Compress: Reducing Area and Latency of Masked Pipelined Circuits

Gaëtan Cassiers[1], Barbara Gigerl[1], Stefan Mangard[1], Charles Momin[2] and Rishub Nagpal[1]

[1] TU Graz, Graz, Austria, firstname.lastname@iaik.tugraz.at
[2] UCLouvain, Louvain-la-Neuve, Belgium, firstname.lastname@uclouvain.be

Abstract. Masking is an effective countermeasure against side-channel attacks. It replaces every logic gate in a computation by a gadget that performs the operation over secret sharings of the circuit's variables. When masking is implemented in hardware, care should be taken to protect against leakage from glitches, which could otherwise undermine the security of masking. This is generally done by adding registers, which stop the propagation of glitches, but introduce additional latency and area cost. In masked pipeline circuits, a high latency further increases the area overheads of masking, due to the need for additional registers that synchronize signals between pipeline stages. In this work, we propose a technique to minimize the number of such pipelining registers, which relies on optimizing the scheduling of the computations across the pipeline stages. We release an implementation of this technique as an open-source tool, Compress. Further, we introduce other optimizations to deduplicate logic between gadgets, perform an optimal selection of masked gadgets, and introduce new gadgets with smaller area. Overall, our optimizations lead to circuits that improve the state-of-the art in area and achieve minimal latency. For example, a masked AES based on an S-box generated by Compress reduces latency by 19 % and area by 10 % over the state of the art.

Keywords: Side-channel · Masking · HPC

1 Introduction

Physical side-channel attacks that exploit information leakage such as the power consumption or the electromagnetic radiation of cryptographic implementations are an important security threat. Masking is a common countermeasure against these attacks [CJRR99]. Its core principle is to replace every variable x in a computation with a secret sharing $\boldsymbol{x} = (x_0, \ldots, x_{d-1})$ such that $x = x_0 \star \cdots \star x_{d-1}$, where $\star$ is a group law and any set of $d-1$ shares x_i. A common example is Boolean masking, where values belong to $\mathbb{F}_2$ and the group operation is $\oplus$. The computations to mask are typically decomposed in elementary operations (e.g., simple logic gates) which are then replaced by gadgets that compute securely over shared data. In this work, all masked circuits are based on Boolean masking, but most of the contributions also work with other kinds of masking.

Securely masking a circuit is a difficult task. A first challenge is that the security of small gadgets may not directly extend to their combination, leading to so-called composition issues [CPRR13, BBD+16]. Another challenge comes from physical defaults such as glitches and transitions, that can break the independence assumptions needed for masking to be secure [MPG05, NRS11]. These issues can also be combined [FGP+18, MMSS19, MKSM22].

The hardware private circuit (HPC) masking scheme [CGLS21] provides a solution to these challenges. Based on the composable notion of glitch-robust probe-isolating non-

interference (PINI) [CS20], it ensures that gadgets are composable in the presence of glitches. Further, HPCs have also been proven secure against transition leakage [CS21]. In [KM22], Knichel and Moradi introduce the HPC3 AND gadget, which has a latency of only 1 clock cycle, whereas the latency of the original HPC1 and HPC2 AND gadgets [CGLS21] was 2 clock cycles. There are other HPC gadgets, such as the GHPC gadgets that implement an arbitrary function at first-order [KSM22] or optimized squaring gadgets [CMM+23b].

The simple composition properties of the HPC masking scheme make it feasible to verifiy the security of implementations at scale (e.g., using the fullVerif tool [Cas20]). It also makes it an interesting target for automated generation of masked implementations. AGEMA [KMMS22] is a tool to automatically generate a masked circuit from an unprotected netlist, when provided with information about the sensitivity of the input wires. In [MCS22], Momin et al. introduce handcrafted architectures for a masked AES implementation with better performance than the ones generated by AGEMA. For the generation of the AES S-box, they introduce a tool[1] that optimizes the latency of the S-box by exploiting the asymmetric latency of the HPC2 gadget: it has a latency of 2 clock cycles w.r.t. one of the input sharings, and only 1 clock cycle w.r.t. the other one. This optimization is also part of the recent AGMNC [WFP+23] tool, which further introduces new AND-XOR HPC gadgets. These gadgets implement the Toffoli gate (computation of $w \oplus (x \wedge y)$) more efficiently than a composition of AND and XOR gadgets.

In this paper, we propose new optimizations for HPCs, working both on the composition of gadgets, and on the gadgets themselves. We introduce Compress[2], an open-source tool[3] which uses our optimizations to generate efficient masked pipeline circuits.

We focus on the generation of pipeline circuits, that is, circuits that are composed of a sequence of combinational logic stages, where the wires that connect a stage to the next are going through registers (typically implemented as D-flip-flops). We build these circuits by composing the gadgets (which are themselves small pipeline circuits) together, with the help of additional registers to ensure proper separation of the pipeline stages. The big advantages of pipeline circuits are their simplicity (e.g., there is no control logic) and high throughput, as they perform one evaluation per clock cycle. This makes them good candidates for the implementation of sub-components in cryptographic algorithms, where the high throughput enables serialized implementation strategies, and a single pipeline circuit is used to perform many parallel computations sequentially (e.g. S-boxes). We show that it is also easy to automatically generate masked pipeline circuits thanks to their simplicity, avoiding tedious design work when such circuits contain dozens of gates with little structure such as the AES S-box by Boyar and Peralta [BP12]. Pipeline circuits can then be integrated in circuits with more complex architectures, either by hand [MCS22], or automatically (e.g., with EasiMask [BSG23]).

Our work is based on the observation that most previous works were focused on finding efficient Boolean circuit representations of functions [BP12, CGLS21], or designing new gadgets with reduced randomness usage or lower latency. However, these works generally leave out "low-hanging fruit" optimizations in the composition of gadgets and inside the gadgets themselves. Most of our optimizations aim at reducing the number of registers in masked pipelines, which actually may represent more than 70 % of the total design area [MCS22]. Securing CMOS logic against glitches generally necessitates some *glitch-stopping* registers, which is the root cause of the high latency of masked circuits. For pipeline circuits, this high latency in turn forces to add *pipelining* registers, in order to properly synchronize signals, beyond the ones needed to prevent glitch-leakage.

At a high level, our main goal is to reduce the number of pipelining registers in a circuit.

[1] Available at https://github.com/simple-crypto/SMAesH/blob/main/hdl/aes_enc128_32bits_hpc2/sbox/hpc_veriloger.py.

[2] Composable Optimizer of Masked Pipelines with Register-Enhanced Staging Selection

[3] Available at ANONYMIZED.

This is achieved by combining multiple techniques. First, at the gadget composition level, we optimize the staging of computations. That is, we assign every gadget in the composition to its pipeline stage(s). Further, we may duplicate gadgets. Indeed, when a gadget is small and its output is used in multiple pipeline stages, it is sometimes more efficient to instantiate the gadget multiple times, instead of having pipelining registers to forward its output to all later pipeline stages where it is used. Second, we tackle the issue of duplicate pipeline registers inside gadgets. This issue comes from the presence of pipelining registers inside the gadgets themselves, and these registers may be redundant (i.e., carry the same value or a closely related value) with pipelining registers inside other gadgets or registers added at the composition level.

We also introduce other optimizations. At the level of individual gadgets, we reduce the area cost of the HPC2 and HPC3 gadgets, mainly through an optimized handling of the so-called *inner-domain* terms (i.e., term of the form $x_i \wedge y_i$, where the input sharings are $\boldsymbol{x} = (x_0, \ldots, x_{d-1})$ and $\boldsymbol{y} = (y_0, \ldots, y_{d-1})$), as well as HPC2 and HPC3 variants that implement the Toffoli gate, in a more efficient way than the AND-XOR gadgets of [WFP+23]. Finally, our circuits are the first ones (to the best of our knowledge) to combine HPC2 and HPC3 gadgets. This allows to efficiently build low-latency circuits by using the single-cycle latency HPC3 where needed, while HPC2, which is more area-efficient than HPC3, can be used when the operands are not both on the critical latency path (thanks to its 1-2 cycle asymmetric latency).

Combined, all these optimizations bring significant performance improvements. In particular, we design a pipeline AES S-box with 33 % latency and 11 % area gain over the state of the art. We further adapt the state-of-the-art masked AES HPC implementation of [MCS22], leading to an overall latency (and throughput) improvement of 19 %, and an area reduction of 10 %. Our tool Compress is not limited to the design of masked S-boxes. As an example, we apply it to multiple architectures of 32-bit adders.

This work is structured as follows: Section 2 introduces the HPC masking scheme and its use to build pipeline circuits from gadgets. Section 3 presents the core ideas behind Compress and the optimization problem it solves. Section 4 discusses the optimizations to deduplicate pipelining registers inside gadgets, and Section 5 details the other optimizations to the HPC2 and HPC3 gadgets. Next, Section 6 discusses the results of the tool and compares it to the state of the art for multiple masked circuits: AES S-box and its integration in a complete masked AES, Skinny S-box and binary adders. Finally, we discuss in more detail the related works (Section 7).

