration of `PQM4`: the system clock and the AHB bus are clocked to 16 MHz and the TRNG peripheral is clocked at 48 MHz as recommended by the manufacturer. The performance measurements are based on the `DWT_CCYCNT` cycle accurate counter (hence also clocked at 16 MHz).

The randomness used in the gadget is taken on-the-fly from the on-chip TRNG, with no buffering, hence the time needed to generate randomness is included in the gadget's execution time. Concretely, the TRNG outputs 32-bit words, which are used as-is when randomness is needed in a bitsliced gadget. When uniform randomness in $\mathbb{Z}_p$ is needed, we extract two k-bit blocks in a 32-bit word from the TRNG ($k = \lceil \log_2 p \rceil \leq 16$) and apply rejection sampling: each block whose value is lower than p is accepted as a fresh $\mathbb{Z}_p$ random element while the other blocks are discarded. When uniform randomness in $\mathbb{F}_2^k$ with $k < 32$ is needed (e.g., in the `Kyber` implementation, $k = 13$ for the KS adder), we generate $\lfloor 32/k \rfloor$ k-bit words from 32 bits of randomness, dropping the remaining bits. The bottleneck in the randomness generation is the TRNG, which outputs four fresh random 32-bit words every 213 cycles with the previously described clock configuration[11], resulting in a throughput of 32 random bits every 53.25 cycles.

In the rest of this Section, we report the performance of concrete implementations, for which we have to fix the value of p. We take the prime of Kyber: $p = 3329$, which implies that most of the gadgets will be benchmarked for $k = \lceil \log_2(p) \rceil = 12$. All the cycle counts reported in this Section are for 256 independent calls to a given gadget since it is the polynomial size of `Kyber`. Since 256 is a multiple of the register width (32 bits), we fully exploit the bitslicing potential of the processor.

4.2 Performance of $\texttt{SecAdd}_k^d$

We first analyze masked adders on k bits. We compare in Figure 2 the Kogge-Stone adder from [BBE+18], which has a complexity of $\mathcal{O}(\log(k)d^2)$ CPU instructions, and the Algorithm 6 which has a complexity of $\mathcal{O}(kd^2)$ bit operations. First, we observe that Algorithm 6 requires fewer cycles than the KS adder. For $k = 13$, Algorithm 6 is about 23 times faster and for $k = 32$, the speedup is about 9x. As expected form the complexities, the gain of Algorithm 6 decreases as k increases. Yet for relevant parameters for lattice-based cryptography, it provides a significant improvement.

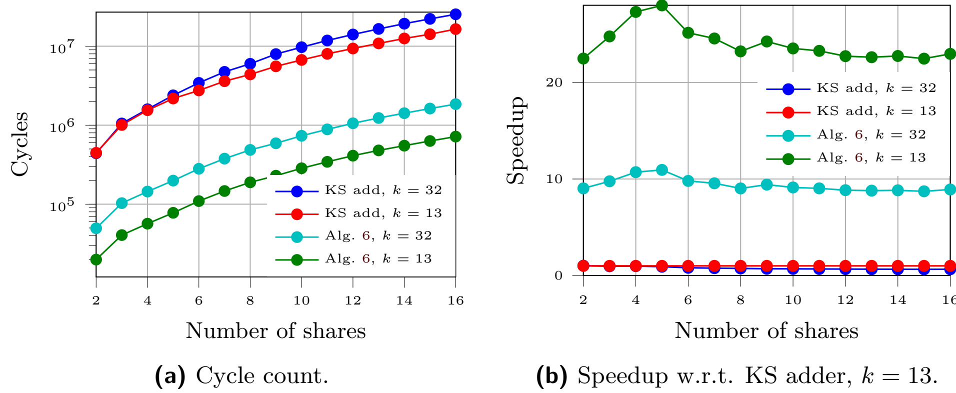
4.3 Performance of $\texttt{SecAddModp}_k^d$

Next, we compare in Figure 3 the execution time for various $\texttt{SecAddModp}_d^k$ gadgets. Concretely, we compare (i) Algorithm 2 when using the KS adder (not bitsliced), (ii) Algorithm 2 with the Algorithm 6 as underlying `SecAdd` (hence leveraging bitslicing), and (iii) Algorithm 7 (also using Algorithm 6). We observe that (ii) has a speedup of about 12x over

[9] The need for assembly implementations is discussed in Section 5 (as well as their performance characteristics). We focus on C implementations in this section to ease the comparison with state-of-the-art gadgets, which were implemented in C.

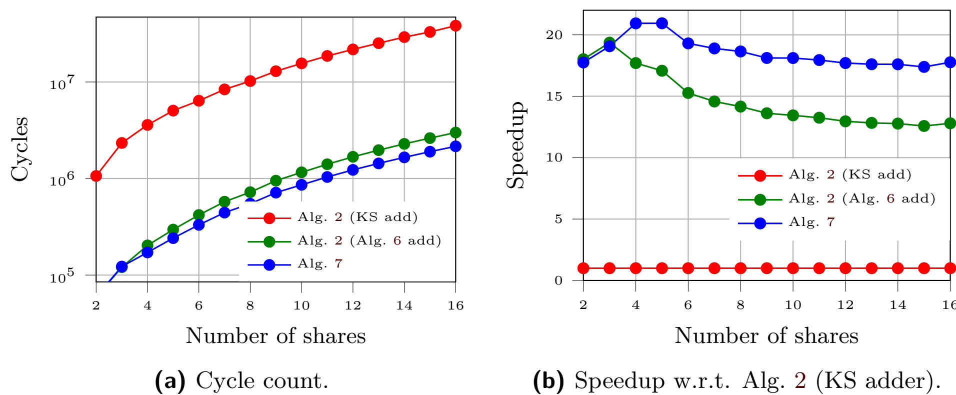
[10] Our benchmarks are compiled with options `-O2 -flto`, and we note that speedup figures for the `-O3` and `-Os` optimization levels are very similar. The `GCC` version is 9.4.0.

[11] As written in Section 3.2 of the datasheet (`https://www.st.com/resource/en/reference_manual/rm0432-stm32l4-series-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`), and confirmed by our experiments.



(a) Cycle count. **(b)** Speedup w.r.t. KS adder, $k = 13$.

Figure 2: Performance comparison of `SecAdd` implementations.



(a) Cycle count. **(b)** Speedup w.r.t. Alg. 2 (KS adder).

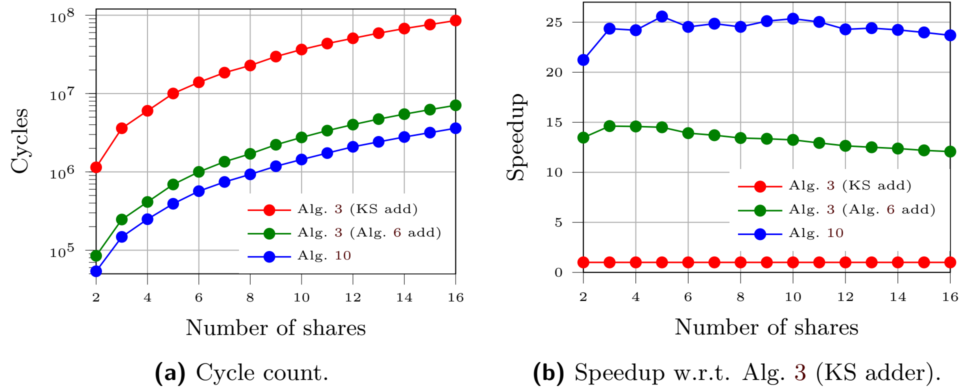
Figure 3: Performance comparison of $\texttt{SecAddModp}^d_{12}$ implementations.

(i), which is smaller than the improvement of 21x on the adder (`SecAdd`) itself. Indeed, the execution time of (ii) is dominated by the `SecAdd` calls and the MUX (Line 9) since both require in total $2(13-1)$ `SecAnd` executions, and while the speedup for the `SecAdd` part is 21x, the one for the MUX part is only the bitslicing gain of $32/12 = 2.7$x. Finally, in case (iii), the dedicated gadget allows to roughly half the cost of the MUX by replacing it with a `SecAdd`, which gives a speedup of about 1.3x over (ii).

4.4 Performance of arithmetic-to-Boolean conversions

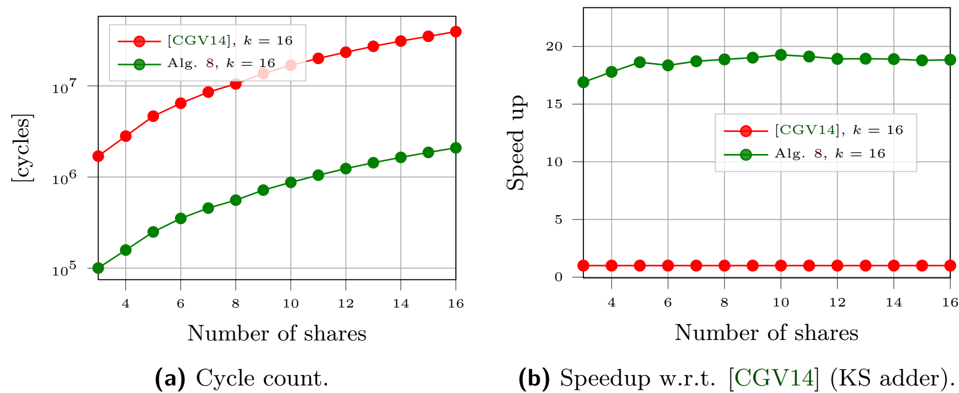
$\texttt{SecA2BModp}^d_k$. Similarly, we compare the performance of $\texttt{SecA2BModp}^d_k$ implementations in Figure 4. The reference implementation (i) is Algorithm 3 (with KS adder). We compare it to (ii) a modified Algorithm 3 using the bitsliced adder (Algorithm 7), and to (iii) the new Algorithm 10. We note that the speedup of (ii) over (i) is similar to the one we got for the corresponding `SecAddModp` gadgets (albeit a bit lower due to the presence of `RefreshSNI` whose bitslicing speedup is only 32/12). The new gadget (iii) has a speedup of 2x over (ii), thanks to the removal of refresh gadgets and the execution of one `SecAdd` with the number of shares halved.

