Formal Verification of Arithmetic Masking in Hardware and Software

Barbara Gigerl[1], Robert Primas[1] and Stefan Mangard[1,2]

[1] Graz University of Technology, Graz, Austria
[2] Lamarr Security Research, Graz, Austria

Abstract. Masking is a popular secret-sharing technique that is used to protect cryptographic implementations against physical attacks like differential power analysis. So far, most research in this direction has focused on finding efficient Boolean masking schemes for well-known symmetric cryptographic algorithms like AES and Keccak. However, especially with the advent of post-quantum cryptography (PQC), arithmetic masking has received increasing attention from the research community. In practice, many PQC algorithms require a combination of arithmetic and Boolean masking, which makes the search for secure and efficient conversion algorithms between these domains (A2B/B2A) an interesting but very challenging research topic. While there already exist lots of tools that can help with the formal verification of Boolean masked implementations, the same cannot be said about arithmetic masking and accompanying mask conversion algorithms.

In this work, we demonstrate the first formal verification approach for (any-order) Boolean and arithmetic masking which can be applied to both hardware and software, while considering side-effects such as glitches and transitions. First, we show how a formal verification approach for Boolean masking can be used in the context of arithmetic masking such that we can verify A2B/B2A conversions for arbitrary masking orders. We investigate various conversion algorithms in hardware and software, and point out several new findings such as glitch-based issues for straightforward implementations of [CGV14]-A2B in hardware, transition-based leakage in Goubin-A2B in software, and more general implementation pitfalls when utilizing common optimization techniques in PQC. We provide the first formal analysis of table-based A2Bs from a probing security perspective and point out that they might not be easy to implement securely on processors that use of memory buffers or caches.

Keywords: Side-Channel Attacks · Arithmetic Masking · Formal Verification · Glitches

1 Introduction

Passive side-channel attacks, including power or electromagnetic analysis, are among the most relevant attack vectors against cryptographic devices like smart cards, that are physically accessible by an attacker [KJJ99, QS01]. A commonly used approach to protect cryptographic implementations against these attacks is to implement algorithmic countermeasures, for example masking [CRB$^+$16, GIB18, GMK16, ISW03, RBN$^+$15]. Masking schemes split input and intermediate values of cryptographic computations into $d + 1$ random shares such that observations of up to d shares do not reveal any information about the native (unmasked) value. Boolean masking schemes, where native values correspond to the XOR-sum over its shares, have received much attention since such schemes are applicable to almost all symmetric cryptographic algorithms. However, especially with the advent of post-quantum cryptography (PQC), arithmetic masking schemes have also gained increased importance in the research community since they generally represent a better fit for these kinds of algorithms. In arithmetic masking, native values correspond to the arithmetic addition over their shares which allows to express arithmetic operations much more efficiently than with Boolean masking. In practice however, arithmetic masking often has to be combined with Boolean masking since PQC algorithms often make use of various symmetric building blocks to achieve CCA2-security or for sampling random numbers [FO99, BGR+21, SPOG19]. Masked versions of these algorithms hence require efficient conversion techniques between Boolean and arithmetic masks, typically referred to as A2B/B2A conversions in literature, and are hence also the main focus of research on arithmetic masking. On top of that, many works in the past have shown that designing masked implementations with practical security requires a detailed understanding of the targeted hardware platform, as architectural side-effects like glitch or transition effects can violate basic assumptions of masking schemes such as independence of leakage. This makes the design of masked cryptographic implementations a notoriously error-prone and time consuming task, which is why there is strong need for verification tooling that supports this effort to the highest possible extend.

While there already exists a vast amount of literature on the verification of Boolean masking, including formal verification approaches like Rebecca [BGI+18], maskVerif [BBC+19], Coco (Alma) [GHP+21, HB21], SILVER [KSM20], or scVerif [BGG+21], the same cannot be said about arithmetic masking.

Limitations of Existing Approaches On the formal verification side, the first works on proving formal security specifically of arithmetic masking schemes were published with QMVerif by Gao et al. [GXZ+19] and `LeakageVerif` by Meunier et al. [MPH21]. While these works already form a good foundation, these tools are limited in several ways.

In 2019, Gao et al. published QMVerif, a tool for the verification of first-order Boolean and arithmetically masked software implementations [Gao20], which works in two steps. First, QMVerif uses *type inference* to efficiently compute the distribution of every internal program variable, which is either uniform, independent of private inputs, dependent on private inputs or unknown. This step leads to inconclusive results due to the lack of completeness guarantees of type inference, and false positives are very likely [MPH21]. In that case, QMVerif moves on to the *model-counting* step, and encodes the verification problem into a SAT equation to compute the exact missing probability distributions. While model-counting is complete, it is does not scale, which is also why they often need to resort to GPU acceleration [GXSC20]. This scalability issue also leads to the author's conclusion that model-counting-based masking verification approaches are generally infeasible in the context of arithmetic masking. Besides that, QMVerif is limited to masked software, the input program must follow a specific high-level syntax, must not include any branches, loops or functions, and variables are limited to 8 bit. The leakage model of QMVerif hence also does not consider hardware side-effects like glitches and transitions, and it is unclear wether QMVerif can be applied to conversions without a power-of-two-modulus. The same authors later propose HOME for higher orders following the same approach, but do not evaluate it for higher-order arithmetic masking. Since the tool is not (yet) open-source, it is not possible to investigate its further functionality.

More recently, Meunier et al. [MPH21] propose `LeakageVerif`, a verification library based on the substitution algorithm suggested by Barthe et al. [BBD+15]. Although the evaluation shows that `LeakageVerif` is more efficient than QMVerif, it does not provide a model-counting step in case the substitution fails, and is therefore not complete and fails to verify common A2B/B2A conversions such as Goubin-A2B [Gou01] and [CGV14]-B2A correctly. Additionally, the tool works for first-order software implementations only, does not consider glitches, cannot verify table lookups and is also not evaluated for moduli which are not a power of two. The masked implementation must be provided in Python.

In general, both QMVerif and `LeakageVerif` are *sound* (leakages are never missed), but can only achieve *completeness* (leaks are only reported if they really exist) in many scenarios if they fall back to expensive and exact model-counting. The evaluation executed by Meunier et al. [MPH21] shows that it is very likely that the model-counting step is necessary due to a high rate of false-positives, which makes both approaches inefficient in some cases.

In 2021, Bos et al. demonstrate the verification of a first-order masked software implementation of Kyber [BGR+21] using an extended version of the tool scVerif [BGG+21] that simulates typical leakage behaviour of ARM microprocessors when executing code that is written in an intermediate language close to ARM assembly. scVerif was not evaluated for other arithmetically masked programs, so no general statement about its efficiency or accuracy can be made. It does consider hardware side-effects but only if they have been identified in prior experiments, which means the method is not sound and binds the evaluation stronger to the microarchitecture, while leaving no potential for masked hardware.

Other existing verification tools for Boolean masking often also perform exact model-counting, and are therefore unlikely to be applicable to arithmetic masking due to scalability issues. For example, maskVerif has been shown infeasible in this context by several works [GXSC20, GZSW19, MPH21], while we expect SILVER [KSM20] to also not be able to deal with the complexity of arithmetic expressions since it tracks exact distributions with the help of binary decision diagrams. In 2018, Bloem et al. suggests to approximate Fourier coefficients of Boolean functions [BGI+18] as a way to perform a cheaper variant of model counting that achieves soundness but not completeness. The resulting approach was evaluated for Boolean masked hardware (Rebecca), and later for software on concrete CPU netlists (Coco) [GHP+21, HB21], and has shown to be efficient with a relatively low rate of false positives. However, it was not evaluated for arithmetic masking in terms of efficiency, accuracy, and general applicability for PQC relevant use cases.

Our Contribution We improve this situation by discussing in detail how the security of arithmetically masked software/hardware can be efficiently verified using verification approaches tailored to Boolean masking. More concretely, we provide the following contributions:

- We show how verification methods based on approximated Fourier coefficients of Boolean functions (as used by Rebecca/Coco) can be efficiently applied in the context of arithmetic masking. The resulting verification approach can successfully be applied to both masked hardware and masked software written in Assembly language, and that its soundness is sufficient for many PQC/ARX applications. This approach is also the first to consider physical defaults (glitches, transitions) and the first to be evaluated for higher-orders in the context of arithmetic masking.

- In case of hardware implementations, we analyze different versions of the [CGV14]-A2B/B2A conversion algorithms and identify potential weaknesses caused by glitches. We then present a proof-of-concept implementation that is secured against glitches and can be fully verified using our approach.

- In the context of software implementations, we analyze various popular A2B/B2A conversion algorithms using power of two or prime moduli and provide new insights on implementation aspects that can reduce their protection order. More concretely, we report new findings of transition leakages in Goubin-A2B [Gou01] and point out more general pitfalls when using lazy-reduction techniques in the context of masking. Additionally, we are the first to investigate architecture side-effects of table-based A2Bs and discuss why they might not be easy to implement securely on processors that make use of memory buffers or caches. Last but not least, we also show applicability of this approach in the context of symmetric cryptographic schemes by verifying the security of masked software implementations of one round of Speck and the ARX-box Alzette.

Outline We cover preliminaries on masking and verification techniques in Section 2. The main idea behind the verification approach is described in Section 3. Section 4 and Section 5 cover our findings of investigating various hardware/software A2B/B2A conversion algorithms. We conclude our paper in Section 6.

Open Source We plan to publish the software and hardware implementations on Github[1].

2 Background

In this section, we cover necessary background on Boolean and arithmetic masking, and corresponding conversion techniques. Since our approach is based on Rebecca/Coco, we briefly describe the verification concept and the applied adversary model.

2.1 Masking Schemes and Applications

Masking is a prominent algorithmic countermeasure against Differential Power Analysis [KJJ99] that splits intermediate values of a computation into $d + 1$ uniformly random shares [CRB+16, GIB18, GMK16, ISW03], such that an attacker who observes up to d shares cannot deduce information about native (unshared) intermediate values. Boolean masking is commonly used for symmetric cryptographic algorithms, and uses the exclusive or ($\oplus$) operation to split a value b into $d + 1$ uniformly random shares $b_0 \ldots b_d$ such that:

$$b = \bigoplus_i b_i = b_0 \oplus \cdots \oplus b_d$$

In arithmetic masking schemes, the relation between shares of a value a is the modular addition:

$$a = \sum_i a_i = a_0 + \cdots + a_d \mod q$$

In both cases, masking linear functions is trivial since they can simply be computed for each share individually. Masking non-linear functions is more challenging since these functions operate on all shares of a native value and thus usually require additional fresh randomness to avoid unintended direct combination of shares. The concrete masking technique, Boolean or arithmetic, determines which operations are (non-)linear.

