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 strongly speed up to masked implementations of symmetric primitives, it has never been used in higher-order arithmetic-to-boolean and boolean-to-arithmetic masking conversion gadgets. 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 $x = x_0 + \cdots + x_{d-1} \mod p$

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 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 a to 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 [CGMZ21b]. 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, CGMZ21b], from which a B2A algorithm for 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, CGMZ21a].

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 the large number of 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 and implementations of Coron et al. [CGMZ21b], 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 1.84 over K1, and up to 8.7 over the best reported performance in the state-of-the-art on an embedded platform [BGR+21]. Similarly S2 achieves a speedup to 3x over S1. In both K2 and S2, the execution time is dominated by hashing respectively by 50% for Kyber and 72% for Saber.

Finally, exploiting the PQM4 [KRSS] framework, we release an easily reusable and extensible implementation of `Kyber768`.[2] These implementations are based on the same gadgets as K2 and S2.

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. Finally, we describe our Kyber and Saber implementations and measure their performance in Section 5.

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.

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.

[1] Available at https://github.com/uclcrypto/pqm4_masked/files/8049274/implems.zip.
[2] Available at https://github.com/uclcrypto/pqm4_masked.

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

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

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 to compute bitwise AND of boolean-shared values at a randomness and computational cost $\mathcal{O}(d^2)$. 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 which are composable. The two following definitions were introduced in [BBD+16].

Definition 1 (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 2 (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 usage of refresh gadgets, which may significantly increase the computationnal and randomness cost. More recently, Cassiers and Standaert [CS20] introduced a new definition which allows to remove those refresh gadgets.

Definition 3 (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., 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 multiplication gadget ([CS20], Algorithm 2) with a similar cost as 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 sharings, 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. 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). Using adequate refreshing [BBE+18], the `SecAdd` gadget is made $(d-1)$-NI.

Algorithm 1 $\texttt{SecAddModp}_k^d$ 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: $\boldsymbol{p}^{B,k+1} \leftarrow \left(2^k - p, 0, \ldots, 0\right)$
2: $\boldsymbol{s}^{B,k+1} \leftarrow \texttt{SecAdd}_{k+1}^d\left(\boldsymbol{x}^{B,k}, \boldsymbol{y}^{B,k}\right)$ ▷ Left 0-extend the input sharings by 1 bit.
3: $\boldsymbol{s'}^{B,k+1} \leftarrow \texttt{SecAdd}_{k+1}^d\left(\boldsymbol{s}^{B,k+1}, \boldsymbol{p}^{B,k+1}\right)$
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: $\boldsymbol{c}^{B,k} \leftarrow \texttt{BitCopyMask}_k^d\left(\boldsymbol{c}^{B,1}, 2^k - 1\right)$ ▷ 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: $\boldsymbol{z}^{B,k} \leftarrow \texttt{SecAnd}_k^d\left(\boldsymbol{s}^{B,k+1}[[\![0, k[\![], \boldsymbol{c}^{B,k}\right) \oplus^{\mathtt{B}} \texttt{SecAnd}_k^d\left(\boldsymbol{s'}^{B,k+1}[[\![0, k[\![], \boldsymbol{c'}^{B,1}\right)$ ▷ MUX

Next, we consider the `SecAddModp` gadget which performs the addition modulo p. The construction of Algorithm 1 (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.

2.4 Arithmetic-to-boolean masking conversion

Coron et al. [CGV14] introduced a simple way to convert from arithmetic to boolean masking (SecA2BModp): mask with boolean masking each arithmetic share, then perform the addition modulo p of the resulting sharings. Remarking that the addition of d' arithmetic shares can be securely masked uinsg d'-shares boolean masking instead of d, and organizing the $d-1$ additions to perform in a binary tree (halving the number of shares at each layer) leads to Algorithm 2 (from [SPOG19]), which 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 [CGMZ21b].

Algorithm 2 $\texttt{SecA2BModp}_k^d$ 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, 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 3). This gadget is $(d-1)$-SNI.[3]

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 (see Algorithm 4). 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.

Finally, Coron et al. [CGMZ21b] introduced recently another conversion algorithm. This algorithm also performs k single-bit conversions, but the single-bit conversion is a table-based gadget.

