Handcrafting: Improving Automated Masking in Hardware with Manual Optimizations

Charles Momin, Gaëtan Cassiers, and François-Xavier Standaert

UCLouvain, ICTEAM, Crypto Group, Louvain-la-Neuve, Belgium
firstname.lastname@uclouvain.be

Abstract. Masking is an important countermeasure against side-channel attacks, but its secure implementation is known to be error-prone. The automated verification and generation of masked designs is therefore an important theoretical and practical challenge. In a recent work, Knichel et al. proposed a tool for the automated generation of masked hardware implementations satisfying strong security properties (e.g., glitch-freeness and composability). In this paper, we study the possibility to improve their results based on manual performance optimizations for the AES algorithm. Our main conclusion is that as the target architecture becomes more serial, such a handcrafted approach gains interest. For example, we reach latency reductions by a factor six for 8-bit architectures. We conclude the paper by discussing the extent to which such optimizations could be integrated in the tool of Knichel et al. As a bonus, we adapt a composition-based verification tool to check that our implementations are robust against glitches & transitions, and confirm the security order of exemplary implementations with preliminary leakage assessment.

Keywords: Side-Channel Attacks · Masking Countermeasure · Efficient Hardware Implementations · AES Rijndael · Formal Verification

1 Introduction

Side-channel attacks relying on the exploitation of physical information leakage such as the power consumption or the electromagnetic radiation of cryptographic implementations are an important security threat. The masking countermeasure is a standard answer to this threat [CJRR99]. Its underlying principle is to compute over secret-shared intermediate variables that are individually independent of the secret data manipulated by a device. For this purpose, the designs to protect are typically split in small operations (e.g., AND and XOR gates) which are then replaced by gadgets able to compute securely over shared data.

The evaluation of masked implementations is a tricky task. On the one hand, the security of small gadgets may not directly extend to their combination, leading to so-called composition issues [CPRR13,BBD+16]. On the other hand, physical defaults such as glitches can break the independence assumptions needed for masking to be secure [MPG05,NRS11]. These issues can also be combined leading to additional challenges [FGP+18,MMSS19]. This state-of-the-art has

motivated an increased interest for the automated verification of masked implementations [BBD+15,BGI+18,BBC+19,KSM20]. The automated generation of these implementations therefore appears as the natural next steps, and first efforts in this direction can be found in [BDM+20,KMMS22].

In this paper, we are in particular interested in the work of Knichel et al., which introduced a tool, AGEMA, allowing inexperienced engineers to generate masked implementations from unprotected ones [KMMS22]. At high-level, the tool leverages the Hardware Private Circuits (HPC) scheme introduced in [CGLS21] and variations thereof. The HPC scheme provides strong composition properties in the presence of hardware defaults such as glitches and comes with the fullVerif tool which allows checking if the requirements needed for the composition theorems to hold are respected in practice. By combining AGEMA and fullVerif, Knichel et al. made a significant step towards improving the usability of masking schemes in hardware. Yet, and as usual when considering automation, these advances also raise the question whether the implementations obtained compete with manually optimized ones. In other words, can all the architecture-level optimizations that an experienced designer would exploit be automated, and if not, what is the performance gap that they lead to?

We contribute to this question by presenting the results of different masked hardware implementations of the AES, designed to take advantage of the inherent pipeline of complex blocks based on the HPC2 gadgets. In particular, we first point out that AGEMA's generated implementations are suboptimal in terms of latency. We propose an optimization strategy and demonstrate that using it gives better result for the implementation of specific blocks such as the AES S-box. Second, we propose optimized masked AES implementations based on 8-bit, 32-bit and 128-bit loop architectures. We show that the latency of our handcrafted designs is significantly improved w.r.t. the ones generated by AGEMA in the 8-bit case, while the gains tend to vanish for the larger architectures. Concretely, our results therefore provide improved masked implementation results for the AES-128 algorithm based on a state-of-the-art masking scheme. More generally, we also use our investigations to discuss tracks that could be used to improve the performances of automatically generated masked implementations for tools like AGEMA. Eventually, and as an additional contribution, we analyze the security of our implementations, first in the robust probing model with glitches and transitions using the fullVerif tool, which we modified to verify the transition-robustness conditions introduced in [CS21].[1] We finally confirm these results by means of leakage detection tests on exemplary implementations.