2 Background

The security of masked circuits is often evaluated in the t-probing model [ISW03], where computations are represented as an abstract arithmetic circuit, and the adversary may probe the values carried by any set of t wires in the circuit (t is known as the masking order). A circuit is secure if the values observed by the adversary are independent of the sensitive values, i.e., all non-masked values represented by sharings in the circuit. When masking with d shares, the security order t is at most $d - 1$.

When considering glitches and transitions, the circuit model is closer to concrete synchronous circuits, where the computation is executed over multiple clock cycles, and registers carry values from one clock cycle to the next [CS21]. For these circuits, the robust probing model [FGP+18] allows the adversary to use extended probes, which allow the observation of multiple wires. For a glitch-extended probe, the observed wires are all the wires that belong to the combinatorial circuit that computes the probed wire, i.e., glitches propagate through combinatorial gates but are stopped by registers. For a transition-extended probe, the value carried by the probed wire is observed at two consecutive clock cycles. A glitch+transition-extended probe represents the combination

Algorithm 1 Sharewise-X with d shares.

Input: Sharings $\boldsymbol{x}$, $\boldsymbol{y}$, binary gate X (e.g., XOR, AND...).
Output: Sharing $\boldsymbol{z}$.

for $i = 0$ to $d-1$ **do**
  $z_i = \mathsf{X}(x_i, y_i)$

Algorithm 2 HPC2 AND gadget with d shares.

Input: Sharings $\boldsymbol{x}$, $\boldsymbol{y}$
Output: Sharing $\boldsymbol{z}$ such that $z = x \wedge y$.

for $i = 0$ to $d-1$ **do**
  for $j = i+1$ to $d-1$ **do**
    $r_{ij} \xleftarrow{\$} \mathbb{F}_2$; $r_{ji} \leftarrow r_{ij}$
for $i = 0$ to $d-1$ **do**
  $p_{ii} \leftarrow \mathsf{PR}(x_i \mathsf{PR}(y_i))$
  for $j = 0$ to $d-1, j \neq i$ **do**
    $p_{ij} \leftarrow \mathsf{R}(\overline{x_i} \wedge \mathsf{PR}(r_{ij})) \oplus \mathsf{R}(x_i \wedge \mathsf{R}(y_j \oplus r_{ij}))$
  $z_i \leftarrow \bigoplus_{j=0}^{d-1} p_{ij}$

Algorithm 3 HPC3 AND gadget with d shares.

Input: Sharings $\boldsymbol{x}$, $\boldsymbol{y}$
Output: Sharing $\boldsymbol{z}$ such that $z = x \wedge y$.

for $i = 0$ to $d-1$ **do**
  for $j = i+1$ to $d-1$ **do**
    $r_{ij} \xleftarrow{\$} \mathbb{F}_2$; $r_{ji} \leftarrow r_{ij}$
    $r'_{ij} \xleftarrow{\$} \mathbb{F}_2$; $r'_{ji} \leftarrow r'_{ij}$
for $i = 0$ to $d-1$ **do**
  $p_{ii} \leftarrow \mathsf{PR}(x_i)\,\mathsf{PR}(y_i)$
  for $j = 0$ to $d-1, j \neq i$ **do**
    $p_{ij} \leftarrow \mathsf{R}\left((\overline{x_i} \wedge r_{ij}) \oplus r'_{ij}\right) \oplus \mathsf{PR}(x_i) \wedge \mathsf{R}(y_j \oplus r_{ij})$
  $z_i \leftarrow \bigoplus_{j=0}^{d-1} p_{ij}$

of these models, giving access to all wires in the combinatorial circuit for two consecutive clock cycles.

Hardware private circuit (HPC) is an arbitrary-order masking scheme with $t = d-1$ probing security against glitches and transitions [CGLS21, CS21]. To mask a circuit with HPC, it must be decomposed in simple gates (typically XOR, AND, NOT). Then, conceptually, each wire is replaced by a sharing and each gate is replaced by a gadget. For linear gates (e.g., XOR), a sharewise gadget implementing the gate can be used (e.g., Sharewise-XOR shown in Algorithm 1). For affine gates, the "offset" term is applied to only one of the shares, for example, a NOT gadget simply applies NOT to the first share. Non-linear gates are more complex, and the design of multiplication/AND gadgets is an active research area. Two common HPC AND gadgets are HPC2 [CGLS21] and HPC3 [KM22][4]. The HPC3 gadget is described in Algorithm 3, where $\mathsf{R}(\cdot)$ denotes a glitch-stopping register and $\mathsf{PR}(\cdot)$ denotes a pipelining register. This gadget has latency of one clock cycle and uses $d(d-1)$ random bits, while the HPC2 gadget (Algorithm 2) has an asymmetric latency of one clock cycle w.r.t. one input, and two clock cycles w.r.t. the other input sharing. It uses less randomness than HPC3 ($d(d-1)/2$ bits), but has a higher logic area. These two gadgets satisfy the glitch-robust probe-isolating non-interference (PINI) security property [CS20], which means that they can be arbitrarily composed (also with sharewise gadgets), and the resulting circuit is $d-1$-probing secure with glitches. The circuit is also secure with glitches and transitions under some additional conditions on its structure, which trivially satisfied in many cases, such as when implementing a substitution-permutation network (SPN) with at least 2 clock cycles per round [CS21].

While masking a circuit with only sharewise gadgets is a simple transformation, using the HPC2 or HPC3 gadget (or, generally, gadgets implementing a non-linear gate) is more complex because these gadgets introduce additional latency in the circuit. This means that masked non-linear sub-circuits such as S-boxes in SPNs often have a high latency, which may greatly diminish the overall efficiency of masked implementations [KMMS22]. Indeed, masking a circuit by simply replacing gates with gadgets will need to cleverly use clock gating to properly synchronize all the signals in the circuit, and pay a high cost in latency, on top of the area overhead of masking. Another strategy for masked implementations

[4]We do not consider here the HPC1 [CGLS21] gadget, since it has no significant advantage over HPC2 and HPC3 when using Boolean masking and a relatively low number of shares.

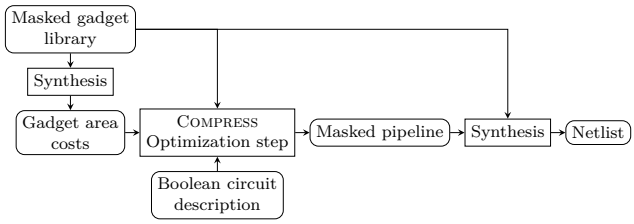


Figure 1: COMPRESS flow. Rectangles denote flow steps, rounded corners denote inputs, outputs and intermediate flow artifacts.

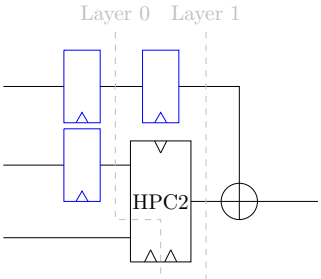


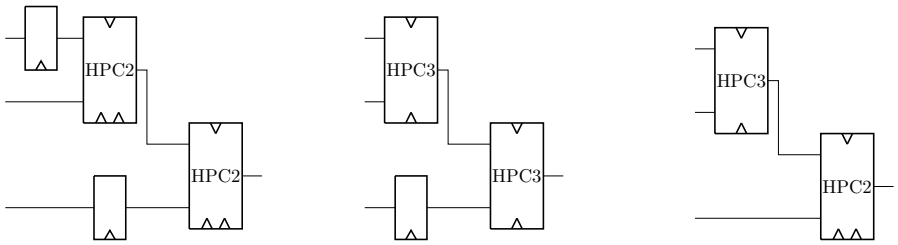
Figure 2: Example of masked pipelined circuit.

is to exploit pipelining: the synchronization is achieved through the addition of registers instead of clock gating. Pipelining does not improve latency and increases area cost, but it increases the throughput of the sub-circuit. When multiple computations can be performed in parallel (e.g., a block cipher in a parallelizable mode of operation), pipelining translates into a large throughput gain over clock gating, at a small area overhead. Another way to exploit pipelining is to switch to a more serialized architecture. For example, in a SPN implementation, the masked S-box may be instantiated only once (or a few times) and be evaluated more than once per round. Serializing the architecture (as shown in Figure 2) reduces the area cost, and it combines well with pipelining: the high throughput of the pipeline minimizes the latency overhead. As a result, masking with pipelining is a good technique to achieve excellent latency/area trade-offs (except for the extreme "very high latency/low area" case) [MCS22].

3 Generic Optimization of Masked Pipelined Circuits

COMPRESS takes as input the Boolean circuit to mask and outputs a netlist. This netlist implements the circuit as a pipeline that has the requested number of stages. As shown in Figure 1, the masked gadget library is another important input for COMPRESS, which takes the areas of the gadgets as parameters of the optimization target function.