$\texttt{SecA2B}^d_k$. We compare the performances of $\texttt{SecA2B}^d_k$ gadgets in Figure 5 for $k = 16$. The reference implementation (i) corresponds to the conversion from [CGV14, Alg. 4] with a KS adder. It is equivalent to Algorithm 3 by replacing the `SecAddModp` with `SecAdd`. It is compared to (ii) the new Algorithm 8 for $\texttt{SecA2B}^d_k$ leveraging the bitsliced adder



(a) Cycle count. **(b)** Speedup w.r.t. Alg. 3 (KS adder).

Figure 4: Performance comparison of $\texttt{SecA2BModp}^d_{12}$ implementations.



(a) Cycle count. **(b)** Speedup w.r.t. [CGV14] (KS adder).

Figure 5: Performance comparison of $\texttt{SecA2B}^d_{16}$ implementations.

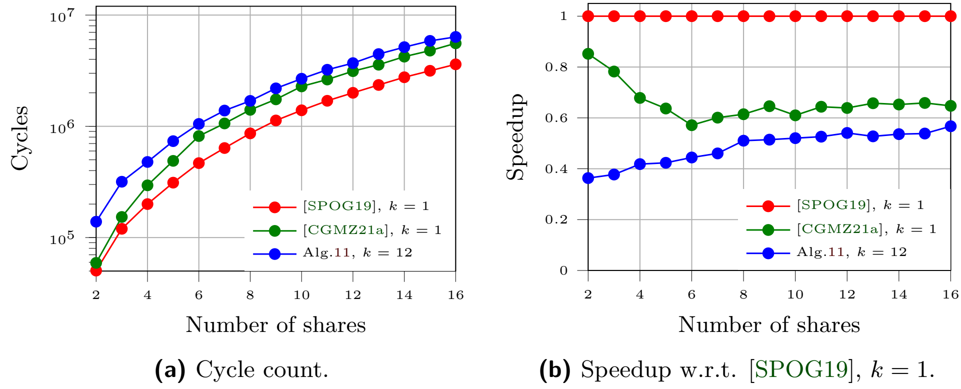
(Algorithm 7). The performance gain is around 18.8x by moving from (i) to (ii). This is expected from the improvements on the underlying $\texttt{SecAdd}^d_k$ (see Figure 2).

4.5 Performance of Boolean-to-arithmetic conversions

$\texttt{SecB2AModp}^d_k$. We next compare in Figure 6 the performance of various implementations of SecB2AModp. We consider as state-of-the-art the algorithms from [SPOG19] and [CGMZ21a] which both implement $\texttt{SecB2AModp}^d_k$ from single-bit conversions. As a result, their computational cost is proportional to k, and we observe that they have comparable cost, with a small advantage for [SPOG19] (which agree with the results on Intel x86 processors of [CGMZ21a], Table 4).

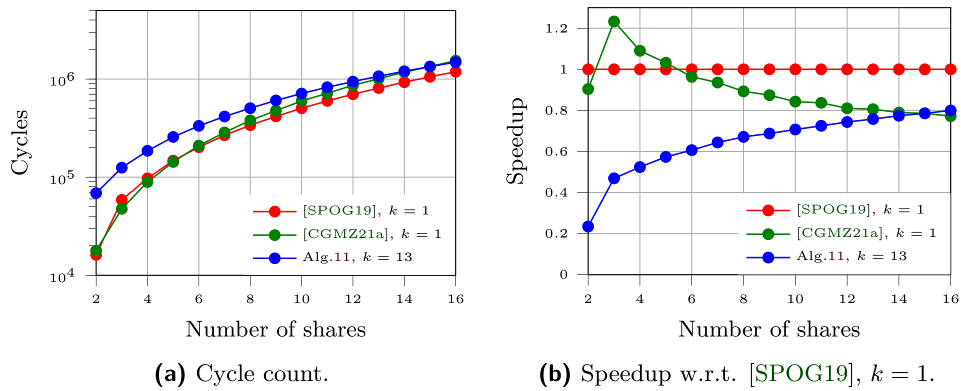
Our bitsliced conversion gadget (Algorithm 11) always operates on $\lceil \log_2(p) \rceil$ bits (here, 12). Concretely, for 16 shares, the bitsliced conversion of any $x \in \mathbb{Z}_p$ is only twice as slow as the state-of-the-art single-bit conversions, and is therefore on par with state-of-the-art 2-bit conversions. For larger k-bit conversions, the advantage of Algorithm 11 grows linearly with k.

$\texttt{SecB2A}^d_k$. Finally, we compare in Figure 7 the performance of various implementations of SecB2A. To do so, instantiate the 2^k variant of the gadgets in the previous experiment. The conclusions are similar: for a single bit ($k = 1$) to convert from Boolean to arithmetic masking, both [SPOG19] and [CGMZ21a] are more efficient than the new gadget. For $k > 1$, our gadget is more efficient.



(a) Cycle count. **(b)** Speedup w.r.t. [SPOG19], $k = 1$.

Figure 6: Performance comparison of `SecB2AModp` implementations.



(a) Cycle count. **(b)** Speedup w.r.t. [SPOG19], $k = 1$.

Figure 7: Performance comparison of `SecB2A` implementations.

5 Side-channel leakage assessment of implementations

The previous section demonstrates that using a Boolean representation (hence using bitslicing for micro-controllers) for masking conversion leads to performance gains. In order to ensure that the proposed gadgets meet their goal of providing concrete $d-1$-order security, we perform leakage assessment. As hinted by the literature [BGG+14, BWG+22], the gadgets from the previous section, which are written in C to ease comparisons, lead to unintended leakage recombination.

In the following section, we first recall the Test Vector Leakage Assessment (TVLA) [GJJ+11, CMG+, SM16] and introduce our side-channel measurement setup. We then show that TVLA confirms the presence of unintended leakage in the gadgets written in C. Next, we present hardened implementations of the new conversion gadgets which, using a mix C and assembly, remove these problematic leakages. Finally, we discuss the overhead of this hardening, answering an open question from [BWG+22].

5.1 Test vector leakage assessment

TVLA. Student's t-test performs hypothesis testing to highlight difference in the i-th order moment of two distributions. In the context of side-channel leakage assessment, these two sets are traces corresponding to two distinct values for the secret input of a cryptographic implementation. This methodology is known as the (fixed-versus-fixed) TVLA, and the commonly adopted threshold for declaring the presence of leakage at a given order is a p-value smaller than 10^{-5}. This p-value is translated to a threshold on the t statistic, taking into account the number of independent tests performed [DZD+17, WO19] (otherwise there is a high risk of false positive).

Concretely, we instantiate the $\texttt{SecA2BModp}_k^p$ and $\texttt{SecB2AModp}_k^p$ gadgets with $d = 2$, $p = 3329$ and $k = 12$ (to match `Kyber` parameters in the next section). We analyze the difference in the means (first-order moment), and in the variance (second-order centered moment) following the algorithm from [SM16].[12] In both cases, we collect 100,000 traces to compute the t statistic. In the following plots, the threshold is denoted with a red horizontal line.

SCA measurement setup. The side-channel evaluation is performed on the `STM32F415` target board of the `CW308 Chipswhisperer`.[13] The target is running at a clock frequency of 80 MHz which is derived from an 8 MHz external crystal. The side-channel traces are captured thanks to a `Picoscope5244D` with a 250 MSamples/sec attached to a `CT1` current probe from `Tektronix`. As a result, the signal-to-noise ratio on a canonical representation of a word over $\mathbb{Z}_p$ within the implementation is around 0.4, showing that the setup provides clean measurements.