Related works. Many hardware masking schemes are proposed in the literature. For example, the proposals in [CRB+16,GMK17] aim at similar goals as HPC, with less formal composability guarantees. Our focus is on the HPC scheme because it has been selected for automation in [KMMS22], but we believe our main conclusions regarding automation are mostly independent of this choice.

[1] The verification of the transition-robustness has been integrated in the latest version of fullVerif at `https://github.com/cassiersg/fullverif`.

Paper structure. The paper is organized in 3 parts. First, we describe the AES-128 algorithm and recall some specificities about the HPC2 masking scheme. We also provide brief explanations about the fullVerif and AGEMA tools. Next, we detail the construction of the protected S-box as well as the three AES architectures considered in this work. Finally, performance metrics and basic side-channel analysis results are presented.

2 Background

Advanced Encryption Standard. The Advanced Encryption Standard (AES) is a symmetric key algorithm standardized by the NIST in 2001. Its variant AES-128 operates on 128-bit state with 128-bit secret key. Both the state and the key can be represented as 4x4 matrices of 16 bytes each. The algorithm is composed of 10 rounds, each one being composed of 4 different operations. First, the `SubByte` operation is a non-linear substitution operating on each byte independently. Second, the `ShiftRows` operation is a cyclic shift of 1, 2 or 3 positions of each byte in the last three rows. Third, the `MixColumns` operates over each column of the state independently. The later are considered as polynomials with coefficients over GF(2^8) and are multiplied modulo $x^4 + 1$ by the fixed polynomial $03x^3 + 01x^2 + 01x + 02$. Finally, the `AddRoundKey` operation adds the round key by performing a bitwise XOR operation. It has to be noted that the `MixColumns` operation is not performed in the 10-th (i.e., last) round and an `AddRoundKey` is performed before the first round.

Each round key is derived with a key scheduling algorithm applied at each round on the previous round key. First, the last column is rotated by one byte up. Next, the SubByte operation is applied on each byte of the resulting column and the round constant is added using a bitwise XOR operation. The first column of the new round key is obtained by applying a bitwise XOR between the column obtained after the round constant addition and the first column of the key. Finally, the new value of the i-th column is obtained by bitwise XORing the i-th column of the key and the $i-1$-th column of the new round key.

HPC2 masking scheme. The hardware private circuits introduced in [CGLS21] come in two flavors: we next focus on the HPC2 variant. In addition to being glitch-robust, its gadgets are proven trivially composable at any order using the Probe Isolation Non Interference (PINI) framework [CS20]. The HPC2 scheme operates over GF(2) (i.e., bitwise) and requires state-of-the-art amount of randomness for its non-linear operation (i.e., AND2 gates). Precisely, each AND2-HPC2 gadget requires $d(d-1)/2$ bits of randomness where d is the number of shares in the design. A specificity of this gadget is its latency asymmetry: if the first input sharing enters the gadgets at cycle t, the second input sharing is expected to enter the gadget at cycle $t+1$ and the output is produced at cycle $t+2$. Other gadgets performing basic operations have been designed (e.g.,

XORs & MUXes).[2] The HPC2 masking scheme requires no refresh gadget, and the affine/linear gadgets do not require randomness and have no latency.

fullVerif. The open source tool fullVerif is an automated composition-based program that can verify that the assumptions of the HPC2 security proofs are fulfilled by a Hardware Description Language (HDL) design. In order to be checked by fullVerif, the modules' definitions as well as their ports should be annotated with specific verilog attributes. These attributes indicates signal's properties such as their types (i.e., *sharing*, *control* or *randomness*) or their sizes and time validity to the tool. Different composition strategies can be specified for each module in the design. The tool works based on a netlist of the architecture together with a simulation file in order to build a dataflow graph. The latter is then used to proceed to different security checks as listed in [CGLS21].

AGEMA is an open source tool presented in [KMMS22]. It automatically generates masked hardware netlists based on unprotected HDL implementations and relies on specific annotations of the IO ports at the top level of the hierarchy in order to identify which part of the circuit should be masked. Working on top of a synthesized netlist of the full architecture, it propagates the signals annotations across the hierarchy before implementing the masked parts of the circuit. It offers the choice between different masking schemes relying on the PINI property (among which HPC2), as well as different optimization strategies depending on the chosen masking scheme. To ensure the proper synchronization of the signals across the hierarchy, the tool relies on two approaches: pipelining (i.e., introducing registers to synchronize all the inputs of each masked gadgets) or clock gating (i.e., modulating the registers' clock signals). As shown in [KMMS22], pipelining implies more area cost while achieving better throughput; clock gating allows reducing the global cost but results in implementations with larger latency.