The goal of COMPRESS is to generate a pipeline of masked gadgets with optimal gadget selection and computation scheduling in order to achieve the requested latency while minimizing area. The tool exploits the following degrees of freedom: gadget selection, scheduling of computations across pipeline stages, and gadget replication. First, COMPRESS selects a suitable gadget for AND gates. There are multiple gadgets with different latency, area and randomness usage characteristics (HPC2, HPC3, etc.) available, as illustrated in Figure 3. The assignment of input sharings is also considered in case of asymmetric gadgets, such as HPC2. Second, COMPRESS optimizes the scheduling of computations by deciding which pipeline stage a computation should best be performed in, and instantiating the pipelining registers that forward the computed data across register stages. Optimized scheduling reduces the number of pipelining registers to be instantiated, thereby reducing area as shown in Figure 4. Third, COMPRESS may perform gadget replication, which means that if a value is used in multiple clock cycles and the gadget that computes it is small (e.g., an XOR gadget), it might be more efficient to replicate the gadget in multiple pipeline stages instead of instantiating pipelining registers (provided that the operands of the gadget are available at the corresponding pipeline stages). For example, in Figure 5, an XOR gadget is duplicated in order to avoid the instantiation of two masked registers (the dashed line indicates a value used elsewhere in the circuit).

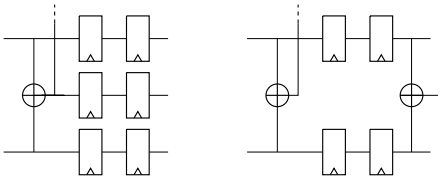


(a) HPC2-only [MCS22] (b) HPC3-only (c) Compress

Figure 3: AND3 implementation with minimal latency: Compress reaches the minimum possible latency, while combining HPC2 and HPC3 gadgets to minimize the total area.

(a) Naive (b) Compress

Figure 4: Scheduling of computations across pipeline stages.



(a) Naive (b) Compress

Figure 5: Gadget replication.

Regarding the security, all the gadgets instantiated by Compress are glitch-robust pipeline PINI gadgets, therefore the overall generated circuit is a glitch-robust PINI pipeline [CGLS21]. Further, under the conditions given in [CS21], this guarantees security against combined glitch and transition leakage.

The core part of Compress consists representing the masked circuit generation as a constraint optimization problem. We then use OR-tools [PF, Gun19] to solve this problem. Compress splits the computation in pipeline stages $0, \ldots, L$, where the inputs are fed in the circuit at stage 0, while the outputs are connected to stage L.

For each intermediate value w in the Boolean circuit and for each pipeline stage s, Compress instandiates a Boolean variable v_s^w ("valid"), which is true iff there is a sharing representing the value w in the pipeline stage s. For each of these variables, there is also a Boolean variable c_s^w that is true iff there is a gadget that outputs w at the stage s. Then, r_s^w indicates the presence of a pipelining register that forwards the value of w from stage s to stage $s+1$, for all w and for $s \in \{0, \ldots, L-1\}$. These variables are connected by the constraint

$$\begin{aligned} v_0^w &= c_0^w \\ v_s^w &= c_s^w \vee \left(v_{s-1}^w \wedge r_{s-1}^w\right) \quad \text{for } s > 0. \end{aligned}$$

The instantiation of gadgets is then represented: each value is computed by a logic gate (e.g. XOR, AND, ...) that can be implemented by one or multiple gadgets. Indeed, for simple gates such as the XOR gate, we only consider the trivial sharewise implementation of the gate (d parallel gates), while for more complex gates, there may be multiple ways to implement them (e.g., HPC2 or HPC3 for the AND gate). For each value v, each stage s, and each gadget type g that implements the required gate, the Boolean variable $g_s^{g^w}$ indicates if a gadget of type g is instantiated to output w in stage s. Since we consider only gadgets with a single output sharing, this is well-defined. We set $g_s^{g^w}$ to false when it

corresponds to a gadget that takes an input before stage 0. Since an instantiated gadget requires valid inputs to produce a valid output, we let

$$g'^{g^w}_s = g^{g^w}_s \wedge \bigwedge_{w' \in \mathsf{op}(w)} v^{w'}_{s-\mathsf{lat}(g^w,w')},$$

where $\mathsf{op}(w)$ is the set of operands of the logic gate that computes w, while $\mathsf{lat}(g^w, w')$ is the relative latency of the input of the gadget g^w connected to w' w.r.t. its output (i.e., the difference in pipeline stage numbers between the input and the output). We can then constrain the "compute" variables:

$$c^w_s = \bigvee_{g^w \in G^w} g'^{g^w}_s$$

where G^w is the set of all possible gadgets g^w to compute w. For asymmetric gadgets such as HPC2, the choice of input sharing assignation is handled by including multiple variants of the gadget in G^w, with different input ordering (e.g., we include a "swapped inputs HPC2" whenever the "normal HPC2" gadget belongs to G^w).

For input variables i we instead set c^i_0 to true and c^i_s to false for $s > 0$. Finally we constrain the outputs: for all output wires o, v^o_L must be true.

The objective of the optimization problem is the minimization of the area used by the masked circuit. Since Compress takes as an input the area cost of each gadget type, including a "pipelining register" gadget, the total cost is the sum of the $g^{g^w}_s$ and r^w_s variables, weighted by the area of the corresponding gadgets, in addition to the area of the pipeling registers:

$$C = \sum_{s=0}^{L} \sum_{w} \sum_{g^w \in G^w} a_g g^{g^w}_s + \sum_{s=0}^{L} \sum_{w} a_{\mathsf{reg}} r^w_s$$

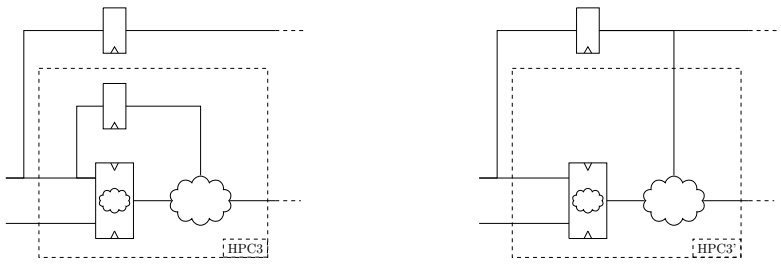
where a_g is the area of the gadget g. Since the solver works only with integers, we use a fixed point representation for the areas a_g. Let us remark that while the constraints of the problem guarantee that a part of the circuit correctly compute the output at the required pipeline stage, it allows nonsensical logic to be instantiated (e.g., gadgets with no valid inputs). The optmimization ensures that such useless logic never appears, as long as all logic costs are strictly positive.

Further, we take into account the cost of randomness generation for the masked gadgets. We assume that a PRNG is instantiated along with the masked circuit, and that it should provide enough randomness to run the pipeline continously: each randomness input of a gadget is connected to an output of the PRNG, and the PRNG should be able to refresh its full output at every clock cycle. Concretely, we use an unrolled Trivium, following [CMM+23a]. Then, we observe that the marginal area cost of one additional bit of randomness per clock cycle from the PRNG is roughly constant. This allows to include the PRNG cost into the area optimization function by simply increasing the areas a_g by the area needed to generate the randomness for each gadget. Overall, this approach ensures that the full cost of the masked pipeline is optimized by Compress.

Let us finally remark that while the optimization problem is order-specific (the gadget costs depend on the masking order), Compress's output is still generic: it can be synthesized at all masking order, provided that the gadgets support it.

4 Register Reuse through Gadget Decomposition

In this section, we further reduce the amount of registers in the masked circuits by looking at registers instantiated inside gadgets. For example, an HPC3 gadget contains pipelining registers for the x_i shares (see Algorithm 3). If two HPC2 gadgets have the same input



(a) Without register separation (b) With register separation

Figure 6: Illustration of register de-duplication thanks to the separation of a pipelining register out of the HPC3 gadget.

sharing $\boldsymbol{x}$, then these registers will be duplicated. There could also be duplication between pipelining registers inside a gadget and pipelining registers outside gadgets. Our approach to avoid such inefficiencies is to decompose gadgets into multiple parts, which eliminates pipelining registers from such gadgets, and instead exposes them as latency constraints. This avoids duplication and COMPRESS's gadget selection (e.g., by choosing the order of the input sharings in a gadget) may bring further optimizations.

4.1 Separate Pipelining Registers

As a first step, we handle pipelining of input sharings. When a gadget uses registers directly on input shares, we add a new input sharing to the gadget, that must have identical share values, but at a different pipeline stage. Concretely, this technique is applied to the input shares y_i of HPC2 (separating d registers), and to the input shares x_i and y_i of HPC3 (separating $2d$ registers). Figure 6 illustrates how this separation works, and how it leads to the de-duplication of registers.

The new gadgets bring the additional constraint of having identical shares on two input sharings, which is required for correctness and also for security. Compared to the standard definition of gadgets, this is a stronger requirement, but it has no significant impact on the correctness and security analysis since it is enough to consider that the classical definitions apply, under the indentical sharings condition. This condition is non-trivial in presence of gadget duplication because a sharing can be computed twice, possibly with different (but equivalent) results. Therefore, we add two constraints to COMPRESS. First, gadgets of different types must not be used to generate the same value w. Therefore, for all w:

$$\mathsf{AtMostOne}(\{\{\mathsf{Any}(g_s^{g^w} : 0 \leq s \leq L\}) : g^w \in G^w\}\}). \tag{1}$$

Second, a sharing can only be computed by multiple identical gadgets if these gadgets do not use randomness:

$$\mathsf{AtMostOne}(\{\{g_s^{g^w} : g^w \in G^w, 0 \leq s \leq L\}\}) \tag{2}$$

for all w where any $g^w \in G^w$ uses randomness. Constraints (1) and (2) guarantee that all sharings of the same wire are equal. While stronger than strictly necessary, these constraint do not increase the overall circuit cost, since gadget duplication is in practice only applied to sharewise gates (which always satisfy the constraints), while the large area of AND gates makes it inefficient to duplicate them.