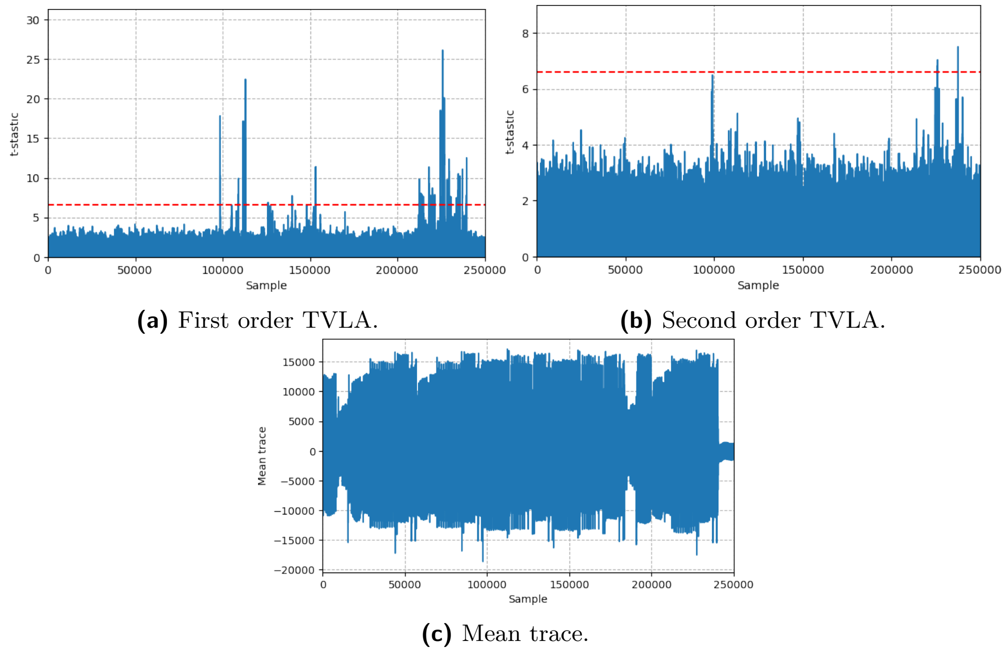
Disclaimer. TVLA is a good tool to detect the presence of unintended lower-order leakage and to perform root cause analysis of weaknesses [GOP21]. It does however not guarantee the security of the implementation [Sta18], especially in the case of low-noise targets [BS20, BS21].

5.2 Leakage assessment of C implementations

Figure 8 shows the TVLA analysis of the (pure) C implementation of the `SecB2AModp` and `SecA2BModp` gadgets with two shares. Namely, Figure 8b highlights evidence of second-order leakage, as expected. However, Figure 8a highlights evidence of first-order leakage,

[12] Using the implementation of the `SCALib` library (https://scalib.readthedocs.io/en/latest/source/scalib.metrics.html).

[13] https://rtfm.newae.com/Targets/UFO%20Targets/CW308T-STM32F/



(a) First order TVLA.

(b) Second order TVLA.

(c) Mean trace.

Figure 8: TVLA results with 100,000 traces for `SecB2AModp` followed by a `SecA2BModp` on both distinct sets of inputs (fixed vs. fixed). Implementation is in plain C.

which should not happen in a proper first-order secure implementation. This is due to so-called "transition leakage" phenomenon, where the leakage depends (for example) on the Hamming distance between the two consecutive values stored in a register [BGG+14]. When these two values are the two shares of a Boolean sharing, this produces first-order leakage, since the Hamming distance of the two shares is equal to the Hamming weight of the shared value.

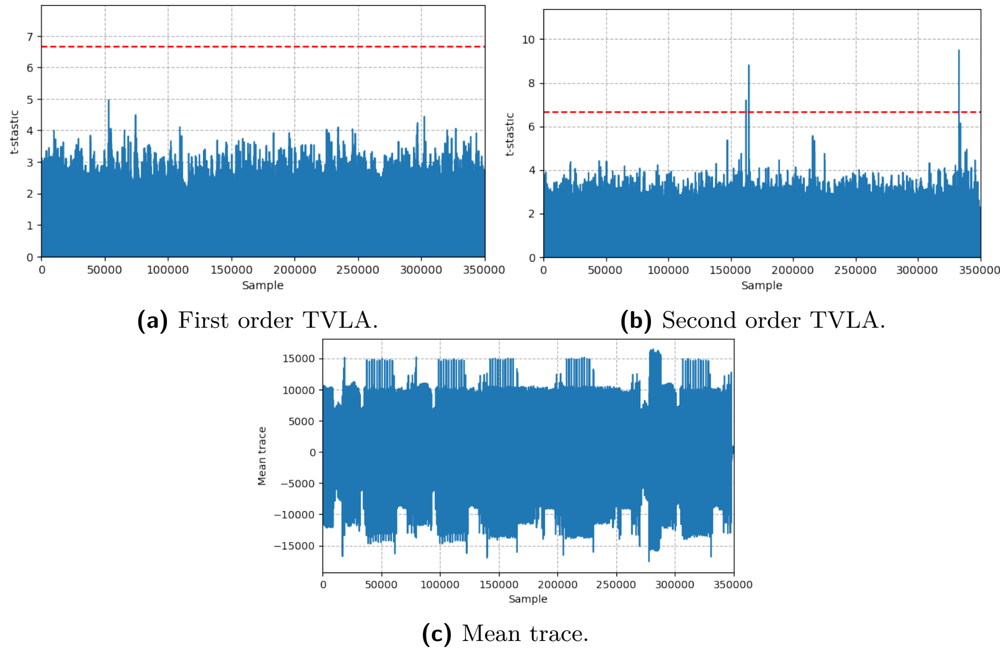
5.3 Implementing masking conversions with C & assembly

Avoiding Hamming distance leakage between the shares of a sharing requires an accurate control of the (micro-)architectural state of a processor. Since the C programming language does not give this level of control, we implement the manipulations of the shares in assembly. However, we keep C implementations for gadgets that compose other gadgets and do not touch the shares directly.[14] This eases the writing and improves the readability of the implementations without degrading its security.

Heuristics for secure assembly gadgets. Based on an abstract understanding of the architecture of a small micro-controller[15], we anticipate transition leakage to appear in the registers, on the ALU inputs and outputs, and in the memory read and write paths. Each assembly gadget ($\texttt{SecAnd}^d$, $\oplus^d$ and $\texttt{BitCopyMask}_k^d$) therefore takes as input a pointer to the shares (avoiding the presence of the shares in the registers when C code is executed) and uses dummy operations to avoid damaging transition leakage. We use a defensive approach: a dummy load (resp. store) of a non-sensitive variable (e.g., a constant) is

[14] We also kept the C implementations for gadgets that manipulate shares but do not exhibit lower-order leakage. This is admittedly not robust to compilation toolchain changes, but we do not see it as an issue since the compiler-generated code can be used as new assembly source. Moreover, our evaluations are specific to a single MCU, hence our code offers anyway to portable security guarantees.

[15] We do not have access to the detailed architecture of the Cortex-M4 MCU.



(a) First order TVLA.

(b) Second order TVLA.

(c) Mean trace.

Figure 9: TVLA results with 100,000 traces for `SecB2AModp` followed by a `SecA2BModp` on both distinct sets of inputs (fixed vs. fixed). Implementation is a mix of C and assembly.

Table 1: Gadget hardening overhead: number of execution cycles of the hardened C & assembly implementation divided by the number of execution cycles for the pure C implementation.

d	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\texttt{SecAdd}_{12}$	1.45	1.57	1.61	1.67	1.69	1.73	1.71	1.71	1.69	1.69	1.68	1.68	1.67	1.66	1.65
$\texttt{SecAddModp}_{12}$	1.36	1.49	1.56	1.63	1.66	1.70	1.69	1.69	1.68	1.67	1.67	1.67	1.66	1.65	1.65
$\texttt{SecB2AModp}_{12}$	1.29	1.36	1.39	1.43	1.46	1.48	1.48	1.50	1.51	1.52	1.52	1.53	1.53	1.53	1.53
$\texttt{SecA2BModp}_{12}$	1.35	1.42	1.42	1.47	1.50	1.51	1.51	1.53	1.54	1.55	1.56	1.57	1.57	1.57	1.57

executed between the loads (resp. stores) of shares. Moreover, we keep a minimum number of shares in the register file at any moment (there are at most three shares in the register file at the same time), erasing any register containing a share as soon as possible.

Leakage assessment. We first applied TVLA to $\texttt{SecAnd}^d$ and $\oplus^d$ in order to ensure that our defensive approach is effectively preventing lower-order leakage. Then, the masked conversions `SecA2BModp` and `SecB2AModp` are evaluated, and the results are reported in Figure 9, showing no first-order leakage with up to 100,000 traces on the evaluated micro-controller (showing that the remaining C code does not cause lower-order leakage).

Performance. The defensive implementation approach has a significant performance overhead (between 1.29x and 1.71x) over the pure C implementation (see Table 1).[16] Formal verification tools that leverage detailed knowledge of the processor's micro-architecture [GHP+21, BGG+21] could be used to improve the performance of these implementations by allowing a less defensive implementation strategy. It would also increase the confidence in the security of these implementations (as well as formally validating the security of the code generated by the C compiler), providing a complementary evaluation to the TVLA.

[16] The benchmarking setup is the same as the one used in Section 4.

Algorithm 12 `Kyber.CCAKEM.Dec` (c, sk)

Input: Ciphertext $c = (c_u, c_v)$, secret key $sk := (\hat{\boldsymbol{s}}, pk, \texttt{H}(pk), z)$.
Output: Decapsulated secret K.

1: $m' :=$ `Kyber.CPAPKE.Dec`$(\hat{\boldsymbol{s}}, c)$
2: $(\bar{K}', \sigma') := \texttt{G}^d(m'||\texttt{H}(pk))$
3: $(c'_u, c'_v) :=$ `Kyber.CPAPKE.Enc`(pk, m', σ')
4: **if** $(c_u = c'_u)$ & $(c_v = c'_v)$ **then**
5: $\quad K := \texttt{KDF}(\bar{K}'||\texttt{H}(c))$
6: **else**
7: $\quad K := \texttt{KDF}(z||\texttt{H}(c))$

Algorithm 13 `Kyber.CPAPKE.Dec` $(\hat{\boldsymbol{s}}, c)$

Input: Secret key $\hat{\boldsymbol{s}} \in \mathbb{R}_p^l$, ciphertext $c = (c_u, c_v)$.
Output: Plaintext m.

