IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 3, pp. 500–529. DOI:10.46586/tches.v2024.i3.500-529

Compress: Generate Small and Fast Masked Pipelined Circuits

Gaëtan Cassiers[1*], Barbara Gigerl[2], Stefan Mangard[2], Charles Momin[1] and Rishub Nagpal[2,3]

[1] UCLouvain, Louvain-la-Neuve, Belgium, firstname.lastname@uclouvain.be
[2] Graz University of Technology, Graz, Austria, firstname.lastname@iaik.tugraz.at
[3] Silicon Austria Labs, TU-Graz SAL DES Lab, Graz, Austria

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 pipeline 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 state-of-the-art latency. For example, a masked AES based on an S-box generated by Compress reduces latency by 19 % and area by 27 % over a state-of-the-art implementation, or, for the same latency, reduces area by 45 %.

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 over any set of $d-1$ shares x_i. A common example is Boolean masking, where variables belong to $\mathbb{F}_2$ and the group operation is the exclusive or ($\oplus$). The computations to mask are typically decomposed in elementary operations (e.g., simple logic gates) which are then replaced by gadgets: small circuits that securely perform computations over shared data.

Masking a circuit in a secure way is a challenging task. Physical defaults such as glitches and transitions can break the independence assumptions required for a secure masked implementation [MPG05, NRS11]. Furthermore, the security of small gadgets may not directly extend to their combination, leading to so-called composition issues [CPRR13, BBD+16]. Physical defaults and composition issues can also arise in a

* Work performed in part while associated with Graz University of Technology, Graz, Austria.

Received: 2024-01-15 Accepted: 2024-03-15 Published: 2024-07-18

combined way [FGP+18, MMSS19, MKSM22]. One possible solution to these challenges is using the hardware private circuits (HPC) masking scheme [CGLS21, CS21], which is composable even in the presence of glitches and transitions. With the HPC scheme, linear and affine operations can be implemented by simple sharewise gadgets, while non-linear gadgets are more complex. Several multiplication (or AND) gadgets have been proposed, including the two-cycle HPC1 and HPC2 [CGLS21], and the single-cycle HPC3 [KM22].

Even though trivially composable masking schemes such as HPC simplify the security analysis of a masked circuit, transforming an unprotected design into a securely masked one remains a challenging task. Indeed, the HPC multiplication gadgets contain sequential logic, hence masking a non-linear combinational circuit leads to a sequential circuit, requiring adjustments to the synchronization logic to take the added latency into account. AGEMA [KMMS22] is an automated masked circuit generation tool that addresses this problem. However, the generated designs are often suboptimal, as pointed out by Momin et al. [MCS22], who introduce handcrafted designs for a masked AES implementation that have better performance than the circuits generated by AGEMA. While their performance gains come from carefully crafted high-level architectures and from an improved Sbox design. The latter is automatically generated by a tool that achieves better performances than AGEMA, but is much more restricted since it only handles pipelined circuits. This tool, as well as the recently-introduced AGMNC [WFP+23] (another tool that generates masked pipelined circuits), do not generate optimal circuits.

Although existing CAD tools implement advanced circuit optimizations, using them to optimize masked designs is non-trivial, since they do not necessarily preserve the required security properties. For example, arbitrary re-timing of registers might allow the propagation of insecure glitches, and logic sharing, along with boundary optimizations may break Threshold Implementation's [NRR06] non-completeness property [CBG+17, ABP+18, ZSS+21, CCGB21, MM22, CMM+23b]. In practice, designers therefore generally take measures to disable or prevent critical CAD tools' optimizations when synthesizing masked circuits.

Contributions In this work, we introduce Compress[1], a tool to generate area-optimized masked pipelined circuits. We make Compress publicly available on Github[2]. Compress takes as an input a Boolean circuit describing the circuit to be implemented, and generates a masked netlist representing the circuit with a latency chosen by the user. Thanks to its optimizations, it generates more efficient circuits than state-of-the-art tools, while keeping security-critical structures for masking. The tool supports any-order Boolean masking, although the techniques it uses also apply to other kinds of masking.

We focus on the generation of pipelined circuits, i.e., 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 HPC gadgets (which are themselves small pipelined circuits) together with the help of additional registers to ensure proper synchronization of the pipeline stages. Compared to the alternatives such as clock gating [KMMS22], the big advantage of pipelined circuits is their simplicity (e.g., there is no control logic) and high throughput (they perform one evaluation per clock cycle). This makes them good candidates for the implementation of subcomponents in cryptographic algorithms, where the high throughput enables serialized implementation strategies, and a single pipelined circuit is used to perform many parallel computations sequentially (e.g. S-boxes). Pipelined circuits can then be integrated in circuits with more complex architectures, either by hand [MCS22], or automatically (e.g., with EasiMask [BSG23]).

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

[2] Available at https://github.com/cassiersg/compress, with all our scripts at https://github.com/cassiersg/compress_artifact.

While most previous works focus rather on finding efficient Boolean circuit representations of functions [BP12, CGLS21] or designing new gadgets with reduced randomness usage or lower latency, these works generally leave out "low-hanging fruit" optimizations in the composition of gadgets and inside the gadgets themselves. Most of our optimizations therefore focus on eliminating and re-timing registers in masked pipelines, which sometimes represent more than 70 % of the area [MCS22]. This is achieved in two steps. First, at the gadget composition level, we optimize the staging of computations, i.e., we assign every gadget in the composition to its pipeline stage(s). If needed, we duplicate gadgets, for example, when a gadget is small and its output is used in multiple pipeline stages, it might be more efficient to instantiate the gadget multiple times, instead of having pipeline registers to forward its output to all later pipeline stages where it is used. Second, we tackle the issue of redundant pipeline registers, that is, multiple registers that store the same value. This issue arises when gadgets store their inputs in registers, which may be redundant with pipeline registers inside other gadgets, or registers added for synchronization between gadgets.

Furthermore, Compress introduces a variety of other optimizations to reduce the area of masked designs. At the level of individual HPC2 and HPC3 gadgets, we employ an optimized handling of the so-called *inner-domain* terms (i.e., term of the form $x_i \wedge y_i$, for input sharings $\boldsymbol{x} = (x_0, \ldots, x_{d-1})$ and $\boldsymbol{y} = (y_0, \ldots, y_{d-1})$). This optimization also leads to new variants of HPC2 and HPC3 that implement the Toffoli gate in a more efficient way than the AND-XOR gadgets of [WFP+23], and to the extension of HPC3 to arbitrary fields, enabling its use in more contexts (such as efficient implementations of the Canright AES Sbox [Can05]). Finally, we show for the first time that saving area is possible through combining HPC2 and HPC3 gadgets, rather than exclusively using one or the other. In general, HPC3 has a lower latency and lower area than HPC2, but it requires double the amount of randomness, leading to a larger total area (i.e., including the area of the randomness generation circuit). Small and low-latency circuits can be obtained by sticking to HPC3 when the operands are both on the critical latency path, but HPC2 elsewhere (thanks to its 1-2 cycle asymmetric latency).

Combining all these optimizations leads to significant area reductions, and makes low-latency circuits (which are generally larger) more practically-relevant. In particular, we design a pipelined AES S-box with up to 50 % latency and 33 % area gain over the smallest HPC2 implementation in the state of the art, and 45 % area gain over the state of the art HPC3 implementation (same latency). We further adapt the 32-bit datapath 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 27 %. Our round-based AES implementations also exhibit similar improvements over the state of the art in area and/or latency. Compress is not limited to the design of masked S-boxes. As an example, we apply it to multiple architectures of 32-bit adders.

Outline Section 2 introduces the HPC masking scheme and its use to build pipelined 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).

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

2 Background

In this section, we first introduce the glitch- and transition-robust probing model for analyzing the security of masking schemes in hardware. We then present the HPC masking schemes and its various multiplication gadgets. Finally, we discuss the issue of synchronization in masked circuits.