Nowadays, one well-known application of arithmetic masking are post-quantum cryptographic (PQC) algorithms that often operate on large matrices or polynomials where each coefficient is an element in $\mathbb{F}_q$. Here, operations like matrix/polynomial multiplication can be efficiently masked in the arithmetic domain when broken down into coefficient-wise modular addition/multiplications using e.g. the number theoretic transform (NTT). However, in practice, arithmetic masking often has to be combined with Boolean masking since building blocks including Gaussian samplers and lattice decoding, or constructions like the Fujisaki-Okamoto transform for achieving CCA2-security are more efficiently masked in the Boolean domain. Therefore, many masked implementations use dedicated conversion algorithms to transform shares from the arithmetic to the Boolean domain (A2B) and vice versa (B2A). Besides PQC, arithmetic masking is also applied to ARX-based symmetric cryptographic algorithms like SHA-256 or ChaCha, but in these cases the resulting runtime overhead is significantly higher compared to Boolean masked variants of non-ARX symmetric algorithms.

[1] TBD

2.2 Mask Conversion Techniques

Many cryptographic schemes require to switch between the Boolean and the arithmetic domain when respective masking techniques are applied. The performance of the protected scheme is mainly determined by the A2B and B2A conversions used, which is why there has been a lot of research in this direction [CGV14, CGTV15, Cor17, Cor17, CGMZ22, CGP+12, SPOG19, SMG15]. Existing conversion algorithms either follow an *algebraic* or a *table-based* approach. An algebraic conversion algorithm performs the whole conversion at once, while table-based approaches first pre-compute a table which is later used during the actual conversion. B2A conversions can be done very efficiently following the algebraic approach, while A2B is less efficient, and therefore often apply a table-based approach.

In 2001, the first algebraic conversion algorithms were proposed by Goubin [Gou01]. The first algebraic conversion algorithms for higher orders were presented by Coron et al. [CGV14]. They propose the SecAdd algorithm, which allows to securely add Boolean shares at any order using a power-of-two modulus. Many follow-up works use [CGV14]-A2B/B2A as a basis, and suggest several performance improvements [CGTV15, BCZ18, HT19, Cor17]. Since PQC applications often require a prime modulus, Barthe et al. [BBE+18], and later Schneider et al. [SPOG19] suggest how to adapt [CGV14]-B2A to work with prime moduli.

Table-based A2B conversion algorithms use pre-computed tables to reduce the computation effort during the actual conversion. In general, A2B conversions transform the shares together with the carry which is produced in an arithmetic addition. The pre-computed tables are used to handle the conversion of the carry, and prevent unintended unmasking of native values. The first table-based A2Bs were suggested by Coron-Tchulkine [CT03] and Neiße-Pulkus [NP04]. They were however shown to be incorrect and insecure by Debraize [Deb12], who suggests several corrected and optimized versions of their algorithms. Recently, Beirendonck et al. [BDV21] show that Debraize-A2B does also not fulfill its security claims, and propose two further table-based A2Bs. So far, the work by Coron et al. [CGMZ22] is the only work on table-based conversion for higher orders.

A2B/B2A conversions are applied to masked implementations of various PQC and ARX schemes against side-channel attacks. For example, the SecAdd algorithm by Coron et al. [CGV14] has been used as a cryptographic primitive in several software [AFM17, GR19, BGR+21, CGTV15, SPOG19, BBE+18] and hardware implementations [FBR+21, CEvMS15]. Debraize-A2B has also been applied recently in works on masking PQC [BGR+21, OSPG18].

2.3 Masking Verification with Rebecca/Coco

Rebecca [BGI+18] is a tool to formally verify Boolean-masked hardware implementations defined by gate-level netlists. In order to verify a circuit, a label is assigned to each circuit input. The label is either a share, fresh randomness or unimportant. During the verification process, these labels are propagated through the circuit and each gate is assigned a correlation set according to the propagation rules. In general, a correlation set contains information about the statistical dependence of the respective gate on the circuit inputs. Tools like Silver [KSM20] compute these dependencies accurately, while Rebecca approximates statistical dependency with non-zero Fourier coefficients [BGI+18]. A term, which is either a label or a combination of labels, with a non-zero Fourier coefficient indicates statistical dependence on the respective circuit input. The approximation is performed by not tracking the exact Fourier coefficient, but only whether a term has a non-zero coefficient or not. A correlation set contains all terms with non-zero coefficients.

Later, an optimized variant of this approach was implemented in Coco, a tool for the formal verification of (any-order) Boolean masked software implementations on concrete CPUs [GHP+21, GPM21]. The main purpose of Coco is to analyze the potential implications of hardware side-effects like glitches within a CPU on masked software implementations. Coco can additionally incorporate control flow logic, which is required for the verification of executed software and iterative hardware circuits. Before the verification, the CPU netlist is simulated together with a masked assembly implementation, in order to obtain a trace of the (constant) data-independent control signals like memory/register access patterns and branches. Next, similar to Rebecca, initial labels are assigned to registers and memory locations, which are further propagated through the netlist for multiple cycles to construct correlation sets, while considering software-specific control signals. The verification fails if there exists a gate in the netlist which directly correlates with a native value. In that case, Rebecca reports the leaking gate, while Coco additionally reports the exact clock cycle.

2.4 Adversary Model

The classic probing model for hardware [ISW03] allows an attacker to observe the values of up to d wires in a masked circuit. The circuit is dth-order secure if the adversary is not able to learn anything about the native value by combining all these observations. The standard probing model for software allows the adversary to probe intermediate program values accordingly. Faust et al. [FGP+18] suggest the robust probing model as an extension of the classic probing model to additionally capture side-effects by the usage of (g, t, c)-extended probes to observe glitches ($g = 1$), transitions ($t = 1$) or coupling ($c = 1$).

In this work we use the so-called *time-constrained probing model*, which was introduced by [GHP+21], and is currently adopted by Coco for masked software implementations executed by a specific CPU. The main difference to the classic/robust probing model is the time restriction of each probe to one clock cycle, which is necessary to correctly model the execution of masked software on netlist level. More concretely, in the time-constrained probing model the attacker uses $(g, t, 0)$-extended probes to observe the value of any specific gate/wire in the CPU netlist for the duration of one clock cycle. The gate/wire and cycle can be chosen independently for each probe. A masked software implementation is dth-order secure in the time-constrained probing model if the attacker cannot learn anything about native values when combining all observed values.

The time-constrained probing model can be applied to masked hardware circuits and allows to handle *iterative* circuits directly without the need to perform *unrolling*[2] thanks to its time-awareness. In Appendix C we give an example of an iterative circuit and its unrolled version based on the suggestion of [BGSD10]. Verification approaches adopting the classic/robust probing model usually unroll the processed iterative circuit, which works well for simple circuits, but is more difficult for circuits with more complex control logic, such as state machines. Iterative circuits can be seen as a reduced version of a CPU, and therefore allows the direct application of the time-constrained probing model.

The original version of Coco provides two different verification modes. *Stable* verification focuses on pure algorithmic security. *Transient* verification uses $(g, t, 0)$-extended probes, and therefore considers algorithmic security and wire/register transitions and glitches within the hardware. For the purpose of this work, we add a third mode, the *Transitions* verification mode, working with $(0, t, 0)$-extended probes, which is convenient since it reports stable and transition leaks, but without the runtime overhead of the transient mode.

3 Verification of Arithmetic Masking in the Boolean Domain

In this section, we explain how one can perform verification of arithmetic masking using a method based on approximating Fourier coefficients of Boolean functions that was

[2]We apply the definitions of [BGSD10], since there exists no official wording yet.

previously used by the tools REBECCA/COCO in the context of Boolean masking. In Section 3.1 we recall how arithmetic expressions, when directly broken down into equivalent Boolean expressions, can be represented at bit-level. In Section 3.2 we discuss optimization strategies that can be used to reduce the complexity of the derived Boolean expressions for the initial labeling and to more efficiently propagate expressions through dedicated arithmetic addition circuits. We also comment on the soundness and completeness of the resulting approach. Finally, we give a small self-contained example using correlation set notation in Section 3.3.

Notation Similar to Coron et al. [CGV14], we identify a specific bit at position i of a variable a by superscript, which denotes with $a^{(i)}$ the i-th bit of variable a. a^0 is the least significant bit (LSB). When considering a Boolean or arithmetic native variable x, we identify the individual shares by subscript, for example $x = x_1 \oplus x_2 \oplus ....$ We apply the correlation set notation of Bloem et al. [BGI+18], which denotes the correlation set of a gate/wire w by $\mathcal{C}(w) = \{...\}$. $\otimes$ denotes the element-wise multiplication of two correlation sets as introduced in [GHP+21]. We use small letters for symbolic expressions, while capital letters are used to identify wires in a circuit.

3.1 Modeling Arithmetic Expressions using Boolean Logic

A netlist represents a circuit design after logic synthesis that models gates as Boolean functions mapping 1-bit inputs to a 1-bit output, and indicates their interconnection. We aim at performing netlist-level verification of a circuit on bit granularity. In the end, a bitwise view on all terms computed by the circuit must still valid in the context of masking. This implies that the dependencies between the shares must be described using Boolean equations on bit granularity. Such a mapping can be obtained based on the definition of the Ripple-carry adder, which represents a cascade of 1-bit full adders, where each carry bit *ripples* to the next full adder. Each full adder takes two 1-bit summands and a 1-bit carry-in, and computes the arithmetic sum and respective carry-out [Man82].

Consider a sum s, which is computed from the summands u and v such that $s = u + v$. If u and v are n-bit values, s is represented by $n + 1$ bits, and hence, $n + 1$ full adders are needed to compute s. Each full adder takes two summand bits $u^{(i)}$ and $v^{(i)}$ together with the carry-in $c^{(i)}$, and computes $s^{(i)}$ as:

$$s^{(i)} = u^{(i)} \oplus v^{(i)} \oplus c^{(i)} \text{ with } c^{(0)}=0 \tag{1}$$

The carry-out bit $c^{(i+1)}$ is then computed based on the carry-in $c^{(i)}$ by the following recursive formula:

$$c^{(i+1)} = (u^{(i)} \oplus v^{(i)}) \wedge c^{(i)} \vee (u^{(i)} \wedge v^{(i)}) \tag{2}$$

Equation 1 already gives a valid first-order Boolean sharing for s using the two shares $x_1 = u$ and $x_2 = v \oplus c$.