[3] 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 3 $\texttt{SecB2AModp}_k^d$ 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: $\boldsymbol{z}_{d-1}^{A_k} \leftarrow \texttt{UnMask}_k^{d-1} \left(\texttt{FullRefresh}_k^{d-1} \left(\boldsymbol{b}^{B,k}\right)\right)$

Algorithm 4 $\texttt{SecB2AModp}_k^d$ based single bit conversion (from [SPOG19])

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: $\boldsymbol{z}^{A_p} \leftarrow \texttt{SecB2AModpBit}_k^d \left(\boldsymbol{x}^{B,k}[k-1]\right)$
2: **for** $i = k-2$ down to 0 **do**
3: $\quad \boldsymbol{b}^{A_p} \leftarrow \texttt{SecB2AModpBit}_k^d \left(\boldsymbol{x}^{B,k}[i]\right)$
4: $\quad \boldsymbol{z}^{A_p} \leftarrow 2 \cdot \boldsymbol{z}^{A_p} +^{\mathtt{A}} \boldsymbol{b}^{A_p} \mod p$

3 New gadgets

As we already mentioned in introduction, our starting point is the observation that high-level cryptographic algorithms such as Kyber have large data parallelism, hence they may benefit from bitsliced implementations for the boolean sharings (while staying non-bitsliced for the arithmetic sharings). We therefore introduce algorithms that work on 1-bit words, and which are therefore well-suited to bitslicing. As main elementary gadgets, we use $\oplus^{\mathtt{B}}$ and PINI `SecAnd`, where the `SecAnd` is more expensive than $\oplus^{\mathtt{B}}$ ($\mathcal{O}\left(d^2\right)$ vs. $\mathcal{O}\left(d\right)$).

The complexity of our algorithms is measured in single-bit operations, which may be amortized over the bit width w of the processor thanks to bitslicing. In order to compare to the algorithms of Section 2 which required k-bit words, the complexity should be divided by k to consider k-bit bitslicing (in practice, if $w > k$, the gain is even larger).

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 c computes $a := x \oplus y$, then outputs $(a \oplus c, x \oplus a \cdot (x \oplus c))$. 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 most significant bits addition), 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 $\texttt{SecFullAdder}^d$ 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: $\boldsymbol{w}^{B,2}[1] \leftarrow \boldsymbol{x}^{B,1} \oplus^{\mathsf{B}} \texttt{SecAnd}_1^d\left(\boldsymbol{a}^{B,1}, \boldsymbol{x}^{B,1} \oplus^{\mathsf{B}} \boldsymbol{z}^{B,1}\right)$ ▷ PINI SecAnd

Algorithm 6 $\texttt{SecAdd}_k^d$ 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: $\quad \boldsymbol{t}^{B,2} \leftarrow \texttt{SecFullAdder}^d\left(\boldsymbol{x}^{B,k}[i], \boldsymbol{y}^{B,k}[i], \boldsymbol{c}^{B,1}\right)$ ▷ 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 1 to use Algorithm 6 as SecAdd. On top of this adaptation, we remark that the MUX in Algorithm 1 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 $\texttt{SecAdd}_k^d$ (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 2). Namely, we also use a recursive structure

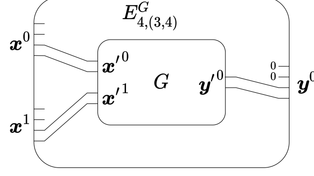


Figure 1: Example of 2-share to 4-share gadget embedding.

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 $\texttt{SecA2B}^d_k$ 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: $\quad\boldsymbol{y}^{B,k} \leftarrow \texttt{SecA2B}^{\lfloor d/2 \rfloor}_k \left(\boldsymbol{x}^{A_{2^k}}[[\![0, \lfloor d/2 \rfloor[\![]\right)$ ▷ $\lfloor d/2 \rfloor$ sharing.
5: $\quad\boldsymbol{y'}^{B,k} \leftarrow \texttt{SecA2B}^{d-\lfloor d/2 \rfloor}_k \left(\boldsymbol{x}^{A_{2^k}}[[\![\lfloor d/2 \rfloor, d[\![]\right)$ ▷ $d - \lfloor d/2 \rfloor$ sharing.
6: $\quad\boldsymbol{s}^{B,k} \leftarrow \left(\boldsymbol{y}^{B,k}_0, \boldsymbol{y}^{B,k}_1, \ldots, \boldsymbol{y}^{B,k}_{\lfloor d/2 \rfloor - 1}, 0, \ldots, 0\right)$ ▷ Expand to d shares.
7: $\quad\boldsymbol{s'}^{B,k} \leftarrow \left(0, \ldots, 0, \boldsymbol{y'}^{B,k}_{\lfloor d/2 \rfloor}, \ldots, \boldsymbol{y'}^{B,k}_{d-1},\right)$ ▷ Expand to d shares.
8: $\quad\boldsymbol{z}^{B,k} \leftarrow \texttt{SecAdd}^d_k \left(\boldsymbol{s}^{B,k}, \boldsymbol{s'}^{B,k}\right)$ ▷ 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 number 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 4 (Gadget embedding)**.** Let G be a d'-share gadget and let $m \in [\![0, d[\![^{d'}$ (with $d \geq d'$) have unique components ($m_i \neq m_j$ for all i, j). The d-share embedding of G with mapping m is the d-share gadget denoted $E^G_{d,m}$ and built as follows. $E^G_{d,m}$ has the same number of input and output sharings as G (with a one-to-one correspondence), and instantiates G internally. Each input sharing $\boldsymbol{x'}^j$ of G is generated as $\left(\boldsymbol{x}^j_{m_i}\right)_{i=0,\ldots,d'-1}$, where $\boldsymbol{x}^j$ is the input sharing of $E^G_{d,m}$ that corresponds to $\boldsymbol{x'}^j$. For each output sharing $\boldsymbol{y}^j$ of $E^G_{d,m}$, we set $\boldsymbol{y}^j_{m_i} = \boldsymbol{y'}^j_i$ for all $i \in [\![0, d'[\![$ where $\boldsymbol{y'}^j$ is the output sharing of G that corresponds to $\boldsymbol{y}^j$. For all $i \in [\![0, d[\![$ that do not appear in m, we set $\boldsymbol{y}^j_i = 0$.