4.2 Separate Inner-domain Terms

While some pipelining registers can be optimized by separating them out of the gadgets, the above optimization does not apply to all pipelining registers. In this section, we look at the

Algorithm 4 HPC2-cross gadget with d shares.

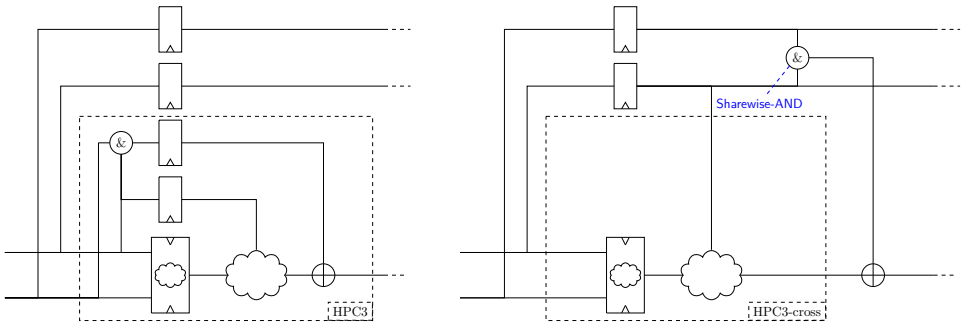
Input: Sharings $\boldsymbol{x}$, $\boldsymbol{y}$
Output: Sharing $\boldsymbol{z}$.

for $i = 0$ to $d-1$ **do**
  for $j = i+1$ to $d-1$ **do**
    $r_{ij} \xleftarrow{\$} \mathbb{F}_2$; $r_{ji} \leftarrow r_{ij}$
for $i = 0$ to $d-1$ **do**
  for $j = 0$ to $d-1, j \neq i$ **do**
    $p_{ij} \leftarrow \mathsf{R}(\overline{x_i} \wedge \mathsf{PR}(r_{ij})) \oplus \mathsf{R}(x_i \wedge \mathsf{R}(y_j \oplus r_{ij}))$
  $z_i \leftarrow \bigoplus_{j=0, j\neq i}^{d-1} p_{ij}$

Algorithm 5 HPC2 AND decomposed in pseudo-gadgets.

Input: Sharings $\boldsymbol{x}$, $\boldsymbol{y}$
Output: Sharing $\boldsymbol{z}$ such that $z = x \cdot y$.

$\boldsymbol{a} \leftarrow$ HPC2-cross $(\boldsymbol{x}, \boldsymbol{y})$
$\boldsymbol{b} \leftarrow$ Sharewise-AND $(\boldsymbol{x}, \boldsymbol{y})$
$\boldsymbol{z} \leftarrow$ Sharewise-XOR $(\boldsymbol{a}, \boldsymbol{b})$



(a) Without separation (b) With full separation

Figure 7: Illustration of register de-duplication thanks to register separation and inner-domain terms separation.

pipelining registers on the so-called *inner-domain* terms in the HPC2 and HPC3 gadgets, namely the terms $x_i \wedge y_i$. Indeed, since these terms perform only sharewise computation, registers are not needed for security, only for proper pipeline staging. We therefore split the AND gadgets into a part that computes these terms (sharewise AND), a part that computes the other terms (HPC2/3-cross), and the XOR their output (sharewise), as illustrated in Figure 7. The decomposition of HPC2 is given in Algorithm 4 and Algorithm 5, and the decomposition of HPC3 follows the same pattern. The three resulting gadgets perform exactly the same computation as the original gadget, but the pipelining registers are now handled by Compress, giving further optimization opportunities.

Regarding the security analysis, there is no change to the leakage, except for the addition or removal of registers that are not needed for security purposes. Formally, there is one technical difficulty in the definitions: the new gadgets (sharewise AND, HPC2/3-cross) do not satisfy the classical definition of a gadget which requires correctness. That is, a gadget must correspond to a function on the unmasked values, which is not the case here (e.g., the unmasked values of Sharewise-AND$((1,0),(1,0))$ and Sharewise-AND$((1,0),(0,1))$ are different). This issue has however no impact on the security definition PINI [CS20], hence we may simply define the notion *pseudo-gadget*, which is the same as *gadget*, without the correctness requirement.

Algorithm 6 HPC2o Toffoli gadget with d shares.

Input: Sharings $\boldsymbol{w}$, $\boldsymbol{x}$, $\boldsymbol{y}$.
Output: Sharing $\boldsymbol{z}$ such that $z = w \oplus (x \wedge y)$.

for $i = 0$ to $d-1$ **do**
  for $j = i+1$ to $d-1$ **do**
    $r_{ij} \xleftarrow{\$} \mathbb{F}_2$; $r_{ji} \leftarrow r_{ij}$
for $i = 0$ to $d-1$ **do**
  $j_i \leftarrow \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases}$
  for $j = 0$ to $d-1, j \neq i$ **do**
    if $j = j_i$ **then**
      $p_{ij} \leftarrow \mathsf{R}(w_i \oplus (x_i \wedge \mathsf{PR}(y_i)) \oplus (\overline{x_i} \wedge \mathsf{PR}(r_{ij}))) \oplus \mathsf{R}(x_i \wedge \mathsf{R}(y_j \oplus r_{ij}))$
    else
      $p_{ij} \leftarrow \mathsf{R}(\overline{x_i} \wedge \mathsf{PR}(r_{ij})) \vee \mathsf{R}(x_i \wedge \mathsf{R}(y_j \oplus r_{ij}))$
  $z_i \leftarrow \bigoplus_{j=0, j\neq i}^{d-1} p_{ij}$

Algorithm 7 HPC3o Toffoli gadget with d shares.

Input: Sharings $\boldsymbol{w}$, $\boldsymbol{x}$, $\boldsymbol{y}$.
Output: Sharing $\boldsymbol{z}$ such that $z = w \oplus (x \wedge y)$.

for $i = 0$ to $d-1$ **do**
  for $j = i+1$ to $d-1$ **do**
    $r_{ij} \xleftarrow{\$} \mathbb{F}_2$; $r_{ji} \leftarrow r_{ij}$
    $r'_{ij} \xleftarrow{\$} \mathbb{F}_2$; $r'_{ji} \leftarrow r'_{ij}$
for $i = 0$ to $d-1$ **do**
  $j_i \leftarrow \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases}$
  for $j = 0$ to $d-1, j \neq i$ **do**
    if $j = j_i$ **then**
      $p_{ij} \leftarrow \mathsf{R}\left(w_i \oplus (x_i \wedge (y_i \oplus r_{ij})) \oplus r'_{ij}\right) \oplus (\mathsf{PR}(x_i) \wedge \mathsf{R}(y_j \oplus r_{ij}))$
    else
      $p_{ij} \leftarrow \mathsf{R}\left((x_i \wedge r_{ij}) \oplus r'_{ij}\right) \oplus (\mathsf{PR}(x_i) \wedge \mathsf{R}(y_j \oplus r_{ij}))$
  $z_i \leftarrow \bigoplus_{j=0, j\neq i}^{d-1} p_{ij}$

5 Optimized Gadgets: HPC2o and HPC3o

5.1 New Gadget Designs

In this section, we go beyond the register re-use of Section 4 by completely eliminating some pipelining registers through optimizations inside the AND gadgets.

Inner-domain term optimization Instead of separating the inner-domain terms for the gadgets, we propose to merge these terms with cross-domain terms. For every $i = 0, \ldots, d-1$, we select a $j_i \neq i$ (e.g., $j_i = 0$ for all $i \neq 0$, and $j_0 = 1$). Then, we integrate the term $x_i \wedge y_i$ into the term p_{ij_i}, by replacing in its computation $\mathsf{R}(\overline{x_i} \wedge \mathsf{PR}(r_{ij_i}))$ with $\mathsf{R}((x_i \wedge \mathsf{PR}(y_i)) \oplus (\overline{x_i} \wedge \mathsf{PR}(r_{ij_i})))$. This transformation removes d registers from the HPC2 gadget and does not damage the security: for the PINI security analysis, we added a term of the domain i to a term that already involves the domain i (see proof in Appendix A).