2.1 Robust Probing Model

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

2.2 Hardware Private Circuit

HPC is an arbitrary-order masking scheme with $t = d-1$ robust 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. HPC is based on the

Table 1: Performance characteristics of multiplication/AND gadgets (randomness and area given for $\mathbb{F}_2$ with the NanGate45 PDK).

Gadget	Latency `ina`	Latency `inb`	d	Random bits	Area w/o PRNG (GE)	Area w/ PRNG (GE)	Group
GHPC [KSM22]	1	2	2	1	95.3	134.7	$\mathbb{F}_2$
GHPC_{LL} [KSM22]	1	1	2	4	86.7	244.1	$\mathbb{F}_2$
HPC1 [CGLS21]	1	2	2	2	50.7	129.4	$\mathbb{F}_n$
			3	5	127.0	323.8	
			4	10	217.3	610.9	
			5	15	325.0	915.4	
HPC2 [CGLS21]	1	2	2	1	82.3	121.7	$\mathbb{F}_2$
			3	3	209.0	327.1	
			4	6	392.7	628.8	
			5	10	633.3	1026.9	
HPC2o (new)	1	2	2	1	55.0	94.4	$\mathbb{F}_2$
			3	3	168.3	286.4	
			4	6	338.7	574.8	
			5	10	566.0	959.6	
HPC3 [KM22]	1	1	2	2	69.3	148.1	$\mathbb{F}_2$
			3	6	165.0	401.2	
			4	12	301.3	773.7	
			5	20	478.3	1265.5	
HPC3o (new)	1	1	2	2	38.7	117.4	$\mathbb{F}_n$
			3	6	119.0	355.2	
			4	12	240.0	712.3	
			5	20	401.7	1188.9	

composable notion of glitch-robust probe-isolating non-interference (PINI) [CS20], which ensures that gadgets are trivially composable in the presence of glitches: if all gadgets are glitch-robust PINI, the masked circuit is glitch-robust PINI as well. Further, HPC has also been proven secure against glitch+transition leakage under some additional conditions on its structure, which are trivially satisfied in many cases, such as when implementing a substitution-permutation network (SPN) with at least 2 clock cycles per round [CS21].

For linear gates (e.g., XOR), there exists simple sharewise gadgets (e.g., Sharewise-XOR shown in Algorithm 1) which are glitch-robust PINI. Similarly, for affine gates, we can use a sharewise gadget where the affine map is applied to one of the shares and the associated linear map is applied to the other shares (e.g., a NOT gadget may simply apply the NOT on the first share). Non-linear gates are more complex, and the design of multiplication/AND gadgets is an active research area, with state-of-the-art gadgets listed in Table 1. The gadgets are characterized by their latency (number of cycles between providing each of the input sharings and generating the output), the number of shares supported, their randomness usage and area requirement.

Given the high randomness requirements of the HPC gadgets, masked circuits using them generally use dedicated PRNGs. Assuming that such a PRNG is used allows us to simplify the gadget comparison: we estimate the area of the PRNG needed to provide enough randomness to run the gadget continuously, and integrate it in the area of the gadget. We therefore compare gadgets based on their area with randomness generation, and do not focus on the randomness usage itself. For the PRNG area of generating one bit of randomness per cycle, we take the area of an unrolled Trivium (as suggested by [CMM+23a]) divided by the unrolling factor (we take an unrolling factor of 512), amounting to 39.4 GE/bit with NanGate45.

In Table 1, it appears that HPC2 [CGLS21] and HPC3 [KM22] have respectively lower (PRNG-included) area than the Generic Hardware Private Circuit (GHPC) and its low latency variant (GHPC_{LL}) [KSM22], for the same latency. HPC1 is similarly less performant than HPC2. We therefore mainly focus on the HPC2 and HPC3 gadgets in

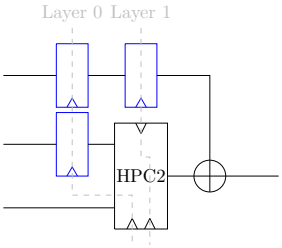


Figure 1: Example of a masked pipelined circuit as a composition of gadgets and masked registers.

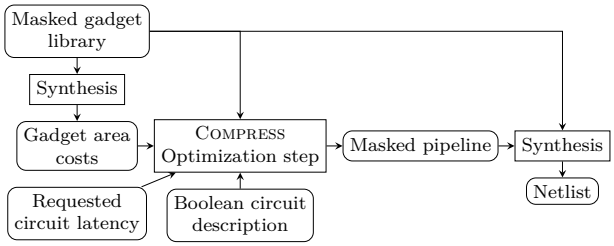


Figure 2: Compress flow. Rectangles denote flow steps, rounded corners denote inputs, outputs and intermediate flow artifacts.

this paper, and use HPC1 only when multiplication in a larger field (i.e., not $\mathbb{F}_2$) is used. The HPC2 and HPC3 gadgets are described in Algorithm 2 and Algorithm 3, where $\mathsf{R}(\cdot)$ denotes a glitch-stopping register (i.e., one that is needed for security) and $\mathsf{PR}(\cdot)$ denotes a pipeline register (i.e., one that is only needed for turning the gadget into a pipeline).

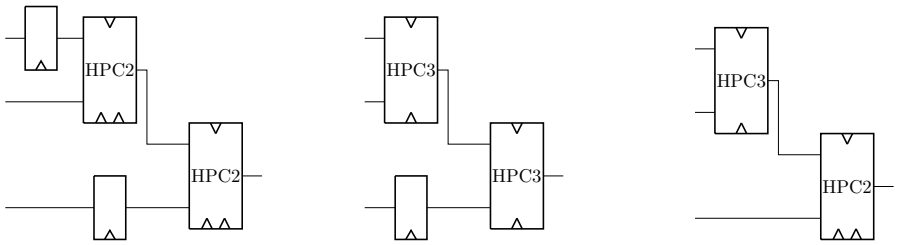
2.3 Synchronization in Masked Hardware Circuits

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 is to exploit pipelining: the synchronization is achieved through the addition of registers instead of clock gating, as shown in Figure 1. 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 round-based (parallel) implementation of an SPN, the masked S-box can be instantiated multiple times such that that all S-boxes can be evaluated in parallel (each instance is evaluated only once per round). By contrast, a serialized implementation may instantiate the masked S-box only once (or a few times) and evaluate it multiple times in order to evaluate a round. Serializing the architecture reduces the area cost, and it combines well with pipelining: the high throughput of the pipeline minimizes the latency overhead of serialization. As a result, masking with pipelining is a technique that can achieve better latency/area trade-offs than clock gating, but may require handcrafted designs [MCS22].

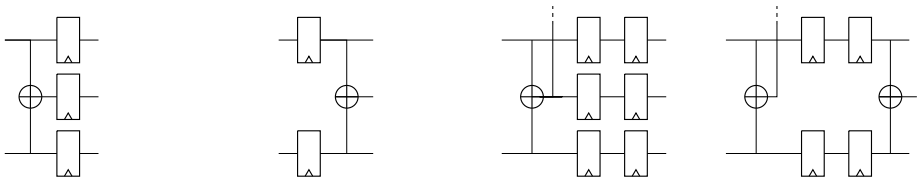
3 Generic Optimization of Masked Pipelined Circuits

Compress takes as input the Boolean circuit to mask and outputs a netlist that implements the circuit as a masked HPC pipeline. As shown in Figure 2, Compress also takes as input a masked gadget library, whose areas (including PRNG cost) are used to parameterize an optimization goal. The latency of the generated circuit is also a parameter of Compress,



(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.



Naive Compress Naive Compress

(a) Scheduling of computations across pipeline stages. **(b)** Gadget replication.

