One-Hot Conversion: Towards Faster Table-based A2B Conversion

Jan-Pieter D'Anvers[0000−0001−9675−7988]

imec-COSIC KU Leuven, Kasteelpark Arenberg 10 - bus 2452, 3001 Leuven, Belgium
`firstname.lastname@esat.kuleuven.be`

Abstract. Arithmetic to Boolean masking (A2B) conversion is a crucial technique in the masking of lattice-based post-quantum cryptography. It is also a crucial part of building a masked comparison which is one of the hardest to mask building blocks for active secure lattice-based encryption. We first present a new method, called one-hot conversion, to efficiently convert from higher-order arithmetic masking to Boolean masking using a variant of the higher-order table-based conversion of Coron et al. Secondly, we specialize our method to perform arithmetic to 1-bit Boolean functions. Our one-hot function can be applied to masking lattice-based encryption building blocks such as masked comparison or to determine the most significant bit of an arithmetically masked variable. In our benchmarks on a Cortex M4 processor, a speedup of 15 times is achieved over state-of-the-art table-based A2B conversions, bringing table-based A2B conversions within the performance range of the Boolean circuit-based A2B conversions.

1 Introduction

A majority of public key cryptographic algorithms are based on factoring or the discrete logarithm problem. These algorithms are no longer secure in the presence of a large-scale quantum computer. The field of Post-Quantum Cryptography (PQC) researches alternative cryptographic algorithms that remain secure in the presence of quantum computers. To replace the soon-to-be-insecure public-key standards, the National Institute of Standards and Technology (NIST) launched a standardization effort in 2016 [30]. In July 2022, NIST announced three lattice-based schemes to be standardized: Kyber [37], Dilithium [28] and Falcon [32].

One of the challenges in replacing the current standards with post-quantum standards is protecting their implementations against side-channel attacks. Side-channel attacks are attacks on a cryptographic implementation that use unwanted effects of computation leaking information, such as power usage, electromagnetic radiation and timing. Several side-channel attacks on lattice-based cryptographic implementations have been demonstrated, including timing attacks [38,19,25] or power consumption and electromagnetic radiation

attacks [3,42,33,2,34,43,39]. These works illustrate the importance of protection mechanisms against side-channel attacks, and in its latest report NIST has emphasized the importance of these protection mechanisms [1], including them as a major evaluation criterion in the standardization process.

Masking is a popular tool to protect against side-channel attacks. The idea of masking is to split a sensitive variable into two or more shares, in such a way that an adversary that is able to see all but one share still can not infer any information about the sensitive value. The ideas behind masking were introduced by Chari et al. [9] and later extended by Barthe et al. [5] to include the notions of Non-Inference (NI) and Strong Non-Inference (SNI), which allow easier composition of building blocks.

To give an example of masking, a sensitive value x can be split into $x^{(1)}$ and $x^{(2)}$ so that $x = x^{(1)} \odot x^{(2)}$ where $\odot$ is a mathematical operation that depends on the type of masking. For Boolean masking $\odot$ is the XOR operation $\oplus$, while arithmetic masking chooses $\odot$ to be addition modulo a predefined integer q. In first-order masking, the sensitive value is split into 2 shares (i.e., an adversary can probe at most 1 share without compromised security), while higher-order masking splits the sensitive variable into more shares. One observation is that some efficient techniques have been developed specifically for first-order masking, which do not scale to higher masking orders.

Several masked implementations of lattice-based cryptographic schemes have been presented. For signature schemes, a masked implementation of the GLP signature scheme was presented by Barthe [6], followed by a Dilithium implementation by Migliore et al. [29]. Passively secure lattice-based encryption was first masked in [35], followed by an active secure scheme by Oder et al. [31] for first-order and Bache et al. [4] for higher-order. Van Beirendonck et al. [40] provided a first-order masked implementation of the NIST PQC finalist Saber [18], and Coron et al. [16], and later Kundu et al. [27] discussed a higher-order implementation. Kyber was implemented at arbitrary order by Bos et al. [8] and for first-order by Heinz et al. [26]. Fritzmann et al. [22] looked at making masked implementations of Saber and Kyber more effective using instruction set extensions. A masked NTRU implementation was proposed by Coron et al. [12].

A2B conversion One recurring property of most masked implementations of lattice-based cryptography is that both Boolean masking and arithmetic masking are used. To integrate both masking domains, arithmetic to Boolean (A2B) and Boolean to arithmetic (B2A) conversions are needed. In this paper, we are specifically interested in arithmetic to Boolean conversion. The first secure A2B conversion was proposed by Goubin [24], which was later extended by Coron et al. [10]. Both methods are focused on first-order and are based on writing the conversion as a Boolean circuit and implementing this Boolean circuit in a secure fashion.

A different approach to first-order A2B conversion is table-based conversion, where the Boolean result is stored in a table that is manipulated based on the arithmetic input. Coron and Tchulkine [14] were the first to propose such a conversion. Debraize [21] discovered a flaw in their algorithm and improved the

overall efficiency of the Coron and Tchulkine approach. Later, Van Beirendonck et al. [41] discovered a security problem in one of the conversions of Debraize, and proposed two new A2B conversions to circumvent this problem.

Higher-order conversions were proposed in [11,10] for arithmetic masking modulo a power-of-two $q = 2^k$. These techniques were extended for arbitrary modulus by Barthe et al [6], which was later refined in [36]. Similar to the first A2B conversion algorithm by Goubin [24], the above techniques rely on a Boolean circuit methodology to perform the conversion.

Coron et al. [16] adapted the first-order table-based approach for higher orders both for A2B and B2A conversion. For large modulus q, the authors split the inputs into different chunks which are converted individually using A2B conversion, and the carries between the chunks are taken into account using an additional arithmetic to arithmetic (A2A with different moduli) conversion. While the B2A conversions in this work are generally efficient, the overall A2B conversions are only efficient in specific applications.

The increased importance of these conversions due to the rise of lattice-based cryptography is emphasized by the CHES 2021 Test of Time Award, which was awarded to Goubin [24] for introducing the first A2B and B2A conversion techniques.

Masked comparison One important application of A2B conversions is masked comparison, which is a vital building block in actively secure implementations of lattice-based cryptography. The goal of such a comparison is to validate an input ciphertext by comparing it with a recomputed ciphertext as part of the Fujisaki-Okamoto transformation [23].

For first-order masking, a hash-based approach was proposed by Oder et al. [31]. The main idea of this approach is to check if a sensitive array is zero by hashing both shares separately and checking the equality of the hash outputs. For schemes that perform ciphertext compression, this comparison additionally needs an arithmetic to arithmetic (A2A) conversion (i.e., a conversion between two arithmetic masking domains with different modulus). Such an A2A conversion can be implemented as a modified A2B conversion, where a table-based conversion is most efficient for first-order. A problem in the security of the hash-based method of [31] was discovered and fixed by Bhasin et al. [7].

Higher-order masked comparisons have to rely on different techniques, as the hash-based method is limited to two shares. The state-of-the-art conversion techniques to perform higher-order masked comparison first perform A2B conversion and then do the comparison in the Boolean domain. The approaches differ in pre- and postprocessing of the A2B conversion. Barthe et al. [6] perform a masked comparison by a simple approach: A2B conversion followed by a masked bitwise comparison. Bache et al. [4] introduced a method based on a random sum to reduce the number of coefficients. This method was broken by Bhasin et al. [7], who introduced a variant random sum compression that is secure but only applicable for cryptographic schemes without compression and with prime order moduli.

D'Anvers et al. [17] adapted the random sum method as a postprocessing method to reduce the cost of the final Boolean circuit. Bos et al. [8] looked at the preprocessing stage and proposed to decompress the input ciphertext instead of compressing the masked recomputed ciphertext. This approach was later adapted by Coron et al. [15] by combining the decompression idea, the random sum method, and some extra masked gadget into a new comparison. These methods were compared and improved in a later work by D'Anvers et al. [20], which we refer to to get an overview of higher-order masked comparison algorithms.

1.1 Our contributions

In this paper, we introduce a new strategy to perform arithmetic to Boolean conversion. Although it is not exactly table-based, our method falls in the table-based category and is indebted to the higher-order table-based A2B conversions of [16], and more specifically to the register-based optimized arithmetic to 1-bit Boolean conversion. We start with introducing an arithmetic to Boolean conversion, and later introduce optimizations to more efficiently perform specific masked operations used in lattice-based cryptography.

Our method works on a register (which can be seen as a table with 1-bit entries). In contrast to previous table-based methods, where the table is used to encode the output values, our register is used as a one-hot encoding of the input values, with which we mean that a value x is represented with a register where the xth bit is 1 and all others are 0. The first advantage of a one-hot encoding is that the register/table size does not grow with the output length, which would be the case in a table-based approach where all possible outputs are stored in the table. Secondly, the input has to be processed only once, and the result can be used to determine both a carry value (as a result of the arithmetic masking) and a Boolean masked output value. Thirdly, we introduce an efficient method to propagate carries by using the properties of the one-hot encoding. An intuitive introduction to these ideas is given in section 3.

In section 4 we formalize our arithmetic to Boolean conversion, followed by a generalization of our method and a security proof. In section 5 we introduce an arithmetic to 1-bit Boolean function calculation. This is a generalization of the aforementioned register-based optimized arithmetic to 1-bit Boolean conversion of [16] in two ways: we allow an arbitrarily large arithmetic masking modulus (instead of 5 or 6 bits in previous works) and we allow multiple masked coefficients to be the input of the function. One of the use-cases of this algorithm is masked comparison, where multiple masked coefficients need to be compared with publicly known reference values and only one bit is returned that indicates whether all coefficients match their reference value(s).

Section 6 details how to obtain a more efficient implementation and how to achieve parallelism in our inherently sequential design at low cost. The resulting A2B implementation is then compared to the state-of-the-art algorithms in section 7. Our measurements show a speedup of approximately a factor of 15 compared to the state-of-the-art table-based A2B comparisons. This brings

higher-order table-based conversion to the performance range of Boolean circuit-based A2B conversions, in some cases outperforming the latter with a cycle count reduction of 27%. The implementations of our algorithms for Cortex M4 are made available at `https://github.com/KULeuven-COSIC/One-hot-masking`.

2 Preliminaries

2.1 Notation

Lists and matrices are denoted in bold text. These are indexed using a subscript, where $\mathbf{X}_i$ indicates the ith element of the list $\mathbf{X}$ and where $\mathbf{X}_{i,j}$ indicates the element on the ith row and j^{th} column of a matrix $\mathbf{X}$. We write $|\mathbf{X}|$ to denote the number of coefficients in the list $\mathbf{X}$. We denote with $\lfloor x \rfloor$ a flooring of a number x to the nearest integer less or equal to x, with $\lceil x \rceil$ ceiling x to the nearest integer greater or equal to x, and with $\lfloor x \rceil$ rounding to the nearest integer with ties rounded upwards. These operations are extended coefficient-wise to lists.