For HPC3, let us first remark that the NOT gate in the computation $\mathsf{R}\left((\overline{x_i} \wedge r_{ij}) \oplus r'_{ij}\right) \oplus \mathsf{PR}(x_i) \wedge \mathsf{R}(y_i \oplus r_{ij})$ is not necessary. Namely, replacing this computation with $p_{ij} \leftarrow \mathsf{R}\left((x_i \wedge r_{ij}) \oplus r'_{ij}\right) \oplus \mathsf{PR}(x_i) \wedge \mathsf{R}(y_i \oplus r_{ij})$ still leads to a correct gadget. This simplified gadget gives $p_{ij} = (x_i \wedge y_j) \oplus r'_{ij}$, instead of $p_{ij} = (x_i \wedge y_j) \oplus r'_{ij} \oplus r_{ij}$, which sill satisfies the property needed for correctness: $p_{ij} \oplus p_{ji} = (x_i \wedge y_j) \oplus (x_j \wedge y_i)$. Regarding the security, the core observation is that $\mathsf{R}\left((x_i \wedge r_{ij}) \oplus r'_{ij}\right)$ has still the distribution of a fresh random value if r'_{ij} is not observed elsewhere. In other words, the cancellation of the random r_{ij} has no security impact. While the removal of a NOT gate in the gadget has no significant performance impact by itself[5], it becomes useful when optimizing the inner-domain terms. Indeed, while we can apply a similar optimization to HPC2 by turning $\mathsf{R}\left((x_i \wedge r_{ij_i}) \oplus r'_{ij_i}\right)$ into $\mathsf{R}\left((x_i \wedge (y_i \oplus r_{ij_i})) \oplus r'_{ij_i}\right)$, which additionally saves one AND gate.

Finally, we remark that we can apply the AND-XOR trick of [WFP+23] to these AND gadgets: they can be turned into Toffoli gate by adding a share w_i of a third input sharing at the place where $x_i y_i$ is computed. Once again, this technique allows to save pipelining registers for w_i. The tweaks does not break the security, they do not add any new domain:

[5] Let us remark that removing the NOT gate makes the gadget trivially generalizable to any field (our security proof is field-agnostic, provided that the gadget is made correct by turning XOR gates into additions and subtractions where needed, and turning AND gates into products). The change also makes the randomness r_{ij} local [CGZ20] to the gadget, which might help with masking randomness re-use, but is beyond the scope of this work. Further, HPC3o work with any field.

Algorithm 8 Identification of Toffoli gates in Boolean circuit

Input: A Boolean circuit
Output: For each AND gate g, a set L_g of XOR-operands for a Toffoli gate instantiation.
for all AND gates g in the circuit **do**
Let $\boldsymbol{z}$ be the output of g.
$\boldsymbol{o} \leftarrow \boldsymbol{z}$
while $\boldsymbol{o}$ is the operand of only one operation **do**
Let $\boldsymbol{o}$ be the the output of that operation.
$S \leftarrow \{\boldsymbol{o}\}$ ▷ $\boldsymbol{o}$ is now the "result" variable.
$L_g \leftarrow \emptyset$
if $\boldsymbol{o} \neq \boldsymbol{z}$ **then**
while $S \neq \emptyset$ **do**
Pop an element $\boldsymbol{x}$ from S
if $\boldsymbol{x}$ is the output of an XOR gate g' and $\boldsymbol{x}$ is the operand of only one operation **then**
Add the operands of g' to S.
else
Add $\boldsymbol{x}$ to L_g.

they add y_i and w_i to a term where x_i already appears. The final gadgets are given in Algorithm 6 and Algorithm 7, and their formal security proofs are given in Appendix A.

Gate area optimization We introduce another optimization in the HPC2 gadget. In the computation of p_{ij} for $j \neq j_i, i$ (i.e., for the cross-domain terms where the above optimization is not applied), $\mathsf{R}(\overline{x_i} \wedge \mathsf{PR}(r_{ij}))$ and $\mathsf{R}(x_i \wedge \mathsf{R}(y_j \oplus r_{ij}))$ are never both 1. Therefore, combining these terms with an XOR gate gives the same results as combining them with a OR gate, which has a lower area in CMOS designs. This optimization is implemented in Algorithm 6.

5.2 Using Toffoli Gates in Compress

Compress takes as input circuits composed of AND, XOR and NOT gates. Therefore, in order to efficiently make use of the HPC2o and HPC3o gadgets, it should extract Toffoli gates from a circuit of AND and XOR gates.

We use the following approach. For each output sharing $\boldsymbol{a}$ of a AND gate, if $\boldsymbol{a}$ is XORed with $\boldsymbol{b}$ ($\boldsymbol{c} = \boldsymbol{a} \oplus \boldsymbol{b}$) and not used in any other gate, then we may instantiate a Toffoli with $\boldsymbol{b}$ as the third ($\boldsymbol{w}$) input. Further, if $\boldsymbol{c}$ is itself used only once in an XOR with $\boldsymbol{d}$, then $\boldsymbol{d}$ (or $\boldsymbol{c} \oplus \boldsymbol{d}$) is also a good candidate as an input to a Toffoli gate. This continues, until the value is not used in an XOR, or if it is an operand of more than one operation (we don't want to force logic duplication). All these variables can be XORed into the input of the Toffoli gate that contains the AND computation of $\boldsymbol{a}$, which may save pipelining registers. However, some of these variables could be more efficiently computed at a later cycle, and we should not adopt a restrictive all-or-none approach. We therefore consider that a subset of these variables may be XORed in the input of the Toffoli gate, while othersmay be XORed to its output. Further, if $\boldsymbol{d} = \boldsymbol{e} \oplus \boldsymbol{f}$ and $\boldsymbol{d}$ is only used once in the circuit, then we should take $\boldsymbol{e}$ and $\boldsymbol{f}$ in our list of XOR operands instead of $\boldsymbol{d}$, in order to maximize flexibility and avoid dependency on the parentheses between the additions in orignial circuit representation.

The flow of Compress is therefore modified as follows. First, for every AND gate in the circuit, Algorithm 8 is executed in order to identify a list of variables that are candidates to be XORed in a Toffoli gate, and the name of a "result" variable, that is, the XOR of all these variables and of the output of the AND gate. Next, we add an alternative way of computing the result (we keep the approach without Toffoli gate as a solution). This alternative way is based on an extended Toffoli gate, which takes as input the operands of the AND gate and the variables in the candidates list, and it outputs the result.

By introducing multiple computations inside a single extended Toffoli gadget, we go against our previous decomposition approach and, as a result, could lose some of the

scheduling optimizations of Compress. We circumvent this issue by making the extended Toffoli gadget very flexible w.r.t. input and output latency, and by providing Compress the knobs to exploit this flexibility (as well as information on the cost of the gadget depending on how it is used). In more details, the Toffoli gadgets take inputs sharings $\boldsymbol{x}$, $\boldsymbol{y}$ and $(\boldsymbol{w}_i)_i$. It is made of either one HPC2o or HPC3o gadget, whose input $\boldsymbol{w}$ is the XOR of a subset of the sharings $\boldsymbol{w}_i$ (computed using XOR gadgets). The output $\boldsymbol{z}$ of the HPC2o/HPC3o gadget is then forwarded to an arbitrary (subject to optimization) stage deeper in the pipeline by means of registers. In the pipeline stages covered by these registers, the other $\boldsymbol{w}_i$ operands are XORed to the forwarded state (again, the staging of these XOR operations is selected by the optimization solver).

6 Case Studies

In this section, we look at the performance characteristics of the masked pipelines generated by Compress and we compare them to the state of the art designs. The area numbers are obtained by synthesizing the designs with Yosys 0.33 and the Nangate 45 PDK. The area with PRNG is based on the assumption that the proposal of [CMM+23a] is followed: using an unrolled Trivium as PRNG. The PRNG area is then computed as the number of bits multiplied with on the area cost per bit per clock cycle of the PRNG for a large unrolling factor (512).

6.1 Optimized S-boxes

As a first case study for Compress, we generated optimized implementations of the AES S-box (based on the 34 AND gate Boyar-Peralta representation [BP12]) and of the 8-bit Skinny S-box. Our results are given in Table 1 and Table 2. For both cases, we provide the masking order (number of shares) and desired latency as parameters to Compress. Furthermore, in order to analyze the individual contributions of our different optimizations, three results are provided for each parameter set. The "Base" case corresponds to Compress (as described in Section 3) with the HPC2 and HPC3 gadgets. For "Sep", we add all gadget decomposition techniques of Section 4. Lastly, for "Opt", all optimizations of this paper are enabled. We also report similar results from related works. We focus on the most comparable works, that is, the ones with provable security in the glitch-probing model and arbitrary security order.

We observe that some low latency designs require less area than the higher latency ones, even when accounting for randomness usage. This may seem surprising at first, since lower latency designs require more randomness due to the use of more low-latency HPC3 gadgets in place of HPC2 gadgets. However, lower latency generally means a lower amount of pipelining registers, which explains the area gain. These two effects mostly cancel each other, resulting in similar area costs (with PRNG) for the AES and Skinny S-boxes with 4, 5 or 6 cycles of latency, at all considered masking orders.

Compared to all similar state of the art designs (excluding the non-pipeline S-box of [KM22]), our AES S-box achieves both lower latency and lower area. Regarding the Skinny S-box, we observe the same trend when compared to the pipeline S-box of [VCS22]: lower latency and lower area. Compared to the other S-box design of [VCS22], the comparison is more difficult, since this S-box is based on an iterative design and performs two S-box evaluations in 9 clock cycles. This amounts to a throughput of 0.22 evaluations per clock cycles (compared to 1 for a pipeline). However, the area (with PRNG) is about half the one of our designs, therefore, we can somewhat fairly compare two instances of their design to one instance of ours. In such a comparison, the pipeline S-box of Compress still has a higher throughput, and it achieves a lower latency when performing 4 S-box evaluations (8 clock cycles vs. 9).