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 of all, 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 to send the inputs required to the simulator of G, hence to simulate 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 sub-gadgets: $E^{\texttt{SecA2B}_k^{\lfloor d/2 \rfloor}}_{d,(0,\dots,\lfloor d/2 \rfloor - 1)}$ (which computes $\boldsymbol{s}^{B,k}$ from $\boldsymbol{x}^{A_{2^k}}$), $E^{\texttt{SecA2B}_k^{d-\lfloor d/2 \rfloor}}_{d,(\lfloor d/2 \rfloor,\dots,d-1)}$ (which computes $\boldsymbol{s'}^{B,k}$ from $\boldsymbol{x}^{A_{2^k}}$) and $\texttt{SecAdd}_k^d$ (which computes $\boldsymbol{z}^{B,k}$ from $\boldsymbol{s}^{B,k}$ and $\boldsymbol{s'}^{B,k}$). Since $\texttt{SecA2B}_k^{\lfloor d/2 \rfloor}$ and $\texttt{SecA2B}_k^{d-\lfloor d/2 \rfloor}$ are PINI (by induction on d), their embeddings are PINI (by Lemma 1). Furthermore, $\texttt{SecAdd}_k^d$ 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).[4] 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.

Proposition 4. *Algorithm 9 is PINI.*

Proof. The proof is almost identical to the proof of Algorithm 9. 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 ($\texttt{SecA2BModp}_k^{\lfloor d/2 \rfloor}$ and $\texttt{SecAdd}_{k+1}^{\lfloor d/2 \rfloor}$), hence it is PINI, and the embedding is PINI. Next, the d-share embedding of $\texttt{SecA2BModp}_k^{d-\lfloor d/2 \rfloor}$ is PINI, as well as the other d-share sub-gadgets (`SecAdd`, `BitCopyMask`). □

3.5 `SecB2AModp`: Bitslice boolean-to-arithmetic conversion modulo p

We now adapt in Algorithm 10 the `SecB2AModp` from [BBE+18] (Algorithm 3) to use our new `SecA2BModp` and `SecAddModp` algorithms.[5] Furthermore, we replace the refresh gadget

[4] Another solution would be to use the compression algorithm (`HOCompress`) from [CGMZ21a] 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.

[5] The conversion modulo 2^k $\texttt{SecB2A}_k^d$ can be implemented following Algorithm 10, using the new `SecA2B` and `SecAdd` instead of `SecA2BModp` and `SecAddModp`. The security proof is not changed.

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

to reduce its cost (from $\mathcal{O}\left(d^2\right)$ to $\mathcal{O}\left(d \log d\right)$). The new refresh gadget is input-output separative (IOS) and is described in Algorithm 17 (Appendix B). It is a generalization (to handle any value of d, and not only power of 2) of the one introduced in [GPRV21].

Definition 5 (IOS ([GPRV21], adapted))**.** A refresh gadget G is t-IOS if it is uniform[6] and if for every admissible pair $(\boldsymbol{x}, \boldsymbol{y})$[7] and every set of probes P with $|P| \leq t$, there exists a simulator which can perfectly simulate the probes 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.

Algorithm 10 $\texttt{SecB2AModp}_k^d$ 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: $\quad \boldsymbol{z}_i^{A_p} \overset{\$}{\leftarrow} \mathbb{Z}_p$
3: $\quad \boldsymbol{z'}_i^{A_p} \leftarrow p - \boldsymbol{z}_i^{A_p}$
4: $\boldsymbol{z'}_{d-1}^{A_p} \leftarrow 0$
5: $\boldsymbol{a}^{B,k} \leftarrow \texttt{SecA2BModp}_k^d\left(\boldsymbol{z'}^{A_p}\right)$ ▷ Use Algorithm 9.
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 17.
8: $\boldsymbol{z}_{d-1}^{A_p} \leftarrow \texttt{UnMask}_k^d\left(\boldsymbol{c}^{B,k}\right)$ ▷ XOR all shares together.