Positive integers are represented in unsigned binary representation unless stated otherwise, with the most significant bit (MSB) at the leftmost position and the least significant bit (LSB) at the rightmost position. $x[i]$ indicates the ith bit of the binary representation of x starting from the least significant bit and $|R|$ indicates the number of bits in the representation of R.

The concatenation operator $x_1\|x_0$ concatenates the bitstrings x_1 and x_0. This representation is extended for non-power-of-two p-ary numbers y_1, y_2 (i.e., numbers represented with an integer value between 0 and $p - 1$) as $y = y_1\|y_0$. More precisely, the value of y equals $y_1 \cdot p + y_0$. In its most generalized sense we can concatenate numbers with different representations: for a p_2-ary number y_2, a p_1-ary number y_1 and a p_0-ary number y_0, we write $y = y_2\|y_1\|y_0$ to signify $y = y_2 \cdot (p_0 \cdot p_1) + y_1 \cdot p_0 + y_0$.

We denote with:

$$\underbrace{x_1}_{b_1} \| \underbrace{x_0}_{b_0} \leftarrow x, \tag{1}$$

splitting the binary representation of x in parts x_1 with bitsize b_1 and x_0 with bitsize b_0 so that $x = x_1\|x_0$. This is generalized for p-ary numbers as:

$$\underbrace{x_1}_{p_1\text{-ary}} \| \underbrace{x_0}_{p_0\text{-ary}} \leftarrow x, \tag{2}$$

where x is split into a p_0-ary symbol x_0 and a p_1-ary symbol x_1 so that $x = x_1\|x_0$. Note that this is a unique way of splitting a number x.

We denote with $x \ll i$ a shift of the binary representation of x to the left with i positions (which equals to $x \cdot 2^i$), and with $x \gg i$ a shift to the right with i positions (which equals $\lfloor x/2^i \rfloor$). A circular shift to the left with i positions is written as $x \overset{|R|}{\lll} i$, with $|R|$ the number of bits involved in the shift. More specifically, $x \overset{|R|}{\lll} i = (x \ll i)\|(x \gg (|R| - i))$.

Sampling a random value x from a distribution χ is denoted $x \leftarrow \chi$. Furthermore, $\mathcal{U}(S)$ denotes the uniform distribution over a set S.

2.2 Masking

In Boolean masking, a sensitive variable x is split into S shares $x^{[0]}$ to $x^{[S-1]}$, so that the XOR of the shares results in the original variable x (i.e., $x = \oplus_{i=0}^{S-1} x^{[i]}$). We write $x^{[i]}$ to denote the value of the ith share of a masked variable x, and $x^{[\cdot]}$ to denote the value of x while explicitly making clear that x is shared. As such, the value $x^{[\cdot]}$ will not be physically represented in a secure implementation and is only implicitly present by combining the different shares.

One can perform Boolean operations on a Boolean masked variable: $z^{[\cdot]} = x^{[\cdot]} \oplus y^{[\cdot]}$ is calculated by an XOR on the corresponding shares as $z^{[i]} = x^{[i]} \oplus y^{[i]}$. An AND with an unmasked variable $z^{[\cdot]} = x^{[\cdot]} \,\&\, m$ is calculated by applying m to each share individually $z^{[i]} = x^{[i]} \,\&\, m$. Similarly, shifts, rotations and concatenations on a Boolean masked variable are applied to each share individually.

Arithmetic masking splits a sensitive variable x in S shares $x^{(0)}$ to $x^{(S-1)}$ so that the sum of the shares modulo a given integer q equals the sensitive value ($x^{(\cdot)} = x^{(0)} + x^{(1)} \bmod q$). As before we denote with $x^{(i)}$ the ith share of a masked variable, and with $x^{(\cdot)}$ the value of x while stressing that this value is not physically present in the implementation.

Arithmetic masking allows easy computation of arithmetic operations, where a sum $z^{(\cdot)} = x^{(\cdot)} + y$ can be calculated by summing y to the zero$^{\text{th}}$ share of $x^{(\cdot)}$ (i.e., $z^{(0)} = x^{(0)} + y$ and $z^{(i)} = x^{(i)}$ for other shares). Multiplication with an unmasked constant is performed on each share individually (i.e., $z^{(\cdot)} = c \cdot x^{(\cdot)}$ can be calculated as $z^{(i)} = c \cdot x^{(i)}$). Concatenation, flooring and rounding are calculated on each share individually. It is important to note that for arithmetic masking $\lfloor x^{(\cdot)} \rfloor$ is not necessarily equal to $\lfloor x \rfloor$ as the former is calculated on each share individually, while the latter is calculated on the unmasked variable. This is also true for rounding and concatenation.

3 Intuitive introduction to one-hot conversion

The goal of our algorithm is to perform arithmetic to Boolean conversion. More specifically, the input is an arithmetically masked number $D^{(\cdot)}$, with masking modulus q. The output is a Boolean masked number $B^{[\cdot]}$ so that $B^{[\cdot]} = D^{(\cdot)}$. For the sake of simplicity, we will assume that the arithmetic masking modulus is a power of two unless stated otherwise. It is trivial to extend our method for different masking moduli and we will later show how to extend the method to non-power-of-two moduli.

We will first give an intuitive explanation of the algorithm before explaining the details in section 4. The algorithm starts by preparing a Boolean masked register $R^{[\cdot]}$ with value 1, i.e., with a one in the zero$^{\text{th}}$ bit and zeros in all other bits. The algorithm then iteratively processes parts of $D^{(\cdot)}$, modifying the register $R^{[\cdot]}$ in two steps: in the first step, the register is converted to a one-hot encoding of the input coefficient $\mathbf{D}_i^{(\cdot)}$ (i.e., the value of $\mathbf{D}_i^{(\cdot)}$ is encoded by setting the register bit at position $\mathbf{D}_i^{(\cdot)}$ to 1, and all other bits to 0) and in the second step, the relevant information is extracted from the register in a sharewise fashion.

A simple example: $q = |R|$ First, imagine that the modulus q equals the number of bits in the register $|R|$. The algorithm first rotates the register with $\sum_k D^{(k)}$ positions using a variant of the secure rotation algorithm described in [16]. This corresponds in practice to a rotation of the register with $\sum_k D^{(k)} \bmod |R| = \sum_k D^{(k)} \bmod q = D^{(\cdot)}$ positions, where the $\bmod |R|$ operation is present due to the limited size of the register and the resulting wraparound. The output of this step can be seen as a one-hot encoding of the input $D^{(\cdot)}$ (i.e., the 1 in the register can be found on the $D^{(\cdot)\text{th}}$ position).

After this operation we effectively associated each position in the register with one value of $D^{(\cdot)}$ (i.e., if the 1 is in the tth position, then $D^{(\cdot)} = t$ and vice versa). We then process the shares of the register individually to obtain the required result. For each share, we take the bit at position p (i.e., $D^{(i)}[t]$) and multiply it with t. The results are all XOR'ed together into a share of the output. The output is thus calculated as:

$$B^{[i]} = \bigoplus_{t=0}^{|R|-1} t \cdot R^{[i]}[t]. \tag{3}$$

Now remembering that $R^{[\cdot]}[t] = 1$ at position $t = D^{(\cdot)}$, but $R^{[\cdot]}[u] = 0$ at all other positions $u \neq t$, we can see that:

$$B^{[\cdot]} = \bigoplus_{i=0}^{S-1} B^{[i]} \tag{4}$$

$$= \bigoplus_{i=0}^{S-1} \left(\bigoplus_{t=0}^{|R|-1} t \cdot R^{[i]}[t] \right) = \bigoplus_{t=0}^{|R|-1} t \cdot \left(\bigoplus_{i=0}^{S-1} R^{[i]}[t] \right) \tag{5}$$

$$= \bigoplus_{t=0}^{|R|-1} t \cdot R^{[\cdot]}[t] \tag{6}$$

$$= D^{(\cdot)}. \tag{7}$$

Thus confirming that the output is indeed $B^{[\cdot]} = D^{(\cdot)}$ as required. In terms of masking security, the first operation can be instantiated as a variant of the secure rotation of [16], while the second operation is performed on each share separately and is thus inherently secure in the masking framework.

This simple example is depicted in figure 1 where in the first step the register is rotated with $\sum_k D^{(k)} \bmod |R| = 3$ positions, and in the second step the output is calculated following Equation 3. Note that it is possible to implement the second operation as given in Equation 3 more efficiently as will be discussed in section 6.

More complicated: $q > |R|$ The problem with the simple approach is that it is typically not efficient to allow an arbitrarily large register size. Therefore, we will adapt the previous algorithm to allow q to be bigger than the register size. We will do this by chopping the input coefficients $D^{(\cdot)}$ with bitlength $\log_2(q)$ in
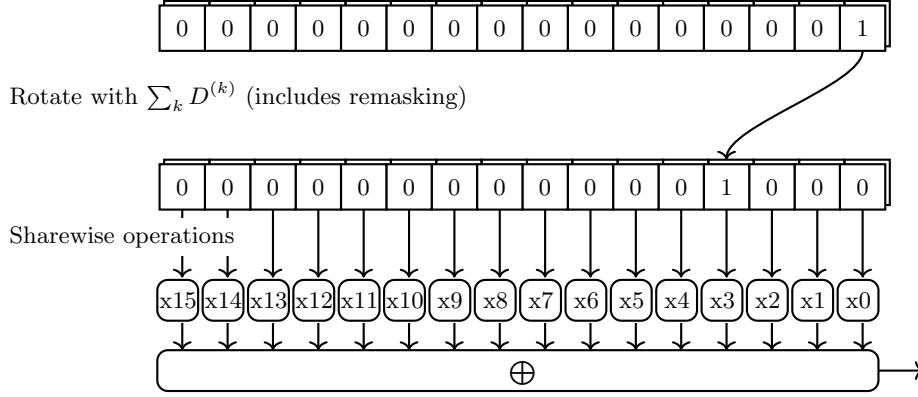
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Rotate with $\sum_k D^{(k)}$ (includes remasking)

0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Sharewise operations

| x15 | x14 | x13 | x12 | x11 | x10 | x9 | x8 | x7 | x6 | x5 | x4 | x3 | x2 | x1 | x0 |

$\oplus$

Fig. 1. Overview of the three steps in InnerLoop for $D^{(\cdot)} = 3$ and $q = 16$. In the first step the register is rotated with $D^{(\cdot)} = 3$ positions using the SecureRotate algorithm. In the second step, all elements of the register are multiplied with their position and the results are XOR'ed together to produce the output.

several smaller chunks with bitlength $\log_2(p)$, with $p < |R| < q$. These smaller chunks are then processed iteratively, starting with the least significant chunk $D_0^{(\cdot)}$. Note that these chunks are not independent as the arithmetic masking entails that there are carries that need to be propagated from the less significant chunks to the more significant chunks. We will have to take care of these carries in our method.

First we choose the smaller power-of-two modulus p so that $p \cdot S < |R|$, with S the number of shares, and split the coefficients of $D^{(\cdot)}$ in chunks $\hat{D}_j^{(\cdot)}$ of $\log_2(p)$ bits. These chunks are then processed iteratively, starting with the least significant chunk. A depiction of the processing of the first chunk is given in figure 2.