1: $\boldsymbol{u} := \texttt{Decompress}^d_{p,d_u}(c_u)$ ▷ $\boldsymbol{u} \in \mathbb{R}_p^l$, $d_u = 10$
2: $v := \texttt{Decompress}^d_{p,d_v}(c_v)$ ▷ $v \in \mathbb{R}_p$, $d_v = 4$
3: $\hat{z} = \hat{\boldsymbol{s}}^T \circ \texttt{NTT}(\boldsymbol{u})$ ▷ $\hat{z} \in \mathbb{R}_p$
4: $w := v - \texttt{NTT}^{-1}(\hat{z})$ ▷ $w \in \mathbb{R}_p$
5: $m := \texttt{Compress}^d_{p,1}(w)$ ▷ m is a 256-bit string

6 Application to lattice-based KEMs

In this section, we put our new gadgets together into an implementation of Kyber. We focus on `Kyber768` to maximize to comparability with the recent works of Coron et al. [CGMZ21a, CGMZ21b]. Eventually, we apply the same methodology to Saber and report the results.[17]

6.1 Overview of masked Kyber

Kyber leverages the Fujisaki-Okamoto (FO) transform to transform a chosen-plaintext attack (CPA) secure public encryption scheme (PKE) intro a chosen-ciphertext attack (CCA) secure KEM [FO99, ABD+19]. Kyber decapsulation is described Algorithm 12 where the ciphertext c is decrypted with `CPAPKE.Dec`$(\cdot)$ to obtain the message m'. This message is then re-encrypted with `CPAPKE.Enc`$(\cdot)$ to derive the ciphertext c' using some pseudo-randomness σ' derived from m' and the public key. The encapsulated secret K is then returned only if c and c' are identical, which ensures that the c has been derived from the public key. We focus on the masked implementation of `Kyber.CCAKEM.Dec` since it is the most sensitive to SCA [RRCB20, UXT+22]. In the following algorithms, green means that no masking is required[18], blue that masking is required and has linear complexity with d (when implemented with arithmetic masking), and red that masking with quadratic complexity is required, which means that bitsliced Boolean masking may be beneficial.

`Kyber.CPAPKE` manipulates polynomial ring $\mathbb{Z}_p[X]/(X^n + 1)$ that we denote as $\mathbb{R}_p$. Vectors of size l of polynomials are next denoted with bold such that $\boldsymbol{x} \in \mathbb{R}_p^l$. Kyber makes also use of `NTT` representation that we denote $\hat{x} := \texttt{NTT}(x)$. The first step (Line 1-2) in decryption is to map the ciphertext c into the corresponding (vector of) polynomial(s). Then, the secret key $\hat{\boldsymbol{s}}$ is multiplied with $\boldsymbol{u}$ and subtracted to v (Line 3-4). Concretely, these operations (addition, multiplications and `NTT`) are performed with arithmetic masking and can be applied share-by-share, hence with linear complexity. Finally, each coefficient (in $\mathbb{Z}_p$) of the resulting polynomial is compressed to a single bit, which represents the rounding to $\lceil p/2 \rfloor$ or 0. We detail the masked implementation of $\texttt{Compress}^d_{p,c}$ in Algorithm 15.

Finally, `Kyber.CPAPKE.Enc` is described in Algorithm 14. This algorithm starts by generating $2l + 1$ noise polynomials (Line 2-4) whose coefficients follow a central binomial distribution (`CBD`, see Algorithm 17) with parameter η, such that they belong to $[\![-\eta, \eta]\!]$. The `CBD` takes as input a pseudo-random string of bits which is computed as the hash `PRF` of the random seed σ and a nonce. Next, the noise ($\boldsymbol{e}_1$ and $\boldsymbol{e}_2$) is added to the product

[17] We implemented the NIST level 2 version of the Saber family, which is called `Saber`.
[18] We focus on long-term security of the Kyber private key, and assume that the exchanged key K can be leaked to a side-channel adversary. Otherwise, the derivation of K should also be protected.

of the public key and the vector of noise polynomials $\boldsymbol{r}$ (Line 5-7). The message m is decompressed to a polynomial with $\texttt{Decompress}^d_{q,1}$ (see Algorithm 16) and added to the sum. The last step is to compress (i.e., rounding then divide) both $\boldsymbol{u}$ to d_u bits and v to d_v bits, which gives the ciphertext (Lines 8-9).

Algorithm 14 `Kyber.CPAPKE.Enc` (pk, m, σ)

Input: $pk = (\hat{\boldsymbol{t}}, \hat{\boldsymbol{A}})$ with $\hat{\boldsymbol{t}} \in \mathbb{R}_p^l$, $\hat{\boldsymbol{A}} \in \mathbb{R}_p^{l \times l}$; message $m \in \{0,1\}^n$, randomness $\sigma \in \{0,1\}^{256}$.
Output: Ciphertext $c = (c_u, c_v)$.

1: **for** $i = 0$ to $l-1$ **do** ▷ Noise sampling
2: $\quad \boldsymbol{r}[i] := \texttt{CBD}^d_{\eta_1}(\texttt{PRF}^d(\sigma, i))$ ▷ $\boldsymbol{r} \in \mathbb{R}_p^l$, $\eta_1 = 2$
3: $\quad \boldsymbol{e}_1[i] := \texttt{CBD}^d_{\eta_2}(\texttt{PRF}^d(\sigma, i+l))$ ▷ $\boldsymbol{e}_1 \in \mathbb{R}_p^l$, $\eta_2 = 2$
4: $e_2 := \texttt{CBD}^d_{\eta_2}(\texttt{PRF}^d(\sigma, 2 \cdot l))$ ▷ $e_2 \in \mathbb{R}_p$, $\eta_2 = 2$
5: $\hat{\boldsymbol{r}} := \texttt{NTT}(\boldsymbol{r})$
6: $\boldsymbol{u} := \texttt{NTT}^{-1}(\hat{\boldsymbol{A}}^T \circ \hat{\boldsymbol{r}}) + \boldsymbol{e}_1$ ▷ $\boldsymbol{u} \in \mathbb{R}_p^l$
7: $v := \texttt{NTT}^{-1}(\hat{\boldsymbol{t}}^T \circ \hat{\boldsymbol{r}}) + e_2 + \texttt{Decompress}^d_{p,1}(m)$ ▷ $v \in \mathbb{R}_p$
8: $c_u := \texttt{Compress}^d_{p,d_u}(\boldsymbol{u})$ ▷ $d_u = 10$
9: $c_v := \texttt{Compress}^d_{p,d_v}(v)$ ▷ $d_v = 4$

6.2 `Kyber768` implementations

Next, we detail our implementation of `Kyber768`, whose parameters are $d_u = 10$, $d_v = 4$, $\eta_1 = \eta_2 = 2$, $l = 3$ and $p = 3329$.[19] For each of the algorithms $\texttt{Compress}^d_{p,c}$, $\texttt{Decompress}^d_{p,c}$ and $\texttt{CBD}^d_\eta$, we will describe our new construction together with the previous state-of-the-art solution.

The implementations K1, K2 and K3 are derived from the PQM4 [KRSS] optimized Kyber implementation for the Cortex-M4: the linear operations (such as the `NTT`) are kept (and applied to all the shares), while the non-linear operations are replaced by masked gadgets. We keep a single noise polynomial in memory at any time in Algorithm 14 to reduce the stack usage. Implementation K1 relies on the C implementation provided by Coron et al. of their gadgets [CGMZ21b] and on new C implementations of the single-bit B2A conversion of [SPOG19]. Implementation K2 is based on a C-only implementation of the new bitsliced gadgets while K3 uses a mix of C and assembly to avoid lower-order leakage (see details in Subsection 5.3).

`BsToC` & `CToBs`. In all implementations, the top-level algorithms (`Kyber.CCAKEM.Dec`, `Kyber.CPAPKE.Dec` and `Kyber.CPAPKE.Enc`) use a canonical (i.e., non-bitslice) representation for all their variables. Therefore, `BsToC` / `CToBs` is executed in the lower-level algorithms (`Decompress`, `Compress` and `CBD`) when needed, that is, for every sharing that is an input or output of a gadget introduced in Section 3 (except the output of Algorithm 11, which is already in canonical representation), while avoiding unnecessary representation changes in `CBD` (i.e., the only representation change is `CToBs` for $\boldsymbol{a}^{B,\eta}$ and $\boldsymbol{b}^{B,\eta}$). Since the masked `CBD`, `Compress` and `Decompress` are applied to vectors whose length is $n = 256$, the parallelism offered by bitslicing the Boolean parts of these algorithms is used to parallelize the operations inside a single vector (this is therefore transparent to the top-level algorithms). Finally, the internal structure of Keccak can be exploited such that a single masked computation of `Keccak-f[1600]` is internally trivially bitsliced [BDPA13].

[19] Note that the proposed construction also applies to both `Kyber512` (with $l = 2$, $\eta_1 = 3$) and `Kyber1024` (with $l = 4$, $d_u = 11$, $d_v = 5$).

Algorithm 15 $\texttt{Compress}^d_{p,c}$, from [CGMZ21b]

Input: d shares arithmetic sharing $\boldsymbol{x}^{A_p}$ such that $p < 2^k$ and $x \in [\![0, p[\![$. Compression factor $c \in [\![1, k[\![$.
Output: d shares Boolean sharing $\boldsymbol{z}^{B,c}$ such that $z = \lfloor (2^c/p) \cdot x \rceil \mod 2^c$.