Figure 4: Examples of Compress optimization of pipeline registers (registers and $\oplus$ represent sharewise gadgets, wires represent sharings).

it must be at least equal to the AND depth of the input circuit (otherwise the circuit cannot be implemented using the HPC2 and HPC3 gadgets).

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 pipeline registers that forward the computed data across register stages. Optimized scheduling reduces the number of pipeline registers to be instantiated, thereby reducing area as shown in Figure 4a. 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 pipeline registers (provided that the operands of the gadget are available at the corresponding pipeline stages). For example, in Figure 4b, 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).

The core part of Compress consists representing the masked circuit generation as a constraint optimization problem. We then use OR-tools [PF] to solve this problem, and the solution is then translated into a verilog netlist. We next give a high-level description of this optimization problem (the complete algorithm is given in Appendix B).

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 instantiates a

"valid" Boolean variable v_s^w, 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 "compute" Boolean variable c_s^w that is true iff there is a gadget that outputs w at the stage s (for now, we consider gadgets that output a single sharing, multi-output gadgets are handled in Appendix B). The "pipeline" variable r_s^w indicates the presence of a pipeline 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 following constraints, which define the "valid" variable as a function of "compute" and "pipeline":

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

Let us next model the instantiation of gadgets, which will determine the value of the "compute" variables. Each value w is computed by a logic gate (e.g. XOR, AND, ...) that can be implemented by one or multiple gadget types (e.g., an XOR gate can only be implemented by an XOR gadget, while an AND gate can be implemented by HPC2 or HPC3). For each value w, each stage s, and each gadget type t the "gadget" Boolean variable $g_s^{t,w}$ indicates if a gadget of type t is instantiated to output w in stage s. In order to ensure correctness of the generated circuit, $g_s^{t,w}$ is set to false (i.e., the gadget is not instantiated) when (i) the gadget implements a different gate than the one computing w, or (ii) when the corresponding gadget would take an input before stage 0 (e.g., for any w, $g_0^{\mathsf{HPC2},w} = g_1^{\mathsf{HPC2},w} = \bot$, i.e., they are set to false). Next, the following constraint requires the inputs of an instantiated gadget to be valid: we let

$$g_s^{t,w} \Rightarrow \bigwedge_{w' \in \mathsf{op}(w)} v_{s-\mathsf{lat}(t,w,w')}^{w'},$$

where $\mathsf{op}(w)$ is the set of operands of the logic gate that computes w, while $\mathsf{lat}(t, w, w')$ is, for the gadget t, the latency of the input sharing w' relative to the output sharing w (i.e., the difference in pipeline stages between the input and the output).

Gadget instantiations determine the value of the "compute" variables:

$$c_s^w = \bigvee_{t \in G} g_s^{t,w}$$

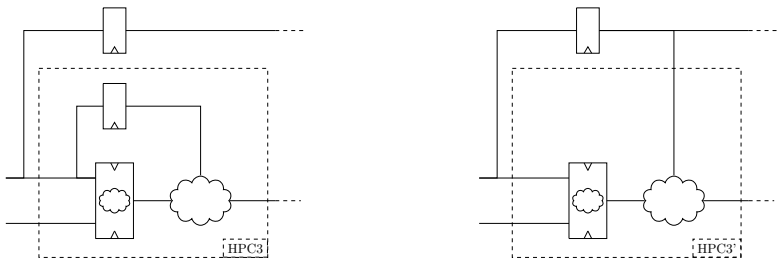
where G is the set of all gadget types[3]. As an exception, the inputs of the circuit are not computed by gadgets, they are provided at stage 0, which we model as follows: for all input wires w, $c_0^w = \top$ and $c_s^w = \bot$ for all $s > 0$.

The last constraint is for outputs: for all output wires w, $v_L^w = \top$. Together, the constraints ensure that any admissible solution to the problem corresponds to a correct masked circuit implementation[4].

Next, the objective of the optimization problem is the minimization of the area used by the masked circuit. Therefore, Compress takes as an input the area cost of each gadget type, including a "pipeline register" gadget. The total cost is then defined as the sum of the areas of the instantiated gadgets (as determined by the $g_s^{t,w}$ and r_s^w variables). We take into account the cost of randomness generation for the masked gadgets as follows. We assume that a PRNG is instantiated along with the masked circuit, and that it should provide enough randomness to run the pipeline continuously: each randomness input of a gadget

[3] In order to provide most optimization opportunities (including the optimizations performed by [MCS22]) for gadgets that have functionally identical input sharings with different latencies (e.g., HPC2), we have multiple variants of these gadgets in G that are all equivalent, up to a re-ordering of functionally identical inputs.

[4] Except that it allows useless and nonsensical pipeline register instantiations, but these never occur in practice thanks to the optimization.



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

Figure 5: Illustration of register de-duplication thanks to the separation of a pipeline register out of the HPC3 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. As a result, the PRNG cost is included into the area optimization function as an increase of the area of each gadget by the area needed to generate the randomness it uses.

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 pipeline registers for the x_i shares (see Algorithm 3). If two HPC3 gadgets have the same input sharing $\boldsymbol{x}$, then these registers will be duplicated. The same can also happen between a register inside a gadget and a pipeline register outside any gadget. Our approach to avoid such inefficiencies is to decompose gadgets into multiple parts, which eliminates pipeline 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 Pipeline Registers

As a first step, we handle the de-duplication of pipeline registers on input sharings, which is illustrated in Figure 5. The de-duplication is implemented in Compress by putting these pipeline registers outside of the gadgets. If a gadget contains registers $\mathsf{PR}(x_i)$ (where $\boldsymbol{x}$ is an input sharing, see e.g. HPC3), these can be removed from the gadget and these values replaced by the shares x'_i of a new input sharing. The new input sharing $\boldsymbol{x'}$ is encoded as taking the same value of $\boldsymbol{x}$, but being used one clock cycle later than $\boldsymbol{x}$. This leads Compress to instantiate a pipeline register to generate $\boldsymbol{x'}$, which may be used as an input for other gadgets. 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).

The new gadgets bring new additional constraints in Compress. Indeed, a naive implementation of the techniques presented above would require that the values represented by the sharings $\boldsymbol{x}$ and $\boldsymbol{x'}$ are the same, while, for the circuit to be correct, the values of the individual shares must be equal. These conditions are not equivalent due to the

Algorithm 4 HPC3-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**
    $r_{ij} \xleftarrow{\$} \mathbb{F}_2$; $r_{ji} \leftarrow r_{ij}$
    $r'_{ij} \xleftarrow{\$} \mathbb{F}_2$; $r'_{ji} \leftarrow r'_{ij}$
for $i = 0$ to $d-1$ **do**
  for $j = 0$ to $d-1, j \neq i$ **do**
    $p_{ij} \leftarrow \mathsf{R}\left((\overline{x_i} \wedge \mathsf{PR}(r_{ij})) \oplus r'_{ij}\right) \oplus \mathsf{R}(x_i \wedge \mathsf{R}(y_j \oplus r_{ij}))$
  $z_i \leftarrow \bigoplus_{j=0, j\neq i}^{d-1} p_{ij}$

Algorithm 5 HPC3 AND decomposed in gadgets.