If a sum t is split into three summands u, v and w such that $t = u + v + w$, basically the same equations apply, and t can be computed in two steps. In the first step, the partial sum $s = u + v$ is computed, which yields the carry c. In the second step, t is computed by adding the partial sum to the remaining summand: $t = s + w$, which produces the carry e. Again, we can represent t in Boolean logic:

$$\begin{aligned} t^{(i)} &= \quad s^{(i)} \quad \oplus w^{(i)} \oplus e^{(i)} && (3) \\ &= u^{(i)} \oplus v^{(i)} \oplus c^{(i)} \quad \oplus w^{(i)} \oplus e^{(i)} && (4) \end{aligned}$$

Equation 4 gives a valid second-order Boolean sharing for t using three shares $x_1 = u, x_2 = v$ and $x_3 = w \oplus c \oplus e$. Formulas for more than three summands can be derived in a similar way, each resulting in a valid higher-order sharing. When working with $d + 1$

Boolean sharing

$$\underset{b_0^{(i)}}{\boxed{a^{(i)} \oplus r^{(i)}}} \oplus \underset{b_1^{(i)}}{\boxed{r^{(i)}}}$$

Arithmetic sharing

$$\underset{b_0^{(i)}}{\boxed{a^{(i)} \oplus r^{(i)} \oplus c^{(i)}}} \oplus \underset{b_1^{(i)}}{\boxed{r^{(i)}}}$$

Figure 1: Initial labeling for Boolean and arithmetic masking as given to the verifier

shares, the first d Boolean shares would always be equal to the first d arithmetic shares, while the last Boolean share needs to additionally include the carry.

3.2 Tailoring the Verification Approach

Arithmetically masked circuits process arithmetic input shares, while REBECCA/COCO expects Boolean input shares. The derived Boolean equations for arithmetic expressions in Section 3.1 can now be used to translate arithmetic shares to the Boolean domain, such that REBECCA/COCO could work with it. In the following, we describe how one can obtain such a translation in a correct and efficient way, how the resulting expressions can be propagated more efficiently in some cases, and comment on soundness, completeness and scalability of the resulting approach.

Initial Labeling Tools for the formal verification of masking require a set of initial labels that specify the location/dependency of shares on circuit inputs, registers or memory cells that are then further tracked throughout a circuit. In the case of (first-order) Boolean masking, each bit of a native value $a^{(i)}$ is initially masked with a random mask $r^{(i)}$. Therefore, the native value $a^{(i)}$ can simply be expressed as the XOR between the two shares $a^{(i)} \oplus r^{(i)}$ and $r^{(i)}$. As shown in Figure 1, the labels assigned prior to the verification would then be $b_0^{(i)} = a^{(i)} \oplus r^{(i)}$ and $b_1^{(i)} = r^{(i)}$.

In the case of (first-order) arithmetic masking, each bit of a native value $a^{(i)}$ is initially masked with a random mask $r^{(i)}$ using modular additions. According to Equation 1, the native value $a^{(i)}$ can be expressed as the XOR between the two shares $a^{(i)} \oplus r^{(i)} \oplus c^{(i)}$ and $r^{(i)}$. In contrast to Boolean masking, we also need to include the carry of the addition $c^{(i)}$, which depends on lower bits of $a^{(i)}$ and $r^{(i)}$. The first option to obtain a valid labeling for arithmetic shares is thus to resolve $c^{(i)}$ recursively according to Equation 2. The initial labels would then be given by $b_0^{(i)} = (a^{(i)} \oplus r^{(i)}) \oplus c^{(i)}$, and $b_1^{(i)} = r^{(i)}$. Here, the carry $c^{(i)}$ is computed recursively for each bit position, which adds already quite complex terms to the correlation set at the beginning of the verification, especially for the more significant bits of the arithmetic shares since the depend in a non-linear way on all lower bits.

It is however also possible to use a different initial labeling that incorporates additional information that is available at the beginning of the verification and significantly simplifies the resulting Boolean expressions. More concretely, with each $c^{(i)}$ being a non-linear combination of all lower bits (including their masks), this expression alone must never be observable by an attacker. Put differently, each bit of a fresh arithmetic share is only independent of any native values because the term $r^{(i)}$ is added in a linear way and does not occur in any of the lower bits (and thus also not $c^{(i)}$). It is hence sufficient to verify if the linear term $r^{(i)}$ in a certain bit of one arithmetic share ever gets in contact with the same $r^{(i)}$ in the corresponding bit of the other share, similarly as in the case of Boolean masking (c.f. Figure 1). This simplification leads to simpler expressions for the initial labels and thus improves verification runtime. Note that this simplification is only used for deriving initial labels but not during mask refresh operations throughout the masked computation where our assumptions on unique usage of fresh randomness does not necessarily hold anymore. This simplification also applies to initial labels of higher order arithmetic masking in a similar manner.

Fourier Expansion of Arithmetic Addition One particularly challenging aspect of verifying arithmetic masking is scalability due to complex dependencies between shares on bit-level, introduced by the carry when an arithmetic addition is computed. In hardware, arithmetic additions are often performed by dedicated sub-circuits. For example, CPUs usually have such an adder circuit in their ALU (Arithmetic Logic Unit). We propose the Fourier expansion of arithmetic additions, which allows to directly obtain correlation sets for the result of an adder circuit, instead of computing an individual correlation set for every gate within the adder, and thus speeds up the verification runtime. In Section 5.1 we give more details about how this can be used to increase the performance of software verification, and how it can be integrated into Coco.

In Equation 5 we propose the Fourier expansion W of an addition.

$$W(s^{(j)}) = \frac{1}{2}u^{(j)} \cdot v^{(j)} \cdot c^{(j)} + \frac{1}{2}u^{(j)} + \frac{1}{2}v^{(j)} - \frac{1}{2}W(c^{(j)}) \tag{5}$$

The expansion of the sum is based on the Fourier expansion of the carry given in Equation 6.

$$W(c^{(j)}) = \frac{1}{2}W(c^{(j-1)}) + \frac{1}{2}v^{(j)} + \frac{1}{2}u^{(j)} - \frac{1}{2}u^{(j)} \cdot v^{(j)} \cdot W(c^{(j-1)}) \tag{6}$$

More details on how we derived both expansions are given in Appendix A.

Soundness and Completeness While masking verification based on approximated Fourier coefficients of Boolean functions is sound, it is not complete. Throughout a masked computation it might happen that certain terms in the exact Fourier representation cancel out or evaluate to constants. Our verification approach might miss such situations since it only keeps track of whether a term occurs in a correlation set or not (for performance reasons), which ultimately results in an overapproximation of the exact Fourier representation. If such a situation occurs, i.e. multiple shares with of a native value with a correlation coefficient of zero are combined, the verifier would report a leak that does however not exist in practice (which implies non-completeness). Soundness is however guaranteed by the fact that the verifier always keeps track of an *over*approximation of all the terms that a register/wire could depend on, hence, a real leak can never be missed.

In case of sound but not complete masking verification approaches, the amount of false positive leakage reports in realistic scenarios plays an important for practicality. Simply speaking, the longer a computation becomes, the more likely a false positive occurs. Note however that after every mask refresh operation, the newly introduced randomness essentially eliminates possible future false-positive leaks caused by over-approximation that has happened thus far. In other words, as long as mask refreshing occurs somewhat frequently (which is generally the case) the occurrence of false positive leak reports will generally be quite low. Later, in Section 4 and Section 5, we show that the soundness of our approach is in fact sufficient to perform meaningful verification of masked SW/HW implementations in many typical PQC/ARX applications.

During our analysis in this work, we only really observe a single false positive when verifying Goubin-A2B [Gou01] in software. We discuss this case in more detail in Section 5.2.

3.3 Example

Assume an example circuit which takes two 2-bit arithmetic shares $a + r$ (input signal A_0) and r (input signal A_1), and two bits of fresh randomness s (input signal S). The ultimate goal is to compute $(A_0 + S) + A_1$ by using two *Full Adders*. In order to verify the first-order security of this circuit, one first assigns the respective labels to the inputs which result in the following correlation sets:

$$\begin{aligned}
\mathcal{C}(A_0^{(0)}) &= \{\{b_0^{(0)}\}\}, & \mathcal{C}(A_0^{(1)}) &= \{\{b_0^{(1)}\}\}\\
\mathcal{C}(A_1^{(0)}) &= \{\{b_1^{(0)}\}\}, & \mathcal{C}(A_1^{(1)}) &= \{\{b_1^{(1)}\}\}\\
\mathcal{C}(S^{(0)}) &= \{\{s^{(0)}\}\}, & \mathcal{C}(S^{(1)}) &= \{\{s^{(1)}\}\}
\end{aligned}$$

The input bits are propagated to the first adder, which computes $(A_0 + S)$. We obtain the following correlation sets at the output signals of the first adder:

$$\begin{aligned}
\mathcal{C}(\text{Adder1}_{sum}^{(0)}) &= \mathcal{C}(A_0^{(0)}) \otimes \mathcal{C}(S^{(0)}) = \{\{b_0^{(0)}, s^{(0)}\}\}\\
\mathcal{C}(\text{Adder1}_{sum}^{(1)}) &= \mathcal{C}(A_0^{(1)}) \otimes \mathcal{C}(S^{(1)}) \otimes \mathcal{C}(\text{Adder1}_{carry}^{(1)})\\
&= \{\{b_0^{(1)}, s^{(1)}\}\} \otimes \{\{1\}, \{b_0^{(0)}\}, \{s^{(0)}\}, \{b_0^{(0)}, s^{(0)}\}\}\\
&= \{\{b_0^{(1)}, s^{(1)}\}, \{b_0^{(1)}, s^{(1)}, b_0^{(0)}\}, \{b_0^{(1)}, s^{(1)}, s^{(0)}\}, \{b_0^{(1)}, s^{(1)}, b_0^{(0)}, s^{(0)}\}\}\\
\mathcal{C}(\text{Adder1}_{sum}^{(2)}) &= \mathcal{C}(\text{Adder1}_{carry}^{(2)})
\end{aligned}$$

Note that the second bit of the adder has to be labeled with the carry of the addition. These correlation sets are then propagated to the second adder:

$$\begin{aligned}
\mathcal{C}(\text{Adder2}_{sum}^{(0)}) &= \mathcal{C}(A_1^{(0)}) \otimes \mathcal{C}(\text{Adder1}_{sum}^{(0)}) = \{\{b_1^{(0)}, b_0^{(0)}, s^{(0)}\}\}\\
\mathcal{C}(\text{Adder2}_{sum}^{(1)}) &= \mathcal{C}(A_1^{(0)}) \otimes \mathcal{C}(\text{Adder1}_{sum}^{(1)}) \otimes \mathcal{C}(\text{Adder2}_{carry}^{(1)})\\
&= \{\{b_1^{(1)}, b_0^{(1)}, s^{(1)}\}, \{b_1^{(1)}, b_0^{(1)}, s^{(1)}, b_0^{(0)}\}, \{b_1^{(1)}, b_0^{(1)}, s^{(1)}, s^{(0)}\},\\
&\qquad \{b_1^{(1)}, b_0^{(1)}, s^{(1)}, b_0^{(0)}, s^{(0)}\}\}\\
\mathcal{C}(\text{Adder2}_{sum}^{(2)}) &= \mathcal{C}(\text{Adder1}_{sum}^{(2)}) \otimes \mathcal{C}(\text{Adder2}_{carry}^{(2)})\\
\mathcal{C}(\text{Adder2}_{sum}^{(3)}) &= \mathcal{C}(\text{Adder2}_{carry}^{(2)})
\end{aligned}$$

Obviously, $(A_0 + S) + A_1$ is a valid operation in the context of arithmetic masking and this is also visible on bit-level. The computation of the carry bits of the second adder combines shares in a non-linear way, which typically leads to a leak. However, the addition is still secure in the end since $(A_0 + S)$ adds randomness to each share bit linearly. When performing an addition of two operands we always conservatively label one bit more than the size of the largest operand to correctly capture bit width of the result independently on the concrete input values. Note that by performing modular reduction one can clear the carry residing in the most significant bit (MSB). This type of computation occurs very frequently in the beginning of A2B algorithms when two arithmetic shares should be added since the addition of fresh randomness is equivalent to a mask refreshing operation.