1: $\alpha \leftarrow \lceil \log_2(p \cdot d) \rceil$
2: $\boldsymbol{y}_{d-1}^{A_{2^{c+\alpha}}} \leftarrow \lfloor (\boldsymbol{x}_{d-1}^{A_p} \cdot 2^{c+\alpha+1} + p)/(2p) \rfloor + 2^{\alpha-1} \mod 2^{c+\alpha}$
3: **for** $i = 0$ to $d-2$ **do**
4: $\quad \boldsymbol{y}_i^{A_{2^{c+\alpha}}} \leftarrow \lfloor (\boldsymbol{x}_i^{A_p} \cdot 2^{c+\alpha+1} + p)/(2p) \rfloor \mod 2^{c+\alpha}$
5: $\boldsymbol{z}^{B,c+\alpha} \leftarrow \texttt{SecA2B}^d_{c+\alpha}\left(\boldsymbol{y}^{A_{2^{c+\alpha}}}\right)$ ▷ Algorithm 8
6: $\boldsymbol{z}^{B,c} \leftarrow \boldsymbol{z}^{B,c+\alpha}[[\![\alpha, \alpha + c[\![]$

Algorithm 16 $\texttt{Decompress}^d_{p,1}$

Input: d shares Boolean sharing $\boldsymbol{x}^{B,1}$, integer p such that $p < 2^k$ and $x \in \{0, 1\}$.
Output: d shares arithmetic sharing $\boldsymbol{z}^{A_p}$ such that $z = x \cdot \lceil p/2 \rfloor \mod p$.

1: $\boldsymbol{y}^{A_p} \leftarrow \texttt{SecB2AModp}^d_1\left(\boldsymbol{x}^{B,1}\right)$
2: $\boldsymbol{x}^{A_p} \leftarrow \lceil p/2 \rfloor \cdot \boldsymbol{y}^{A_p} \mod p$

$\texttt{Compress}^d_{p,c}$. The Compress allows to map an element in $\mathbb{Z}_p$ to $z = \lfloor (2^c/p) \cdot x \rceil \mod 2^c$. We leverage the masked compression algorithm from [CGMZ21b] (Algorithm 15) for the implementation of $\texttt{Compress}^d_k$ in all Kyber768 implementations (see below details for K1). Our $\texttt{Compress}^d_{p,c}$ algorithm takes as input an arithmetic sharing $\boldsymbol{x}^{A_p}$ and transforms it into an arithmetic sharing $\mod 2^{c+\alpha}$ (where $\alpha = \lceil \log_2(p \cdot d) \rceil$) using sharewise operations. The result is then converted into a $(c + \alpha)$-bit Boolean sharing with the bitsliced SecA2B (Algorithm 8). Finally, the c most significant bits of the Boolean sharing are taken as output.

For K2 and K3, the polynomial comparison is fully based on Compress. That is, we test the joint equality to the ciphertext of all the compressed polynomial coefficients (c'_u and c'_v) using bitsliced Boolean $\oplus^{\mathsf{B}}$ (for individual bit equality testing) then SecAnd (to summarize all equality test results in a single bit).

For K1, each of the polynomial comparison are detailed in [CGMZ21b]. More precisely, we consider as reference for their hybrid-method. For the test of c_u, Coron et al. compare (in arithmetic masking) $\boldsymbol{u}'$ with all the possible candidates $\boldsymbol{u}$ that could lead to the compression c_u. For the test of c_v, Coron et al. uses Algorithm 15 without bitslicing. Eventually, the $\texttt{Compress}^d_{p,1}$ in K1 is performed with the table-based conversion from [CGMZ21a].

$\texttt{Decompress}^d_{p,1}$. Decompress is mapping a single bit to $\lceil p/2 \rfloor$ or 0, and we implement it with Algorithm 16, in which single-bit Boolean sharing $\boldsymbol{x}^{B,1}$ is converted to arithmetic sharing $\boldsymbol{y}^{A_p}$ with the single-bit dedicated conversion from [SPOG19]. We do not use our generic $\texttt{SecB2AModp}^d_k$ for this purpose since, as shown in Figure 6, it is slower by a factor 2 for single-bit conversions.

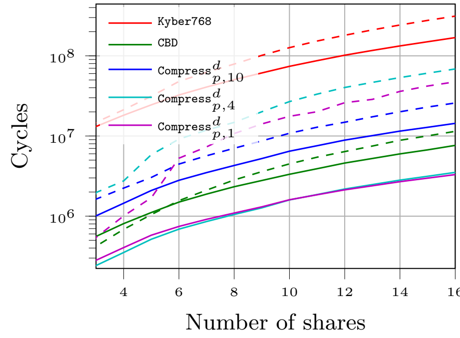
$\texttt{CBD}^d_2$. The CBD takes as input two random strings a and b of η bits and outputs $\texttt{HW}(a) - \texttt{HW}(b) \mod p$. For K1, we use the implementation from [SPOG19] which computes $\texttt{HW}(a) - \texttt{HW}(b) + \eta$ in Boolean masking (using their $\texttt{SecAdd}^d_k$), then converts it to arithmetic masking using their $\texttt{SecB2AModp}^d_k$, and finally subtracts η. For K2 and K3, we use Algorithm 17, which is close to the gadget of [SPOG19], but uses an optimal full adder composition for the addition of the bits of a and $\neg b$, and furthermore uses our new SecFullAdder and SecB2AModp bitslice gadgets. The new $\texttt{CBD}^d_\eta$ uses $\lfloor 2\eta/2 \rfloor + \lfloor 2\eta/4 \rfloor + \lfloor 2\eta/8 \rfloor + \ldots$ full-adders to compute $\texttt{HW}(a) - \texttt{HW}(b) + \eta$, which amounts to 3 SecAnd when $\eta = 2$, instead of 8 SecAnd for the implementation of [SPOG19].

G, H and PRF. All the hash functions used are based on SHA-3 and therefore all use the Keccak-f[1600] permutation. Concretely, we developed a masked Keccak-f[1600]

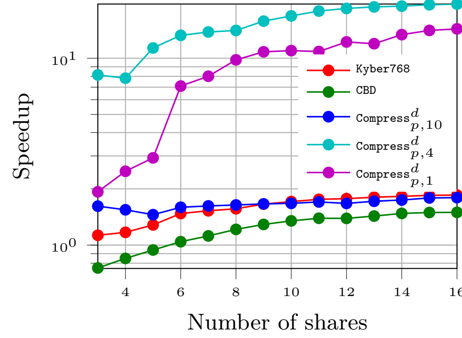
Algorithm 17 $\mathtt{CBD}^d_\eta$ New (PINI, by composition)

Input: d shares Boolean sharing $\boldsymbol{a}^{B,\eta}$ and $\boldsymbol{b}^{B,\eta}$, integer p such that $p < 2^k$ and $x \in [\![0, p[\![$.
Output: d shares arithmetic sharing $\boldsymbol{z}^{A_p}$ such that $z = \mathtt{HW}(a) - \mathtt{HW}(b) \mod p$.

1: $\left(\boldsymbol{s}^{B,2\eta}[[\![0,\eta[\![], \boldsymbol{s}^{B,2\eta}[[\![\eta,2\eta[\![]\right) \leftarrow \left(\boldsymbol{a}^{B,\eta}, \neg\boldsymbol{b}^{B,\eta}\right)$ ▷ $\mathtt{HW}(s) = \mathtt{HW}(a) - \mathtt{HW}(b) + \eta$
2: $\ell \leftarrow 2\eta$
3: $k \leftarrow \lceil \log_2(\ell + 1) \rceil$
4: **for** $i = 0$ to $k - 1$ **do** ▷ Iterate from output LSB to MSB.
5: $\quad \boldsymbol{x}^{B,1} \leftarrow$ **if** $\ell \mod 2 = 1$ **then** $\boldsymbol{s}^{B,2\eta}[\ell - 1]$ **else** $(0, 0, \ldots, 0)$
6: $\quad \ell \leftarrow \lfloor \ell/2 \rfloor$
7: $\quad$ **for** $j = 0$ to $\ell - 1$ **do** ▷ Accumulate all bits of weight i.
8: $\quad\quad \boldsymbol{t}^{B,2} \leftarrow \mathtt{SecFullAdder}^d\left(\boldsymbol{s}^{B,2\eta}[2j], \boldsymbol{s}^{B,2\eta}[2j+1], \boldsymbol{x}^{B,1}\right)$ ▷ Algorithm 5
9: $\quad\quad \left(\boldsymbol{x}^{B,1}, \boldsymbol{s}^{B,2\eta}[j]\right) \leftarrow \left(\boldsymbol{t}^{B,2}[0], \boldsymbol{t}^{B,2}[1]\right)$ ▷ Sum bit goes to $\boldsymbol{x}^{B,1}$ and carry to $\boldsymbol{s}^{B,2\eta}[j]$.
10: $\quad \boldsymbol{y}^{B,k}[i] \leftarrow \boldsymbol{x}^{B,1}$
11: $\boldsymbol{z}^{A_p} \leftarrow \mathtt{SecB2AModp}^p_d\left(\boldsymbol{y}^{B,k}\right)$ ▷ Algorithm 11, $y = \mathtt{HW}(a) - \mathtt{HW}(b) + \eta$
12: $\boldsymbol{z}_0^{A_p} \leftarrow \boldsymbol{z}_0^{A_p} - \eta \mod p$



(a) Cycle count (K1: dashed, K2: solid line).