To process a chunk $\hat{D}_j^{(\cdot)}$ we perform the following three operations: first, we rotate the register, then we compute the relevant output bits and finally we prepare the carry for the next iteration.

In the first operation, the register is rotated with $\sum_k \hat{D}_j^{(k)}$ positions. Note that in contrast to the previous method $\sum_k \hat{D}_j^{(k)} \bmod |R| \neq \hat{D}_j^{(\cdot)}$, more specifically, the modulo operation is no longer relevant and can be ignored as long as we choose p to be small enough to avoid any possible wrap-around of the 1 in the register.

The position of the one in the register can now be described in function of two components: the value of the chunk, $\hat{D}_j^{(\cdot)} = \sum_k \hat{D}_j^{(k)} \bmod p$, and the carry $c_j = \lfloor \sum_k \hat{D}_j^{(k)}/p \rfloor$ that needs to be propagated to the next chunk. These two components are represented in the position as follows: the register can be subdivided into multiple 'carry parts' of $\log_2(p)$ bits as given in figure 2 with the red lines. The carry is then encoded by the part containing the 1 (in figure 2,

$c = 2$), while the chunk value is encoded as the relative position of the one in its part (in figure 2, $\hat{D}_j^{(\cdot)} = 1$).

In the second operation, the relevant output bits corresponding to the chunk $\hat{D}_j^{(\cdot)}$ are calculated. Similar to the above technique, we perform a sharewise calculation, but this time multiplying with the value $(t \bmod p)$:

$$\hat{B}_j^{[i]} = \bigoplus_{t=0}^{|R|-1} (t \bmod p) \cdot R^{[i]}[t], \tag{8}$$

where analogous to before we can check our method for the first chunk as:

$$\hat{B}_0^{[\cdot]} = \bigoplus_{i=0}^{S-1} \hat{B}_0^{[i]} = \bigoplus_{t=0}^{|R|-1} (t \bmod p) \cdot R^{[\cdot]}[t] \tag{9}$$

$$= \hat{D}_0^{(\cdot)} \bmod p, \tag{10}$$

which means that the first $\log_2(p)$ bits are converted correctly. However, for subsequent iterations, we will have to take into account the carry c_j that needs to be propagated from chunk j to chunk $j+1$. This is done in the third operation.

The third operation propagates the carry and is again performed on each share separately. At the end of the third operation, the register contains a one-hot encoding of the carry c_j that needs to be propagated. This register is then used as the starting register in the next rotation. This means that the rotation already has an initial rotation with c_j, before the rotation with $\sum_k \hat{D}_{j+1}^{(k)}$ is applied. The total rotation is then $c_j + \sum_k \hat{D}_{j+1}^{(k)}$, thus effectively taking the carry into account.

The method to obtain the one-hot encoding of the carry can be best understood using figure 2. For each share of the register, we xor together all bits within the same carry bin c, and place it at position c in the register. Or more specifically, for each possible carry value c and each share k we calculate:

$$R^{[k]}[c] = \bigoplus_{m=0}^{p-1} {R_{tmp}}^{[k]}[c \cdot p + m] \tag{11}$$

The last iteration is slightly different, as the last chunk to be processed does not need to take into account further propagation of the carries. This case can thus be performed analogous to the simple example above (see figure 1) and can use any bitsize $\log_2(p_L)$ as long as $p_L \leq |R|$ (assuming the register size is also a power of two).

4 Arithmetic to Boolean conversion

In this section, we will go into detail on the arithmetic to Boolean conversion technique, as well as generalize the technique and formulate a security proof.
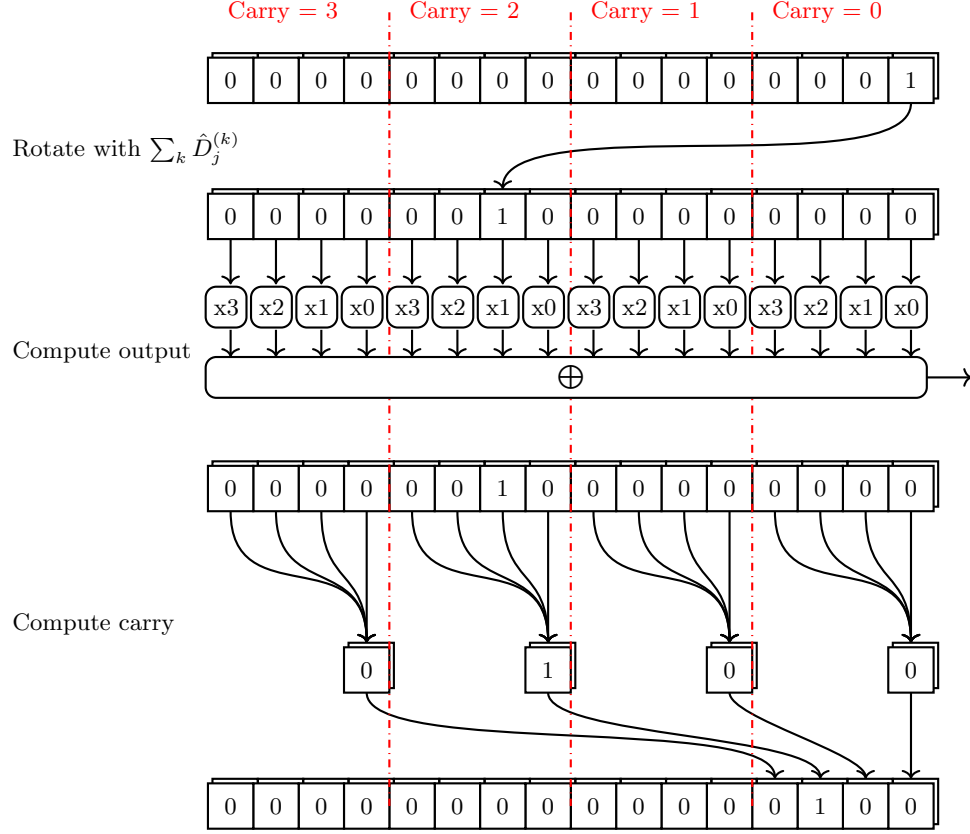
Fig. 2. Overview of an iteration of the A2B conversion for $\hat{D}_j^{(\cdot)} = 1$, with modulus $p = 4$, a carry value $c = \lfloor \sum_k \hat{D}_j^{(k)}/p \rfloor = 2$ and $S = 4$ shares. Note that all registers are masked during execution and that the values depicted are the corresponding unmasked values which should never be revealed during the computation. In the first step the register is rotated with $\sum_k \hat{D}_j^{(k)} = 9$ positions. In the second step the partial output is computed as in figure 1. In the third step the carry is propagated by XORing the values per carry and putting them in the relevant position of the output register. The latter step already gives an initial rotation of the register with 2, which is equal to the carry value and as such effectively propagates the carry to the next block.

Algorithm 1 gives a generalized algorithm to perform A2B conversion using the secure rotation method, which is given in Algorithm 2. Remember that operations on $R^{[\cdot]}$ are performed sharewise. A graphical overview of one iteration of the loop is given in figure 2, while the last iteration only performs the operations in figure 1.

For the parameter setting, we need to choose a register size $|R|$, which should be an integer of size at least S^2, with S the number of shares. For software implementations, one would typically choose $|R|$ to be the bit width of the processor. From $|R|$, we can derive the chunk modulus p as the largest power of two such that $p \cdot S \leq |R|$. The final chunk size p_L can be computed as the largest power of two under the conditions that $p_L \leq |R|$ and $\log_2(p_L) = \log_2(q) - L \cdot \log_2(p)$ for L a positive integer. In this case, $L + 1$ will be the number of chunks into which a coefficient is split.

We will first provide a t-SNI security proof of our method, and then explain how to generalize our method to non-powers of two, or to calculate arbitrary functions. Our security proof extends the table-based conversion proofs of [13,16].

Algorithm 1: A2B$(D^{(\cdot)})$

// Setup
1 $R^{[0]} = 1;\ B^{[0]} = 0$
2 **for** $i = 1, \ldots, S-1$ **do** $R^{[i]} = 0;\ B^{[i]} = 0$
3 $\underbrace{\hat{D}_L^{(\cdot)}}_{<\log_2 |R|} \,\|\ldots\|\, \underbrace{\hat{D}_1^{(\cdot)}}_{\log_2(p)} \,\|\, \underbrace{\hat{D}_0^{(\cdot)}}_{\log_2(p)} \leftarrow D^{(\cdot)}$

// calculate
4 **for** $j = 0, \ldots, L-1$ **do**
5 $\quad$ $R_{tmp}^{[\cdot]} = \texttt{SecureRotate}(R^{[\cdot]}, \hat{D}_j^{(\cdot)})$
6 $\quad$ $R^{[\cdot]} = 0$
7 $\quad$ **for** $c = 0$ **to** $S-1$ **do**
8 $\quad\quad$ $R^{[\cdot]}[c] = \bigoplus_{m=0}^{p-1} R_{tmp}^{[\cdot]}[c \cdot p + m]$
9 $\quad$ $B_{tmp}^{[\cdot]} = \bigoplus_{t=0}^{|R|-1} (t \bmod p) \cdot R^{[\cdot]}[t]$
10 $\quad$ $B^{[\cdot]} = B_{tmp}^{[\cdot]} \| B^{[\cdot]}$
11 $R_{tmp}^{[\cdot]} = \texttt{SecureRotate}(R^{[\cdot]}, \hat{D}_L^{(\cdot)})$
12 $B_{tmp}^{[\cdot]} = \bigoplus_{t=0}^{|R|-1} t \cdot R^{[\cdot]}[t]$
13 $B^{[\cdot]} = B_{tmp}^{[\cdot]} \| B^{[\cdot]}$
14 **return** $B^{[\cdot]}$

Theorem 1 ($(S-1)$-SNI of Algorithm 1). *For any set of $t_c < S$ intermediate variables and for any subset $O \in [1, n]$ where $t_c + |O| < S$, we can perfectly simulate the output variables $R^{[O]}$ and the t_c intermediate values using the input values $D^{(i)}$ for each $i \in I$, with $|I| \leq t_c$.*

Algorithm 2: SecureRotate($R^{[\cdot]}, \hat{D}^{(\cdot)}$)

    // Rotate + remask

1 for $sh_D = 0$ **to** $S - 1$ **do**

2 $R^{[0]} = R^{[0]} \overset{|R|}{\lll} \hat{D}^{(sh_D)}$

3   **for** $sh_R = 1$ **to** $S - 1$ **do**

4 $R^{[sh_R]} = R^{[sh_R]} \overset{|R|}{\lll} \hat{D}^{(sh_D)}$

5 $U \leftarrow \mathcal{U}(2^{|R|})$

6 $R^{[sh_R]} = R^{[sh_R]} \oplus U$

7 $R^{[0]} = R^{[0]} \oplus U$

Proof. Within this proof we will refer to line x of *Algorithm* 1 with lx, and to line x of Algorithm 2 with l_rx. Before we delve into the details we choose which input coefficients will be used to simulate the intermediate values. All operations described in Algorithm 1 are performed sharewise, and so at most one share of the registers $R^{[\cdot]}$ and $R_{tmp}{}^{[\cdot]}$ is involved; and at most one share of $D^{(\cdot)}$ and $\hat{D}_j^{(\cdot)}$ is involved. For each probe during these lines, we will add the share number sh_R of the involved share of $R^{[\cdot]}$ or $R_{tmp}{}^{[\cdot]}$ (if applicable) to the set SHR; and similarly add the share number sh_D to the set SHD, if a share of $D^{(\cdot)}$ or $\hat{D}_j^{(\cdot)}$ is involved. For the intermediate values in the rotation (Algorithm 2) we make the sets as described in table 1.