4 Application to Masked Hardware Implementations

In this section we apply our verification approach to hardware implementations of [CGV14]-A2B. While it has already been shown in the past that this algorithm is secure in the stable setting, which is also confirmed by our verifier, we want to put our focus mainly on settings where we also consider transition and glitch effects. We show, both via a formal analysis, and in empirical evaluations, that hardware side-effects can reduce the protection order of the implementation. While the straight-forward approach of adding additional register stages whenever needed can eliminate this problem, we also want to point out that this comes with a noticeable increase of latency. More research on more efficient constructions may be useful.

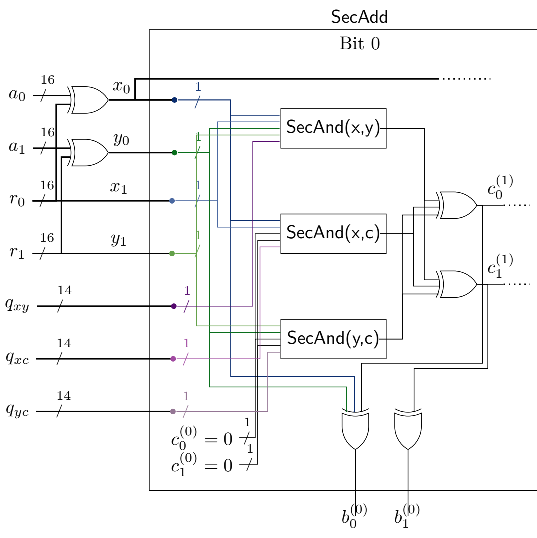

Figure 2: Schematic image of [CGV14]-A2B when implemented in hardware. The arithmetic input shares a_0, a_1 are transformed into Boolean shares b_0, b_1. The carry computation happens in the SecAdd module, from which we draw the first part responsible for bits 0 of the final result.

[CGV14]-A2B/B2A In 2014, Coron et al. have proposed the first higher-order mask conversion algorithm, which we refer to as [CGV14]-A2B/B2A in the following. This algorithm is based on the SecAdd function and allows to perform arithmetic additions in a Ripple-carry fashion on Boolean shares. The carry bit is computed based on Equation 2, which can be rewritten as:

$$c^{(i+1)} = u^{(i)} \wedge v^{(i)} \oplus u^{(i)} \wedge c^{(i)} \oplus v^{(i)} \wedge c^{(i)} \text{ with } c^{(0)} = 0 \tag{7}$$

Figure 2 shows [CGV14]-A2B conversion for the 16-bit native value a, split into the 16-bit shares a_0 and a_1, which starts with refreshing the arithmetic shares a_0 into x_0, and a_1 into x_1, and then computes the shared carries c_0 and c_1 bit by bit using secure masked AND gadgets (SecAnd). During the conversion, a is expressed as $a = b_0 \oplus b_1$ with $b_0 = x_0 \oplus x_1 \oplus c_0$ and $b_1 = y_0 \oplus y_1 \oplus c_1$. The resulting Boolean shares are b_0 and b_1.

The corresponding B2A conversion chooses the first arithmetic share a_0 randomly, and computes $a_1 = (b_0 \oplus b_1) - a_0$ using SecAdd. The algorithm is very efficient for hardware implementations [FBR+21], since both A2B and B2A are based on SecAdd. Both conversion algorithms can also be applied to higher orders. In Section 5 we formally evaluate both [CGV14]-B2A, and a second-order masked software implementation of [CGV14]-A2B.

4.1 Formal Analysis

We implement [CGV14]-A2B with 16-bit shares in hardware. We store all inputs in registers, and implement the remaining parts as a pure combinatorial circuit, which takes a single cycle to finish and therefore does not require a state machine. Figure 2 corresponds to the resulting hardware module. The input shares as well as the 16-bit random values r_0, r_1 and three 14-bit random values q_{xy}, q_{xc} and q_{yc} are stored in registers. The verifier confirms algorithmic security for this single-cycle implementation, while in the transient case under the consideration of glitches, first-order side-channel protection is not given. More concretely, glitches in the initial remasking phase and the SecAnd modules, which are part of the bigger SecAdd, may lead to a temporary combination of shares due to delayed addition of randomness.

Table 1: Verification of [CGV14]-A2B (broken and fixed) in Hardware

Algorithm	Input shares	Runtime (cycles)	Verification result/runtime					
			Stable		Transitions		Transient	
[CGV14]	16 bit	1	✓	11 s	✓	10 s	✗	1 s
[CGV14]	16 bit	34	✓	56 s	✓	2 min	✓	3 min

Initial Remasking Figure 2 shows that in the first step, the arithmetic shares a_0 and a_1 are refreshed with the fresh random values r_0 and r_1, resulting in x_0, x_1, y_0 and y_1, which are further passed to the SecAdd module. Two XOR gates at the circuit's inputs are used to perform the remasking. In the worst case, a glitch at the output of the XOR gate propagates the pure values of a_0 and a_1. The input of SecAdd will then be the arithmetic shares ($x_0 = a_0$, $y_0 = a_1$, $x_1 = y_1 = 0$), and the circuit computes $a_0 + a_1 = a$ for a short time frame in the beginning of the clock cycle, until all wires stabilize and the randomness *arrives* at the gates. As a solution, we add a single additional register stage to store the result of these XOR computations. This ensures that the SecAdd module's input comes out of a register instead of combinatorial logic, and will therefore not glitch.

SecAnd [CGV14] suggest to use the masked AND gate proposed by Ishai et al. [ISW03], called ISW-AND, and provide a row of formal security arguments. We integrate ISW-ANDs as SecAnd-blocks in our hardware implementation. Formal verification however reports a leak due to glitches in the SecAnd module because the ISW AND gate is not glitch-resistant, and also does not fulfill the required composability properties. These findings have been confirmed by several prior works [MPG05,FGP+18,CPRR13]. As a solution, we suggest to insert two register stages to the SecAnd component. Works like [MMSS19,FGP+18] confirm our observation that these two register stages are indeed needed in this case. Combined with the register stage inserted for the initial remasking, this results in a high latency overhead, i.e., for n-bit input shares, the implementation now requires $34 = 2 + 2 \times n$ cycles to complete, and also utilizes a state machine in order to control the execution.

We evaluate our verification approach for masked hardware circuits in Table 1 by comparing the broken single-cycle implementation to the one which adapts our fixes. All experiments are run using a 64-bit Linux Operating System on an Intel Core i7-7600U CPU with a clock frequency of 2.70 GHz and 16 GB of RAM. The security on algorithmic level of both implementations can be shown in 11 seconds and respectively 56 seconds in the stable case. We need around a second to find the issues in the transient case, and about three minutes to prove that our fixes indeed provide first-order protection.

Our implementation serves the purpose of a proof-of-concept. In general, there exist several ways to further optimize [CGV14]-A2B in hardware. For example, one possibility would be to replace SecAnd with another masked AND gadget, which has less latency. However, we consider the discussion of these optimizations along with the evaluation of area and performance overhead out of scope for this paper.

4.2 Empirical analysis

In the last section we discuss the outcome of the formal analysis which indicates that glitches in the design are problematic in the context of masking. As a second step, we show practical evidence for the proposed statements.

Evaluation Setup We practically evaluate [CGV14]-A2B using a first-order t-test on the NewAE CW305 Artix-7 FPGA evaluation board connected to a PicoScope 6404C at 312.5 Ms/s sampling rate. The hardware design operates at a clock frequency of 1 MHz. In order to show whether or not the implementation exhibits first-order leakage, we perform Welch's t-test following the guidelines of Goodwill et al. [GJJR11]. Welch's t-test is a standard method to measure information leakage of masked implementations. The basic idea is to create two sets of measurements, one representing the power consumption of the

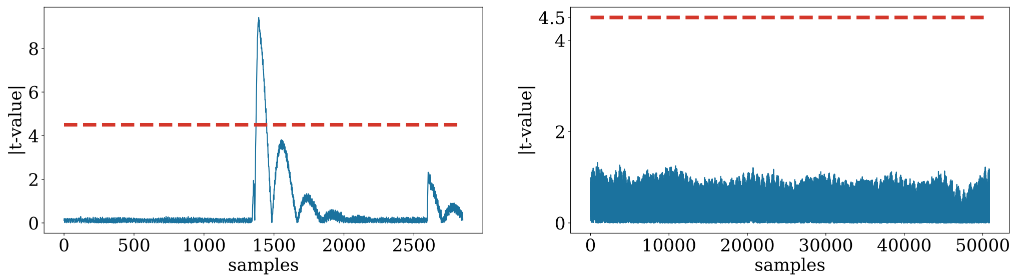

Figure 3: T-test scores of the original (left) and the secured (right) implementation of [CGV14]-A2B using 400 000 power traces

design with random inputs, and one with constant inputs. In our case, we set the native value a to 0 and choose a_0 randomly and a_1 such that $a_0 + a_1 = a$ for the fixed set. For the random set, we generate a_0 and a_1 randomly. We use fresh random values for r_0, r_1 and all q in both cases. From these trace sets, one can compute Welch's t-score to measure the significance of the difference of means of the two distributions. The null-hypothesis is that both trace sets have equal means. It is rejected with a confidence greater than 99.999% if the absolute t-score does not exceed 4.5. In that case, the trace sets cannot be distinguished from each other.

Discussion Figure 3 shows our leakage assessment using 400 000 traces. The results for the original, unprotected single-cycle implementation are presented on the left. The t-test score shows significant peaks over the 4.5 border, indicating first-order leakage. On the right side, the leakage evaluation of our 34-cycle fixed implementation is shown, in which the t-score does not cross the significance boarder. Thus, these measurements confirm the security claim made by the formal tool. In Appendix B we show the functionality of the measurement setup by turning the random number generator off.

Note that Coco verifies ASIC netlists of masked implementations and identifies wires where dangerous glitches might occur. The exact structure of this netlist must be reflected on the final FPGA layout to make concrete security statements, which is why we cannot simply synthesize the hardware design to the FPGA. The synthesis process will possibly merge multiple ASIC gates into a single lookup table (LUT) on the FPGA, and the original netlist structure will not be preserved. Consequently, one might see artifacts in the measurements stemming from this merging process, e.g. because the strict separation of shares is lost in the translation process [CBG+17]. Therefore, we must ensure to map each gate in the verified ASIC netlist to a functionally equivalent FPGA LUT, in order to preserve the original netlist structure as good as possible.We achieve this by mapping each ASIC gate to a LUT with 2 inputs and one output, by putting a `dont_touch = "true"` on every gate/wire in the netlist. Additionally, we ensure that the final FPGA layout facilitates the measurement of transient effects such as glitches.