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

$\boldsymbol{a} \leftarrow$ HPC3-cross $(\boldsymbol{x}, \boldsymbol{y})$
$\boldsymbol{b} \leftarrow$ Sharewise-AND $(\boldsymbol{x}, \boldsymbol{y})$
$\boldsymbol{z} \leftarrow$ Sharewise-XOR $(\boldsymbol{a}, \boldsymbol{b})$

duplication of gadgets discussed in Section 3, when a non-deterministic gadget (i.e., one whose output sharing depends on some randomness) gets duplicated[5]. To ensure that such a case does not happen, we add constraints to the optimization problem of Compress, enforcing all sharings of the same value to be identical[6]. Concretely, this means that a sharing cannot be computed by gadgets of different types: for all wires w,

$$\mathsf{AtMostOne}\left(\left(\mathsf{Any}\left(\left(g_s^{t,w}\right)_{s=0,\ldots,L}\right)\right)_{t\in G}\right) = \top, \tag{1}$$

where $\mathsf{Any}(\cdot)$ is true if any element of its input tuple is true, and $\mathsf{AtMostOne}(\cdot)$ is true if at most one element of its input tuple is true. Further, a gadget that uses randomness cannot be duplicated: for all wires w and all gadget types t that use randomness,

$$\mathsf{AtMostOne}\left(\left(g_s^{t,w}\right)_{s=0,\ldots,L}\right) = \top. \tag{2}$$

Regarding security, the only change we apply to the circuit is the de-duplication of registers that store identical values, which does not change the set of probes in the probing model (including glitches and transitions).

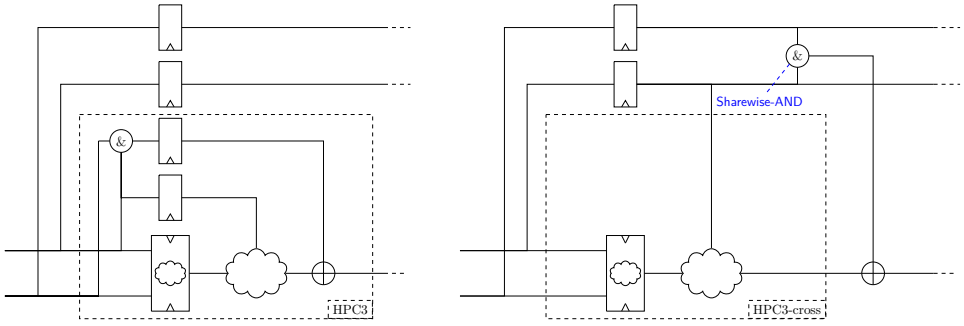
4.2 Separate Inner-domain Terms

While some pipeline registers can be optimized by separating them out of the gadgets, the above optimization does not apply to all pipeline registers. In this section, we look at the pipeline 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. Again, our goal is to move these registers outside of the gadgets, such the Compress can optimize them.

We achieve this by splitting the AND gadgets into a part that computes the inner-domain terms (a sharewise AND gadget), a part that computes the other terms (HPC2/3-cross), and a gadget that XORs their outputs (sharewise), as illustrated in Figure 6. The decomposition of HPC3 is given in Algorithm 4 and Algorithm 5, and the decomposition of HPC2 follows the same pattern. The three resulting gadgets perform exactly the same computation as the original gadget, except for the now-unspecified pipeline registers.

[5] The issue can also appear if a logic gate can be implemented by multiple types of gadgets (with or without randomness), and the "duplication" instantiates two different types.

[6] These constraints are stronger than strictly necessary since we could apply them (recursively) only to inputs of gadgets with pipeline register separation. This is not an issue in practice, since the only logic gate for which we have multiple gadget types is the AND gate, and these gadgets (which are also the only ones using randomness) have a large area. Therefore, it is unlikely that duplicating these gadgets is more efficient than adding pipeline registers.



(a) Without separation **(b)** With full separation

Figure 6: Illustration of register de-duplication thanks to register separation and inner-domain terms separation.

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
      $p_{ij} \leftarrow \mathsf{R}\left((x_i \wedge r_{ij}) \oplus r'_{ij}\right) \oplus (\mathsf{PR}(x_i) \wedge \mathsf{R}(y_j \oplus r_{ij}))$
  $z_i \leftarrow \bigoplus_{j=0, j\neq i}^{d-1} p_{ij}$

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 (the original security proofs of HPC2 and HPC3 did not assume the presence of these registers, and adding registers can only make the adversary weaker).

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 pipeline registers through optimizations inside the AND gadgets.

Inner-domain term optimization Instead of separating the inner-domain terms in HPC2 and HPC3, we propose to merge these terms with cross-domain terms. For HPC2, we modify Algorithm 2 as follows. For every $i = 0, \ldots, d-1$, we select an arbitrary $j_i \neq i$ (e.g., $j_i = 0$ for all $i \neq 0$, and $j_0 = 1$). Then, we replace the computation $\mathsf{R}(\overline{x_i} \wedge \mathsf{PR}(r_{ij_i}))$ with $\mathsf{R}((x_i \wedge \mathsf{PR}(y_i)) \oplus (\overline{x_i} \wedge \mathsf{PR}(r_{ij_i})))$ (see Algorithm 6), which integrates the term $x_i \wedge y_i$ into the term p_{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 (Algorithm 3), we first apply a simplification to the gadget, by remarking that the NOT gate in the computation $\mathsf{R}\left(\left(\overline{x_i} \wedge r_{ij}\right) \oplus r'_{ij}\right) \oplus \mathsf{PR}\left(x_i\right) \wedge \mathsf{R}\left(y_j \oplus r_{ij}\right)$ can be removed. Namely, replacing this computation with $\mathsf{R}\left(\left(x_i \wedge r_{ij}\right) \oplus r'_{ij}\right) \oplus \mathsf{PR}\left(x_i\right) \wedge \mathsf{R}\left(y_j \oplus r_{ij}\right)$ still leads to a correct gadget: while the original computation results in the value $\left(x_i \wedge y_j\right) \oplus r'_{ij} \oplus r_{ij}$ and the new one gives $\left(x_i \wedge y_j\right) \oplus r'_{ij}$, both lead to a correct gadget (i.e., the output is a sharing of $x \wedge y$). Regarding the security, the core observation is that, similarly to the original gadget, the new computation $\mathsf{R}\left(\left(x_i \wedge r_{ij}\right) \oplus r'_{ij}\right)$ has 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[7], it becomes useful when optimizing the inner-domain terms. Indeed, we can apply a similar optimization to HPC2 by turning $\mathsf{R}\left(\left(x_i \wedge r_{ij_i}\right) \oplus r'_{ij_i}\right)$ into $\mathsf{R}\left(\left(x_i \wedge \left(y_i \oplus r_{ij_i}\right)\right) \oplus r'_{ij_i}\right)$, which additionally saves one AND gate (see Algorithm 7).

Finally, we remark that, where $x_i y_i$ is computed, a share w_i of a third input w can be XORed without breaking the PINI property (intuitively, this is because the set of manipulated "share domains" is not changed). This addition turns the HPC2o and HPC3o gadgets into Toffoli gadgets, saving pipeline registers for w_i. The final gadgets are given in Algorithm 6 and Algorithm 7, their performance characteristics are listed in Table 1 and their formal security proofs are given in Appendix A.[8] The new gadgets have the same functionality as the AND-XOR gadgets of [WFP+23], but are more efficient.[9]

Gate area optimization We introduce another optimization in the HPC2 gadget. In the computation of p_{ij} for $j \neq j_i$ and $j \neq i$ (i.e., for the cross-domain terms where the above optimization is not applied), $\mathsf{R}\left(\overline{x_i} \wedge \mathsf{PR}\left(r_{ij}\right)\right)$ and $\mathsf{R}\left(x_i \wedge \mathsf{R}\left(y_j \oplus r_{ij}\right)\right)$ 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 an 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 gate 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 do not 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,

[7] 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.

[8] As a sanity-check (and also to avoid implementation bugs), our implementations of HPC2o and HPC3o have been verified with SILVER [KSM20] for $d = 2, 3$.

[9] Actually the AND-XOR gadgets share the registers between the w_i and $x_i y_i$ terms. However, they do not share these registers with cross-domain terms, as is done in HPC2o and HPC3o.