Proposition 5. *Algorithm 10 is PINI.*

[6]Its output is an uniformly distributed sharing of x for any fixed input sharing $\boldsymbol{x}$. This implies that the distribution of the output sharing $\boldsymbol{y}$ is independent of $\boldsymbol{x}$, conditioned on x.
[7]The pair $(\boldsymbol{x}, \boldsymbol{y})$ is admissible if there exists an assignment for the randomness used is G such that, on input $\boldsymbol{x}$, the output of G is $\boldsymbol{y}$.

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` (Proposition 7, Appendix B). The probes in these gadgets can thus be simulated by knowing at most $|P|$ shares of $\boldsymbol{x}^{B,k}$ 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.

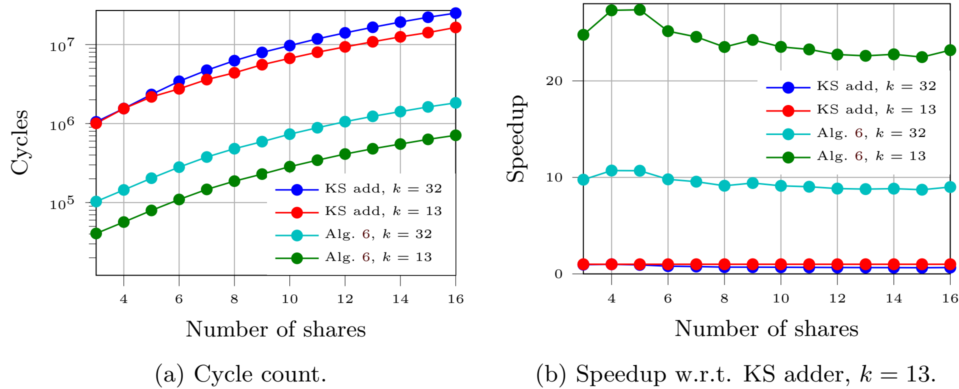
The PINI simulator for `SecB2AModp` first selects randomly $\boldsymbol{z}_{d-1}^{A_p}$, then it generates a uniformly random sharing $\boldsymbol{c}^{B,k}$ of $\boldsymbol{z}_{d-1}^{A_p}$, from which it can simulate any probe in P_u. Next, using the IOS simulator, it determines the set of share indexes 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 $B^* = B_r \cup B_0 \cup (B \setminus \{d-1\})$, an 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 10. 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. □

4 Gadgets performance

In this Section, we will compare the performance of each of our new gadgets to the state of the art gadgets implementing the same feature (ignoring the difference in security property).



(a) Cycle count.

(b) Speedup w.r.t. KS adder, $k = 13$.

Figure 2: Performance comparison of `SecAdd` implementations.

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

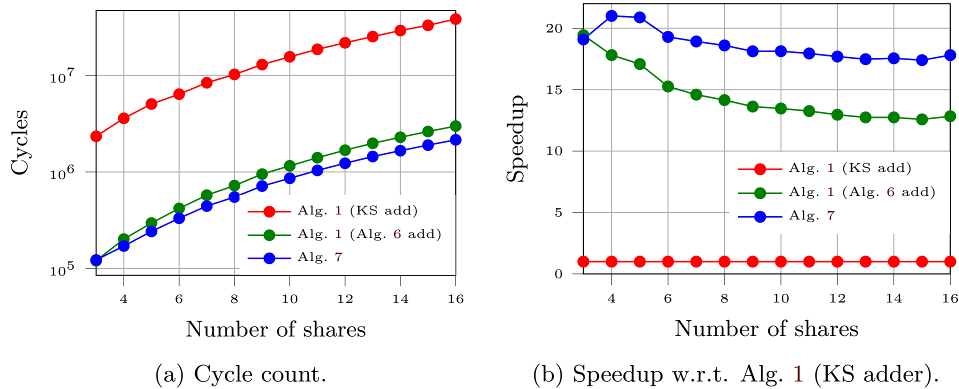
We implement all the gadgets in C and measure the performance on a ARM Cortex-M4, a 32-bit micro-controller. Concretely, we target the NUCLEO-L4R5ZI development board which is used by the `PQM4` benchmarking library [KRSS].[8] The firmware and clock tree are the same as the one used in `PQM4`, as well as the performance measurement method (i.e., we use the cycle accurate counter `DWT_CYCNT`). In order to generate randomness, we use the dedicated TRNG. According to the datasheet, a fresh random 32-bit word is available in a dedicated register every 40 cycles. In practice, access to this TRNG is wrapped with checks to ensure that randomness is not used twice, which increases the cost of loading 32 bits of randomness to 53 cycles. In order to generate uniform randomness in $\mathbb{Z}_p$, we generate two random elements based on 32 random bits and use rejection sampling. Overall, one random over $\mathbb{Z}_p$ is available every 26.5 cycles. 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. Finally, we used naive implementation of the recursive algorithms, only forcing inlining at a few places where the control flow overhead was identified as a bottleneck.