Variable:	Action: add $\cdots$	Simulated by:
l_r2/l_r4: $\hat{D}^{(sh_D)}$	sh_D to SHD	Corresponding bits of: $D^{(sh_D)}$
l_r2: $R^{[0]}$	sh_D to SHD; 0 to SHR	$R_{in}{}^{(0)} \overset{q \cdot C}{\lll} \hat{D}^{(sh_D)}$
l_r4: $R^{[sh_R]}$	sh_D to SHD; sh_R to SHR	$R_{in}{}^{(sh_R)} \overset{q \cdot C}{\lll} \hat{D}^{(sh_D)}$
l_r5: U_{sh_D, sh_R}	sh_D to SHD; sh_R to SHR	U_{sh_D, sh_R}
l_r6: $R^{[sh_R]}$	sh_D to SHD; sh_R to SHR	$(R_{in}{}^{(sh_R)} \overset{q \cdot C}{\lll} \hat{D}^{(sh_D)}) \oplus U_{sh_D, sh_R}$
l_r7: $R^{[0]}$	sh_D to SHD; 0 to SHR	$(R_{in}{}^{(0)} \overset{q \cdot C}{\lll} \hat{D}^{(sh_D)}) \oplus_{k=1}^{sh_R} U_{sh_R, k}$

Table 1. List of variables and their simulatability.

After building the sets SHD and SHR we know that $|SHD| \leq t_c$ and $|SHR| \leq t_c$, as each intermediate probe adds at most one item to each set. We then choose the input set to simulate all probed values as $D^{(sh_D)}$ for each share $sh_D \in SHD$. The set SHD then acts as the input set I and as such we have the asked condition $|I| \leq t_c$. Now rests to show that we can perfectly simulate all probed values.

The general overview of our proof will proceed as follows: first, we will argue that $R^{[\cdot]}$ and all other variables are simulatable during the setup of the algo-

rithm, then we will argue that all asked intermediate values in the subblocks are simulatable if the input $R^{[\cdot]}$ is simulatable, and finally, we will show that $R^{[\cdot]}$ at the output of the block is simulatable if $R^{[\cdot]}$ at the input of the block (further denoted $R_{in}^{(\cdot)}$) is simulatable.

Simulatability of variables during the setup of the algorithm (l1 to l3): This first step is easy, as $R^{[\cdot]}$ is deterministic and can thus be easily simulated by the adversary. The probed $\hat{D}_{i,j}^{(sh_D)}$ values can be simulated as $sh_D \in SHD$ due to our construction of SHD.

Simulatability of variables during SecureRotate: This part will perform induction on the outer loop sh_D in SecureRotate. We will show that if we can simulate the values at the start of one loop iteration, we can also simulate the output variables of that loop iteration and all probed variables.

If $sh_D \notin SHD$, then the adversary has no information on the U_{sh_D,sh_R} and as such the output will look uniformly random, thus rendering $R^{[\cdot]}$ simulatable at the end of the iteration (i.e., it can be simulated by drawing from a uniformly random distribution $\mathcal{U}(\{0,1\}^{|R|})$).

If $sh_D \in SHD$, then we can simulate any probed intermediate variable as given in table 1, where $R_{in}^{[\cdot]}$ is the value at the start of that outer loop iteration. For the latter two variables in the table, if the corresponding U_{sh_D,sh_R} is not probed we can replace it with a uniformly random value.

We have shown that if we can simulate the intermediate values at the start of the first loop iteration, then we can also simulate the intermediate values in the following loop iterations and therefore also at the end of the SecureRotate operation.

Simulatability of variables after SecureRotate (l6-l10 and l12-13): Next we will show that if the register $R_{tmp}^{[\cdot]}$ is simulatable at the end of the SecureRotate then all variables at l6-l10 and l12-13 are simulatable. Note that the operations on these lines only work on one share at a time and are perfectly deterministic if the input $R_{tmp}^{[\cdot]}$ is known. As such, if $R_{tmp}^{[\cdot]}$ can be simulated, then any intermediate variable in these lines can be simulated.

To conclude, we have shown that $R^{[\cdot]}$ is simulatable at the start of the algorithm, that it is simulatable at the end of each block if it is simulatable at the start, and that all probed intermediate values can be simulated if $R^{[\cdot]}$ is simulatable at the start of the block. This means that both the probed intermediate values and the probed output variables are simulatable.

4.1 Generalization

The algorithm presented above can be generalized to have broader applicability. Firstly, the algorithm is not bound by calculating the identity function (i.e., $A^{(\cdot)} = B^{[\cdot]}$). Instead one can replace the multiplications with the value $(t \bmod p)$ in Equation 8, which calculates a unity function, with any function $f()$ as:

$$\hat{B}_j^{[i]} = \bigoplus_{t=0}^{|R|-1} f(t \bmod p) \cdot R^{[i]}[t], \tag{12}$$

13

thus creating a more elaborate A2B conversion that allows calculating on the data for free.

Secondly, the modulus does not need to be a power of two but can be any positive integer q. In this case, we select different modulus p for each chunk. The selection of the p_i needs to fulfill the following conditions:

$$\prod_{i=0}^{L} p_i = q \tag{13}$$

$$\forall_{i=0}^{L} \; p_i \cdot S \le |R| \tag{14}$$

Note that the latter condition can be relaxed for p_L, as we don't need to determine the carry location and can thus allow an overflow at positions that are a multiple of p_L. As such, a p_L value is also valid if p_L divides $|R|$.

Similar as before $D_i^{(\cdot)}$ is split into chunks. However, this time we represent $D^{(\cdot)}$ as a series of p_i-ary numbers:

$$\underbrace{\hat{D}_L^{(\cdot)}}_{p_L\text{-ary}} \| \dots \| \underbrace{\hat{D}_1^{(\cdot)}}_{p_1\text{-ary}} \| \underbrace{\hat{D}_0^{(\cdot)}}_{p_0\text{-ary}} \leftarrow D_i^{(\cdot)} \tag{15}$$

This representation immediately gives us the different chunks, as each symbol corresponds to a chunk $\hat{D}_j^{(\cdot)}$, with $\hat{D}_0^{(\cdot)}$ the least significant symbol.

5 Arithmetic to 1-bit Boolean

In this section, we will specialize our method toward calculating a function $f()$ that takes one or more arithmetically masked variables and outputs one Boolean masked bit. Theoretically, our method can calculate any such function, however, when the input modulo q is split into smaller chunks modulo p (i.e. $L > 0$), only functions that can be described as:

$$f(D^{(\cdot)}) = f_L(\hat{D}_L^{(\cdot)}) \quad \& \quad \dots \quad \& \quad f_0(\hat{D}_0^{(\cdot)}), \tag{16}$$

are implementable. However, as we will show in subsection 5.3, this restriction does not pose a problem for typical applications in lattice-based encryption, such as masked comparison or extraction of the MSB.

Our method is similar to the arithmetic to Boolean conversion described above, where the calculation of $B^{[\cdot]}$ is not performed. The main idea is that in iteration i, if $f_i(\hat{D}_i^{(\cdot)}) = 1$, the register is propagated as before, while if $f_i(\hat{D}_i^{(\cdot)}) = 0$, a register with only zero is propagated. This can be achieved during a 'compute carry' step, by only propagating positions t where:

$$f_i(t \bmod p) = 1. \tag{17}$$

This means that if the 1 in the register is at a location where $f_i(t \bmod p) = 0$, the 1 in the register is not propagated and the register will have only 0's for the rest of the algorithm. At the end of the algorithm, we can check if the one is still present in the algorithm, which is the case if and only if $f(D^{(\cdot)}) = 1$. Note that the register remains masked throughout the algorithm and thus it is not revealed if and during which iteration the one is discarded.

5.1 Method description

More specifically, the input is a list of arithmetically masked numbers $\mathbf{D}^{(\cdot)}$, with corresponding masking modulus q. The output is a boolean masked bit 1 if $\forall_i : f(\mathbf{D}_i^{(\cdot)}) = 1$, and 0 otherwise. The parameter setting (i.e., setting $|R|$, p, L and p_L) proceeds identical to the procedure explained in section 4.

The setup phase of the algorithm similarly consists of two steps: initializing the Boolean masked register $R^{[\cdot]}$ to the value 1 and dividing each coefficient of $\mathbf{D}^{(\cdot)}$ into chunks of $\log_2(p)$ bits (with exception of the most significant chunk, which has $\log_2(p_L)$ bits).

The algorithm then iterates over all coefficients, and for each coefficient over all chunks starting with the least significant chunk. For each chunk first a secure rotation [16] is performed, as depicted in Algorithm 2. Then, instead of propagating all positions as in the full A2B conversion, only positions t where $f_i(t \bmod p) = 1$, are propagated to the output register in a step we will refer to as bit selection. More specifically, for each possible carry value c and each share k we calculate:

$$R^{[k]}[c] = \bigoplus_{t:f(t)=1} R_{tmp}{}^{[k]}[c \cdot p + t]. \tag{18}$$

The bit selection operation performs two functions: first, for all values of $\hat{\mathbf{D}}_{i,j}^{(\cdot)} \bmod p$ where $f(\hat{\mathbf{D}}_{i,j}^{(\cdot)}) = 1$, the 1 in the register is passed to the next iteration (othwerwise, the 1 is not passed to the next iteration). Secondly, the value of the carry (i.e., $c = \lfloor \sum_k \hat{\mathbf{D}}_{i,j}{}^{(k)}/p \rfloor$) is represented in the fact that the 1, if still present in the register, can be found at the cth position of the output register.

In the final iteration of a coefficient $\mathbf{D}_{i,L}{}^{(\cdot)}$, the carry is no longer relevant. We thus map all allowed positions to the zero$^{\text{th}}$ bit of $R^{[\cdot]}$ without distinguishing between the different carry values. Then the algorithm proceeds with the next coefficient in the input array. At the end of the algorithm, the zero$^{\text{th}}$ bit of $R^{[\cdot]}$ contains a Boolean masking of the output.

Side-Channel Security Our security proof proceeds similarly to the security proof of the A2B conversion, as the secure-rotate function is still the only non-sharewise component that needs special attention.