Algorithm 8 Identification of Toffoli gates in Boolean circuit

Input: A Boolean circuit, g a AND gate in the circuit.
Output: (L_g, R_g) where L_g is a set of XOR-operands for a Toffoli gate instantiation and R_g .

```
for all AND gates g in the circuit do
  Let z be the output of g.
  o ← z
  while o is the operand of only one operation do
    Let o be the output of that operation.
  R_g ← o                                   ▷ o is now the "result" variable.
  S ← {o}
  L_g ← ∅
  if o ≠ z then
    while S ≠ ∅ do
      Pop an element x from S
      if x is the output of an XOR gate g' and x is the operand of only one operation then
        Add the operands of g' to S.
      else
        Add x to L_g.
```

while others may 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 original 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 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 $(\boldsymbol{w}_i)_i$. They are based on the HPC2o or the 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, while the latency for S-boxes and adders is the number or register layers.

6.1 Optimized S-boxes

As a first case study for Compress, we generated optimized implementations of the AES S-box and of the 8-bit Skinny S-box. For the AES S-box, we considered two of the most commonly-used and state-of-the-art representations for masking: the 34 AND gate Boyar-Peralta representation [BP12] and the Canright tower field representation [Can05], while

Table 2: Performance characteristics of new masked AES S-box implementations (Boyar-Peralta representation [BP12], AND depth: 4).

Security order	Latency	Design	Random bits	Area w/o PRNG (kGE)	Area w/ PRNG (kGE)
1	4	Base	46	3.53	5.34
		Sep		3.24	5.05
		Opt		2.78	4.59
	5	Base	37	3.81	5.26
		Sep		3.51	4.97
		Opt		3.08	4.54
	6	Base	34	4.06	5.40
		Sep		3.77	5.11
		Opt		3.34	4.67
2	4	Base	138	7.91	13.34
		Sep		7.47	12.90
		Opt		6.79	12.22
	5	Base	111	8.54	12.91
		Sep		8.10	12.47
		Opt		7.47	11.84
	6	Base	102	9.00	13.01
		Sep		8.56	12.57
		Opt		7.92	11.94
3	4	Base	276	14.03	24.89
		Sep		13.44	24.30
		Opt		12.96	23.82
	5	Base	222	15.17	23.91
		Sep		14.58	23.32
		Opt		14.14	22.87
	6	Base	204	15.87	23.90
		Sep		15.28	23.31
		Opt		14.88	22.91
4	4	Base	460	21.89	40.00
		Sep		21.15	39.26
		Opt		20.45	38.55
	5	Base	370	23.68	38.25
		Sep		22.95	37.51
		Opt		22.24	36.80
	6	Base	340	24.69	38.07
		Sep		23.95	37.33
		Opt		23.27	36.65

Table 3: Performance characteristics of new masked AES S-box implementations (Canright representation [Can05], multiplication depth: 4).

Security order	Latency	Design	Random bits	Area w/o PRNG (kGE)	Area w/ PRNG (kGE)
1	4	Opt	36	1.95	3.37
	5		36	2.04	3.46
	6		36	2.13	3.55
2	4		96	4.56	8.34
	5		92	4.57	8.19
	6		90	4.72	8.27
3	4		192	8.06	15.62
	5		184	7.74	14.98
	6		180	7.83	14.92
4	4		300	12.48	24.29
	5		280	11.63	22.65
	6		270	11.59	22.22

Table 4: Performance characteristics of state-of-the-art masked AES S-box implementations.

Security order	Latency	Design	Random bits	Area w/o PRNG (kGE)	Area w/ PRNG (kGE)
1	6	[MCS22]	34	4.24	5.58
2			102	9.27	13.29
3			204	16.24	24.27
4			340	25.14	38.52
1	8	AGEMA -HPC2* (Boyar-Peralta)	34	5.06	6.39
2			102	10.49	14.51
3			204	17.87	25.89
4			340	27.18	40.56
1	4	AGEMA -HPC3* (Boyar-Peralta)	68	3.48	6.16
2			204	7.30	15.33
3			408	12.49	28.55
4			680	19.07	45.84
1	8	AGEMA -HPC2* (Canright)	40	5.60	7.17
2			120	11.80	16.52
3			240	20.28	29.72
4			400	31.04	46.78
1	4	AGEMA -HPC3* (Canright)	80	3.95	7.10
2			240	8.34	17.79
3			480	14.36	33.25
4			800	22.00	53.49
1	8	AGMNC [WFP+23]†	33	3.97	5.06
2			99	9.08	12.37
3			198	16.24	22.81
4			330	25.47	36.43
1	6	[ADN+22] Design I‡	54	2.71	4.84
1	5	[ADN+22] Design II‡	36	4.33	5.75
1	4	[BGN+15]‡§	32	2.84	4.10
1	4	[Sug19]‡	0	3.50	3.50
2	8	[DSM22]†	155‖	6.21	12.31
1	7	DOM [GMK16]¶	18	2.45	3.16
2			54	4.80	6.93
3			108	7.90	12.15
4			180	11.76	18.84

*AGEMA [KMMS22] in pipeline mode.
†Results reported by the designers. Synthesized with Synopsis Design Compiler on NanGate45.
‡Results reported by [ADN+22]. Synthesized with Synopsis Design Compiler on UMC 0.18μm GenericII Logic Process.
§Optimized synthesis by [BGN+15] achieves 2.22 kGE with Synopsis Design Compiler on UMC 0.18μm GenericII Logic Process. The latency of 4 clock cycles is composed of 2 clock cycles for the "core" 3-shares S-box design, and 2 clock cycles for the adaptation to a 2-shares AES design (input refreshing and output compression).
‖Includes 139 bits that can be shared.
¶Does not have a security proof.

Table 5: Performance characteristics of masked 8-bit Skinny S-box implementations, AND depth: 4.

Security order	Latency	Design	Random bits	Area w/o PRNG (kGE)	Area w/ PRNG (kGE)
1	4	Base	12	1.01	1.48
		Sep		0.97	1.45
		Opt		0.84	1.32
	5	Base	9	1.14	1.49
		Sep		1.12	1.47
		Opt		0.98	1.34
	6	Base	8	1.25	1.57
		Sep		1.23	1.55
		Opt		1.10	1.42
2	4	Base	36	2.10	3.51
		Sep		2.04	3.46
		Opt		1.85	3.27
	5	Base	27	2.36	3.43
		Sep		2.33	3.39
		Opt		2.13	3.20
	6	Base	24	2.56	3.50
		Sep		2.53	3.47
		Opt		2.34	3.28
3	4	Base	72	3.57	6.41
		Sep		3.50	6.34
		Opt		3.32	6.16
	5	Base	54	4.03	6.15
		Sep		3.98	6.11
		Opt		3.86	5.99
	6	Base	48	4.32	6.21
		Sep		4.28	6.17
		Opt		4.18	6.07
4	4	Base	120	5.44	10.16
		Sep		5.36	10.08
		Opt		5.11	9.83
	5	Base	90	6.13	9.67
		Sep		6.08	9.62
		Opt		5.89	9.43
	6	Base	80	6.54	9.69
		Sep		6.49	9.63
		Opt		6.33	9.47
1	6	[MCS22]*	8	1.33	1.65
2			24	2.68	3.62
3			48	4.48	6.37
4			80	6.74	9.89
1	8	AGEMA -HPC2	8	1.60	1.92
2			24	3.09	4.03
3			48	5.03	6.92
4			80	7.42	10.57
1	4	AGEMA -HPC3	16	1.00	1.63
2			48	1.99	3.88
3			96	3.30	7.08
4			160	4.93	11.23
1	9	[VCS22]‡	2	0.72	0.80
2			6	1.25	1.49
3			12	1.90	2.37
4			20	2.66	3.44
1	4	S_{2222} [CCGB21]	0	0.60	0.60
	3	S_{232} [CCGB21]		0.82	0.82
	3	S_{222} [CCGB21]		0.63	0.63
	2	S_{33} [CCGB21]		1.59	1.59

*This design is generated by the tool of [MCS22], and is used in [VCS22].
†AGEMA [KMMS22] in pipeline mode [KM22].
‡This design is not pipelined, it is a serial implementation that performs 2 S-box evaluations in 9 clock cycles.

the other state-of-the-art representations for masking [RP10, GPS14] use multiplications in $\mathbb{F}_{256}$, which is inefficient in hardware. For the Canright S-box, we implemented the $\mathbb{F}_4$ and $\mathbb{F}_{16}$ multiplications with the HPC1 and HPC3o gadgets, which can be viewed as Boolean gadgets that use multiple input and multiple output bit sharings.