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 gadgets 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 $\mathcal{O}(\log(k)d^2)$ word operations, and the Algorithm 6 which has a complexity of $\mathcal{O}(kd^2)$ bit operations. First, we observe that Algorithm 6 requires less cycles than the KS adder. For $k = 13$, Algorithm 6 is about 23 times faster and for $k = 32$, the speedup is about 9x. Interestingly, Algorithm 6 is faster than the KS adder despite its higher asymptotic complexity, thanks to the bitslicing gain.

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



(a) Cycle count.

(b) Speedup w.r.t. Alg. 1 (KS adder).

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

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 1 when using the KS adder (not bitsliced), (ii) Algorithm 1 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 (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 halve the cost of the MUX by replacing it with a `SecAdd`, which gives a speedup of about 1.3x over (ii).

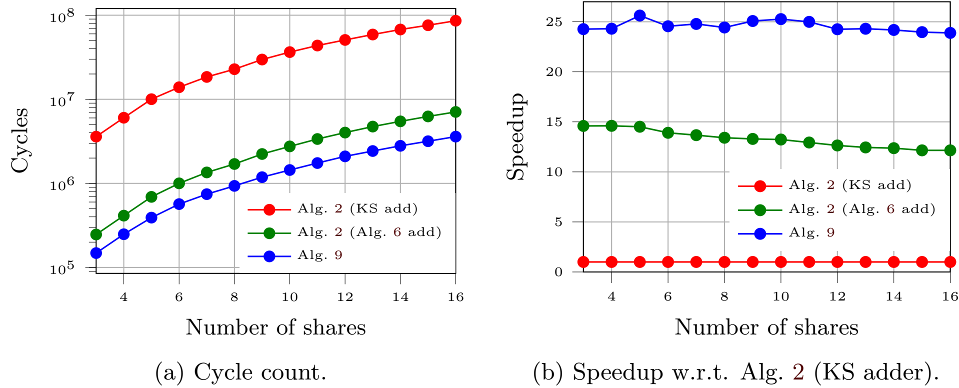
4.4 Performance of $\texttt{SecA2BModp}_k^d$

Similarly, we compare the performance of $\texttt{SecA2BModp}_k^d$ implementations in Figure 4. The reference implementation (i) is Algorithm 2 (with KS adder). We compare it to (ii) a modified Algorithm 2 using the bitsliced adder (Algorithm 7), and to (iii) the new Algorithm 9. 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.

4.5 Performance of $\texttt{SecB2AModp}_k^d$

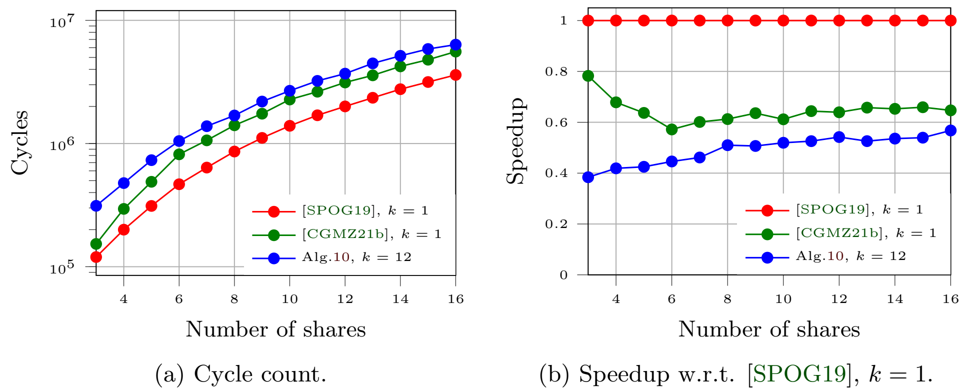
Finally, we compare in Figure 5 the performance of various implementations of `SecB2AModp`. We consider as state of the art the algorithms from [SPOG19] and [CGMZ21b] which both implement $\texttt{SecB2AModp}_k^d$ from single-bit conversions using Algorithm 4. As a result, their computational cost is 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 [CGMZ21b], Table 4).

Our bitsliced conversion gadget (Algorithm 10) 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



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

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



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

Figure 5: Performance comparison of SecB2AModp implementations.

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 10 grows linearly with k.