Theorem 2 $((S{-}1)$**-SNI of Algorithm 3).** *For any set of $t_c < S$ intermediate variables and for any subset $O \in [1, n]$ where $t_c + |O| < S$, we can perfectly simulate the output variables $R^{[O]}$ and the t_c intermediate values using the input values $\mathbf{D}^{(i)}$ for each $i \in I$, with $|I| \le t_c$.*

Proof. The t-SNI security proof of the Arithmetic to 1-bit Boolean function method is similar to the proof of the Arithmetic to Boolean conversion. The difference in both algorithms is only in the sharewise parts (l8 to l10 and l12-l13), which can be simulated deterministically using the knowledge on $R_{tmp}{}^{[\cdot]}$. As such one can essentially reuse the security proof of Theorem 1.

Algorithm 3: A → 1-bit B($\mathbf{D}^{(\cdot)}, \mathbf{M}$)

```
    // Setup
1   R^[0] = 1
2   for i = 1, ..., S − 1 do  R^[i] = 0
3   for i = 1 to N − 1 do
4   │   D̂_{i,L}^{(·)} ‖ ... ‖ D̂_{i,1}^{(·)} ‖ D̂_{i,0}^{(·)} ← D_i^{(·)}
    │   (<log₂|R|)      (log₂(p))   (log₂(p))

    // calculate
5   for i = 0, ..., N − 1  do
6   │   for j = 0, ..., L − 1  do
7   │   │   R_{tmp}^[·] = SecureRotate(R^[·], D̂_{i,j}^{(·)})
8   │   │   R^[·] = 0
9   │   │   for c = 0 to S − 1 do
10  │   │   │   R^[·][c] = ⊕_{∀t:f_i(t)=1} R_{tmp}^[·][c · p + t]
11  │   R_{tmp}^[·] = SecureRotate(R^[·], D̂_{i,L}^{(·)})
12  │   R^[·] = 0
13  │   R^[·][0] = ⊕_{∀t:f_i(t)=1, ∀c∈[0,...,|R|/p_L)} R_{tmp}^[·][c · p_L + t]
14  return R^[·]
```

5.2 Generalization

As with the arithmetic to Boolean conversion, our method can be generalized. First, the masking modulus q is not required to be a power of two. This generalization is similar to the non-power-of-two modulus generalization in subsection 4.1 and we refer to this section for an explanation on how to achieve this.

Secondly, the masking modulus q does not have to be equal for all coefficients. To allow different masking moduli q_i associated with their respective coefficients $D_i^{(\cdot)}$, one performs the determination of the parameters p, L, p_L for each coefficient separately. The rest of the algorithm then proceeds as usual, with each coefficient using its specific set of p, L, p_L.

5.3 Applications to Lattice-Based Encryption

The method presented above can be used as a building block for the masking of lattice-based encryption. In this section we will specifically look into two building blocks for lattice-based encryption: comparison of the (uncompressed) recomputed ciphertext with the input ciphertext in the Fujisaki-Okamoto transformation, and A2B for extraction of the most significant bit(s) during decryption. We will show how both these functionalities can be achieved using our methodology by choosing the appropriate input parameters.

16

Comparison The comparison is an essential part of the Fujisaki-Okamoto transformation. The goal of this comparison is to validate the input ciphertext against a recomputed ciphertext. Several works have looked at optimizing higher-order masked comparison [4,7,17,16,20]. We will consider a recomputed ciphertext that has not been compressed, as the compression is generally expensive and we can include the compression in our solution at almost no cost. This is the same setup as used in previous works.

In previous works, the comparison is typically done in at least two steps containing an A2B conversion and the comparison itself. In this work, the comparison itself is already performed in the A2B conversion. Moreover, the adaptation of the A2B conversion even makes the A2B conversion more efficient as the Boolean output is not calculated.

The first input to the comparison is the input ciphertext, which consists of two arrays $(\mathbf{B}, \mathbf{C})$, with coefficients modulo q_b and q_c respectively. The second input is an uncompressed recomputed masked ciphertext $(\mathbf{B}^{*(\cdot)}, \mathbf{C}^{*(\cdot)})$, both with coefficients modulo q. The comparison then should return true if and only if:

$$\forall i : \lfloor q_b/q \cdot \mathbf{B}_i^{*(\cdot)} \rceil = \mathbf{B}_i \text{ and } \forall i : \lfloor q_c/q \cdot \mathbf{C}_i^{*(\cdot)} \rceil = \mathbf{C}_i \tag{19}$$

Power of two q For q, q_b and q_c powers of two, such a function can be instantiated by calculating the list with coefficients $\mathbf{D}_i^{(\cdot)}$:

$$\mathop{\forall}_{i \in 0,\dots,|B|-1} : \mathbf{D}_i^{(0)} = \mathbf{B}_i^{*(0)} + \frac{q}{2q_b} - \frac{q}{q_b} \cdot \mathbf{B}_i \quad ; \quad \mathop{\forall}_{\substack{i \in 0,\dots,|B|-1 \\ j>0}} : \mathbf{D}_i^{(j)} = \mathbf{B}_i^{*(j)} ; \tag{20}$$

$$\mathop{\forall}_{i \in 0,\dots,|C|-1} : \mathbf{D}_{i+|B|}^{(\cdot)} = \mathbf{C}_i^{*(0)} + \frac{q}{2q_c} - \frac{q}{q_c} \cdot \mathbf{C}_i \quad \text{and} \quad \mathop{\forall}_{\substack{i \in 0,\dots,|C|-1 \\ j>0}} : \mathbf{D}_{i+|B|}^{(j)} = \mathbf{C}_i^{*(j)}, \tag{21}$$

where the $\frac{q}{2q_b}$ and $\frac{q}{2q_c}$ terms are used to convert the rounding operation into a flooring operation. Note that this is the same input preparation as step 0 of Algorithm 7 in [17].

We furthermore prepare the functions $f_0, \dots, f_L$ as:

$$f_{b,i}(x) = \begin{cases} 1 \text{ if: } x \le (\frac{q}{q_b} - 1)/p^i \\ 0 \quad \text{otherwise} \end{cases} \quad \text{and:} \quad f_{c,i}(x) = \begin{cases} 1 \text{ if: } x \le (\frac{q}{q_c} - 1)/p^i \\ 0 \quad \text{otherwise} \end{cases},$$

$$\tag{22}$$

for the coefficients of $\mathbf{B}$ and $\mathbf{C}$ respectively.

Prime q For prime moduli conversion, we follow the approach of Fritzmann et al. [22], where the compression is explicitly calculated for each share individually. This would result in an infinitely long bitstring, but Fritzmann et al. showed that it is sufficient to take into account a certain number of bits $f > \log_2(S) + \log_2\left(\frac{\lceil q/2 \rceil}{q} - 0.5\right)$, with S the number of shares. We end up with the following inputs:

$$\underset{\substack{i \in 0,\ldots,|B|-1}}{\forall} : \mathbf{D}_i{}^{(0)} = \lfloor \frac{q_b \cdot 2^f}{q} B_i^{*(0)} \rfloor + \frac{2^f}{2} - 2^f \cdot \mathbf{B}_i \ ; \ \underset{\substack{i \in 0,\ldots,|B|-1 \\ j>0}}{\forall} : \mathbf{D}_i{}^{(0)} = \lfloor \frac{q_b \cdot 2^f}{q} B_i^{*(j)} \rfloor \ ;$$

(23)

$$\underset{\substack{i \in 0,\ldots,|C|-1}}{\forall} : \mathbf{D}_{i+|B|}{}^{(\cdot)} = \lfloor \frac{q_c \cdot 2^f}{q} C_i^{*(0)} \rfloor + \frac{2^f}{2} - 2^f \cdot \mathbf{C}_i \ \text{and} \ \underset{\substack{i \in 0,\ldots,|C|-1 \\ j>0}}{\forall} : \mathbf{D}_{i+|B|}{}^{(0)} = \lfloor \frac{q_c \cdot 2^f}{q} C_i^{*(j)} \rfloor,$$

(24)

and moduli $q_b \cdot 2^f$ and $q_c \cdot 2^f$ respectively. The functions are constructed as:

$$f_{b,i}(x) = \begin{cases} 1 \text{ if: } x \le (2^f-1)/p^i \\ 0 \qquad \text{otherwise} \end{cases} \quad \text{and:} \quad f_{c,i}(x) = \begin{cases} 1 \text{ if: } x \le (2^f-1)/p^i \\ 0 \qquad \text{otherwise} \end{cases},$$

(25)

Again note that this is the same input preparation as step 0 of Algorithm 7 in [17].

A2B compression / MSB extraction Our arithmetic to 1-bit Boolean can also be used to securely implement the A2B conversion in lattice-based encryption schemes. To be more precise, it can replace the A2B conversion where one is only interested in the most significant bit, which is typically the case in the decoding for schemes like Saber and Kyber. To find the most significant bit of a number $A^{(\cdot)}$ in case of a power of two moduli, one inputs $D_0{}^{(\cdot)} = A^{(\cdot)}$ with the modulus q equal to the arithmetic sharing modulus. The functions $f_0(), \ldots, f_L()$ can be constructed as $f_i(x) = 1$, with the exception of $f_L()$, which equals:

$$f_{b,i}(x) = \begin{cases} 0 \text{ if: } x < p_L/2 \\ 1 \quad \text{otherwise} \end{cases}$$

(26)

Note that the input $D^{(\cdot)}$ is in this case an array with only one coefficient.

Again, for prime moduli, we can perform a similar technique. The goal is to calculate the modulus switching function $\lfloor \frac{2}{q} x \rceil$ on a masked variable. To do this, one also has the option to convert to power-of-two moduli using a trick similar to D'Anvers et al. [20] inspired by the technique of Fritzmann et al. [22]. In this case, we have:

$$D_0{}^{(\cdot)} = \lfloor \frac{2^{f+1}}{q} B^{(\cdot)} + S \rceil$$

(27)

with modulus 2^{f+1}. The function is calculated similar to before as $f_i(x) = 1$, again with the exception of $f_L()$, which equals:

$$f_{b,i}(x) = \begin{cases} 0 \text{ if: } x < p_L/2 \\ 1 \quad \text{otherwise} \end{cases}$$

(28)

18

The reason for the multiplication with 2^{f+1} and addition of S is to preserve correctness even in the presence of flooring errors. The division with q creates an infinitely long fractional part, which the subsequent operation floors down. This means that an error in $(-1, 0]$ is introduced to all shares:

$$D_0^{(\cdot)} = \frac{2^{f+1}}{q} B^{(\cdot)} + S \cdot (1 + e) \tag{29}$$

To prove that this operation always gives the correct result, we investigate the border cases $B^{(\cdot)} = 0$ and $B^{(\cdot)} = \lfloor q/2 \rfloor$, which should result in $D_0^{(\cdot)} \in [0, 2^f)$; and $B^{(\cdot)} = \lceil q/2 \rceil$ and $B^{(\cdot)} = q - 1$, which should result in $D_0^{(\cdot)} \in [2^f, 2^{f+1})$. If these conditions are fulfilled the top bit is correct and the value of $D_0^{(\cdot)}$ will be valid. Note that since q is uneven we have $\lfloor q/2 \rfloor = (q-1)/2$ and $\lceil q/2 \rceil = (q+1)/2$.