Our results are given in Table 2, Table 3 and Table 5. We provide the masking order 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 (the Canright S-box requires "Opt", otherwise it has no single-cycle multiplication gadget). We also report similar results from related works in Table 4 for AES and Table 5 for Skinny.

Generally, the circuits generated by Compress achieve better performance than the state of the art, and all our optimizations ("Sep" over "Base" and "Opt" over "Sep") bring significant improvements. Further, we observe that some low latency designs require less area than the higher latency ones, even when taking into account the PRNG. 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 pipeline 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.

Regarding the AES designs, the "Opt" circuits generated by Compress for the Boyar-Peralta representation are smaller and lower-latency than the AGEMA-HPC2, [MCS22] and AGMNC designs, and have lower area than the AGEMA-HPC3 designs. Using the Canright representation further reduces the size of the circuit generated by Compress, which is the opposite of what happens with AGEMA, due the usage of bit-level operations for the implementation of finite-field multiplications in AGEMA's Canright representation [KMMS22]. We conclude that Compress has better results than the state of the art (AGEMA and the tool of [MCS22]) when implementing the same circuit representation, that natively handling larger field multiplication brings significant benefits, and that when using good representations such as Boyar-Peralta or Canright, Compress produces better results than AGMNC (which generates its own optimized representation of the S-box).

Extending the comparison beyond automated tools, Compress's Canright AES S-box also outperforms all comparable[10] first- and second-order state-of-the-art threshold implementation (TI) designs [ADN+22, BGN+15, Sug19, DSM22] w.r.t. area, while having equal or better latency. The arbitrary-order DOM-indep S-box [GMK16] is based on the Canright representation and was previously the smallest-known AES masked S-box but, in contrast with all other designs discussed above, it does not have a security proof. Compared to this design, Compress achieves a large latency reduction (4 clock cycles instead of 7) and has lower S-box area, but requires more randomness, which results in a larger total area when taking the Trivium PRNG into account.

For the Skinny S-box, we observe the same trend when comparing to AGEMA and [MCS22]: lower latency and/or 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 2 S-box evaluations in 9 clock cycles. In 9 clock cycles, Compress's circuit (with latency 4) performs 5 evaluations, while using only 65 % more (PRNG-included) area. Finally, the first-order TI designs of [CCGB21], achive a lower area and/or lower latency

[10] At the first order, S-boxes with 3 shares (or more) cannot be directly compared with 2-shares S-boxes, since they either increase the cost of the non-S-box layers in the cipher (state and key registers, MixColumns, MUXing logic... are then implemented with 3 shares instead of 2), or need to be adapted to 2-shares inputs and outputs by wrapping them with refreshing and compression layers, which increases the S-box area and latency. In our comparison, we consider the latter option which, e.g., increases the latency of the S-box of [BGN+15] from 2 to 4 clock cycles.

Table 6: Performance characteristics of masked AES-128 implementations (encrypt only).

Design	Datapath width	Latency	Security order	Area w/ PRNG (kGE)
[MCS22]	32-bit	106	1	33.4
			2	67.3
			3	115.9
			4	176.7
	128-bit	71	1	132.9
			2	293.3
			3	520.3
			4	815.2
AGEMA-HPC2 (Boyar-Peralta)	128-bit	91	1	170.9
			2	354.2
			3	602.9
			4	917.2
AGEMA-HPC3 (Boyar-Peralta)	128-bit	51	1	148.4
			2	344.2
			3	621.1
			4	979.2
Opt (Boyar-Peralta)	32-bit	86	1	29.0
			2	64.1
			3	114.2
			4	175.9
	128-bit	51	1	108.4
			2	268.3
			3	504.3
			4	813.0
Opt (Canright)	32-bit	86	1	24.4
			2	47.4
			3	81.2
			4	120.0
	128-bit	51	1	85.0
			2	190.9
			3	343.6
			4	521.5

than COMPRESS, the latter being achieved by building implementations for degree-3 functions directly.

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 (Canright and Boyar-Peralta) to two architectures implementing a masked AES-128 encryption (including the key schedule). The architectures 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 throughput (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 schedule operations are performed in parallel during a round

execution. Overall, the architecture is similar to the 32-bit one from [MCS22].[11] For this architecture, the modifications are 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. More details are provided in supplementary material.

Table 6 includes the post-synthesis implementation results for the two architectures (the Trivium PRNG is included in the masked AES designs), whose security has been verified by fullVerif [Cas20]. The performance comparison of the implementations based on the Canright representation with the implementation of [MCS22] shows that a latency reduction of roughly 19 % for the 32-bit architecture (resp. 29 % for the 128-bit architecture) is achieved by the new implementations. With regard to area, a reduction of about 27 % is achieved at the first order for the 32-bit architecture (36 % for the 128-bit one), with similar results at higher orders. For AGEMA, using the round-based architecture of [KMMS22] (which is similar to our 128-bit architecture with the Boyar-Peralta S-box), we achieve a 43 % area reduction over the HPC3 implementation (same latency) and 50 % area reduction over the HPC2 implementation (44 % latency reduction).

In order to test the scalability of our tool to larger circuits, we automatically generated a full AES round using 20 Boyar-Peralta S-boxes. For all $d \leq 5$, Compress completed in less than 1 minute, with a result almost identical to the round of our 128-bit implementation (there is an area overhead below 1 %, explained by the Compress-generated round having masked pipeline registers for the round constants which can be avoided in our implementation).

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 and have a higher AND depth, therefore they test the scalability 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

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

Table 7: Performance characteristics of 32-bit adder implementations.

Design	Architecture	Security order	Latency	Random bits	Area w/o PRNG (kGE)	Area w/ PRNG (kGE)
Opt	RC	1	31	32	19.04	20.30
			32	31	19.76	20.98
		2	31	96	31.20	34.98
			32	93	32.31	35.97
	KS	1	5	374	16.72	31.44
			6	309	18.61*	30.77*
		2	5	1122	43.30	87.47
			6	927	47.77*	84.26*
	Sklansky	1	6	172	12.88	19.65
			7	161	13.65*	19.98*
		2	6	516	31.71	52.02
			7	483	33.09*	52.10*
	BK	1	9	128	12.13	17.17
			10	122	13.03*	17.83*
		2	9	384	27.96	43.08
			10	366	29.23*	43.63*
[MCS22]	RC	1	32	31	20.61	21.83
		2	32	93	33.57	37.23
	KS	1	10	249	28.31	38.11
		2	10	747	63.75	93.16
	Sklansky	1	9	151	18.88	24.82
		2	9	453	41.23	59.06
	BK	1	12	117	17.32	21.93
		2	12	351	35.98	49.80
AGEMA -HPC2	RC	1	62	31	37.10	38.32
		2	62	93	58.30	61.96
	KS	1	10	249	31.71	41.51
		2	10	747	68.85	98.26
	Sklansky	1	12	151	23.40	29.34
		2	12	453	48.01	65.84
	BK	1	18	117	22.38	26.98
		2	18	351	43.57	57.38
AGEMA -HPC3	RC	1	31	62	19.49	21.93
		2	31	186	31.13	38.45
	KS	1	5	498	21.83	41.43
		2	5	1494	47.94	106.74
	Sklansky	1	6	302	15.37	27.26
		2	6	906	32.27	67.93
	BK	1	9	234	14.05	23.26
		2	9	702	28.22	55.85
[SMG15] (TI)†	RC	1	32	4	N/A‡	N/A‡
		2		8		
[BG22] (HPC2)	KS	1	12	249		
		2		747		
[BG22] (TI)		1	6	31		
[BG22] (HPC2)	Sklansky	1	12	119		
		2		357		
[BG22] (TI)		1	6	41		
[BG22] (HPC2)	BK	1	18	74		
		2		222		
[BG22] (TI)		1	9	31		

*Might not be optimal area since the solver stopped early due to the 1 h timeout.
†Threshold Implementation, not an HPC design (first- and second-order security).
‡Designs are not open-source and area numbers for ASIC designs are not given.