5 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. [CGMZ21b, CGMZ21a]. Eventually, we applied the same methodology to Saber and report the results.

5.1 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 11 where the ciphertext c is decrypted with $\texttt{CPAPKE.Dec}(\cdot)$ to obtain the message m'. This message is then re-encrypted with $\texttt{CPAPKE.Enc}(\cdot)$ 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

Algorithm 11 $\texttt{Kyber.CCAKEM.Dec}(c, sk)$

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

1: $m' := \texttt{Kyber.CPAPKE.Dec}(\hat{\boldsymbol{s}}, c)$
2: $(\bar{K}', \sigma') := \texttt{G}^d(m'||\texttt{H}(pk))$
3: $(c'_u, c'_v) := \texttt{Kyber.CPAPKE.Enc}(pk, m', \sigma')$
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 12 $\texttt{Kyber.CPAPKE.Dec}(\hat{\boldsymbol{s}},c)$

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

that no masking is required[9], 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.

$\texttt{Kyber.CPAPKE}$ 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 $\texttt{NTT}$ 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 $\texttt{NTT}$) are performed with arithmetic masking and can be applied share-by-share, hence with a linear complexity. Finally, each coefficient (in $\mathbb{Z}_p$) of the resulting polynomial is compressed to a single bit, which represents the rounding to $\lceil p/2 \rceil$ or 0. We detail the masked implementation of $\texttt{Compress}^d_{p,c}$ in Algorithm 14.

Finally, $\texttt{Kyber.CPAPKE.Enc}$ is described in Algorithm 13. This algorithm starts by generating $2l + 1$ noise polynomials (Line 2-4) whose coefficients follow a central binomial distribution ($\texttt{CBD}$, see Algorithm 16) with parameter η, such that they belong to $[\![-\eta, \eta]\!]$. The $\texttt{CBD}$ takes as input a pseudo-random string of bits which is computed as the hash $\texttt{PRF}$ of the random seed σ and a nonce. Next, the noise ($\boldsymbol{e}_1$ and $\boldsymbol{e}_2$) is added to the product 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 15) 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 (Line 8-9).

Algorithm 13 $\texttt{Kyber.CPAPKE.Enc}(pk,m,\sigma)$

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: $c_v := \texttt{Compress}^d_{p,d_v}(v)$ ▷ $d_v = 4$

[9] 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.

Algorithm 14 $\mathtt{Compress}^d_{q,c}$, from [CGMZ21a]

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

Algorithm 15 $\mathtt{Decompress}^d_{q,1}$

Input: d shares Boolean sharing $\boldsymbol{x}^{B,1}$, integer p such that $p < 2^k$ and $x \in [\![0, p[\![$.

Output: d shares arithmetic sharing $\boldsymbol{z}^{A_p}$ such that $z = x \cdot (p/2) \mod p$.

1: $\boldsymbol{y}^{A_p} \leftarrow \mathtt{SecB2AModpBit}^d_k\left(\boldsymbol{x}^{B,1}\right)$
2: $\boldsymbol{x}^{A_p} \leftarrow (p/2) \cdot \boldsymbol{y}^{A_p} \mod p$

5.2 Implementations: K1 and K2

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

The implementations K1 and K2 based on the one developed by Coron et al. [CGMZ21b, CGMZ21a]. We changed the NTT and arithmetic operations with the optimized version for Cortex-M4 from [KRSS]. We also keep a single noise polynomial in memory at any time in Algorithm 13 to reduce the stack usage. Implementation K1 relies on the original gadget provided by Coron et al., but we replaced some of them with our implementations of the gadgets of [SPOG19] which perform better. Implementation K2 leverages our new bitsliced gadgets.

$\mathtt{Compress}^d_{p,c}$. 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 [CGMZ21a] (Algorithm 14) for the implementation of $\mathtt{Compress}^d_k$ in K2. Our $\mathtt{Compress}^d_k$ 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. After compression, 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 [CGMZ21a]. 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 14 without bitslicing. Eventually, the $\mathtt{Compress}^d_{p,1}$ in K1 is performed with the table-based conversion from [CGMZ21b].

$\mathtt{Decompress}^d_{p,1}$. Decompress is mapping a single bit to $\lceil p/2 \rceil$ or 0, and we implement it with Algorithm 15, 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

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

generic $\texttt{SecB2AModp}_k^d$ for this purpose since, as shown in Figure 5, it is slower by a factor 2 for single-bit conversions.

$\texttt{CBD}_2^d$. 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 $\texttt{SecAdd}_k^d$), then converts it to arithmetic masking using Algorithm 4 and their $\texttt{SecB2AModpBit}_k^d$, and finally subtracts η. For K2, we use Algorithm 16, 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 $\texttt{CBD}_\eta^d$ 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].