Table 1: Performance characteristics of HPC AES S-box implementations.

d	Latency	Design	Random bits	Area (kGE)	Area with PRNG (kGE)
2	4	Base	46	3.58	5.10
		Sep		3.28	4.81
		Opt		2.97	4.50
	5	Base	37	3.85	5.08
		Sep		3.56	4.79
		Opt		3.29	4.51
	6	Base	34	4.11	5.24
		Sep		3.81	4.94
		Opt		3.54	4.67
3	4	Base	138	8.04	12.62
		Sep		7.60	12.18
		Opt		7.09	11.67
	5	Base	111	8.63	12.31
		Sep		8.18	11.87
		Opt		7.71	11.40
	6	Base	102	9.07	12.45
		Sep		8.62	12.01
		Opt		8.15	11.54
4	4	Base	276	14.28	23.45
		Sep		13.69	22.86
		Opt		12.96	22.12
	5	Base	222	15.30	22.67
		Sep		14.71	22.08
		Opt		14.00	21.37
	6	Base	204	15.96	22.74
		Sep		15.37	22.15
		Opt		14.65	21.42
5	4	Base	460	22.30	37.58
		Sep		21.57	36.84
		Opt		20.58	35.85
	5	Base	370	23.87	36.16
		Sep		23.13	35.42
		Opt		22.14	34.43
	6	Base	340	24.80	36.09
		Sep		24.06	35.35
		Opt		23.04	34.33
2	6	[MCS22]	34	4.29	5.42
3			102	9.34	12.73
4			204	16.33	23.10
5			340	25.25	36.54
2	8	[KMMS22]*	34	5.34	6.47
3			102	11.21	15.59
4			204	19.22	25.99
5			340	29.27	40.56
2	6	[WFP+23]*	33	3.97	5.06
3			99	9.08	12.37
4			198	16.24	22.81
5			330	25.47	36.43
2	4	[KM22]*†	68	1.85	4.11
3			204	4.86	11.63
4			408	9.26	22.81

*Compiled with Synopsis Design Compiler.
†These designs are not pipelined and are not directly comparable.

Table 2: Performance characteristics of HPC 8-bit Skinny S-box implementations.

d	Latency	Design	Random bits	Area (kGE)	Area with PRNG (kGE)
2	4	Base	12	1.02	1.42
		Sep		0.99	1.38
		Opt		0.89	1.29
	5	Base	9	1.15	1.45
		Sep		1.13	1.43
		Opt		1.03	1.33
	6	Base	8	1.26	1.53
		Sep		1.24	1.51
		Opt		1.15	1.42
3	4	Base	36	2.13	3.33
		Sep		2.08	3.28
		Opt		1.93	3.12
	5	Base	27	2.38	3.28
		Sep		2.35	3.25
		Opt		2.19	3.09
	6	Base	24	2.58	3.37
		Sep		2.54	3.34
		Opt		2.39	3.19
4	4	Base	72	3.65	6.04
		Sep		3.58	5.97
		Opt		3.37	5.76
	5	Base	54	4.06	5.86
		Sep		4.02	5.81
		Opt		3.79	5.58
	6	Base	48	4.34	5.94
		Sep		4.30	5.89
		Opt		4.08	5.67
5	4	Base	120	5.57	9.55
		Sep		5.48	9.47
		Opt		5.20	9.19
	5	Base	90	6.18	9.17
		Sep		6.13	9.12
		Opt		5.82	8.81
	6	Base	80	6.57	9.23
		Sep		6.51	9.17
		Opt		6.21	8.86
2	6	[MCS22]*	8	1.26	1.52
3			24	2.56	3.36
4			48	4.33	5.92
5			80	6.54	9.2
2	9	[VCS22]†	2	0.73	0.79
3			6	1.26	1.46
4			12	1.90	2.30
5			20	2.66	3.33

*This design is generated by the tool of [MCS22], and is used in [VCS22].
†This design is not pipelined, it is a serial implementation that performs 2 S-box evaluations in 9 clock cycles.

Table 3: Performance characteristics of HPC AES-128 implementations (encrypt only), including PRG.

Design	Datapath width	Latency	d	Area (kGE)
Opt	32-bit	85	2	32.2
			3	72.6
			4	128.4
			5	202.3
	128-bit	51	2	120.6
			3	303.4
			4	571.0
			5	918.0
[MCS22]	32-bit	105	2	35.8
			3	74.3
			4	127.6
			5	195.4
	128-bit	71	2	134.2
			3	310.0
			4	561.6
			5	888.4

6.2 Optimized AES

Let us now investigate the impact of the optimized S-boxes generated by Compress on masked cipher implementations. For this purpose we integrate the new latency 4 "Opt" AES S-box to two architectures implementing a masked AES-128 encryption (including the key schedule). The architecutres are based on the ones of [MCS22].

The first case study is a 128-bit (round-based) pipelined architecture. It instantiates 20 S-boxes among which 16 are dedicated to the round computation and 4 to the key-scheduling operating in parallel. The architecture considered is the same as the round-based architecture presented in [MCS22] where the S-boxes instances have been replaced (together with some minor control logic modifications). This architecture performs 5 parallel encryptions to fill its pipeline, achieving a high troughput (0.1 encryption per clock cycle).

The second architecture is a 32-bit serial implementation instantiating 4 S-boxes that are shared between the computation of the rounds and the key scheduling algorithm. In particular, the data routed to the S-boxes is interleaved appropriately such that the round operations and the key evolution mechanism are performed in parallel during a round execution. Overall, the architecture is similar to the 32-bit one from [MCS22].[6] For this architecture, the modifications we a bit more substantial. Indeed, only integrating the (4 cycles) new S-boxes in the key holder described in [MCS22, Figure 8] leads to a situation where the computation are not performed properly anymore. In particular, the implementation computes a round by first feeding into the S-boxes a column of the round key, preparing the update of the key for the next round. Then, in the next four clock cycles, each column of the state is added to a part of the round key and sent to the S-box. During this process, the shift register that holds the key is rotated to ensure that the correct part of the round key is added to the state. Then, once the whole state has been fed to the S-boxes, the round key is updated, in parallel with the MixColumns and ShiftRows operations. While this procedure works with a latency of 6 clock cycles for the S-box, it does not work with 4 clock cycles: the lower latency means that the state update for the round is finished before the key update process is completed. We therefore modify the handling of the round key to make its update start earlier in the round, which allows it to be completed at the same time as the state computation.

Table 3 includes the post-synthesis implementation results for the two architecures (the

[6] Our implementation is derived from the open-source one by the authors of [MCS22]: https://github.com/simple-crypto/SMAesH.

Table 4: Performance characteristics of HPC 32-bit adder implementations (Opt strategy).

Design	Security Order	Number of Shares	Latency	Random bits	Area (kGE)	Area with PRNG (kGE)
RC (Opt)	1	2	31	32	19.23	20.29
			32	31	19.95	20.98
	2	3	31	96	31.42	34.60
			32	93	32.51	35.60
KS (Opt)	1	2	5	374	18.30	30.55
			12	249	26.11	34.38
	2	3	5	1122	45.8	83.06
			12	747	59.84	84.65
Sklansky (Opt)	1	2	6	172	13.77	19.48
			12	151	18.42	23.44
	2	3	6	516	32.9	50.03
			12	453	40.63	55.67
BK (Opt)	1	2	9	115	12.07	15.89
			18	105	18.73	22.22
	2	3	9	345	26.69	38.15
			18	315	36.81	47.27
RC [SMG15]*	1	3	32	4	N/A†	N/A†
	2	5/10		8		
KS [BG22]	1	2	12	249		
	2	3		747		
Sklansky [BG22]	1	2	12	119		
	2	3		357		
BK [BG22]	1	2	18	74		
	2	3		222		

*Threshold Implementation, not an HPC design.
†Designs are not open-source and area numbers for ASIC designs are not given.

Trivium PRNG is included in the masked AES designs). The comparison with [MCS22] shows that a latency reduction of roughly 19 % for the 32-bit architecture (resp. 28 % for the 128-bit architecture) is achieved by the new implementations. With regard to area, a reduction of about 10 % is achieved at the first order, while the difference for higher orders is below 5 %. Regarding security, the implementations have been formally verified by fullVerif [Cas20].

6.3 Optimized Adder Implementations

Modular additions are often used by cryptographic algorithms such as post-quantum schemes and ARX-based designs. When applying a Boolean masking scheme in such cases to protect against side-channel attacks, the modular addition is usually implemented as a masked binary adder computing the sum of Boolean masked operands. In a third case study, we investigate four different 32-bit modular adder architectures to realize masked binary adders, as such a building block is commonly needed in cryptographic algorithms. Such circuits are interesting study cases since they are larger than the S-boxes, challenging the complexity limits of Compress. These cases are also practically relevant and, despite being more regular than S-box circuits, the complexity of some adders makes it non-obvious how to best implement them.