(b) Speedup of K2 over K1.

Figure 10: Comparison of the performance of various components of C-only implementations of `Kyber768`: K1 (state-of-the-art gadgets) and K2 (new).

implementation based on the PINI `SecAnd`.

Probing security The Kyber implementations K2 and K3 are a composition of PINI gadgets, hence they are PINI, and therefore probing secure.

6.3 Kyber performance

We show in Figure 10 the performance[20] of the top-level masked components of the Kyber K1 (based on state-of-the-art gadgets) and K2 (new), both written only C to ease comparison.

First, we remark that $\mathtt{Compress}^d_{p,1}$ in K2 achieves a speedup of more than 10x over K1, showing that Algorithm 15 (bitsliced) is faster than the table-based approach by Coron et al. For $\mathtt{Compress}^d_{p,4}$, the speedup (about 20x) is exactly the one of our new $\mathtt{SecA2B}^d_k$ since both implementations implement the same algorithm and `SecA2B` is the bottleneck. Next, the speedup for the compressed comparison of c_u and c'_u is smaller. Indeed, Coron et al. have already vastly improved this polynomial comparison in [CGMZ21b], which limits the speedup of K2 to 1.8x. Finally, regarding the `CBD` (which includes the Boolean to arithmetic

[20]The benchmarking setup is the same as the one described in Subsection 4.1.

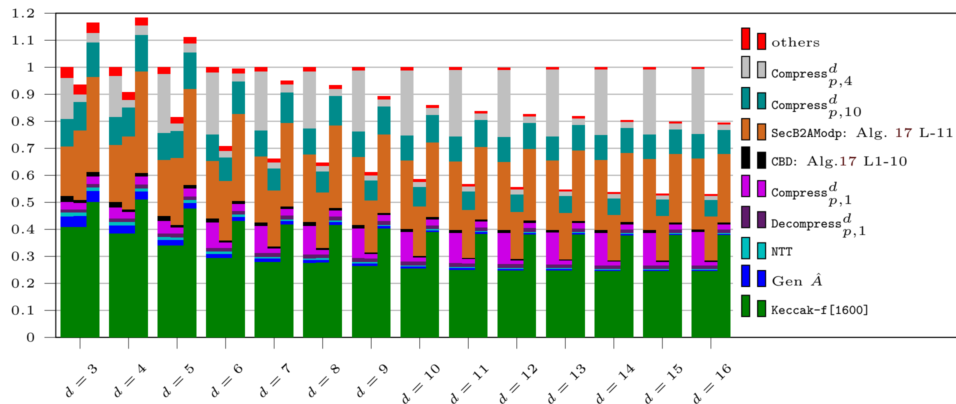


Figure 11: Performance comparison of `Kyber768` implementations: K1 (state-of-the-art gadgets, C-only, left) and K2 (new, C-only, middle) and K3 (new, C and assembly, right). Performance is normalized w.r.t. K1. For better performance and small d, users should swap $\texttt{SecB2AModp}^d_k$ conversions.

masking conversion of the noise), the gain in performance is directly dependent on the gain for $\texttt{SecB2AModp}^d_k$ that we discussed in Figure 6, since this gadget is the bottleneck. For number of shares up to 6, the `CBD` based only on gadgets from [SPOG19] is faster, while for a larger number of shares, the gain is around 1.5.

Overall, our new gadgets lead to a speedup of about 1.8x for the entire `Kyber768`. As shown in the decomposition of Figure 11, the speedup mostly comes from the improvement on polynomial compressions and comparisons (reduced from 45% to about 10% of the total execution time). This leaves the implementation K2, dominated by the masked `Keccak-f[1600]` (for 50% of the cycles) whose implementation is already efficiently bitsliced in the state-of-the-art, and by the $\texttt{SecB2AModp}^d_k$ conversion of the noise polynomials (in Algorithm 17) for about 30% of the cycles.

The K3 implementation, which is hardened to avoid lower-order leakages, implies overheads compared to K2 as expected from Subsection 5.3. Eventually, we report the exact cycle count for K3 and each of its top-level components in Table 2. In that table, we note that the total number of cycles spent in representation changes (`CToBs` and `BsToC`) takes 4.9% of the total execution of a $d = 2$ `Kyber.CCAKEM.Dec` while it is only 1.8% for $d = 16$. This confirms the interest of changing the data representation to take advantage of new gadgets in their application to lattice-based KEMs.

6.4 Saber performance

We implement and benchmark Saber [BBMD+19] with the methodology we used for Kyber. Indeed, the structure of Saber is very similar to the one of Kyber, the main difference being the use of a field of characteristic two instead of a prime order field. We developed all the implementations starting from the unprotected implementation provided by PQM4 and integrating the masked gadgets. That is for S1, we use the gadgets proposed by Coron et al. [CGMZ21b], except for the `SecB2A` in `CBD` which is more efficient by leveraging the algorithms from [SPOG19] (see Figure 7). For the implementation S2, we make use of a C-only implementations of the bitsliced gadgets `SecA2B`, `SecB2A` and `CBD`. Implementation S2 is trivially probing secure thanks to PINI composition.

Overall, implementation S2 achieves a speedup of about 3x over S1 for the entire Saber as reported in Figure 12. Concretely, our new gadgets reduce the execution time of the conversions by a large factor such that the fraction of runtime dedicated to them is

Table 2: Performance of the K3 `Kyber768` implementation: number of clock cycles when running on a `STM32L4R5` and using the TRNG for masking randomness (32-bit randomness every 53 cycles). Reported numbers are in `kCycles`. The cost of the `CToBs` and `BsToC` operations is included in the gadgets that perform it, and the total time of their execution is also given separately.

d	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`Kyber.CCAKEM.Dec`	10018	16747	24709	34683	45950	58473	72512	88203	106040	124598	144757	166094	189034	213064	238865
`Keccak-f[1600]`	4579	7225	10664	14897	19923	25752	32384	39816	48032	57042	66871	77500	88915	101111	114108
Gen $\hat{A}$	537	588	619	667	718	758	804	853	902	949	996	1041	1088	1135	1181
`NTT`	132	187	243	298	354	409	465	520	575	631	686	742	797	853	908
$\texttt{Decompress}^d_{p,1}$	97	168	278	442	652	834	1086	1379	1685	2032	2403	2818	3243	3612	4097
$\texttt{Compress}^d_{p,1}$	214	414	616	909	1184	1479	1784	2151	2660	3078	3508	3972	4444	4938	5444
`CBD`: Alg.17 L1-1	167	229	306	396	497	619	751	896	1056	1228	1415	1615	1831	2059	2302
`SecB2A`: Alg. 17 L-11	2687	5052	7822	11123	14903	18992	23609	28572	34135	39948	46401	52927	60124	67524	75678
$\texttt{Compress}^d_{p,10}$	941	1858	2822	4169	5502	6949	8466	10298	12593	14672	16826	19154	21550	24062	26647
$\texttt{Compress}^d_{p,4}$	255	492	735	1082	1416	1776	2151	2603	3202	3716	4247	4820	5408	6023	6654
`CToBs` /`BsToC`	490	773	1056	1341	1624	1907	2190	2474	2760	3043	3327	3610	3894	4177	4461

Table 3: Performance of the S3 `Saber` implementation: number of clock cycles when running on a `STM32L4R5` and using the TRNG for masking randomness (32-bit randomness every 53 cycles). Reported numbers are in `kCycles`. The cost of the `CToBs` and `BsToC` operations is included in the gadgets that perform it, and the total time of their execution is also given separately.

d	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`Saber.CCAKEM.Dec`	5947	9324	13409	18395	24115	30653	37895	46086	54978	64685	75085	86379	98350	111141	124649
`Keccak-f[1600]`	3432	5412	7986	11155	14933	19298	24266	29837	35998	42756	50119	58095	66648	75801	85528
Gen $\hat{A}$	313	313	313	313	313	313	313	313	313	313	313	313	313	313	313
`NTT`	704	971	1238	1506	1773	2040	2307	2574	2841	3109	3376	3643	3910	4177	4444
$\texttt{Compress}^d_{2^{10},2^1}$	93	177	270	385	507	642	785	954	1130	1321	1514	1730	1946	2181	2420
`CBD`: Alg.17 L1-10	148	207	282	369	470	589	719	861	1019	1190	1374	1573	1786	2013	2254
`SecB2A`: Alg. 17 L-11	534	1012	1541	2196	2903	3695	4534	5510	6539	7649	8796	10056	11360	12756	14185
$\texttt{Compress}^d_{2^{13},2^{10}}$	340	677	1066	1565	2111	2729	3400	4194	5039	5957	6913	7970	9062	10232	11456
$\texttt{Compress}^d_{2^{10},2^4}$	86	167	259	377	502	645	798	980	1171	1379	1593	1832	2076	2338	2610
`CToBs` /`BsToC`	269	407	546	684	822	960	1098	1237	1375	1513	1651	1790	1928	2066	2204