Algorithm 16 $\texttt{CBD}_\eta^d$ 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 $z = \texttt{HW}(a) - \texttt{HW}(b) \mod p$.

1: $\left(\boldsymbol{s}^{B,2\eta}[[\![0,\eta[\![], \boldsymbol{s}^{B,2\eta}[[\![\eta,2\eta[\![]\right) \leftarrow \left(\boldsymbol{a}^{B,\eta}, \neg \boldsymbol{b}^{B,\eta}\right)$ ▷ $\texttt{HW}(s) = \texttt{HW}(a) - \texttt{HW}(b) + \eta$
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: $\quad\quad \boldsymbol{t}^{B,2} \leftarrow \texttt{SecFullAdder}^d\left(\boldsymbol{s}^{B,2\eta}[2j], \boldsymbol{s}^{B,2\eta}[2j+1], \boldsymbol{x}^{B,1}\right)$ ▷ 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: $\boldsymbol{z}^{A_p} \leftarrow \texttt{SecB2AModp}_d^p\left(\boldsymbol{y}^{B,k}\right)$ ▷ Algorithm 10, $y = \texttt{HW}(a) - \texttt{HW}(b) + \eta$
12: $\boldsymbol{z}_0^{A_p} \leftarrow \boldsymbol{z}_0^{A_p} - \eta \mod p$

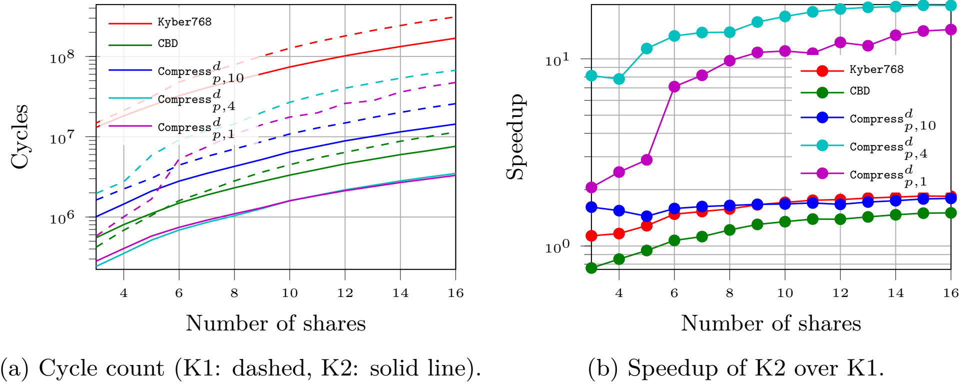
G, H and PRF. All the hash functions used are based on SHA-3 and therefore all use the Keccak-f[1600] permutation. Concretely, we use the masked Keccak-f[1600] provided by Coron et al. and a SNI SecAnd for K1, while we use a PINI SecAnd for K2.

Probing security The Kyber implementation K2 is a composition of PINI gadgets, hence it is PINI itself, and therefore probing secure.

5.3 Kyber performance

We show in Figure 6 the performance of the top-level masked components of the Kyber implementations K1 (based on state of the art gadgets) and K2 (new).

First, we remark that $\texttt{Compress}_{p,1}^d$ in K2 achieves a speedup of more than 10x over K1, showing that Algorithm 14 (bitsliced) is faster than the table-based approach by Coron et al. For $\texttt{Compress}_{p,4}^d$, the speedup (about 20x) is exactly the one of our new $\texttt{SecA2B}_k^d$ 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 [CGMZ21a], which limits the speedup of K2 to 1.8x. Finally, regarding the CBD (which includes the boolean to arithmetic masking conversion of the noise), the gain in performances is directly dependent on the gain for $\texttt{SecB2AModp}_k^d$ that we discussed in Figure 5, since this gadget is the bottleneck.



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

(b) Speedup of K2 over K1.

Figure 6: Comparison of the performance of various components of `Kyber768`: implementations K1 (state of the art gadgets) and K2 (new).