We study both ripple-carry (RC) and parallel-prefix designs (the Kogge-Stone adder (KS) [KS73], the Sklansky adder [Skl60], and the Brent-Kung adder (BK) [BK78]). In general, an RC architecture performs addition by chaining 1-bit full adders, where each carry bit *ripples* to the next full adder. Every 1-bit full adder takes two summands and a carry-in and computes the respective sum bit and carry-out. Since every carry-out c_i depends on the previous carry-in c_{i-1}, the carry-part of the sum needs to be computed

iteratively, leading to a logic depth of $n-1$ AND gates for a n-bit masked adder. Parallel-prefix adders [BL01, BK82, HC87] aim at reducing the depth by computing the carry-part in parallel using a tree-like structure. To do so, they split the carry generation into generate and propagate functions. A generate function determines if two input bits generate a carry-out, while the propagate function determines if a carry-in will be propagated to the computation of the next carry-out. Both functions can be combined to span larger blocks (groups) of bits, which can be combined again on the next levels, leading to a tree-like structure. KS, Sklansky and BK adders differ in the way of creating these groups, and therefore target different optimization goals.

The results of our case study are given in Table 4. For every adder, we give the security order, the number of shares, the desired latency, the amount of random bits required and the resulting area. We focus on first- and second-order designs (higher-order designs are not more difficult to generate and do not bring significantly different results than low-order ones), and give the design with the lowest possible latency, i.e., the minimal latency required to obtain a secure design. We compare our results with the designs of Schneider *et al.* [SMG15] and Bache *et al.* [BG22], and add the respective data for the latencies analyzed in their work. We put a timeout of 1 h on Compress, i.e., if the optimal solution cannot be found within that time frame, the solver will return the best solution found so far. From our experiments, the 12-cycle KS and Sklansky and the 18-cycle BK adders reached the timeout. Since these clearly correspond to sub-optimal cases, since the adders can be implemented using half the latency and less area, we consider that Compress scales successfully to 32-bit adders. In order to mask even larger adders (e.g., 64-bit adders), further optimizations regarding the handling of the solver might be necessary, such as optimizing the representation of the problem or exploring alternative solvers.

Compared to the RC design proposed in [SMG15], our generated design uses only 2 shares instead of 3 for first-order security, and 3 shares instead of 5/10 for second-order security, although requiring more online randomness. Our KS design requires the same amount of randomness at a latency of 12 compared to [BG22]. However, Compress is able to generate a KS design with less latency (5 cycles), which has a lower overall area consumption than the 12-cycle variant. For both the Sklansky and BK design, Compress also finds variants requiring only half the latency compared to [BG22], resulting in a lower area consumption than the high-latency designs.

7 Related Works

AGEMA The AGEMA tool [KMMS22] was the first tool introduced to perform automated masked hardware circuit generation. AGEMA is a very flexible tool that takes any netlist as an input and masks it, using a Mealy machine representation. That is, contrary to this work and to the other tools discussed in this section, it is not limited to pipeline computations. AGEMA can work in a "naive" mode, where the generated circuit follows the structure of the input circuit, or in "BDD" modes where the logic representation is re-synthesized from a lookup table representation. The naive mode generally performs better when the input circuit is already optimized, e.g., with the Boyar-Peralta AES S-box or the Skinny S-box. The circuits generated by AGEMA can be either in a pipeline structure, or exploit clock gating. The latter enables some area reduction (fewer registers are needed), at the cost of a lower throughput, which is typically not interesting when the logic circuit processes many parallel computations (e.g., an S-box in a block cipher).

AGEMA can further generate circuit using HPC1, HPC2 or GHPC gates (this was later extended to HPC3 in [KM22]). It appears that AGEMA does not perform any latency optimization: every HPC1 or HPC2 instance leads to a latency cost of 2 clock cyles. Further, it does not optimize the scheduling of the computations. Overall, it appears that AGEMA and Compress offer complementary feature sets: AGEMA handles general

circuit masking and interaction between masked an non-masked parts of the circuits, while Compress optimizes masked pipelines.

Handcrafting In [MCS22], Momin et al. demonstrate that handcrafted architectures for masked AES implementations may lead to more efficient circuits than automated masking. This result comes from designing serialized AES architectures that efficiently exploit the high-latency pipeline S-box (the AES of Section 6.2 is based on that architecture). Regarding the S-box design itself, the authors develop an automated masking tool that generates a pipeline, i.e., a tool with a similar purpose to ours. This tool works exclusively with the HPC2 gadget, and exploits its asymmetric latency chararcteristic to minimize the overall latency. This minimization can be performed by a simple greedy algorithm, and the tool performs no further optimization of the pipeline scheduling (every operation is started as soon as its operands are computed).

AGMNC Recently, Wu et al. [WFP+23] introduced the AGMNC tool, which also generates masked pipelines. Their tool is based on two steps. The first step consist in a logic synthesis from a lookup table representation. In the second step, the circuit is implemented into a masked pipeline. This pipeline is then optimized for latency, using the same technique as [MCS22]. A pipeline staging optimization step is also performed. Finally, AGMNC also comes with new masked gadgets. These gadgets, named AND-XOR1 and AND-XOR2 are variants of HPC1 and HPC2 that perform the same operation as our Toffoli gadgets.

The pipeline implementation and optimization steps of AGMNC fullfill the same function as Compress. A detailed comparison of the two tools is difficult given the lack of details in how the optimizations are performed in AGMNC. However, Compress appears to have more features than AGMNC (e.g., selection between multiple kinds of AND gadgets, duplication of gadgets) and it further guarantees an optimial solution, while the algorithm of AGMNC is not described. Regarding the AND-XOR gadgets, these save d registers over an HPC1/HPC2 composition with an XOR gadget. This optimization is a subset of the optimizations enabled by the inner-domain term separation in Compress (Section 4.2).

EasiMask EasiMask [BSG23] is another recent tool for autmating masked circuit generation. Similarly to AGEMA, this is a high-level tool that transfroms a description of a relatively complex operation into a masked circuit. This tool is mainly concerned with high-level architecture decisions, e.g., its user can choose betwen unrolled, round-based or serial architectures. EasiMask comes with a library of masked S-boxes to choose from, and does not generates S-boxes itself. Therefore, the feature sets of EasiMask and Compress do not overlap. In fact, the output of Compress could be integrated to EasiMask's library.

8 Conclusion

Compress optimizes the area of masked pipelines by minimizing the amount of pipelining registers and by the choice of efficient masked gadgets adapted to the latency constraint. Further, the separation of gadgets in smaller components allows the deduplication of some logic, and the new HPC2o and HPC3o gadgets have identical characteristics as HPC2 and HPC3, except for a smaller area footprint and the added Toffoli gate feature. These optimizations, along with the combination of HPC2 and HPC3 gadgets in a single circuit, lead to implementations with minimal latency while improving the state-of-the art area requirements. Our methodology takes into account the amount of randomness required

by the different gadgets: it includes the area cost of generating the required randomness using a PRNG.

Since Compress generates only pipeline circuits, it does not in itself provide a full solution to mask complete cryptographic operations, whose implementations are typically not fully unrolled. However, Compress's output can easily be integrated into a handcrafted design (as done in this work), or into automated workflows. For example, tools that exploit libraries of masked components (e.g., masked S-boxes in EasiMask [BSG23]) could be easily integrated with Compress in a design flow.

A Security Proof of HPC2o and HPC3o

Let us now prove the security of the HPC2o and HPC3o gadgets. We work in the glitch-robust probing model for hardware circuits, which are modeled as directed acyclic graphs whose edges are wires and whose nodes are gates. Gates include logic gates, registers, and input/output gates. In a gadget with d shares, input and output gates are grouped into d-tuples named sharings (while the individual input and output are shares). The index of a share is its (zero-indexed) position in the sharing tuple. We refer to [ISW03, CS21] for a more detailed discussion of this model, and the definition of correctness. In this model, the adversary places glitch-extended probes on the wires of a circuit. When the circuit is evaluated, a probe one a wire leaks to the adversary the values of all the wires in the combinatorial circuit that computes that wire. In other terms, the glitch-extended leakage of a probe on a wire, is made of the value carried by that wire and, if the gate that produces the value on the wire is not a register or an input of the gadget, of the glitch-extended leakage of probes on the input wires of that gate.

Let us now define the security notions.

Definition 1 (Glitch-robust simulatability [BBD+16, FGP+18])**.** Let P be a set of l glitch-extended probes in a gadget G. Let $\mathcal{I}$ be a set of k input shares of G. Let $G_P(x)$ be the random variable denoting the values observed by the adversary when x is the value of the input shares of the gadget, and let $x|_\mathcal{I}$. The set of probes P can be *simulated* with the set of input wires $\mathcal{I}$ if, for any x and x' such that $x|_\mathcal{I} = x|_\mathcal{I}$, the distributions of $G_P(x)$ and $G_P(x')$ are identical.

In particular, if there exists a (randomized) function $\mathcal{S}$ (named the simulator) such that the distribution of $\mathcal{S}(x|_\mathcal{I})$ (where $x|_\mathcal{I}$ denotes the values of the input shares x that belong to $\mathcal{I}$) and $G_P(x)$ are equal for any x, then the glitch-robust probes P can be simulated by the input shares $\mathcal{I}$ [BBP+16].