5 Application to Masked Software Implementations

In this section we discuss how Coco can be used to identify leaks in arithmetically masked RISC-V assembly implementations. In the beginning, we discuss the software verification setup. Then, we focus on algebraic conversions, including [CGV14], [SPOG19] for prime moduli and Goubin-A2B/B2A [Gou01], for which we point out several register overwrite leaks. We discuss the table-based conversion algorithms of Debraize [Deb12] and Beirendonck et al. [BDV21], and explain how table lookups can be formally verified from a probing-security perspective. For each algebraic/table-based algorithm, we give a brief introduction, discuss our findings and - in case a leak is reported - give a pseudocode description for better understanding. To conclude the section, we verify the masked ARX-based schemes Speck 32/64 and Alzette.

5.1 Software Verification Setup

When analyzing masked software implementations, potential issues are either caused by flaws in the algorithmic design, or due to microarchitectural side-effects of the processor's hardware. Flaws in the algorithmic design are mainly attributed to non-uniform sharings of intermediate variables, accidental combinations of masks, or transition leakage caused by variable overwrites. However, even if such issues are taken into account there is still no guarantee that such an algorithm, once implemented for a specific processor, will be free of leaks. For example, a recent work by Gigerl et al. [GHP+21] has analyzed the RISC-V Ibex core in terms of architecture side-effects for masked software, and has pointed out multiple additional potential sources of leakage due to the design of the register file, the SRAM, the ALUs, and the load-store unit. As one of their results, they created a *secured* Ibex[3] that incorporates some relatively cheap hardware fixes that mostly eliminate glitch-related issues that are otherwise difficult to deal with purely on software-level.

For the purpose of this paper we are not so much interested into further netlist modifications, but rather focus on potential flaws in the algorithmic design of masked software implementations. We want to use their *secured* Ibex core as a reference platform that comes with a concrete list of hardware side-effects that do or do not need to be taken into consideration in software, thus allowing for an even playing field when evaluating and comparing different masked software implementations of A2B/B2A conversion algorithms. More specifically, the certain common microarchitectural leakages do not need to be addressed in software because the *secured* Ibex already has appropriate fixes on netlist-level. These fixes include:

- A glitch-resistant register file which allows to read and write shares without combination, as long as the respective software constraints are met
- No hidden registers or always-active computation units
- A glitch-resistant model of the SRAM (similar to the register file)

For more details on these fixes, we refer to the work of Gigerl et al. [GHP+21]. When a masked assembly implementation is executed by the *secured* Ibex and the software constraints are met, the leakages which are left are primarily register/memory overwrites and leaks caused by algorithmic flaws. The results of the following analysis can therefore be ported to any other microprocessor, as long as the respective device-specific fixes against these leaks, either in hardware or in software, are implemented.

The synthesis process will transform the adder, which lies in the ALU of the *secured* Ibex, to a set of logic gates. Theoretically, each gate is each assigned a correlation set during verification, which is very time-consuming. We wrap up the addition into a custom `adder` "gate" instead of splitting it up, which means only the output wires of the adder must be assigned correlation sets. In order to achieve this, we identify the addition in the CPU design before synthesis (which is trivial), move it into a distinct module, and apply keep hierarchy on this module, which results in a single `adder` gate on netlist level. In case of the *secured* Ibex core, the `adder` gate is represented by 2×32-bit inputs, and creates a 33-bit output, for which we can compute the correlation set quite efficiently using Equation 5 and Equation 6. Without this optimization, the synthesizer would split the adder up into individual logic gates and one would check the correlation of each of these gates individually. Consequently, especially verification in transient mode would then not

[3] https://github.com/IAIK/coco-ibex

Table 2: Verification of masked software implementations. The verification result is either ✓(no issues were found), (✓) (no issues were found except for potentially insecure table lookups), ✗(algorithmically insecure implementations) or ✗(false positive).

Algorithm	Input shares	Runtime (cycles)	Verification result/runtime					
			Stable		Transitions		Transient	
A2Bs								
[CGV14]	4 bit	225	✓	41 s	✓	67 s	✓	3 min
[CGV14]	16 bit	984	✓	4 min	✓	5 min	✓	16 min
[CGV14] (2nd order)	4 bit	1240	✓	3.8 min	✓	6 min	✓	20 min
Debraize [Deb12] ⊞	4 bit ($n = 2$, $k = 2$)	140	✗	35 s	-		-	
Debraize [Deb12] ⊞	16 bit ($n = 4$, $k = 4$)	450	✗	118 s	-		-	
[BDV21]-fixed-Debraize ⊞	4 bit ($n = 2$, $k = 2$)	180	(✓)	38 s	(✓)	48 s	-	
[BDV21]-Dual-Lookup ⊞	4 bit ($n = 2$, $k = 2$)	105	(✓)	28 s	(✓)	30 s	-	
Goubin [Gou01]	16 bit	170	✗	37 s	✗		-	
B2As								
Goubin [Gou01]	16 bit	23	✓	5 s	✓	8 s	✓	19 s
[CGV14]	4 bit	650	✓	6 min	✓	4 min	✓	11 min
[CGV14]	16 bit	2475	✓	11 min	✓	16 min	✓	38 min
[SPOG19], without final reduction	4 bit, ($q = 257$, $\log_2 q = 9$)	400	✓	2 min	✓	21 min	-	
ARX-based schemes								
SPECK 32/64 (Goubin-B2A, [CGV14]-A2B) (1 round)	6 × 16 bit	1465	✓	6 min	✓	13 min	✓	5.13 h
Alzette (Goubin-B2A, [CGV14]-A2B) (1 round)	2 × 32 bit	3082	✓	29 min	✓	2.48 h	✓	27 h

be possible in a feasible time frame [4]. It is important to note that this does not affect the soundness guarantees of our approach because the correlation sets computed for the outputs of the `adder` gates are identical to the correlation sets of an adder which is split up.

5.2 Verification of Algebraic Share Conversions

[CGV14]-A2B/B2A In Section 4 we discuss the verification of [CGV14] conversion algorithms in hardware. When verified on a CPU netlist, the algorithm in general behaves very similar. As shown in Table 1, we implement [CGV14]-A2B and -B2A in software and verify it successfully. We provide 16-bit A2B and B2A implementations which we verify in all three verification modes. Additionally, we implement 4-bit first- and second-order implementations, which can also successfully be verified with our approach. Compared to the results of Section 4, where we verify a 34-cycle implementation in 3 min, we can verify the respective software implementation (~1000 cycles) in 20 min, which shows the efficiency of our tool. Interestingly, both QMVERIF and `LeakageVerif` have to fall back to exhaustive enumeration when verifying [CGV14]-B2A, while the other direction (A2B) is possible [MPH21].

[SPOG19]-B2A Various existing A2Bs/B2As work with power-of-two moduli exclusively, while many lattice-based constructions require a prime modulus. To address this issue, one can first transform the shares computed in $\mathbb{F}_q$ to $\mathbb{F}_{2^k}$, and then apply conversion algorithms working with power-of-two moduli, as shown by Oder et al. [OSPG18]. Another possibility is to alter a $\mathbb{F}_{2^k}$-conversion algorithm to work in $\mathbb{F}_q$. For example, [SPOG19]-B2A is an adaption of [CGV14]-B2A, such that it can handle arbitrary moduli. The authors' goal was to build a masked binomial sampler, which is why they did not propose an A2B. The

[4]Runtime of a few hours for a single 32-bit addition

Algorithm 1 [SPOG19] B2A (simplified for 1st order)

```
Input: k-bit shares $b_0, b_1$ such that $b = b_0 \oplus b_1$
Output: Shares $a_0, a_1 \in \mathbb{F}_q$ such that $b = a_0 + a_1 \mod q$
 1: $b'_0 \leftarrow b_0^{(k-1)}$
 2: $b'_1 \leftarrow b_1^{(k-1)}$
 3: $a_0, a_1 \leftarrow$ B2A_Bit$(b'_0, b'_1)$
 4: $R \overset{\$}{\leftarrow} \mathcal{R}_q$
 5: $a_0 \leftarrow (a_0 + R) \mod q$
 6: $a_1 \leftarrow (a_1 - R) \mod q$
 7: for $j = 2$ to $k-1$ do
 8:     $b'_0 \leftarrow b_0^{(k-j)}$
 9:     $b'_1 \leftarrow b_1^{(k-j)}$
10:     $C_0, C_1 \leftarrow$ B2A_Bit$(b'_0, b'_1)$
11:     $R \overset{\$}{\leftarrow} \mathcal{R}_q$
12:     $C_0 \leftarrow (C_0 + R) \mod q$
13:     $a_0 \leftarrow ((a_0 << 1) + C_0) \mod q$
14:     $C_1 \leftarrow (C_1 - R) \mod q$
15:     $a_1 \leftarrow ((a_1 << 1) + C_1) \mod q$
16: end for
17: return $a_0, a_1$
```

Algorithm 2 [SPOG19] B2A_Bit (simplified for 1st order)

```
Input: 1-bit shares $b'_0, b'_1$ such that $b = b'_0 \oplus b'_1$
Output: $E_0, E_1$ such that $E_0 + E_1 = b \mod q$
 1: $E_0 \overset{\$}{\leftarrow} \mathcal{R}_q$
 2: $E_1 \leftarrow b'_1 - E_0 \mod q$
 3: $E_1 \leftarrow E_1 - 2 \cdot (E_1 \cdot b'_0) \mod q$
 4: $E_0 \leftarrow E_0 - 2 \cdot (E_0 \cdot b'_1) \mod q$
 5: $E_1 \leftarrow E_1 + b'_1 \mod q$
 6: return$E_1, E_0$
```

transformation process handles the Boolean shares bit by bit, and was proven secure against side-channel adversaries at arbitrary orders. We construct a first-order implementation of [SPOG19] B2A with a modulus $q = 257$ and 4-bit input shares according to Algorithm 1 and Algorithm 2.