The cases of $B^{(\cdot)} = 0$ and $B^{(\cdot)} = \lceil q/2 \rceil$ can only go wrong in negative wrap around, and so the worst-case scenario is $e = -1$. This results in:

$$D_0^{(\cdot)} = S \cdot (1 - 1) \geq 0 \quad \text{and} \quad D_0^{(\cdot)} = 2^f \frac{q+1}{q} + S \cdot (1 - 1) \geq 2^f \tag{30}$$

which is always fulfilled.

The cases of $B^{(\cdot)} = q - 1$ and $B^{(\cdot)} = \lfloor q/2 \rfloor$ can only go wrong in positive wrap around, and so the worst-case scenario is $e = 0$. This results in:

$$D_0^{(\cdot)} = 2^{f+1} \frac{q-1}{q} + S < 2^{f+1} \quad \text{and} \quad D_0^{(\cdot)} = 2^f \frac{q-1}{q} + S < 2^f \tag{31}$$

which results in conditions $S < \frac{2^{f+1}}{q}$ and $S < \frac{2^f}{q}$, of which the latter is the most restrictive. Therefore, as long as $f > \log_2(S) + \log_2(q)$ we have a correct most significant bit and thus a correct MSB extraction.

6 Implementation aspects

The algorithms given above are not necessarily the most efficient way to implement one-hot conversions on a variety of computing platforms. In this section, we detail methods to speed up these conversion algorithms. We first look at possible tweaks in software implementations and then look at parallelization possibilities, which are typically more useful in hardware.

6.1 Software optimizations

The inner loop of our technique consists of two parts: secure rotation and bit selection. The secure rotation itself consists of two main instructions: a rotation and an XOR operation on the register. As such it is relatively easy to optimize in both software and hardware. The bit selection warrants a more in-depth look, and we will first look into the bit selection of the arithmetic to 1-bit Boolean conversion, and then look into the A2B conversion.

Bit selection In this paragraph we will specifically look at the bit selection of $R^{[\cdot]}$ (line 11, and line 14 in Algorithm 3). In a hardware implementation, one can implement these operations using a simple Boolean hardware circuit.

For software implementations, as we are working within a register, an efficient implementation is more challenging. To get a feel for the cost we will describe the cost of algorithms in the number of XOR that needs to be performed, taking this measure because it is the main operation in the innermost loop in the code. We will specifically look at the power-of-two q case and a subfunction that considers each bit individually, i.e. a function $f_i(x)$ that can be written as:

$$f_i(x) = f_{i,0}^*(x[0]) \text{ AND } f_{i,1}^*(x[1]) \text{ AND } \dots \text{ AND } f_{i,|X|-1}^*(x[|X|-1]). \quad (32)$$

This is the case that covers the typical applications from subsection 5.3.

A straightforward approach would be to perform the XORs one by one, which would lead to $S^2 \cdot (|f_i| - 1)$ XOR operations, where $|f_i|$ denotes the number of inputs to which the function f_i returns 1. This can be brought back to less than $S^2 + S \cdot \log_2(|f_i|)$ XOR operations using two tricks: exploiting inherent parallelism and a divide-and-conquer combination approach.

Firstly, the inherent parallelism comes from the fact that the XOR for positions in different carry bits but with the relative position can be calculated at the same time, by exploiting the fact that the different carry bins are exactly p positions separated. As such, when performing the XOR operation on the full register on line 11 and line 14, one is not only calculating the result for carry $c = 0$, but also for all other carries c, the result of which can be found $c \cdot p$ positions further in the register.

Secondly, one can speed up the calculations using a divide-and-conquer strategy. There are three possible instantiations for $f_i^*(x)$:

$$f_i^*(x) = x, \quad f_i^*(x) = NOT(x), \quad f_i^*(x) = 1 \quad (33)$$

Note that $f_i^*(x) = 0$ is not an option, as this would mean that $f(x) = 0$ which is a useless function to implement. The number of positions that needs to be propagated during bit selection in loop l can be calculated as:

$$\prod_{i=0}^{\log_2(p)-1} |f_i^*(x[i])| \quad (34)$$

For the functions $f_i^*(x[i]) = x$ and $f_i^*(x[i]) = NOT(x)$, $|f_i^*(x[i])|$ is one and thus the number of positions to be considered is not increased. However, for function of the form $f_i^*(x) = 1$, the number of positions is doubled. More specifically, for each position that is propagated, a position exactly 2^i further is also propagated.

We address such an instance by shifting the register $R^{[\cdot]}$ with 2^i positions and XORing it with the original register. This operation essentially combines the scenario where $x[i] = 0$, with the scenario where $x[i] = 1$, and puts both options at the position as if $x[i] = 0$. Thus, after this operation, the original function $f_i^*(x) = 1$ needs to be replaced with $f_i^*(x) = x$ to obtain the same result. Once

all $f_i^*(x) = 1$ are replaced by $f_i^*(x) = x$, there is only one position left to be considered, more specifically this is position $F = \sum_{i=0}^{\log_2(p)-1} 2^i \cdot f_i^*(0)$

Algorithm 4 gives a faster implementation of the bit selection, where in lines 1-6 the inherent parallelism and the divide-and-conquer combination are exploited. Line 7 is a cleanup where the XORed value for each carry c is placed at the $c \cdot p^{\text{th}}$ position and all other positions are set to zero, after which lines 8-11 copy the carry bits to their final position c.

Postprocessing A2B The bit selection in the A2B conversion can be optimized in the same ways as in the arithmetic to 1-bit Boolean conversion detailed above. However, for the A2B conversion, there is an additional step to calculate $B_{tmp}{}^{[\cdot]}$ which can be optimized significantly. In this paragraph, we will discuss two optimizations.

The first algorithm uses the same divide-and-conquer combination to combine the different carry bins, after which the multiplication operation is calculated p times. This algorithm is depicted in Algorithm 5 and is efficient as long as p is a small value. It takes $S \cdot (p-1)$ multiplications and $S \cdot (S+p-2)$ XOR operations.

The second algorithm is aimed at a higher value p. For this, we take a step back at the bits of $B_{tmp}{}^{[\cdot]}$, which are calculated as:

$$B_{tmp}{}^{[\cdot]} = \bigoplus_{t=0}^{p-1} t \cdot \left(\bigoplus_{c=0}^{|R|/p-1} R_{tmp}{}^{[\cdot]}[c \cdot p_L + t] \right) \tag{35}$$

Note that the second term of the multiplication is a single bit with a value of 0 or 1. When looking at a specific bit of the output $B_{tmp}{}^{[\cdot]}[i]$, this equation can be further simplified:

$$B_{tmp}{}^{[\cdot]}[i] = \bigoplus_{\substack{\forall t=0,\ldots,p-1 : t[i]=1 \\ c=0,\ldots,|R|/p_L-1}} R_{tmp}{}^{[\cdot]}[c \cdot p + t] \tag{36}$$

$$= \texttt{parity}(R^{[\cdot]} \mathbin{\&} F_i) \quad \text{with: } F_i = \bigoplus_{\substack{\forall t=0,\ldots,p-1 : t[i]=1 \\ c=0,\ldots,|R|/p_L-1}} 2^{c \cdot p+t}. \tag{37}$$

In essence, F_i is a mask that selects all terms involved in the XOR operation. For example, for $i = 0$, $F_i = 0101\ldots01$ and for $i = 1$, $F_i = 00110011\ldots0011$. The resulting algorithm is depicted in Algorithm 6.

The cost of this second algorithm heavily depends on the instruction set of the processor. If a parity instruction (or Hamming weight instruction) is present, the algorithm takes $S \cdot \log_2(p)$ parity instructions. If this instruction is not present, one can compute the parity with a divide-and-conquer strategy which would cost $\lceil \log_2(S \cdot p) \rceil$ operations.

6.2 Parallelization

Our algorithm is inherently serial, as the output $R^{[\cdot]}$ of the previous chunk is necessary to start the calculations on the next chunk. This might be a bottleneck

Algorithm 4:

```
   // Get valid positions
1  F = 0
2  for i = 0 to log₂(p) − 1 do
3  │   if f*ᵢ(x) = 1 then
4  │   │   R[·] ⊕= R[·] ≫ 2ⁱ
5  │   else if f*ᵢ(x) = NOT(x) then
6  │   │   F = F + 2ⁱ
7  R[·] = (R[·] ≫ F) & Σᵢ₌₀ˢ⁻¹ 2ⁱ·ᵖ
   // Set carries
8  for i = 1 to log₂(S − 1) do
9  │   R[·] ⊕= R[·] ≫ (p − 1) · i
10 R[·] = R[·] & (2ˢ − 1)
```

Algorithm 4:

// Get valid positions
1. $F = 0$
2. **for** $i = 0$ **to** $\log_2(p) - 1$ **do**
3.   **if** $f^*_i(x) = 1$ **then**
4. $R^{[\cdot]} \oplus= R^{[\cdot]} \gg 2^i$
5.   **else if** $f^*_i(x) = NOT(x)$ **then**
6. $F = F + 2^i$
7. $R^{[\cdot]} = (R^{[\cdot]} \gg F)\ \&\ \sum_{i=0}^{S-1} 2^{i \cdot p}$
// Set carries
8. **for** $i = 1$ **to** $\log_2(S - 1)$ **do**
9. $R^{[\cdot]} \oplus= R^{[\cdot]} \gg (p - 1) \cdot i$
10. $R^{[\cdot]} = R^{[\cdot]}\ \&\ (2^S - 1)$

Algorithm 5:

1. **for** $i = 1$ **to** $\log_2(S - 1)$ **do**
2. $R^{[\cdot]} \oplus= R^{[\cdot]} \gg 2^i \cdot p$
3. **for** $i = 1$ **to** $p - 1$ **do**
4. $R^{[\cdot]} \oplus= i \cdot R^{[\cdot]}[i]$

Algorithm 6:

1. **for** $i = 0$ **to** $\log_2(p) - 1$ **do**
2. $B^{[\cdot]}[i] = \texttt{parity}(R^{[\cdot]}\ \&\ F_i)$

for the masked comparison operation as used in lattice-based cryptography as described in subsection 5.3. In such a scenario one has typically an input array $\mathbf{D}^{(\cdot)}$ that has between 768 and 1280 coefficients that need to be validated. In this section, we will show how to make a parallel implementation on n 'cores' with minimal overhead.

At the start, one divides the array $\mathbf{D}^{(\cdot)}$ in n arrays of approximately $|\mathbf{D}^{(\cdot)}|/n$ elements. These sub-arrays are then validated separately on the n cores, which results in n registers $R_0^{[\cdot]}$ to $R_{n-1}^{[\cdot]}$. The LSB of each of these registers is a Boolean masked bit representing the result of the comparison of the corresponding sub-array (i.e., $R_i^{[\cdot]}[0] = 1$ if the corresponding sub-array was valid, and 0 if it was invalid).