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 7. 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, and for higher latencies. We compare our results with the designs of Schneider *et al.* [SMG15] and Bache *et al.* [BG22] in addition to automatically-generated HPC2 and HPC3 ones using the tools of [MCS22] and [KMMS22] respectively. 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, only KS, Sklansky and BK adders with non-minimal latency reached the timeout (these are indeed the largest circuits, and non-minimal latency greatly increases the solution space). Since these clearly correspond to suboptimal cases, since the adders can be implemented using half the latency and less area, we consider that Compress scales successfully to 32-bit adders.

Compared to the TI designs of [SMG15, BG22], Compress achieves equal or better latency, while having at most half the latency of the HPC2 designs of [BG22]. ASIC area is not given in [SMG15, BG22]. It can however be noted that the RC adder uses 3 shares at first-order and a mix of 5 and 10 at second-order, whereas the non-TI designs use only the minimal amount (respectively 2 and 3 shares). The comparison with the automatically-generated designs yields a similar conclusion as for the S-boxes.

7 Related Works

In this section, we compare Compress to other automated tools for the generation of masked designs.

AGEMA AGEMA [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 circuits 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 cycles.

Further, it does not optimize the scheduling of the computations. Overall, it appears that AGEMA and Compress focus on different goals: AGEMA handles general circuit masking and interaction between masked and non-masked parts of the circuits, while Compress optimizes masked pipelines.

Momin et al.'s tool 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[12] that generates a pipeline, which has a purpose similar to Compress. This tool works exclusively with the HPC2 gadget, and exploits its asymmetric latency characteristic (it has a latency of 2 clock cycles regarding one of the input sharings, and only one clock cycle regarding the other one) 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 consists 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 fulfill 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 optimal solution, while the algorithm of AGMNC is not described.

To the best of our knowledge, this work and [WFP+23] overlap in their goal, but not in the contributions, except for gadgets that implement a Toffoli gate. AGMNC introduces new AND-XOR gadgets for this purpose, saving 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 described in Compress (Section 4.2). We also introduce the HPC2o and HPC3o, which are even more optimized than AND-XOR (in particular, both AND-XOR2 and HPC2o are based on HPC2, but HPC2o has a lower area).

EasiMask EasiMask [BSG23] is another recent tool for automating masked circuit generation. Similarly to AGEMA, this is a high-level tool that transforms 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 between unrolled, round-based or serial architectures. EasiMask comes with a library of masked S-boxes to choose from, and does not generate 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.

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

8 Conclusion

Compress optimizes the area of masked pipelines by minimizing the amount of pipeline 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 low 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.

Acknowledgements

Gaëtan Cassiers is a postdoctoral researcher of the Belgian Fund for Scientific Research (FNRS-F.R.S.). This work was supported in part by the FWF SFB project SpyCoDe F8504, by the Walloon Region through the FEDER project USERMedia (501907-379156) and the Win2Wal project PIRATE (1910082), by the ERC Advanced Grant 101096871 (BRIDGE) and by SGS. Views and opinions expressed are those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[ABP+18] Victor Arribas, Begül Bilgin, George Petrides, Svetla Nikova, and Vincent Rijmen. Rhythmic keccak: SCA security and low latency in HW. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):269–290, 2018.

[ADN+22] Amund Askeland, Siemen Dhooghe, Svetla Nikova, Vincent Rijmen, and Zhenda Zhang. Guarding the first order: The rise of AES maskings. In *CARDIS*, volume 13820 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2022.

[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.

[BGN+15] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Trade-offs for threshold implementations illustrated on AES. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 34(7):1188–1200, 2015.

[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.

[Can05] David Canright. A very compact s-box for AES. In *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.

[Cas20] Gaëtan Cassiers. FullVerif, 2020.

[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 *COSADE*, volume 10348 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2017.

[CCGB21] Andrea Caforio, Daniel Collins, Ognjen Glamocanin, and Subhadeep Banik. Improving first-order threshold implementations of SKINNY. In *INDOCRYPT*, volume 13143 of *Lecture Notes in Computer Science*, pages 246–267. Springer, 2021.

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

[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.

[DSM22] Siemen Dhooghe, Aein Rezaei Shahmirzadi, and Amir Moradi. Second-order low-randomness d + 1 hardware sharing of the AES. In *CCS*, pages 815–828. ACM, 2022.

[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.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *TIS@CCS*, page 3. ACM, 2016.

[GPS14] Vincent Grosso, Emmanuel Prouff, and François-Xavier Standaert. Efficient masked s-boxes processing - A step forward -. In *AFRICACRYPT*, volume 8469 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2014.

[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.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In *ASIACRYPT (1)*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[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.

[MM22] Nicolai Müller and Amir Moradi. PROLEAD A probing-based hardware leakage detection tool. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):311–348, 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.

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

[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.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.

[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.

[Sug19] Takeshi Sugawara. 3-share threshold implementation of AES s-box without fresh randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):123–145, 2019.

[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.

[ZSS+21] Sara Zarei, Aein Rezaei Shahmirzadi, Hadi Soleimany, Raziyeh Salarifard, and Amir Moradi. Low-latency keccak at any arbitrary order. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):388–411, 2021.

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, and a glitch+transition-extended probe leaks all the inputs of the combinatorial circuit that generates a value, for two consecutive clock cycles.

Let us now formally introduce the notion of gadget.

Definition 1 (Gadget)**.** A gadget G is a sub-circuit whose input and output wires a grouped in d-tuples named *sharings*. The index of a share is its (zero-indexed) position in its sharing. A gadget with input sharings $(x_0^1, \dots x_{d-1}^1), \dots, (x_0^m, \dots x_{d-1}^m)$ and output sharings $(y_0^1, \dots y_{d-1}^1), \dots, (y_0^n, \dots y_{d-1}^n)$ implements a function f iff $(y_0^1 \oplus \dots \oplus y_{d-1}^0, \dots, y_0^n \oplus \dots \oplus y_{d-1}^n) = f(x_0^1 \oplus \dots \oplus x_{d-1}^0, \dots, x_0^m \oplus \dots \oplus x_{d-1}^m)$ for all possible values of the input shares and randomness sampled inside the gadget.

Unlike many previous works (e.g., [ISW03, CS21]), we consider gadgets that do not implement any function (e.g., Sharewise-AND, HPC2-Cross, HPC3-Cross). This change in the definition of a gadget does not impact the following security notions.

Definition 2 (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 3 (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 $A \subseteq \{0, \dots, d-1\}$ and any set of glitch-extended probes P such that $|A| + |P| \leq t$, there exists a set $B \subseteq \{0, \dots, d-1\}$ 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.[13] □

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 consequence, r_{ij} is not observed except through the value v_{ij}, which appears as a fresh random.[14] □

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

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

B Optimization problem

In this section, we describe in more detail the optimization problem generated by Compress. This problem is modelled using the CPMpy [Gun19] modeling library and solved using OR-Tools [PF]. Thanks to the expressive modeling features of CPMpy, the constraint given below can be straightforwardly implemented.