3 Architectures descriptions

We now describe our optimized constructions. First, we introduce a generic latency optimization methodology for pipeline blocks based on the HPC2 AND gadget. It is applied to a bit-level AES S-box, reducing its latency by 25 %. Second, our three AES architectures are detailed, differing by their level of parallelism, and more particularly the number of S-boxes instantiated in the design. Precisely, we next consider an architecture using one S-box instance (8-bit serial), 4 S-box instances (32-bit serial) and 20 S-box instances[3] (128-bit serial).

3.1 Masked AES S-box Implementation

Because of its significant cost both in logic and randomness, the masked implementation of the non-linear `SubByte` layer requires a particular attention. First,

[2] All the gadgets are available in a public verilog library at `https://github.com/cassiersg/fullverif/tree/release/lib_v`.

[3] 16 S-boxes for the rounds computation and 4 for the key schedule.

taking into account the constraint that the HPC2 masking scheme instantiates gadgets operating over GF(2) only, we need a bit-level S-box representation. We follow the design choice of [KMMS22] for this purpose, and use the S-box design proposed by Boyar and Peralta [BP12]. It is only composed of basics operation over GF(2) (i.e., XOR2, AND2 and NOT gates). We selected this design as a good trade-off between the number of AND2 gates (it uses 34 of them) and the AND depth (which is 4). The design with the lowest number of AND2 gates has been introduced in [BMP13] and uses 32 AND2 gates but has a AND depth of 6.

Due to the asymmetry at their inputs, naively replacing basic unprotected operations in complex circuit with HPC2 gadgets may result in a sub-optimal area and latency. For example, two different implementations (i.e., unprotected and masked) of an AND3 gate are shown Figure 1. While the area and latency for the two unprotected implementations are identical, they differ for the protected ones. On the left, 4 additional registers are required to ensure the circuit functionality, and the computation is performed in 4 cycles. On the right, only 2 registers are required and the computation is performed in 3 cycles. Such area and latency overheads are incurred by implementations (such as AGEMA) that add a register on one of the inputs of the AND2 gadget to achieve symmetric latency (which simplifies the design process).

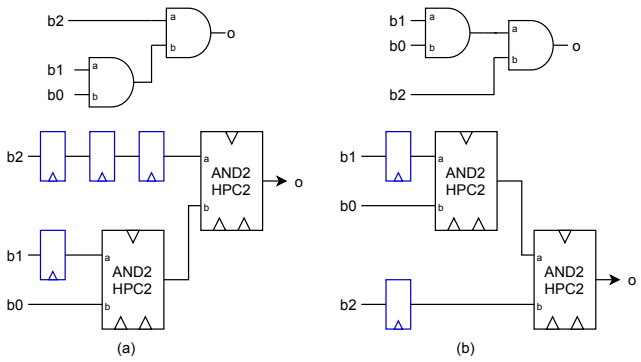

Fig. 1. Example of AND3 gate implementation using HPC2 gadgets.

Based on this observation, it may be tempting to look for optimal symmetric-latency gadget-based circuits for more complex operations (e.g., AND3 gates). However, using a composition of such larger gadgets is still not optimal. For example, two implementations of an AND5 operation are shown in Figure 2. On the left, the implementation directly instantiates twice the optimized AND3 construction of Figure 1 and requires 10 additional registers to compute the results in 6 cycles. On the right, a more optimized implementation only requires 3 additional registers and performs the computation in 4 cycles. Instead of the

"larger gadgets" approach, we optimize directly the latency of a full logic block (the full S-box), which helps reducing the number of registers in our designs.