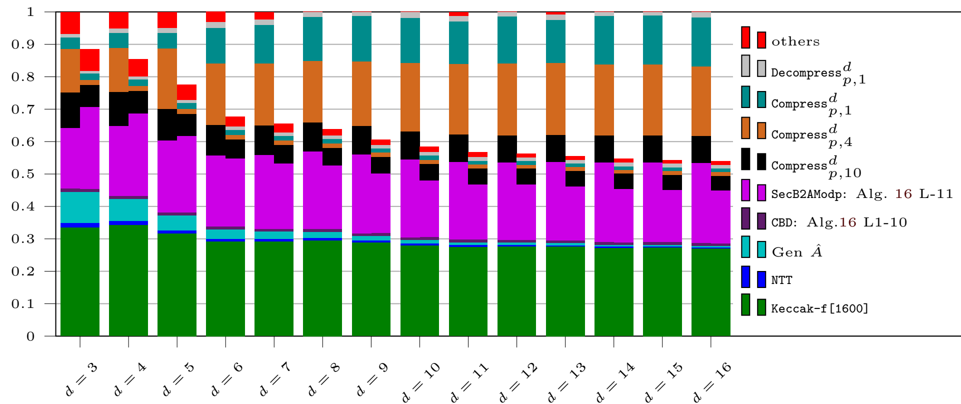


Figure 7: Performance comparison of `Kyber768` implementations: K1 (state of the art gadgets, left) and K2 (new, right). Performance is normalized w.r.t. K1. For better performances and small d, users should swap $\texttt{SecB2AModp}_k^d$ conversions.

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 7, 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 $\texttt{SecB2AModp}_k^d$ conversion of the noise polynomials (in Algorithm 16) for about 30% of the cycles.

5.4 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 the implementation S1 starting from the one of Coron et al. [CGMZ21a], adapting and optimizing it for the Cortex-M4, and finally integrating the best state of

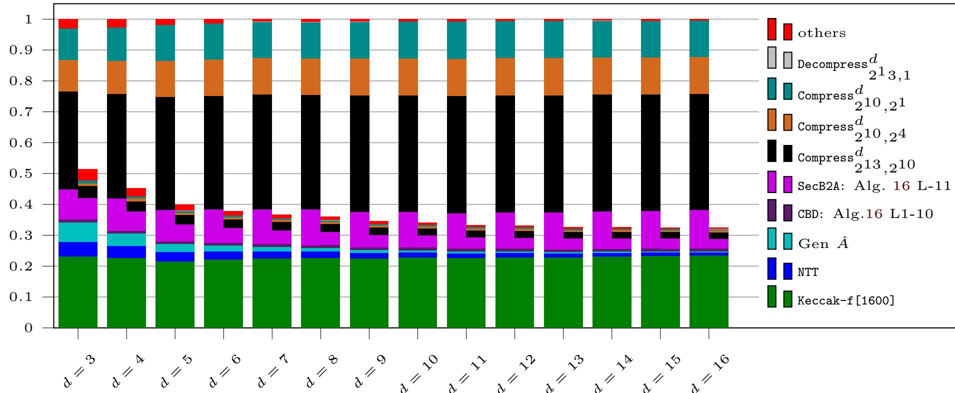


Figure 8: Performance comparison of `Saber` implementations: S1 (state of the art gadgets, left) and S2 (new, right). Performance is normalized w.r.t. S1.

the art gadgets.[11] We then developed implementation S2, replacing the `SecA2B`, `SecB2A` and `CBD` implementations by our new bitslice algorithms. 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 8. Concretely, our new gadgets reduces the execution time of the conversions by a large factor such that the fraction of runtime dedicated to them is 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.

6 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 7). 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.

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 for 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 4 and Lemma 1), which can be viewed as an extension of trivial PINI composition to the composition of sub-gadgets with mixed number of shares. Futher, 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.

[11] We only replaced the `CBD` with the from [SPOG19].

Acknowledgments. Gaëtan Cassiers is a Research Fellow of the Belgian Fund for Scientific Research (FNRS-F.R.S.). 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. Cryptology ePrint Archive, Report 2022/036, 2022. https://ia.cr/2022/036.

[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.

[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.

[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.

[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.

[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 polynomial comparison and masking lattice-based encryption. Cryptology ePrint Archive, Report 2021/1615, 2021. https://ia.cr/2021/1615.

[CGMZ21b] 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.

[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.

[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.

[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.

[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.

[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.

[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`.

[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.

[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.

[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.

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 $(a_j)_{j=0,\dots,i-1}$ and $(b_j)_{j=0,\dots,i-1}$ 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 17. 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 17 $\texttt{RefreshIOS}_k^d$

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: $\quad r \xleftarrow{\$} \mathbb{F}_2^k$
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: $\quad\quad r_i \xleftarrow{\$} \mathbb{F}_2^k$
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 17 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 $\boldsymbol{z}_{d-1}^{B,k} = \boldsymbol{y}_{d-1}^{B,k}$. Simulation then proceeds as in the original proof.

Re-ordering operations The execution of Algorithm 17 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.

PINI Finally, we prove that *Algorithm* 17 is PINI.

Proposition 7. *Algorithm 17 is PINI.*

Proof. Algorithm 17 it can be partitioned in one randomness generation circuit G, and d sub-circuits that each take as input randomness (produced by G) and one input share of `RefreshIOS`, and output the corresponding output share. The simulator can simply simulate the randomness generation circuit, as well as any sub-circuit in which there is a probe (or for which the output must be simulated). □