Definition 2 (Probe Isolating Non-Interference (PINI) [CS20])**.** A d-shares gadget G is glitch-robust t-probe-isolating non-interferent (t-PINI) if, for any set $A \subseteq \{0, \ldots, d-1\}$ and any set of glitch-extended probes P such that $|A| + |P| \leq t$, there exists a set $B \subseteq \{0, \ldots, d-1\}$ with $|B| \leq |P|$ such that the glitch-extended probes P and glitch-extended probes on all output shares of G with index in A can be simulated by the inputs of G with index in $A \cup B$.

The security proof for HPC2o is very similar to the proof for HPC2 [CGLS21].

Proposition 1. *The HPC2o gadget (Algorithm 6) is glitch-robust PINI.*

Proof. Let us build a glitch-robust PINI simulator. We assume wlog that only the input wires of registers and the outputs of the gadgets are probed, (since the other extended probes are less powerful). Namely, these probes can be z_i, $u_{ij} := \overline{x_i} \wedge r_{ij}$, $v_{ij} := y_j \oplus r_{ij}$ and $x_i \wedge v_{ij}$. For $j = j_i$, we instead have $u_{ij} := w_i \oplus (x_i \wedge y_i) \oplus \overline{x_i} \wedge r_{ij}$. Given a set of probes adversarial extended probes P and probed output shares A, the set of required

input shares X is computed as follows: for each probed z_i, add i to X. Then, for each $i \neq j$ pair, if two out of u_{ij}, v_{ij} and $x_i v_{ij}$ are probed, or if i of j belongs to X: add i and j to X. Otherwise, if u_{ij} or $x_i \wedge v_{ij}$ is probed, add i to X, and if v_{ij} is probed, add j to X. The set B is computed as $X \setminus A$.

We observe that the set B satisfies the PINI definition: $|B| \leq |P|$ by construction. All the values to be simulated that depend only on input shares with index in X and on randomness are computed as specified by Algorithm 6 (the required randomness is generated by the simulator). The allows to simulate all the extended probes on u_{ij} and v_{ij}, by construction of X. Then, for all remaining extended probes (z_i (for which $i \in A$) and $x_i v_{ij}$), we observe that $i \in X$. They can therefore be computed as it is done by the gadget, except when the simulation of $v_{ij} = y_j \oplus r_{ij}$ is needed and $j \notin X$. In this case, the simulator simulates v_{ij} by sampling a fresh random r'_{ij} (we say that the simulator *cheats* for ij).

Let us show that this algorithm is indistinguishable from the true gadget. The behavior of the simulator is identical to the behavior of the gadget, except when it cheats for ij. We therefore only need to prove that if the simulator cheats for ij, then r_{ij} is not observed in the set of probes, except through v_{ij}, therefore v_{ij} is indistinguishable from a fresh r'_{ij} and simulation is correct.

The simulator cheats for ij only if $j \notin X$ and a value depending on v_{ij} is probed. The first condition implies that none of z_j, u_{ji}, $x_j v_{ij}$ and v_{ij} are probed, and at most one of z_i, $x_i v_{ij}$, u_{ij} and v_{ji} can be probed. The second condition implies that z_i, or $x_i v_{ij}$ is probed (v_{ij} cannot be probed due to the previous observation). Therefore, the only values depending on r_{ij} that can be probed are z_i or $x_i v_{ij}$, and exactly one of those is probed. If $x_i v_{ij}$ is probed, then the simulation is correct: the extended probe expands to $\{x_i, v_{ij}, x_i v_{ij}\}$, which are the only observations depending on r_{ij}. If z_i is probed, then observations depending on r_{ij} are u_{ij} and $x_i v_{ij}$, and functions of these values. If $x_i = 0$, then $x_i \wedge v_{ij} = 0$ does not depend on r_{ij}, which is thus only observed through u_{ij}, hence the simulation is correct. Otherwise, we have $\overline{x_i} = 0$, which implies that $u_{ij} = 0$ for $j \neq j_i$ or $u_{ij} = w_i \oplus (x_i \wedge y_i)$ for $j = j_i$, thus r_{ij} is only observed through v_{ij}, which is correctly simulated as a fresh random.[7] □

Proposition 2. *The HPC3o gadget (Algorithm 7) is glitch-robust PINI.*

Proof. We the proof is similar to the proof of Proposition 1. We again consider only the probes z_i, $u_{ij} := (x_i \wedge r_{ij}) \oplus r'_{ij}$ and $v_{ij} := y_j \oplus r_{ij}$. For $j = j_i$, we instead have $u_{ij} := w_i \oplus (x_i \wedge r_{ij}) \oplus r'_{ij}$. Given a set of probes adversarial extended probes P and probed output shares A, the set of required input shares X is computed in the same way as in the proof of Proposition 1 (except that $x_i \wedge v_{ij}$ does not exist as a possible probe). The set B is again computed as $X \setminus A$, and satisfies the PINI definition.

Similarly to HPC2o, the simulation follows Algorithm 7, except when the simulation of $v_{ij} = y_j \oplus r_{ij}$ is required and $j \notin X$. In this case, the simulator cheats for ij, by simulating both v_{ij} and u_{ij} as fresh randoms. Since cheating on ij occurs only when simulation of v_{ij} is needed, this means that z_i is probed, hence $i \in X$. Further, since $j \notin X$, there is no other probe than z_i through which the adversary may observe r_{ij} or r'_{ij}. Therefore, the value u_{ij} appears as a uniform random to the adversary since r'_{ij} is not observed otherwise. As a consequece, r_{ij} is not observed except trough the value v_{ij}, which appears as a fresh random.[8] □

[7] This argument does not work in larger fields, in which the HPC2o multiplication gadget is therefore not glitch-robust PINI.

[8] Let us remark that, unlike the proof for HPC2o, this proof is not specific to $\mathbb{F}_2$.

References

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *CCS*, pages 116–129. ACM, 2016.

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

[BG22] Florian Bache and Tim Güneysu. Boolean masking for arithmetic additions at arbitrary order in hardware. *Applied Sciences*, 12(5), 2022.

[BK78] Richard P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *J. ACM*, 25(4):581–595, 1978.

[BK82] Richard P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Computers*, 31(3):260–264, 1982.

[BL01] Andrew Beaumont-Smith and Cheng-Chew Lim. Parallel prefix adder design. In *IEEE Symposium on Computer Arithmetic*, page 218. IEEE Computer Society, 2001.

[BP12] Joan Boyar and René Peralta. A small depth-16 circuit for the AES s-box. In *SEC*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2012.

[BSG23] Fabian Buschkowski, Pascal Sasdrich, and Tim Güneysu. Easimask-towards efficient, automated, and secure implementation of masking in hardware. In *DATE*, pages 1–6. IEEE, 2023.

[Cas20] Gaëtan Cassiers. FullVerif, 2020.

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

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

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

[CMM+23a] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. Randomness generation for secure hardware masking - unrolled trivium to the rescue. *IACR Cryptol. ePrint Arch.*, page 1134, 2023.

[CMM+23b] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, and François-Xavier Standaert. Prime-field masking in hardware and its soundness against low-noise SCA attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(2):482–518, 2023.

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

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

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

[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.

[Gun19] Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

[HC87] Tack-Don Han and David A. Carlson. Fast area-efficient VLSI adders. In *IEEE Symposium on Computer Arithmetic*, pages 49–56. IEEE Computer Society, 1987.

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

[KM22] David Knichel and Amir Moradi. Low-latency hardware private circuits. In *CCS*, pages 1799–1812. ACM, 2022.

[KMMS22] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. Automated generation of masked hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):589–629, 2022.

[KS73] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers*, 22(8):786–793, 1973.

[KSM22] David Knichel, Pascal Sasdrich, and Amir Moradi. Generic hardware private circuits towards automated generation of composable secure gadgets. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):323–344, 2022.

[MCS22] Charles Momin, Gaëtan Cassiers, and François-Xavier Standaert. Handcrafting: Improving automated masking in hardware with manual optimizations. In *COSADE*, volume 13211 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2022.

[MKSM22] Nicolai Müller, David Knichel, Pascal Sasdrich, and Amir Moradi. Transitional leakage in theory and practice unveiling security flaws in masked circuits. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):266–288, 2022.

[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.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In *CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptol.*, 24(2):292–321, 2011.

[PF] Laurent Perron and Vincent Furnon. Or-tools.

[Skl60] Jack Sklansky. Conditional-sum addition logic. *IRE Trans. Electron. Comput.*, 9(2):226–231, 1960.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic addition over boolean masking - towards first- and second-order resistance in hardware. In *ACNS*, volume 9092 of *Lecture Notes in Computer Science*, pages 559–578. Springer, 2015.

[VCS22] Corentin Verhamme, Gaëtan Cassiers, and François-Xavier Standaert. Analyzing the leakage resistance of the nist's lightweight crypto competition's finalists. In *CARDIS*, volume 13820 of *Lecture Notes in Computer Science*, pages 290–308. Springer, 2022.

[WFP+23] Lixuan Wu, Yanhong Fan, Bart Preneel, Weijia Wang, and Meiqin Wang. Automated generation of masked nonlinear components: From lookup tables to private circuits. Cryptology ePrint Archive, Paper 2023/831, 2023. https://eprint.iacr.org/2023/831.