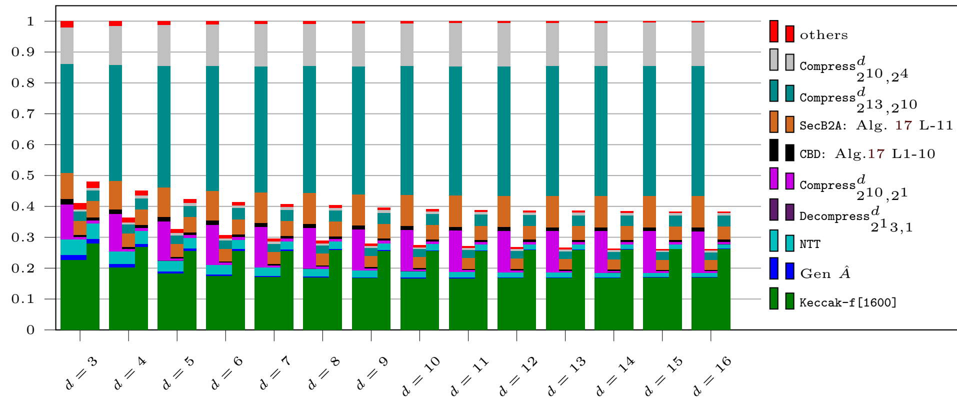


Figure 12: Performance comparison of `Saber` implementations: S1 (state-of-the-art gadgets, C-only, left) and S2 (new, C-only, middle) and S3 (new, C and assembly, right). Performance is normalized w.r.t. S1.

reduced from 78% down to 20%. In implementation S2 (for $d = 16$), 72% of the execution is spent in masked `Keccak-f[1600]`, 12% in $\texttt{SecB2A}^d_k$ and around 10% in $\texttt{SecA2B}^d_k$ to perform polynomial compression. Similarly to K3, we also propose a hardened `Saber` implementation called S3 using C and assembly. We report the cycle count of the S3 implementation in Table 3.

7 Conclusion

We begin our conclusion with the performance improvements. Thanks to very large performance improvement (about 20x) on arithmetic-to-Boolean masking conversion gadgets and to various smaller improvements (notably on Boolean-to-arithmetic conversions), our `Kyber768` implementation K2 based on new gadgets achieves a speedup of 1.8x over the implementation K1 based on state-of-the-art gadgets (see Figure 11). Similarly, we improve the performance of Saber by a factor 3x. The bottleneck of both new implementations of Kyber and Saber is the computation of masked Keccak, meaning that without improvement on the masked hash function, further speedup opportunities are limited. Eventually, we apply a best-effort methodology, using gadgets implemented in assembly language to harden our implementations against lower-order attacks: it induces an ≈ 1.6x overheads.

Next, we remark that in addition to improving performance in software by 1.3x to 25x, our bitsliced gadgets are very amenable to simple and efficient hardware implementations thanks to their bit-level structure, compared to tabled-based gadget or to other non-bitsliced gadgets. Additionally, we expect that the use of PINI as security property will help with security against glitches and transitions [CGLS21, CS21].

Finally, we note that most of the security proofs of this paper are simple: their sole argument is that a gadget is a composition of PINI sub-gadgets. We next discuss the takeaways of the more interesting security proofs. The proofs of Propositions 3 and 4 (arithmetic-to-Boolean conversion) rely on the new definition of gadget embedding (Definition 5 and Lemma 1), which can be viewed as an extension of trivial PINI composition to the composition of sub-gadgets with a mixed number of shares. Further, the proof of Proposition 5 (`SecB2AModp`) shows that one may securely "unmask" a sharing using only a `RefreshIOS`, instead of the FullRefresh which was used in previous works.

Acknowledgments. Gaëtan Cassiers is a Research Fellow of the Belgian Fund for Scientific Research (FNRS-F.R.S.). This research was supported by the Belgian Fund for

Scientific Research (F.R.S.-FNRS) through the Equipment Project SCALAB. This work has been funded in parts by the Walloon Region through the FEDER project USERMedia (convention number 501907-379156).

References

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 3:4, 2019.

[ABH+22] Melissa Azouaoui, Olivier Bronchain, Clément Hoffmann, Yulia Kuzovkova, Tobias Schneider, and François-Xavier Standaert. Systematic study of decryption and re-encryption leakage: The case of kyber. In *COSADE*, volume 13211 of *Lecture Notes in Computer Science*, pages 236–256. Springer, 2022.

[AP21] Alexandre Adomnicai and Thomas Peyrin. Fixslicing aes-like ciphers new bitsliced AES speed records on arm-cortex M and RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):402–425, 2021.

[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 *CCS*, pages 116–129. ACM, 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 *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 354–384. Springer, 2018.

[BBMD+19] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. Saber: Mod-lwr based kem. *NIST PQC Round*, 3, 2019.

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 616–648. Springer, 2016.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In *CHES*, volume 9813 of *Lecture Notes in Computer Science*, pages 23–39. Springer, 2016.

[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. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):334–359, 2021.

[BDK+21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel-resistant implementation of SABER. *ACM J. Emerg. Technol. Comput. Syst.*, 17(2):10:1–10:26, 2021.

[BDM+20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In *EUROCRYPT (3)*, volume 12107 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.

[BDPA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013.

[BGG+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *CARDIS*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014.

[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 Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):189–228, 2021.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First- and higher-order implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):173–214, 2021.

[Bih97] Eli Biham. A fast new DES implementation in software. In *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.

[BMP13] Joan Boyar, Philip Matthews, and René Peralta. Logic minimization techniques with applications to cryptology. *J. Cryptol.*, 26(2):280–312, 2013.

[BPO+20] Florian Bache, Clara Paglialonga, Tobias Oder, Tobias Schneider, and Tim Güneysu. High-speed masking for polynomial comparison in lattice-based kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):483–507, 2020.

[BS20] Olivier Bronchain and François-Xavier Standaert. Side-channel countermeasures' dissection and the limits of closed source security evaluations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):1–25, 2020.

[BS21] Olivier Bronchain and François-Xavier Standaert. Breaking masked implementations with many shares on 32-bit software platforms or when the security order does not matter. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):202–234, 2021.

[BWG+22] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. Provable secure software masking in the real-world. In *COSADE*, volume 13211 of *Lecture Notes in Computer Science*, pages 215–235. Springer, 2022.

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

[CGMZ21a] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order table-based conversion algorithms and masking lattice-based encryption. *IACR Cryptol. ePrint Arch.*, page 1314, 2021.

[CGMZ21b] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order polynomial comparison and masking lattice-based encryption. Cryptology ePrint Archive, Report 2021/1615, 2021. https://ia.cr/2021/1615.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In *FSE*, volume 9054 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2015.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *CHES*, volume 8731 of *Lecture Notes in Computer Science*, pages 188–205. Springer, 2014.

[CGZ20] Jean-Sébastien Coron, Aurélien Greuet, and Rina Zeitoun. Side-channel masking with pseudo-random generator. In *EUROCRYPT (3)*, volume 12107 of *Lecture Notes in Computer Science*, pages 342–375. Springer, 2020.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[CMG+] Jeremy Cooper, Elke De Mulder, Gilbert Goodwill, Josh Jaffe, Gary Kenworthy, and Pankaj Rohatgi. Test vector leakage assessment (tvla) methodology in practice. In *International Cryptographic Module Conference (ICMC 2013)*, page 13.

[Cor18] Jean-Sébastien Coron. Formal verification of side-channel countermeasures via elementary circuit transformations. In *ACNS*, volume 10892 of *Lecture Notes in Computer Science*, pages 65–82. Springer, 2018.

[CPRR13] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In *FSE*, volume 8424 of *Lecture Notes in Computer Science*, pages 410–424. Springer, 2013.

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

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

[DBV22] Jan-Pieter D'Anvers, Michiel Van Beirendonck, and Ingrid Verbauwhede. Revisiting higher-order masked comparison for lattice-based cryptography: Algorithms and bit-sliced implementations. *IACR Cryptol. ePrint Arch.*, page 110, 2022.

[DHP+22] Jan-Pieter D'Anvers, Daniel Heinz, Peter Pessl, Michiel Van Beirendonck, and Ingrid Verbauwhede. Higher-order masked ciphertext comparison for lattice-based cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):115–139, 2022.

[DZD+17] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In *CARDIS*, volume 10728 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2017.

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

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

[GHP+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on cpus. In *USENIX Security Symposium*, pages 1469–1468. USENIX Association, 2021.

[GJJ+11] Gilbert Goodwill, Benjamin Jun, 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.

[GLSV14] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. Ls-designs: Bitslice encryption for efficient masked software implementations. In *FSE*, volume 8540 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2014.

[GOP21] Si Gao, Elisabeth Oswald, and Dan Page. Reverse engineering the micro-architectural leakage features of a commercial processor. *IACR Cryptol. ePrint Arch.*, page 794, 2021.

[GPRV21] Dahmun Goudarzi, Thomas Prest, Matthieu Rivain, and Damien Vergnaud. Probing security through input-output separation and revisited quasilinear masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):599–640, 2021.

[GR16] Dahmun Goudarzi and Matthieu Rivain. On the multiplicative complexity of boolean functions and bitsliced higher-order masking. In *CHES*, volume 9813 of *Lecture Notes in Computer Science*, pages 457–478. Springer, 2016.

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

[Jr.13] Henry S. Warren Jr. *Hacker's Delight, Second Edition.* Pearson Education, 2013.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical cca2-secure and masked ring-lwe implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):142–174, 2018.

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *E-smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based PKE and kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):307–335, 2020.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-lwe implementation. In *CHES*, volume 9293 of *Lecture Notes in Computer Science*, pages 683–702. Springer, 2015.

[SM16] Tobias Schneider and Amir Moradi. Leakage assessment methodology - extended version. *J. Cryptogr. Eng.*, 6(2):85–99, 2016.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *Public Key Cryptography (2)*, volume 11443 of *Lecture Notes in Computer Science*, pages 534–564. Springer, 2019.