When verifying such an implementation on bit-granularity, reductions mod q are a major obstacle since they create very complex dependencies between the individual bits of a share that would drastically increase verification runtime. Fortunately however, many practical implementations already use efficient reduction methods like Montgomery [Mon85] or Barret [Bar86] reductions in combination with lazy reduction, i.e., skipping reductions as long as intermediate values are guaranteed to fit inside 32-bit words (on 32-bit architectures) [BKS19]. These tricks not only improve runtime of the algorithms but also help to reduce the verification runtime drastically. We consider an implementation of [SPOG19] B2A that delays the Barret reduction until the end of the conversion. We verify its security in the stable and transition case (c.f. Table 2). In this setting, we want to point out and interesting pitfall that should be avoided when using lazy reduction techniques in the context of masking. For example, Algorithm 2 first generates a random number $E_0 \in \mathbb{F}_q$, and then computes E_1 based on both shares b'_0, b'_1, using E_0 as fresh randomness. If one now lazily skips the reductions in Lines 2 and 3, the upper bits of E_1 will not be masked by E_0 anymore due to the smaller bit width. Hence, to mitigate this potential pitfall on 32-bit architectures, one could simply always use 32-bit words of randomness whenever mask refreshing is required.

Goubin-A2B/B2A Goubin-A2B algorithm [Gou01], given in Algorithm 3, converts the arithmetic shares $a = a_0 + a_1$ into Boolean shares b_0, b_1 with the help of a recursion formula. The second Boolean share b_1 is set to the second arithmetic share a_1. The first Boolean share b_0 is computed by choosing $b_0 = a \oplus b_1 = (a_0 + a_1) \oplus b_1$ and rewriting the term recursively: $b_0 = a_0 \oplus u_{n-1}$ with $u_0 = 0, u_{i+1} = 2(u_i \wedge (a_0 \oplus a_1) \oplus (a_0 \wedge a_1))$. In a similar spirit, [Gou01] B2A algorithm transforms the Boolean shares $b = b_0 \oplus b_1$ into arithmetic shares $b = a_0 + b_1 \mod 2^k$ based on the property $a_0 = (b \oplus b_1) - b_1$. If we verify these implementations, we can already successfully verify the security of the B2A conversion in the stable, transition, and transient case. However, we do encounter several issues with the A2B conversion that we now describe in more detail. To the best of our knowledge, these findings have not been reported so far.

First, Goubin-A2B has several problems regarding insecure register transitions. For example, even if we ensure that our assembly implementation uses dedicated registers for each of the variables in Algorithm 3, the intermediate Y, after being updated in Line 9

Algorithm 3 Goubin-A2B [Gou01]

```
Input: $n$-bit shares $a_0, a_1$ such that $a = a_0 + a_1 \bmod 2^n$
Output: $n$-bit shares $b_0, b_1$ such that $a = b_0 \oplus b_1$
 1: $Y \leftarrow \mathcal{U}(0,1)^n$
 2: $T \leftarrow 2Y$
 3: $b_0 \leftarrow Y \oplus a_1$
 4: $\Omega \leftarrow Y \wedge b_0$
 5: $b_0 \leftarrow T \oplus a_0$
 6: $Y \leftarrow Y \oplus b_0$
 7: $Y \leftarrow Y \wedge a_1$
 8: $\Omega \leftarrow \Omega \oplus Y$
 9: $Y \leftarrow T \wedge a_0$
10: $\Omega \leftarrow \Omega \oplus Y$
11: for $i \leftarrow 0$ to $n-1$ do
12:     $Y \leftarrow T \wedge a_1$
13:     $Y \leftarrow Y \oplus \Omega$
14:     $T \leftarrow T \wedge a_0$
15:     $Y \leftarrow Y \oplus T$
16:     $T \leftarrow 2Y$
17: end for
18: $b_0 \leftarrow b_0 \oplus T$
19: $b_1 \leftarrow a_1$
20: return $b_0, b_1$
```

will leak the XOR between the old (Y_{old}) and the new (Y_{new}) value. More concretely, the attacker observes:

$$\begin{aligned} Y_{\text{old}} &= Y_{\text{line }6} \wedge a_1 \\ &= (Y_{\text{line }1} \oplus b_{0\text{line }5}) \wedge a_1 \\ &= (Y_{\text{line }1} \oplus (T \oplus a_0)) \wedge a_1 \\ Y_{\text{new}} &= T \wedge a_0 \\ Y_{\text{old}} \oplus Y_{\text{new}} &= ((Y_{\text{line }1} \oplus (T \oplus a_0)) \wedge a_1) \oplus (T \wedge a_0) \\ &= (a_0 \wedge a_1) \oplus (a_0 \wedge T) \oplus (a_1 \wedge Y) \end{aligned}$$

Hence, for every bit $>= 0$, this expression will correlate with native value a. Another similar situation occurs in Line 12 where $Y_{\text{old}} = T \wedge a_0$ is overwritten by $Y_{\text{new}} = T \wedge a_1$ in the first loop iteration.

Second, the verifier indicates that Goubin-A2B might not be executed securely due to the computation of $\Omega \oplus Y$ in Line 10. Taking a closer look at this concrete problem, we determine that in this case we experience a false positive as already mentioned in Section 3.2. In Line 10, when the processor computes $\Omega \oplus Y$, an attacker might probe the least significant bit of the expression, which is

$$(Y^{(0)} \wedge (Y^{(0)} \oplus a_1^{(0)})) \oplus (a_1^{(0)} \wedge (Y^{(0)} \oplus a_0^{(0)}))$$

The exact Fourier expansion of this expression does not contain a single term which depends on both $a_0^{(0)}$ and $a_1^{(0)}$ alone, but only in connection with $Y^{(0)}$, and is therefore properly masked. However, it does look like a leak to the verifier because the approximated correlation set contains a set $\{a_0^{(0)}, a_1^{(0)}\}$ where the mask $Y^{(0)}$ is not contained, which represents a leak. We give the exact calculation in Appendix D.

According to [MPH21], both QMVerif and `LeakageVerif` also fail to verify Goubin-A2B correctly because their tools produce false positives. Unfortunately, they do not discuss the exact issue, and therefore we were not able to make further investigations.

5.3 Verification of Table-based Share Conversions

Besides algebraic approaches, several A2Bs utilize table lookups, such as the ones from Debraize [Deb12] and Beirendonck et al. [BDV21]. These algorithms start with computing one or multiple lookup-tables which they then use for the actual conversion. A table lookup represents a data-dependent memory access, i.e., an operation that loads data from

Algorithm 4 Table T generation [Deb12]

```
Input: $k$
Output: Conversion table $T$, random variables $r, \rho$
1: $r \leftarrow \mathcal{U}(0,1)^k$
2: $\rho \leftarrow \mathcal{U}(0,1)$
3: for $i \leftarrow 0$ to $2^k - 1$ do
4:     $T[\rho||i] \leftarrow (i + r) \oplus (\rho||r)$
5:     $T[(\rho \oplus 1)||i] \leftarrow (i + r + 1) \oplus (\rho||r)$
6: end for
7: return $T, r, \rho$
```

Algorithm 5 Debraize-A2B [Deb12]

```
Input: $(n \cdot k)$-bit shares $a_0, a_1$ such that $a = a_0 + a_1 \mod 2^{(n \cdot k)}, T, r, \rho$
Output: $(n \cdot k)$-bit shares $b_0, b_1$ such that $a = b_0 \oplus b_1$
1: $a_0 \leftarrow a_0 - (r||...||r||...||r) \mod 2^{n \cdot k}$
2: $\beta \leftarrow \rho$
3: for $i \leftarrow 0$ to $n - 1$ do
4:     Split $a_0$ into $(a_{0h}||a_{0l})$, split $a_1$ into $(a_{1h}||a_{1l})$
5:     $a_0 \leftarrow a_0 + a_{1l} \mod 2^{(n-i) \cdot k}$
6:     $\beta||x'_i \leftarrow T[\beta||a_{0l}]$
7:     $x'_i \leftarrow x'_i \oplus a_{1l}$
8:     $a_0 \leftarrow a_{0h}, a_1 \leftarrow a_{1h}$
9: end for
10: $b_0 = (x'_0||...||x'_i||...||x'_{n-1}) \oplus (r||...||r||...||r)$
11: $b_1 = a_1$
12: return $b_0, b_1$
```

memory address that is data-dependent. Coco was mostly intended to verify symmetric cryptography, where table lookups are not common and have therefore not been considered previously. However, our study shows that the verification approach can be successfully applied under specific conditions, which we will discuss in the following. All table-based A2Bs that we are aware of fulfill these conditions, and can therefore be successfully verified.

First, it must be possible to compute all entries in the table with a single unique function $f(i)$, with i being the table index. For example, Debraize A2B initially fixes the random values r and p, which will stay the same for the table generation and for all future A2B conversions. The table then is generated based on the function $f(i) = i + r + p \oplus (p||r)$, which is unique for Debraize-A2B since r and p are already generated before. This ensures that every table entry is assigned the same label during the verification independently of the address such that effectively it is not relevant which memory location is considered.

Second, the evaluation platform must guarantee constant-time memory accesses, i.e., memory accesses always require the same amount of cycles independently of the memory address. For example, the Ibex core contains a state machine in the LSU to handle misaligned memory accesses. If such a memory access occurs, multiple memory locations must be fetched, and therefore the further program execution is paused. Hence, the later execution of the CPU depends on the memory address, which does not fulfill the constant control-flow requirement any more. Therefore, we simply disable the *secured* Ibex core's ability to perform misaligned memory accesses. We argue that, for the purpose of verifying masked software implementations, this modification is quite reasonable since constant-time implementations are anyway a desired property of cryptographic implementations.

The verified A2B algorithms usually pre-generate a table, and later perform loads to this table with an address which is labeled as a share. However, our approach can also potentially be used to verify stores to memory at a labeled address. The result of such a load instruction however correlates with both the address, and the data which was written to memory. Likewise, when performing a store to a memory location using an address which corresponds to a share of a native value, we expect that the memory location correlates with the address and the data. Note that this behavior was also observed by Bos et al. [BGR+21].

5.4 Application to Table-based Conversion Algorithms

Table-based A2Bs use an arithmetic sharing $a = a_0 + a_1$ which should be converted into a Boolean sharing of the form $a = b_0 \oplus a_1$. One Boolean share is immediately taken over from the arithmetic sharing (a_1), while the main challenge of the conversion is to derive b_0. From a mathematical perspective, b_0 can be obtained by computing $b_0 = a \oplus a_1 = (a_0 + a_1) \oplus a_1$. From a masking perspective, $(a_0 + a_1)$ will however immediately leak the native value a, which is prevented by using a pre-computed look-up table T. The look-up table is used to store $T[a_0] = (a_0 + r) \oplus r$ for a fixed r [BDV21]. Generating $T[a_0]$ for each possible value of a_0 is however often not efficient, which is why many A2B algorithms split up the arithmetic shares into smaller chunks and generate T for each of these chunks.

Debraize-A2B In 2012, Debraize [Deb12] propose an improved table-based A2B based on the works of Coron-Tchulkine [CT03] and Neiße-Pulkus [NP04]. Algorithm 4 shows the generation of the lookup-table, while the actual conversion is shown in Algorithm 5. The ultimate goal is to compute $b_0 = (a_0 + a_1) \oplus a_1$. Debraize-A2B splits up the input shares into n parts of k bits each, and then iterates over these parts. Debraize-A2B refers to these parts as a_{0l} and a_{1l}, which consist of k bits each and are updated in every iteration. Every transformation yields a carry bit c_i, that has to be considered in the next iteration respectively. The precomputed table T is used to look up the value of $(a_{0l} + a_{1l}) \oplus a_{1l}$. We implement Debraize-A2B with $n = 2, k = 2$ as well as $n = 4, k = 4$ and verify its execution as shown in Table 2. Two leaks are already reported in the stable verification mode (indicated by ✖), which points towards algorithmic errors.