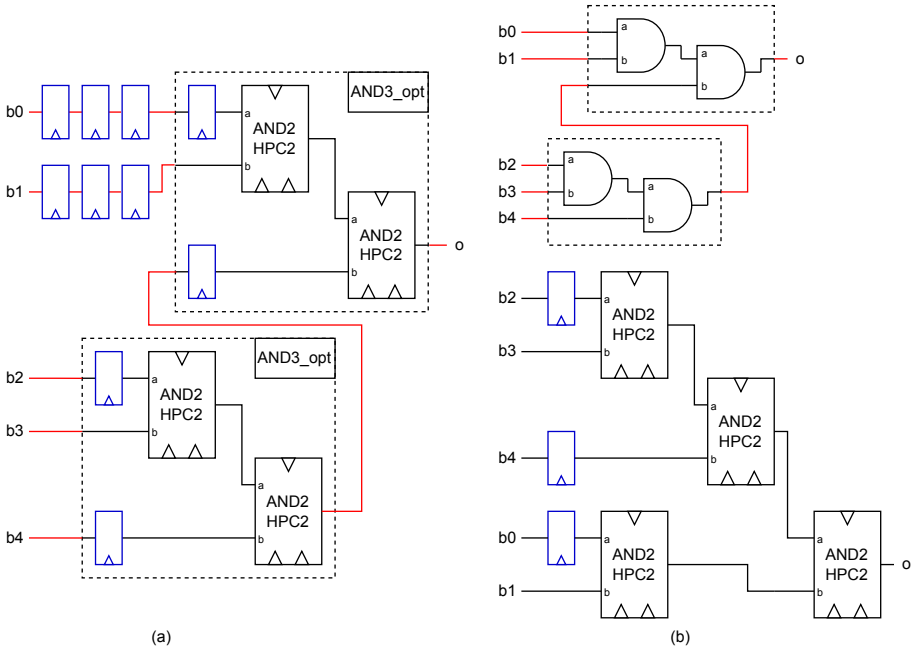

Fig. 2. Example of AND5 gate implementation using HPC2 gadgets.

Generic block latency optimization algorithm. The optimization of a block is efficiently performed as follows. Starting from the input signals of the logic block to optimize, the information about the exact time of validity of each signal is propagated across the circuitry, level by level. For some gadgets that have more than one input, the latency requirements of all the inputs may not be met. In such cases, a solution is to insert registers on the input path in order to meet the timing constraints. In the specific case where the latest signal (i.e., the signal being valid with the largest latency) is connected to the input port with the lowest timing requirement (i.e., the input port at which a signal is first entering the gadget), then the input connections are switched. This optimization naturally leads to a solution with a reduced amount of additional registers while keeping the same functionality as the original circuit.

This (easy to automate) procedure has been used in order to generate the architecture of the implementations (b) on Figure 1 and Figure 2 . It leads to significant improvements for the AES S-box implementation. Considering a

latency-equalized version of the AND2-HPC2 gadget from [CGLS21] (as in Figure 1), the implementation of the Boyar-Peralta S-box requires $156d$ additional registers and operates in 8 cycles. With our optimization, it only requires $94d$ additional registers and operates in 6 cycles.

3.2 8-bit serial implementation

As depicted in Figure 3, the 8-bit serial implementation takes as input the shared value of the key `sh_key` and the (unshared) plaintext `pt`. The later is first represented as a valid sharing by concatenating $d-1$ zero shares to each bit. The architecture of this implementation is organized around two mains blocks: the `KeyHolder` and the `StateHolder` which store and order the processing of the round key and of the state. The blocks feed each byte serially during the `SubByte` and `AddRoundKey` layers. Unless otherwise noted, each bus in the architecture is 8-bit wide. In addition, one S-box is instantiated with the block `MSKSbox`. The block `MSKmc` is a combinatorial logic that implements the `MixColumns` operation for a full column (i.e., 4 bytes) and consists in d instances of the unprotected `MixColumns` operation logic, where each instance processes one state share.

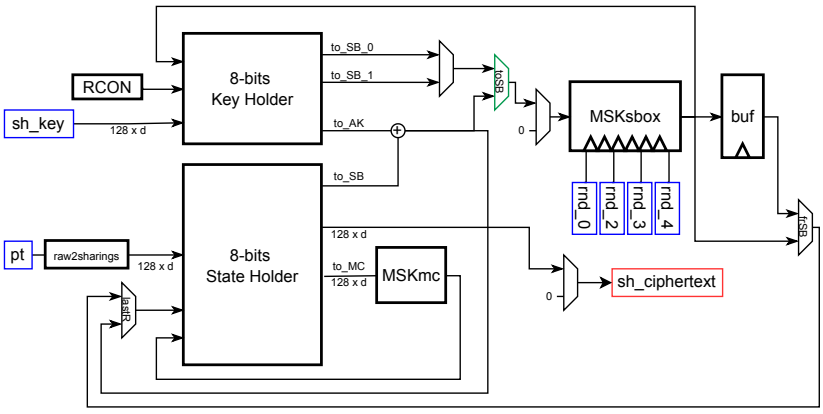

Fig. 3. Global Architecture of the 8-bit serial implementation.