[SS16] Peter Schwabe and Ko Stoffelen. All the AES you need on cortex-m3 and M4. In *SAC*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2016.

[Sta18] François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations. In *CARDIS*, volume 11389 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2018.

[UXT+22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/em analysis on post-quantum kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):296–322, 2022.

[WO19] Carolyn Whitnall and Elisabeth Oswald. A critical analysis of ISO 17825 ('testing methods for the mitigation of non-invasive attack classes against cryptographic modules'). In *ASIACRYPT (3)*, volume 11923 of *Lecture Notes in Computer Science*, pages 256–284. Springer, 2019.

A Minimum number of AND gates for a k-bit adder

In the following, we name k-bit adder the Boolean function with $2k$ inputs and k coordinates that implements addition modulo 2^k when its inputs and outputs are viewed as k-bit binary representations of integers.

Proposition 6. *A Boolean circuit implementing a k-bit adder, When implemented with only 2-input AND, XOR and NOT gates, uses at least $k-1$ AND gates.*

Proof. We next prove the lower bound of $k-1$ AND gates for the addition of two k-bit integers. Let B_0 be the set of all linear and affine Boolean functions whose inputs are the $2k$ adder input bits, then by induction, c_i be the product of two elements a_i and b_i of B_i, and B_{i+1} be the span (in the vector space of Boolean functions) of $B_i \cup \{c_i\}$. We remark that for any (vectorial) Boolean function f that can be implemented with i 2-input AND gates and any number of XOR and NOT gates, there exists $(a_j)_{j=0,\ldots,i-1}$ and $(b_j)_{j=0,\ldots,i-1}$ such that f has all its coordinates in B_i.

Let D_i be the set of all the degrees of the functions in B_i. We have $D_0 = \{0, 1\}$, and for any i, $|D_{i+1}| \leq |D_i| + 1$, thus $|D_i| \leq i + 2$. The induction inequality can be proven

as follows: by construction, any function in B_{i+1} can be written as $\alpha_0 c_i \oplus \bigoplus_{j=1}^{k} \alpha_j f_i$ where all coefficients α belong to $\mathbb{F}_2$ and all f_j belong to B_i. Since B_i is a vector subspace, there exists $f \in B_i$ such that $f = \bigoplus_{j=1}^{k} \alpha_j f_j$. Therefore, all elements of $B_{i+1} \setminus B_i$ can be written as $c_i \oplus f$ for some $f \in B_i$. If the degree of c_i (denoted $\mathsf{deg}(c_i)$) does not belong to D_i, then $\mathsf{deg}(c_i \oplus f)$ is either $\mathsf{deg}(c_i)$ or $\mathsf{deg}(f)$, thus $D_{i+1} \subset D_i \cup \{\mathsf{deg}(c_i)\}$ and the inequality follows. Let us now assume that $\mathsf{deg}(c_i) \in D_i$. Let $f, f' \in B_i$ such that $\mathsf{deg}(c_i \oplus f) \neq \mathsf{deg}(c_i \oplus f')$, let $d = \max(\mathsf{deg}(c_i \oplus f), \mathsf{deg}(c_i \oplus f'))$ and assume by contradiction that both degrees do not belong do D_i. Therefore, $\mathsf{deg}(c_i \oplus f) \leq \mathsf{deg}(c_i)$ and the sets of terms in the algebraic normal forms (ANF) of c_i and f whose degree belong to $[\![\mathsf{deg}(c_i \oplus f), \mathsf{deg}(c_i)]\!]$ are equal. The same goes for f', and furthermore the sets of terms of degree d of f and f' are distinct. As a result, $\mathsf{deg}(f \oplus f') = d \in D_i$, which contradicts the hypothesis.

Numbering from 0 to $k-1$ (from least to most significant) the output bits of the adder, the bit i is a function of degree $i+1$ of the input bits. Therefore, the k-bit adder vectorial Boolean function has coordinates of all degrees in $[\![1, k]\!]$. Hence, the adder does not belong to any B_{k-2}: since $0 \in D_{k-2}$, $|D_{k-2} \setminus \{0\}| \leq k-1 < |[\![1, k]\!]|$, and therefore $D_{k-2} \not\subset [\![1, k]\!]$. We conclude that the k-bit adder cannot be implemented with $k-2$ AND gates (or less). □

B Generalized IOS refresh gadget

In this Section, we generalize the IOS refresh algorithm of [GPRV21] to deal with any number of shares (instead of only power-of-2). In a nutshell, we take the SNI refresh of [BCPZ16] and apply the same changes as [GPRV21] applied to the power-of-2 special case, resulting in Algorithm 18. The main difference with [GPRV21] is that the recursive call do not necessarily have the same number of shares, and that the last share is not re-randomized in the final layer when d is odd. For the sake of simplicity and consistency of notations, we specialize the gadget to Boolean masking, but the generalization of the gadget and the proofs to linear masking are trivial.

Algorithm 18 $\texttt{RefreshIOS}_k^d$

Input: Boolean sharing $\boldsymbol{x}^{B,k}$.
Output: Boolean sharing $\boldsymbol{y}^{B,k}$ such that $x = y$.

1: **if** $d = 1$ **then**
2: $\quad \boldsymbol{y}^{B,k} \leftarrow \boldsymbol{x}^{B,k}$
3: **else if** $d = 2$ **then**
4: $\quad r \overset{\$}{\leftarrow} \mathbb{F}_2^k$
5: $\quad \boldsymbol{y}_0^{B,k} \leftarrow \boldsymbol{x}_0^{B,k} \oplus r$
6: $\quad \boldsymbol{y}_1^{B,k} \leftarrow \boldsymbol{x}_1^{B,k} \oplus r$
7: **else**
8: $\quad \boldsymbol{z}_{[\![0,\lfloor d/2 \rfloor[\![}^{B,k} \leftarrow \texttt{RefreshIOS}_k^{\lfloor d/2 \rfloor}\left(\boldsymbol{x}_{[\![0,\lfloor d/2 \rfloor[\![}^{B,k}\right)$
9: $\quad \boldsymbol{z}_{[\![\lfloor d/2 \rfloor, d[\![}^{B,k} \leftarrow \texttt{RefreshIOS}_k^{d-\lfloor d/2 \rfloor}\left(\boldsymbol{x}_{[\![\lfloor d/2 \rfloor, d[\![}^{B,k}\right)$
10: $\quad$ **for** $i \in [\![0, \lfloor d/2 \rfloor[\![$ **do**
11: $\quad\quad r_i \overset{\$}{\leftarrow} \mathbb{F}_2^k$
12: $\quad\quad \boldsymbol{y}_i^{B,k} \leftarrow \boldsymbol{z}_i^{B,k} \oplus r_i$
13: $\quad\quad \boldsymbol{y}_{\lfloor d/2 \rfloor + i}^{B,k} \leftarrow \boldsymbol{z}_{\lfloor d/2 \rfloor + i}^{B,k} \oplus r_i$
14: $\quad$ **if** $d \mod 2 = 1$ **then**
15: $\quad\quad \boldsymbol{y}_{d-1}^{B,k} \leftarrow \boldsymbol{z}_{d-1}^{B,k}$

Security proof We now prove that Algorithm 18 is input-output separative for $d \geq 2$. Since the proof is very similar to the original proof of [GPRV21], we only mention the few significant differences. Throughout the proof we denote $L = [\![0, \lfloor d/2 \rfloor[\![$ and $H = [\![\lfloor d/2 \rfloor, d[\![$. Furthermore, we replace $d/2$ by $\lfloor d/2 \rfloor$ everywhere and adapt the indices (from 0 to $d-1$ instead of 1 to n).

Uniformity The proof is still by induction, and the base cases are $d = 1$ and $d = 2$. The proof for $d = 2$ is unchanged, while the case $d = 1$ is trivial since there is only one admissible output sharing for a fixed input. Next, for $d \geq 3$, the original induction proof still holds.

IOS The case $d = 1$ is trivial: the full input and output sharings are known if there is at least one probe. The case $d = 2$ is not changed. The induction case only requires changes when d is odd, in order to handle the share $\boldsymbol{z}_{d-1}^{B,k}$ (wlog we assume that $\boldsymbol{y}_{d-1}^{B,k}$ is not probed): we define $\mathcal{V}_{d-1}$ as $\{\boldsymbol{z}_{d-1}^{B,k}\}$ and add $d-1$ for J if $\mathcal{V}_{d-1}$ is not empty, and in that case the simulator sets $\boldsymbol{z}_{d-1}^{B,k} = \boldsymbol{y}_{d-1}^{B,k}$. The simulation then proceeds as in the original proof.

Re-ordering operations The execution of Algorithm 18 can be re-written in the following manner. Let first L_d be well a well-chosen list of pairs (x_i, y_i) (formally, $L_d \in \left([\![0, d[\![^2\right)^*$). Then, for each (x_i, y_i) in L_d, generate $r_i \in \mathbb{F}_2^k$ and update the shares with index x_i and y_i by XORing r_i to them. We remark that L_d may be shuffled without impacting the set of internal variables if we preserve the relative order of any pairs (x_i, y_i) and (x_j, y_j) such that $\{x_i, y_i\} \cap \{x_j, y_j\} \neq \emptyset$. This gives freedom in the implementation to choose the order that minimizes control flow and spilling (i.e., copies from registers to the RAM) overheads.