To combine these registers, one can use the fact that one Boolean masked bit is essentially an arithmetic masked bit modulo 2. To combine $R_0^{[\cdot]}$ and $R_1^{[\cdot]}$ we perform another iteration of the arithmetic to 1-bit Boolean with these inputs: $\forall_k \mathbf{D}^{(k)} = R_1^{[k]}[0]$ with arithmetic masking modulus 2, $R^{[\cdot]} = R_0^{[\cdot]}$ and $f(x) = NOT(x)$. The output of this iteration is a register $R^{[\cdot]}$ that is 1 if both $R_0^{[\cdot]}$ and $R_1^{[\cdot]}$ were 1, and 0 otherwise.

Taking a step back we can see that the above paragraph uses the arithmetic to 1-bit Boolean technique to construct a masked AND gate on Boolean masked bits. By applying this AND gate on all $R_0^{[\cdot]}$ to $R_{n-1}^{[\cdot]}$ we end up with one register denoting the result of the masked comparison.

In a serial implementation, given $|\mathbf{D}^{(\cdot)}|$ coefficients and L chunks for each coefficient, the masked comparison takes $L \cdot |\mathbf{D}^{(\cdot)}|$ iterations (we count `SecureRotate` and the bit selection as one iteration). In the parallelized method we additionally have to perform $n - 1$ iterations to combine the sub-array $R_i^{[\cdot]}$, which increases the cost only slightly to $L \cdot |\mathbf{D}^{(\cdot)}| + n - 1$ iterations. For masked comparison of Saber, where $|\mathbf{D}^{(\cdot)}| = 1024$ and $L = 4$, performing the calculations in parallel on 4 cores would increase the cost from 4096 to 4099 iterations.

7 Validation

In this section, we compare the one-hot A2B conversion and masked comparison to state-of-the-art alternatives. We benchmarked the algorithms on an STM32F407 board with an ARM-Cortex M4F using `arm-none-eabi-gcc` version 9.2.1 with -O3. The system clock was set to 24 Mhz and TRNG clock to 48 Mhz, following the popular benchmarking framework PQM4 [KRSS]. One important factor in the benchmarking of these conversions is the limited throughput of the TRNG available on our processor. Therefore, we provide both benchmarks where the randomness cost is disregarded (i.e., it is sourced from a precomputed array of random elements), and where the randomness is sampled from the on-chip TRNG and its sampling cost is included in the cycle counts. Note that these implementations are only for reference and are not side-channel secured, as such implementations are outside the scope of this work but would be interesting for future work.

7.1 A2B conversion

In table 2 and table 3, our one-hot A2B algorithm is compared with the state-of-the-art table-based conversion by Coron et al. [16], using their publicly available code. We also compare with the Boolean circuit-based A2B algorithm by Coron et al. [11], and additionally with the optimized bitsliced implementation of D'Anvers et al. [17]. The top results in the table give the cycle counts without the waiting effect of the TRNG, while the bottom results include the TRNG wait time.

bits	8-bit		16-bit		32-bit	
order	2	3	2	3	2	3
Bool. circ. [11]	228.7	402.4	442.6	767.1	862.5	1484.7
Bool. circ. (optimized bitsliced) [11,17]	37.3	55.1	72.3	**108.2**	142.6	214.6
Table-based [16]	427.2	916.2	847.2	1806.6	1647.8	3514,8
One-hot [ours]	**27.3**	**51.2**	**54.3**	109.6	**103.3**	**206.4**
When sampling the randomness from the on-chip TRNG generator:						
Bool. circ. [11]	294.1	532.9	560.2	1002.0	1084.5	1928.6
Bool. circ. (optimized bitsliced) [11,17]	**43.2**	**67.1**	**84.8**	**133.3**	**168.2**	**265.9**
Table-based [16]	767.8	1617.4	1524.1	3213.0	3005,8	6338.3
One-hot [ours]	47.0	90.4	103.3	207.5	201.3	408.2

Table 2. Cost to perform 32 A2B conversions on Cortex M4 in 1000 cycles. The top results ignore randomness sampling using the on-chip TRNG generator, the bottom results include the randomness sampling.

We first compare our one-hot conversion to the state-of-the-art table-based conversion, as they are in the same family. As you can see from table 2, our new conversion improves the state-of-the-art table-based conversion with approximately a factor of 15. Similarly, the randomness usage is also reduced with a

bits	8-bit		16-bit		32-bit	
order	2	3	2	3	2	3
Bool. circ. [11]	5,120	10,240	9,216	18,432	17,408	34,816
Bool. circ. (opt. bitsliced) [11,17]	**464**	**928**	**976**	**1,952**	**2,000**	**4,000**
Table-based [16]	26,624	55,296	53,248	110,592	106,496	221,184
One-hot [ours]	1,536	3,072	3,840	7,680	7,680	15,360

Table 3. Randomness cost to perform 32 A2B conversions in bytes.

factor of 15 in the one-hot encoding. From this, we can conclude that the one-hot conversion is an improved version of the table-based conversion of Coron [16] in both cycle count (x15) and randomness usage (x15), and as such it is the fastest table-based full A2B conversion algorithm available at the moment[1].

A comparison to the Boolean circuit method is more complex. First one can notice that the optimized and bitsliced method significantly outperforms a straightforward implementation of the Boolean circuit method. Note that in contrast to the optimized bitsliced implementation, the one-hot implementation provided in this paper is a proof of concept and not a fully optimized implementation, which we leave for future work.

Compared to the one-hot encoding, while reducing the randomness cost of table-based methods by around a factor of 15, the randomness required for the one-hot encoding is still approximately 4 times higher compared to the optimized bitsliced Boolean circuit implementation. Further reducing this randomness cost can thus be identified as an interesting focus for future work. Regarding the cycle count, the one-hot encoding is in most situations slightly faster (up to 27%) than the Boolean circuit method if the limited throughput of the TRNG is ignored. If using the on-chip TRNG, the Boolean circuit method becomes (up to 35%) faster.

The bitsliced optimized Boolean circuit implementation makes very efficient use of the processor instructions available through the use of the bitslicing. Similar optimizations are not implemented for the one-hot encoding. One example of an operation that could be optimized using the appropriate hardware support would be the sharewise operations. These operations are essentially a Boolean circuit with mostly (unmasked) XOR gates, which would be much more efficient in hardware or with the appropriate hardware support (e.g., a parity count or hamming weight instruction as discussion in section 6). It would be interesting for future work to compare both techniques in a hardware implementation.

One advantage of the one-hot A2B over a Boolean circuit-based A2B is that the security-critical non-linear part is fully contained in the small and elegant SecureRotate function, as all other operations are linear and thus can be performed

---

[1] Note that the numbers given in [16] (Table 6) depict algorithmic operation counts and not cycles in an actual implementation. As there is no one-to-one match between the algorithmic operation count and the cycle count (e.g., memory accesses might be more expensive than local operations) one should be careful in comparing these numbers.

share-wise. As such, implementors have a more clear view of the security-critical parts of the algorithm, which should make side-channel secure implementations easier.

7.2 Masked Comparison

We compare our one-hot masked comparison with the state-of-the-art comparison techniques as identified in [20]. These implementations are optimized versions of the comparison by Barthe et al. [6] optimized in [20] (simple optimized) and Coron et al. [15] optimized in [20] (streamlined hybrid). Table 4 gives an overview of the cycle and randomness cost of the various comparison algorithm for usage in Saber and Kyber. For these techniques we use the optimized bitsliced A2B implementation of [17]. The implementation of the one-hot comparison follows the design of Algorithm 3 with the optimizations discussed in section 6.

		Cycles w/o TRNG		Cycles with TRNG		Randomness	
Order		2	3	2	3	2	3
simple optimized [6,20]	Kyber	2.5M	4.1M	3.1M	5.3M	48K	100K
streamlined hybrid [15,20]	Kyber	2.4M	3.4M	3.3M	4.4M	80K	95K
one-hot (ours)	Kyber	2.3M	4.3M	4.6M	8.9M	184K	369K
simple optimized [6,20]	Saber	1.3M	2.0M	1.6M	2.6M	26K	53K
one-hot (ours)	Saber	1.0M	2.0M	2.2M	4.2M	92K	184K

Table 4. Cycle and randomness cost of the state-of-the-art higher-order comparison methods

Due to the improvements of the one-hot conversion, masked comparison based on table-based A2B now performs with similar performance to Boolean circuit A2B based solutions. However, the randomness consumption is still a factor 2 to 4 higher than the Boolean circuit A2B based conversions, which confirms the importance of future work on randomness reduction or reuse as stated above.

One difference between the techniques is the code complexity. The one-hot comparison only consists of one main loop that loops over all chunks and for each chunk performs a secure rotation and a bit selection. As such this technique has a low implementation complexity even compared to the simple optimized method. Moreover, the bit selection is performed share-wise and should therefore be relatively easy to implement securely. Therefore, the critical part for secure implementations is mainly contained in the secure rotation and as such limits the scope of critical code parts that need to be addressed for a secure implementation.

Additionally, the streamlined hybrid method is prone to a small collision probability in which the comparison returns an incorrect result. This probability can be made arbitrarily small at the cost of losing efficiency. The simple

optimized and one-hot comparisons are always correct and thus do not suffer from collisions.

8 Conclusions and Future work

In this paper, we introduced a new table-based arithmetic to Boolean conversion. We also showed how to adapt our new method to efficiently perform masked comparison or extraction of the most significant bit, operations which are important for the masking of lattice-based post-quantum schemes. Additionally, an interesting property of our conversion is that one can perform a wide range of functions on the masked data during the transformation at low to no cost, which could be useful in future applications.

Our A2B method is 15 times faster than state-of-the-art table-based conversions and reduces the randomness consumption by a factor of 15. The resulting scheme still consumes approximately 4 times more randomness than the state-of-the-art bitsliced optimized Boolean circuit based A2B, but can be (depending on the throughput of the TRNG) up to 27% faster. Given that higher-order A2B conversion algorithms using Boolean circuit-based A2B have been around for longer and that they have undergone more optimizations both on an algorithmic and implementation level, the relatively new higher-order table-based A2B conversions might be able to bridge the remaining performance gap in the future.

Future work could include looking at adaptations to make one-hot conversions more efficient, or to apply them in different contexts and for different types of conversions. Reduction of the randomness usage by the one-hot conversion might be an interesting research topic. One could also look at algorithmic or implementation optimizations. Note that possible optimizations to the algorithm will be different on different platforms, for example in a microprocessor the register size is typically fixed by the bitwidth of the processor, while hardware implementation has more slack in choosing the size. For hardware implementations, the Boolean circuit nature of the sharewise operations might lead to significant speedups. Implementing and lab verification of a practically secure one-hot conversion might also be interesting future work.

Another point of interest could be specific first-order versions of the one-hot conversion. In first-order table-based implementations, one can typically re-use randomness over multiple encodings, and thus the randomness cost can be reduced dramatically. As in this scenario the randomness is no longer a limiting factor, this could possibly lead to very efficient designs for first-order.

In terms of extending the reach of the algorithm, one could look into applying the one-hot conversion ideas to improve Boolean to arithmetic conversion or first-order comparison methods. In the future, other more exotic functions might be implementable using the technique (e.g., checking smallness of a vector). It would also be interesting to integrate the one-hot conversion algorithms in post-quantum schemes such as Kyber, Dilithium and Falcon.