Compress takes as input an integer latency $L \geq 0$ (the number of register stages in the generated pipeline), a description of the circuit, and a set of gadgets. The circuit contains a list of input variables (variables are represented by their label v and are Boolean), a list of output variables, and a set of operations. Each operation is represented by a function call under the form $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$, where all x_i's and y_i's are variables. The set of computations must represent a well-formed logic circuit: there must be no cyclic dependency and each variable must be assigned to exactly once (inputs count as assignments). Computations can be simple logic gates such as AND, XOR or NOT, but they can also be more complex, multi-output functions (e.g., a multiplication in $\mathsf{GF}(2^4)$ using a particular representation of $\mathsf{GF}2^4$ as 4 Booleans). Each gadget in the set G is a masked implementation of a computation, and it is characterized by its area (in GE), randomness usage (in bits), and the input-to-output latency for each input sharing (for simplicity, we assume that all output shares have the same latency).

Algorithm 9 is executed by Compress to generate the optimization problem. We use the following notations: The $\mathsf{Variable}\ x$ instruction instantiates a Boolean variable x in the optimization problem (if the x variable has already been instantiated, this does nothing), and $\mathsf{Constraint}\ \cdot$ adds a constraint to the problem (a Boolean predicate that must be true). Let I be the set of input variable labels, O be the set of variable labels, and V be the set of all variable labels that appear in a computation or in the inputs/outputs. For $w \in V$, we denote by $f(w)$ the computation that computes w (i.e., $f(w) = f(w')$ iff w and w' are generated by the same computation), and $\mathsf{op}(w)$ denotes the set of variables that are the operands of the computation $f(w)$. For a gadget $t \in G$, $\mathsf{lat}(t, w, w') \geq 0$ is the latency of the input sharing w' relative to the output sharing w (i.e., the difference in pipeline stages between the input and the output), and $\mathsf{maxlat}(t) = \max_{w,w'} \mathsf{lat}(t, w, w')$.

Finally, a_t is the area (including randomness) of the masked gadget t (a_{reg} and a_{xor} are the areas of the sharewise register and XOR, respectively). Since the solver works only with integers, we use a fixed point representation for the areas a_t.

Algorithm 9 Generation of the optimization problem.

$A \leftarrow \emptyset$ ▷ Set of all gadget instances.

▷ "valid", "compute" and "pipeline" variables described in Section 3.

for $s = 0$ to L, $w \in V$ **do**

  Variable v_s^w

  Variable c_s^w

  Variable r_s^w

  $C_s^w \leftarrow \emptyset$ ▷ All possible ways to compute w in stage s.

  for $t \in G$ **do**

    Variable $g_s^{t,f(w)}$

    Add $g_s^{t,f(w)}$ to C_s^w and to A.

▷ No pipelining from before the first pipeline stage.

for $w \in V$ **do**

  Constraint $v_0^w = c_0^w$

▷ Input wires are "computed" only at the first stage.

for $w \in I$ **do**

  Constraint $c_0^w = \top$

  for $s = 1$ to L, $w \in V$ **do**

    Constraint $c_s^w = \bot$

  ▷ Output wires must be valid at the last stage.

for $w \in O$ **do**

  Constraint $v_L^w = \top$

▷ A gadget can only be instantiated if its inputs are valid and its type matches the expected computation.

for $s = 0$ to L, $w \in V$, $t \in G$ **do**

  if t does not implement $f(w)$ or $\mathsf{maxlat}(t) > s$ **then**

    Constraint $g_s^{t,f(w)} = \bot$

  else

    Constraint $g_s^{t,f(w)} \Rightarrow \bigwedge_{w' \in \mathsf{op}(w)} v_{s-\mathsf{lat}(t,w,w')}^{w'}$

▷ Add Toffoli gates: due to the possibility of XORing terms at stages after the output of the AND gate, we add a series of registers on the output of the Toffoli gate.

$TR \leftarrow \emptyset$, $TX \leftarrow \emptyset$ ▷ Sets of registers and XOR gadgets related to Toffoli gates.

for $s = 0$ to L, $w \in V$, $t \in \mathrm{TG}$ **do**

  if $f(w)$ is a AND operation and $\mathsf{maxlat}(t) > s$ **then**

    Let $(\mathsf{XOR\_OPS}_w, \mathsf{RES}_w)$ be the output of Algorithm 8 running on $f(w)$.

    Variable $\mathsf{tof}_s^{t,f(w)}$

    Add $\mathsf{tof}_s^{t,f(w)}$ to A and $(\mathsf{tof}_s^{t,f(w)}, |\mathsf{XOR\_OPS}_w|)$ to TX.

    Constraint $\mathsf{tof}_s^{t,f(w)} \Rightarrow \bigwedge_{w' \in \mathsf{op}(w)} v_{s-\mathsf{lat}(t,w,w')}^{w'}$ ▷ AND inputs must be valid.

    for $s' = s + 1$ to L **do**

      Variable $\mathsf{rtof}_{s,s'}^{t,f(w)}$

      Add $\mathsf{rtof}_{s,s'}^{t,f(w)}$ to TR. ▷ Instantiate register chain on Toffoli gate's output.

    Constraint $\mathsf{rtof}_{s,s+1}^{t,f(w)} \Rightarrow \mathsf{tof}_s^{t,f(w)}$

    for $s' = s + 2$ to L **do**

      Constraint $\mathsf{rtof}_{s,s'}^{t,f(w)} \Rightarrow \mathsf{rtof}_{s,s'-1}^{t,f(w)}$

    ▷ XOR operand must be valid at the input or output stage of the Toffoli gate, or sometime within the register chain.

    for $w' \in \mathsf{XOR\_OPS}_w$ **do**

      Constraint $\mathsf{tof}_s^{t,f(w)} \Rightarrow v_{s-1}^{w'} \vee v_s^{w'} \vee \bigvee_{s' \in \{s+1,\ldots,L\}} v_{s'}^{w'} \wedge \mathsf{rtof}_{s,s'}^{t,w}$

    for $s' = s + 1'$ to $L - 1$ **do**

      Add $\mathsf{rtof}_{s,s'}^{t,f(w)} \wedge \neg \mathsf{rtof}_{s,s'+1}^{t,f(w)}$ to $C_{s'}^{\mathsf{RES}_w}$.

    Add $\mathsf{tof}_s^{t,f(w)} \wedge \neg \mathsf{rtof}_{s,s+1}^{t,f(w)}$ to $C_s^{\mathsf{RES}_w}$.

    Add $\mathsf{rtof}_{s,L}^{t,f(w)}$ to $C_L^{\mathsf{RES}_w}$.

▷ Define the "compute" variables from sets of possible computations.

for $s = 0$ to L, $w \in V$ **do**

  Constraint $c_s^w = \bigvee_{x \in C_s^w} x$

▷ Avoid duplication of gadgets that would break sharing's equalities (see (1) and (2)).

for $w \in V$ **do**

  Constraint $\mathsf{AtMostOne}\left(\left(\mathsf{Any}\left(\left(g_s^{t,f(w)}\right)_{s=0,\ldots,L}\right)\right)_{t \in G \cup \mathrm{TG}}\right)$

for $s = 0$ to L, $w \in V$ **do**

  if at least on of the gadgets that implement $f(w)$ use randomness **then**

    Constraint $\mathsf{AtMostOne}\left(\left(g_s^{t,f(w)}\right)_{s=0,\ldots,L}\right)$

The goal to minimize is

$$\sum_{g_s^{t,f(w)} \in A : g_s^{t,f(w)} = \top} a_t + \sum_{s=0}^{L} \sum_{w \in V : r_s^w = \top} a_{\mathsf{reg}} + \sum_{(x,n) \in TX : x = \top} a_{\mathsf{xor}} + \sum_{x \in TR : x = \top} a_{\mathsf{reg}}$$