First, a leak occurs when performing the table lookup in Line 6, due to a combination of the address bits and the memory content. Figure 4 sketches a simplified table lookup with 4-bit addresses and 8-bit data on gate level. Performing such an access means comparing the memory address (indicated by $addr$) to the memory address of every memory cell (indicated by $Data$), and reading the respective data value in case of a match. The address bits are first compared to the address of the data, which is 4 ($(0110)_b$) in this case. The comparison is realized by four XNOR and one AND gate, leading to the signal eq, which is 1 in case the address matches the data address, or 0 otherwise. eq is then fed into the eight AND gates which determine whether the respective data is read or not. When performing the table lookup in Debraize-A2B in the first iteration, the address bits of depend on ρ, and $(a_{0l} - r + a_{1_l})$. Furthermore, eq depends on $\beta \vee \bigvee_i (a_{0l} - r + a_{1_l})^{(i)}$ and the content of the lookup-table is determined by $(i + r) \oplus (\rho || r)$. i is a constant in this case. The AND gates at the lookup-table read output combine eq with a single bit from the lookup-table, which cancels out the random values r and ρ and allows to probe a value depending on the native value.

One can argue that an SRAM module is constructed in a way such that eq and the memory cell content will never be combined. However, in bigger CPUs, the memory access logic is much more complicated and might contain buffers or caches, which employ such an addressing mechanism. For example, data caches usually require the computation of a tag based on the address, and compare this tag to the one in the cache, which - on netlist level - roughly corresponds to what is illustrated in Figure 4.

Second, the value obtained from the lookup-table in the first iteration is not uniformly distributed, but used as a mask in the algorithm. For more details we refer to the work of Beirendonck et al. [BDV21], who already report and discuss the leak in detail. They discover the issue by empirical measurements, and provide a theoretical analysis afterwards. We want to emphasize that another advantage of our verification approach is the fast discovery of such bugs, which happens in 35 s and 118 s according to Table 2 in this case, which is much quicker than empirical/theoretical evaluations. Due to the formal approach, we immediately see which instruction causes the leak, since Coco reports the leaking cycle and netlist gate, and therefore one does not need to carry out a laborious empirical analysis.

[BDV21]-A2Bs In their work, Beirendonck et al. propose two new secure table-based A2Bs. [BDV21]-fixed-Debraize A2B represents a secured version of Debraize-A2B, which replaces the non-uniform mask with a fresh mask in every loop iteration. [BDV21]-Dual-Lookup A2B works with two precomputed tables, which are however smaller than the lookup-table proposed by Debraize, and therefore more efficient.

We verify both algorithms by choosing parameters $n = 2, k = 2$. As shown in Table 2,

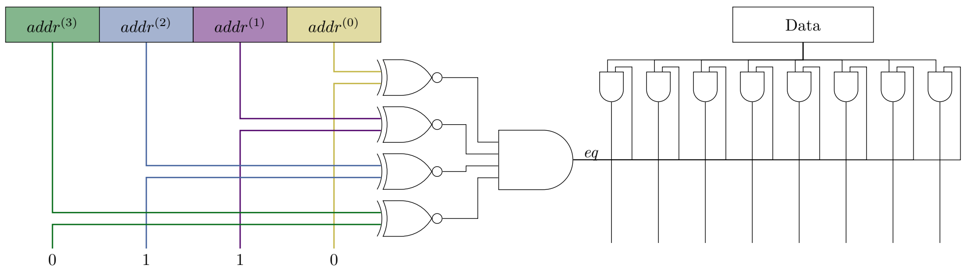

Figure 4: Table lookup on netlist level using 4-bit addresses and 8-bit data words. The address *addr* is compared to the constant address of the SRAM cell ($(0110)_b$). If both values are equal, the resulting 1-bit signal *eq* is 1, and 0 otherwise. *eq* is further used to decide whether the respective data word should be read or not.

table lookups cause a similar leak as we already discussed for Debraize-A2B. Since the issue however strongly depends on the underlying microarchitecture, and no further issues were found, we mark it with (✔) in the table.

5.5 Application to ARX-based Constructions

The ARX (Addition-Rotation-XOR) design principle has been used for several well-known symmetric cryptographic constructions like the block cipher Speck [BSS+13], the stream cipher ChaCha [Ber08], or the hash function SHA-256 [Nat02]. Masking these implementations requires both Boolean masking (for the Rotation and XOR) and arithmetic masking (for the addition), which adds a lot of overhead. There are basically two options to deal with this. First, one can apply an algorithm like SecAdd, which implements modular addition directly on Boolean shares [DGC17, CGV14, SMG15, KRJ14]. Another possibility is to first convert the Boolean shares to arithmetic shares, then perform the addition on arithmetic shares, and convert the shares back to the Boolean domain.

We focus on first-order implementations of Speck 32/64 [BSS+13], and the 64-bit ARX-based S-box Alzette [BBdS+20]. Alzette is a central building block of Sparkle, a lightweight cryptographic permutation which is currently one of the finalists of the NIST LWC Standardization Process [TMC+21]. Furthermore, the Schwaemm AEAD cipher and the Esch family of hash functions are based on Sparkle, and therefore use Alzette. We verify one round of Speck 32/64, and one round of the Alzette S-box. In both cases, we decide to stick to Boolean masking in general and perform conversions where necessary. We switch to arithmetic masking using Goubin-B2A before each addition, perform the addition on arithmetic shares, and switch back to the Boolean domain using [CGV14]-A2B. As shown in Table 2, we verify both algorithms with our approach. We are able to verify algorithmic security in under 30 minutes for both schemes (stable mode). For the transient mode, the verification requires several hours, which is mostly spent by solving the SAT equation, and therefore offers several possibilities for further optimization.

6 Conclusion

In this paper, we presented an approach for the formal verification of masked software and hardware implementations, which supports both arithmetic and Boolean masking schemes of any order. On the hardware side, we show that glitches may cause issues in the context of masking for a straightforward implementation of [CGV14]-A2B. We demonstrate that this issue exists in practice using empirical measurements. On the software side, we first analyze algebraic share conversions, report a previously unknown register transition issue in Goubin-A2B and provide new insights on the security of lazy reduction, a popular optimization technique in PQC. Second, we discuss table-based conversions and demonstrate that table lookups might not be secure due to architectural side-effects. Last but not least, we underline the scalability of our approach by applying it to entire round functions of masked ARX-based ciphers.

Acknowledgements

This work was supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments", and by the European Research Council (ERC) through the European Union's (EU) Horizon 2020 Research and Innovation Program under Grant 681402.

References

[AFM17] Alexandre Adomnicai, Jacques J. A. Fournier, and Laurent Masson. Bricklayer attack: A side-channel analysis on the chacha quarter round. In Arpita Patra and Nigel P. Smart, editors, *Progress in Cryptology - INDOCRYPT 2017 - 18th International Conference on Cryptology in India, Chennai, India, December 10-13, 2017, Proceedings*, volume 10698 of *Lecture Notes in Computer Science*, pages 65–84. Springer, 2017.

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[BBC$^+$19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[BBD$^+$15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.

[BBdS$^+$20] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. Alzette: A 64-bit arx-box - (feat. CRAX and TRAX). In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 419–448. Springer, 2020.

[BBE$^+$18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent

Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 354–384. Springer, 2018.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. Improved high-order conversion from boolean to arithmetic masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):22–45, 2018.

[BDV21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. Analysis and comparison of table-based arithmetic to boolean masking. *IACR Cryptol. ePrint Arch.*, 2021:67, 2021.

[Ber08] Daniel J Bernstein. ChaCha, a variant of Salsa20. In *Workshop record of SASC*, volume 8, pages 3–5, 2008.

[BGG$^+$21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):189–228, 2021.

[BGI$^+$18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

[BGR$^+$21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First- and higher-order implementations. *IACR Cryptol. ePrint Arch.*, 2021:483, 2021.

[BGSD10] Shivam Bhasin, Sylvain Guilley, Laurent Sauvage, and Jean-Luc Danger. Unrolling cryptographic circuits: A simple countermeasure against side-channel attacks. In Josef Pieprzyk, editor, *Topics in Cryptology - CT-RSA 2010, The Cryptographers' Track at the RSA Conference 2010, San Francisco, CA, USA, March 1-5, 2010. Proceedings*, volume 5985 of *Lecture Notes in Computer Science*, pages 195–207. Springer, 2010.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of kyber on cortex-m4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje-eddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2019.

[BSS$^+$13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptol. ePrint Arch.*, page 404, 2013.

[CBG$^+$17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. Does coupling affect the security of masked implementations? In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017,* *Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2017.

[CEvMS15] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt. Masking large keys in hardware: A masked implementation of mceliece. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*, pages 293–309. Springer, 2015.

[CGMZ22] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order table-based conversion algorithms and masking lattice-based encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):1–40, 2022.

[CGP$^+$12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. Conversion of security proofs from one leakage model to another: A new issue. In *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, volume 7275 of *Lecture Notes in Computer Science*, pages 69–81. Springer, 2012.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, 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 Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 188–205. Springer, 2014.

[Cor17] Jean-Sébastien Coron. High-order conversion from boolean to arithmetic masking. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 93–114. Springer, 2017.

[CPRR13] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shiho Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 410–424. Springer, 2013.

[CRB$^+$16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 shares in hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2016.

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. A new algorithm for switching from arithmetic to boolean masking. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 89–97. Springer, 2003.

[Deb12] Blandine Debraize. Efficient and provably secure methods for switching from arithmetic to boolean masking. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2012.

[DGC17] Daniel Dinu, Johann Großschädl, and Yann Le Corre. Efficient masking of arx-based block ciphers using carry-save addition on boolean shares. In Phong Q. Nguyen and Jianying Zhou, editors, *Information Security - 20th International Conference, ISC 2017, Ho Chi Minh City, Vietnam, November 22-24, 2017, Proceedings*, volume 10599 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2017.

[FBR$^+$21] 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 Cryptol. ePrint Arch.*, 2021:479, 2021.

[FGP$^+$18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

[Gao20] Pengfei Gao. Formal verification of masking countermeasures for arithmetic programs. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 1385–1387. IEEE, 2020.

[GHP$^+$21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. *30th USENIX Security Symposium, USENIX Security 2021*, 2021.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. Generic low-latency masking in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):1–21, 2018.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST Non-Invasive Attack Testing Workshop*, 2011.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 3. ACM, 2016.

[Gou01] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. In *Advances in Cryptology - ASIACRYPT 2021*, 2021.