Acknowledgements

References

1. Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Kelsey, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D.: Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process (2020), `https://csrc.nist.gov/publications/detail/nistir/8309/final`

2. Amiet, D., Curiger, A., Leuenberger, L., Zbinden, P.: Defeating NewHope with a single trace. In: Ding, J., Tillich, J.P. (eds.) Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020. pp. 189–205. Springer, Heidelberg (2020). `https://doi.org/10.1007/978-3-030-44223-1_11`

3. Atici, A.C., Batina, L., Gierlichs, B., Verbauwhede, I.: Power analysis on ntru implementations for RFIDs: First results (2008)

4. Bache, F., Paglialonga, C., Oder, T., Schneider, T., Güneysu, T.: High-speed masking for polynomial comparison in lattice-based kems. IACR TCHES **2020**(3), 483–507 (2020). `https://doi.org/10.13154/tches.v2020.i3.483-507`, `https://tches.iacr.org/index.php/TCHES/article/view/8598`

5. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.A., Grégoire, B., Strub, P.Y., Zucchini, R.: Strong non-interference and type-directed higher-order masking. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 116–129. ACM Press (Oct 2016). `https://doi.org/10.1145/2976749.2978427`

6. Barthe, G., Belaïd, S., Espitau, T., Fouque, P.A., Grégoire, B., Rossi, M., Tibouchi, M.: Masking the GLP lattice-based signature scheme at any order. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 354–384. Springer, Heidelberg (Apr / May 2018). `https://doi.org/10.1007/978-3-319-78375-8_12`

7. Bhasin, S., D'Anvers, J.P., Heinz, D., Pöppelmann, T., Van Beirendonck, M.: Attacking and defending masked polynomial comparison. IACR TCHES **2021**(3), 334–359 (2021). `https://doi.org/10.46586/tches.v2021.i3.334-359`, `https://tches.iacr.org/index.php/TCHES/article/view/8977`

8. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., van Vredendaal, C.: Masking kyber: First- and higher-order implementations. IACR TCHES **2021**(4), 173–214 (2021). `https://doi.org/10.46586/tches.v2021.i4.173-214`, `https://tches.iacr.org/index.php/TCHES/article/view/9064`

9. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (Aug 1999). `https://doi.org/10.1007/3-540-48405-1_26`

10. Coron, J.S., Großschädl, J., Tibouchi, M., Vadnala, P.K.: Conversion from arithmetic to Boolean masking with logarithmic complexity. In: Leander, G. (ed.) FSE 2015. LNCS, vol. 9054, pp. 130–149. Springer, Heidelberg (Mar 2015). `https://doi.org/10.1007/978-3-662-48116-5_7`

11. Coron, J.S., Großschädl, J., Vadnala, P.K.: Secure conversion between Boolean and arithmetic masking of any order. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 188–205. Springer, Heidelberg (Sep 2014). `https://doi.org/10.1007/978-3-662-44709-3_11`

12. Coron, J.S., Gérard, F., Trannoy, M., Zeitoun, R.: High-order masking of ntru. Cryptology ePrint Archive, Paper 2022/1188 (2022), `https://eprint.iacr.org/2022/1188`, `https://eprint.iacr.org/2022/1188`

13. Coron, J.S., Rondepierre, F., Zeitoun, R.: High order masking of look-up tables with common shares. IACR TCHES **2018**(1), 40–72 (2018). `https://doi.org/10.13154/tches.v2018.i1.40-72`, `https://tches.iacr.org/index.php/TCHES/article/view/832`

14. Coron, J.S., Tchulkine, A.: A new algorithm for switching from arithmetic to Boolean masking. In: Walter, C.D., Koç, Çetin Kaya., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 89–97. Springer, Heidelberg (Sep 2003). `https://doi.org/10.1007/978-3-540-45238-6_8`

15. Coron, J.S., Gérard, F., Montoya, S., Zeitoun, R.: High-order polynomial comparison and masking lattice-based encryption. Cryptology ePrint Archive, Report 2021/1615 (2021), `https://ia.cr/2021/1615`

16. Coron, J.S., Gérard, F., Montoya, S., Zeitoun, R.: High-order table-based conversion algorithms and masking lattice-based encryption. Cryptology ePrint Archive, Report 2021/1314 (2021), `https://ia.cr/2021/1314`

17. D'Anvers, J.P., Heinz, D., Pessl, P., van Beirendonck, M., Verbauwhede, I.: Higher-order masked ciphertext comparison for lattice-based cryptography. Cryptology ePrint Archive, Report 2021/1422 (2021), `https://ia.cr/2021/1422`

18. D'Anvers, J.P., Karmakar, A., Roy, S.S., Vercauteren, F.: SABER. Tech. rep., National Institute of Standards and Technology (2019), available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`

19. D'Anvers, J.P., Tiepelt, M., Vercauteren, F., Verbauwhede, I.: Timing attacks on error correcting codes in post-quantum schemes. In: Proceedings of ACM Workshop on Theory of Implementation Security Workshop. p. 2–9. TIS'19, Association for Computing Machinery, New York, NY, USA (2019). `https://doi.org/10.1145/3338467.3358948`, `https://doi.org/10.1145/3338467.3358948`

20. D'Anvers, J.P., Van Beirendonck, M., Verbauwhede, I.: Revisiting higher-order masked comparison for lattice-based cryptography: Algorithms and bit-sliced implementations. Cryptology ePrint Archive, Report 2022/110 (2022), `https://ia.cr/2022/110`

21. Debraize, B.: Efficient and provably secure methods for switching from arithmetic to Boolean masking. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 107–121. Springer, Heidelberg (Sep 2012). `https://doi.org/10.1007/978-3-642-33027-8_7`

22. Fritzmann, T., Van Beirendonck, M., Roy, D.B., Karl, P., Schamberger, T., Verbauwhede, I., Sigl, G.: Masked accelerators and instruction set extensions for post-quantum cryptography. Cryptology ePrint Archive, Report 2021/479 (2021), `https://eprint.iacr.org/2021/479`

23. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 537–554. Springer, Heidelberg (Aug 1999). `https://doi.org/10.1007/3-540-48405-1_34`

24. Goubin, L.: A sound method for switching between Boolean and arithmetic masking. In: Koç, Çetin Kaya., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 3–15. Springer, Heidelberg (May 2001). `https://doi.org/10.1007/3-540-44709-1_2`

25. Guo, Q., Johansson, T., Nilsson, A.: A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part II. LNCS, vol. 12171, pp. 359–386. Springer, Heidelberg (Aug 2020). `https://doi.org/10.1007/978-3-030-56880-1_13`

26. Heinz, D., Kannwischer, M.J., Land, G., Pöppelmann, T., Schwabe, P., Sprenkels, D.: First-order masked kyber on arm cortex-m4. Cryptology ePrint Archive, Report 2022/058 (2022), `https://ia.cr/2022/058`

27. Kundu, S., D'Anvers, J.P., Beirendonck, M.V., Karmakar, A., Verbauwhede, I.: Higher-order masked saber. Cryptology ePrint Archive, Paper 2022/389 (2022), `https://eprint.iacr.org/2022/389`, `https://eprint.iacr.org/2022/389`

28. Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., Stehlé, D., Bai, S.: CRYSTALS-DILITHIUM. Tech. rep., National Institute of Standards and Technology (2020), available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions`

29. Migliore, V., Gérard, B., Tibouchi, M., Fouque, P.A.: Masking Dilithium - efficient implementation and side-channel evaluation. In: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. (eds.) ACNS 19. LNCS, vol. 11464, pp. 344–362. Springer, Heidelberg (Jun 2019). `https://doi.org/10.1007/978-3-030-21568-2_17`

30. NIST Computer Security Division: Post-Quantum Cryptography Standardization (2016), `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`

31. Oder, T., Schneider, T., Pöppelmann, T., Güneysu, T.: Practical CCA2-secure masked Ring-LWE implementations. IACR TCHES **2018**(1), 142–174 (2018). `https://doi.org/10.13154/tches.v2018.i1.142-174`, `https://tches.iacr.org/index.php/TCHES/article/view/836`

32. Prest, T., Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: FALCON. Tech. rep., National Institute of Standards and Technology (2020), available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions`

33. Primas, R., Pessl, P., Mangard, S.: Single-trace side-channel attacks on masked lattice-based encryption. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 513–533. Springer, Heidelberg (Sep 2017). `https://doi.org/10.1007/978-3-319-66787-4_25`

34. Ravi, P., Roy, S.S., Chattopadhyay, A., Bhasin, S.: Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. IACR TCHES **2020**(3), 307–335 (2020). `https://doi.org/10.13154/tches.v2020.i3.307-335`, `https://tches.iacr.org/index.php/TCHES/article/view/8592`

35. Reparaz, O., Roy, S.S., Vercauteren, F., Verbauwhede, I.: A masked ring-LWE implementation. In: Güneysu, T., Handschuh, H. (eds.) CHES 2015. LNCS, vol. 9293, pp. 683–702. Springer, Heidelberg (Sep 2015). `https://doi.org/10.1007/978-3-662-48324-4_34`

36. Schneider, T., Paglialonga, C., Oder, T., Güneysu, T.: Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In: Lin, D., Sako, K. (eds.) PKC 2019, Part II. LNCS, vol. 11443, pp. 534–564. Springer, Heidelberg (Apr 2019). `https://doi.org/10.1007/978-3-030-17259-6_18`

37. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehlé, D.: CRYSTALS-KYBER. Tech. rep., National Institute of Standards and Technology (2020), available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions`

38. Silverman, J.H., Whyte, W.: Timing attacks on NTRUEncrypt via variation in the number of hash calls. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 208–224. Springer, Heidelberg (Feb 2007). `https://doi.org/10.1007/11967668_14`

39. Ueno, R., Xagawa, K., Tanaka, Y., Ito, A., Takahashi, J., Homma, N.: Curse of re-encryption: A generic power/em analysis on post-quantum kems. Cryptology ePrint Archive, Report 2021/849 (2021), `https://ia.cr/2021/849`

40. Van Beirendonck, M., D'Anvers, J., Karmakar, A., Balasch, J., Verbauwhede, I.: A side-channel-resistant implementation of SABER. ACM JETC **17**(2), 10:1–10:26 (2021)

41. Van Beirendonck, M., D'Anvers, J.P., Verbauwhede, I.: Analysis and comparison of table-based arithmetic to boolean masking. IACR TCHES **2021**(3), 275–297 (2021). `https://doi.org/10.46586/tches.v2021.i3.275-297`, `https://tches.iacr.org/index.php/TCHES/article/view/8975`

42. Wang, A., Zheng, X., Wang, Z.: Power analysis attacks and countermeasures on ntru-based wireless body area networks. KSII Transactions on Internet and Information Systems (TIIS) **7**(5), 1094–1107 (2013)

43. Xu, Z., Pemberton, O., Roy, S.S., Oswald, D.: Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. Cryptology ePrint Archive, Report 2020/912 (2020), `https://eprint.iacr.org/2020/912`