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

[GXSC20] Pengfei Gao, Hongyi Xie, Fu Song, and Taolue Chen. A hybrid approach to formal verification of higher-order masked arithmetic programs. *CoRR*, abs/2006.09171, 2020.

[GXZ$^+$19] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song, and Taolue Chen. Quantitative verification of masked arithmetic programs against side-channel attacks. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 155–173. Springer, 2019.

[GZSW19] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. Verifying and quantifying side-channel resistance of masked software implementations. *ACM Trans. Softw. Eng. Methodol.*, 28(3):16:1–16:32, 2019.

[HB21] Vedad Hadzic and Roderick Bloem. COCOALMA: A versatile masking verifier. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, pages 1–10. IEEE, 2021.

[HT19] Michael Hutter and Michael Tunstall. Constant-time higher-order boolean-to-arithmetic masking. *J. Cryptogr. Eng.*, 9(2):173–184, 2019.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, 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.

[KRJ14] Mohamed Karroumi, Benjamin Richard, and Marc Joye. Addition with blinded operands. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, volume 8622 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2014.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[Man82] M. Morris Mano. *Computer system architecture*. Prentice Hall, 1982.

[MMSS19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-resistant masking revisited or why proofs in the robust probing model are needed. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):256–292, 2019.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.

[MPH21] Quentin L. Meunier, Etienne Pons, and Karine Heydemann. Leakageverif: Scalable and efficient leakage verification in symbolic expressions. *IACR Cryptol. ePrint Arch.*, page 1468, 2021.

[Nat02] National Institute of Standards and Technology (NIST). FIPS-180-2: Secure Hash Standard, 2002.

[NP04] Olaf Neiße and Jürgen Pulkus. Switching blindings with a view towards IDEA. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 230–239. Springer, 2004.

[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.

[RBN$^+$15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic addition over boolean masking - towards first- and second-order resistance in hardware. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, volume 9092 of *Lecture Notes in Computer Science*, pages 559–578. Springer, 2015.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In Dongdai Lin and Kazue Sako, editors, *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*, volume 11443 of *Lecture Notes in Computer Science*, pages 534–564. Springer, 2019.

[TMC+21] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, ÇagdaËs Çalık, Lawrence Bassham, Jinkeon Kang, and John Kelsey. Status report on the second round of the nist lightweight cryptography standardization process. Technical report, Tech. rep. https://doi. org/10.6028/NIST. IR. 8369. Gaithersburg, MD, USA . . . , 2021.

A Fourier Expansion of the Arithmetic Addition

Recall the Fourier expansion of the AND, OR and XOR functions:

$$\begin{aligned}
\textsc{And} \quad & W(a \wedge b) = \frac{1}{2} + \frac{1}{2}a + \frac{1}{2}b - \frac{1}{2}ab \\
\textsc{Or} \quad & W(a \vee b) = -\frac{1}{2} + \frac{1}{2}a + \frac{1}{2}b + \frac{1}{2}ab \\
\textsc{Xor} \quad & W(a \oplus b) = ab
\end{aligned}$$

Additionally, note that Fourier expansions represent Boolean functions as a polynomial over the real domain $\{1, -1\}$, where 1 represents FALSE and -1 represents TRUE. Consequently, monomials x^c with even exponents c evaluate to 1 in Fourier expansions. The Fourier expansion of the carry and sum can hence be expressed as:

$$\begin{aligned}
\textsc{Carry} \quad & W(c^{(j)}) = W((u^{(j)} \oplus u^{(j)}) \wedge c^{(j-1)}) \vee (u^{(j)} \wedge u^{(j)})) \\
& = -(0.25u^{(j)})^2(u^{(j)})^2c^{(j-1)} - 0.25(u^{(j)})^2(u^{(j)})^2 - 0.25(u^{(j)})^2u^{(j)}c^{(j-1)} - 0.25u^{(j)}(u^{(j)})^2c^{(j-1)} \\
& + (0.25u^{(j)})^2u^{(j)} + 0.25u^{(j)}(u^{(j)})^2 - 0.5u^{(j)}u^{(j)}c^{(j-1)} + 0.25u^{(j)}c^{(j-1)} + 0.25u^{(j)}c^{(j-1)} \\
& + 0.25u^{(j)} + 0.25u^{(j)} + 0.25c^{(j-1)} + 0.25 \\
& = 0.25c^{(j-1)} - 0.25 - 0.25u^{(j)}c^{(j-1)} - 0.25u^{(j)}c^{(j-1)} + 0.25u^{(j)} + 0.25u^{(j)} \\
& - 0.5u^{(j)}u^{(j)}c^{(j-1)} + 0.25u^{(j)}c^{(j-1)} + 0.25u^{(j)}c^{(j-1)} \\
& + 0.25u^{(j)} + 0.25u^{(j)} + 0.25c^{(j-1)} + 0.25 \\
& = 0.5c^{(j-1)} + 0.5u^{(j)} + 0.5u^{(j)} - 0.5u^{(j)}u^{(j)}c^{(j-1)} \\
& W(c[0]) = 1 \\
\textsc{Sum} \quad & W(sum^{(j)}) = W(W(u^{(j)} \oplus u^{(j)}) \oplus c^{(j)}) \\
& = W(u^{(j)}u^{(j)} \oplus c^{(j)}) \\
& = u^{(j)}u^{(j)}W(c^{(j)}) \\
& = 0.5u^{(j)}u^{(j)}c^{(j)} + 0.5u^{(j)} + 0.5u^{(j)} - 0.5c^{(j)}
\end{aligned}$$

B Sanity Check Measurement Setup (RNG Off)

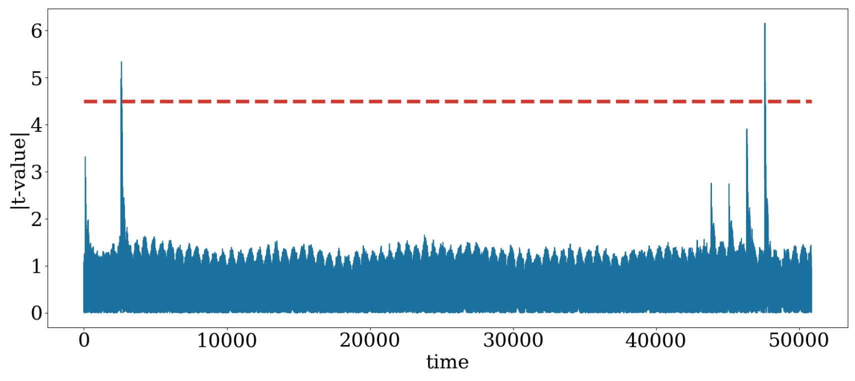

Figure 5: T-test statistics of the fixed version of [CGV14] A2B with 400 000 traces and RNG off.

C Iterative and unrolled circuits

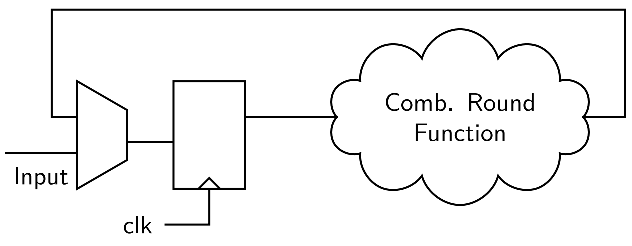

Figure 6: Iterative circuit [BGSD10]

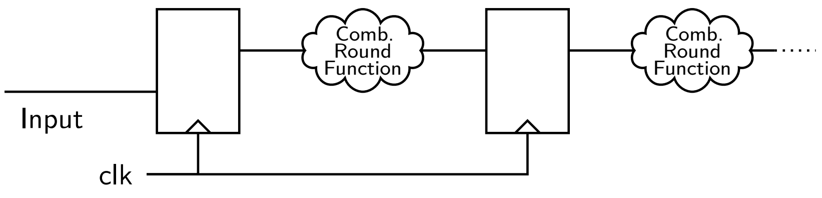

Figure 7: Unrolled circuit [BGSD10]

D False positive in Goubin-A2B

For reasons of readability, we omit to indicate that we always refer to the LSB, i.e., skip $^{(0)}$.

Approximated Fourier expansion

$$\begin{aligned}
\mathcal{C}((Y \wedge (Y \oplus a_1)) \oplus ((Y \oplus a_0) \wedge a_1)) &=? \\
\mathcal{C}(Y \oplus a_0) &= \{\{Y, a_0\}\} \\
\mathcal{C}(Y \oplus a_1) &= \{\{Y, a_1\}\} \\
\mathcal{C}(Y \wedge (Y \oplus a_1)) &= \{\{1\}, \{Y\}, \{Y, a_1\}, \{a_1\}\} \\
\mathcal{C}((Y \oplus a_0) \wedge a_1)) &= \{\{1\}, \{Y, a_0\}, \{a_1\}, \{Y, a_0, a_1\}\} \\
\mathcal{C}((Y \wedge (Y \oplus a_1)) \oplus ((Y \oplus a_0) \wedge a_1)) &= \mathcal{C}((Y \oplus a_0) \wedge a_1)) \otimes \mathcal{C}(Y \wedge (Y \oplus a_1))\} \\
&= \{\{1\}, ...\{Y^2, a_0, a_1\}, ...\}
\end{aligned}$$

Note: $Y^2 = 1$ because in Fourier expression each element is either 1 (False) or -1 (True).

Exact fourier expansion

$$\begin{aligned}
W((Y \wedge (Y \oplus a_1)) \oplus ((Y \oplus a_0) \wedge a_1)) &=? \\
W(Y \oplus a_0) &= Y a_0 \\
W(Y \oplus a_1) &= Y a_1 \\
W(Y \wedge (Y \oplus a_1)) &= -0.5\, Y^2 a_1 + 0.5\, Y a_1 + 0.5\, Y + 0.5 \\
&= -0.5\, a_1 + 0.5\, Y a_1 + 0.5\, Y + 0.5 \\
W((Y \oplus a_0) \wedge a_1)) &= -0.5\, Y a_0 a_1 + 0.5\, Y a_0 + 0.5\, a_1 + 0.5 \\
W((Y \wedge (Y \oplus a_1)) \oplus ((Y \oplus a_0) \wedge a_1)) &= -0.25\, Y^2 a_0 a_1^2 + 0.25\, Y a_0 a_1^2 + 0.25\, Y^2 a_0 - 0.5\, Y a_0 a_1 \\
&\quad + 0.25\, Y a_1^2 + 0.25\, Y a_0 + 0.50\, Y a_1 - 0.25\, a_1^2 + 0.25\, Y + 0.25 \\
&= -0.25\, a_0 + 0.25\, Y a_0 + 0.25\, a_0 - 0.5\, Y a_0 a_1 \\
&\quad + 0.25\, Y + 0.25\, Y a_0 + 0.50\, Y a_1 - 0.25 + 0.25\, Y + 0.25
\end{aligned}$$