The shared state and round key are stored in dedicated shift registers of shared bytes, as shown in Figure 4 and Figure 5. The rounds are computed serially by performing the `SubByte` and `AddRoundKey` operations byte per byte. The state is processed row per row (i.e., starting with the bytes 0,4,8,12 and ending with the bytes 3,7,11,15). The four S-box executions occurring during the key scheduling are interleaved between the processing of the state. To do so, the mux `toSB` in Figure 3 is used to feed the S-box instance with a shared byte coming either from the key, either from the `AddRoundKey` result. This interleaving of key scheduling and round processing necessitates the insertion of a buffer at

the output of the S-box to avoid loosing data in cases where the output of the S-box is valid and a key byte is provided at its input. Indeed, in such cases, the shift register holding the state is stalled, making any feedback from the S-box to the state holder impossible. We rely on the pipeline structure of the S-box instance to reduce the overall latency of the execution. That is, we do not wait for full S-box execution to be finished before feeding the S-box with valid data. Instead, we feed it at each clock cycle when its input data is available. Considering the latency of our S-box design, the full `AddRoundKey`, `SubByte` and the key scheduling operations of a round are performed in $\mathsf{SB}_{lat} + 16 + 4 = 26$ cycles.

The `ShiftRows` operation is performed in a single cycle by enabling the data flowing through the state holder with an appropriate routing defined by the MUXes `sh-i` (depicted in green in Figure 4). Finally, the `MixColumns` operation is performed in 4 cycles, using the MUXes `mc-i` to route the signals back from the `MC` instance (depicted in red in Figure 4). The last `AddRoundKey` operation is performed byte per byte in 16 cycles, using the mux `lastR` in Figure 3 to bypass the S-box instance. Overall, the latency of a full execution of our 8-bit implementation is equal to $9 \times (26 + 1 + 4) + (26 + 1) + 16 = 322$ cycles.[4]

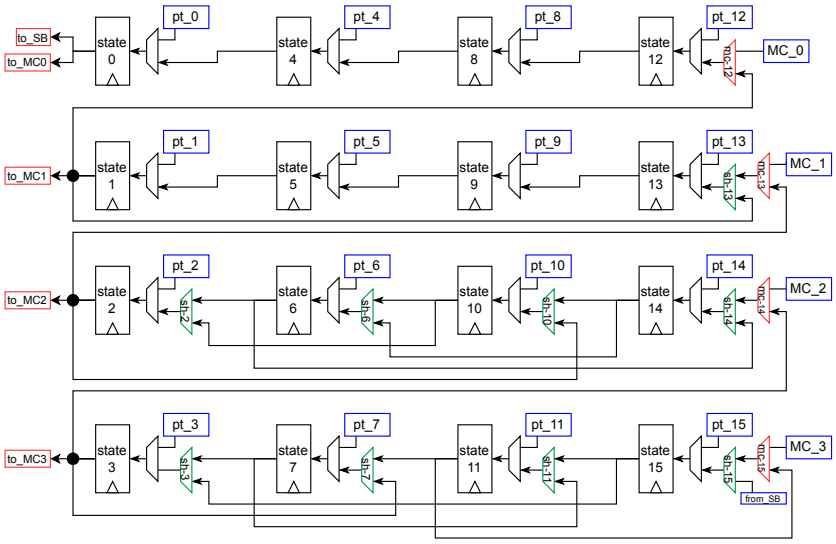

Fig. 4. 8-bit implementation: architecture of the state holder.

[4] A MUX located at the output of the global core is used to control the proper release of the valid ciphertext. This is not strictly required in the context of our work, however practical integrations will likely add a logic block computing the recombined ciphertext. Without our output gating, this would lead to leaking unmasked internal states of the AES, defeating the masking countermeasure.

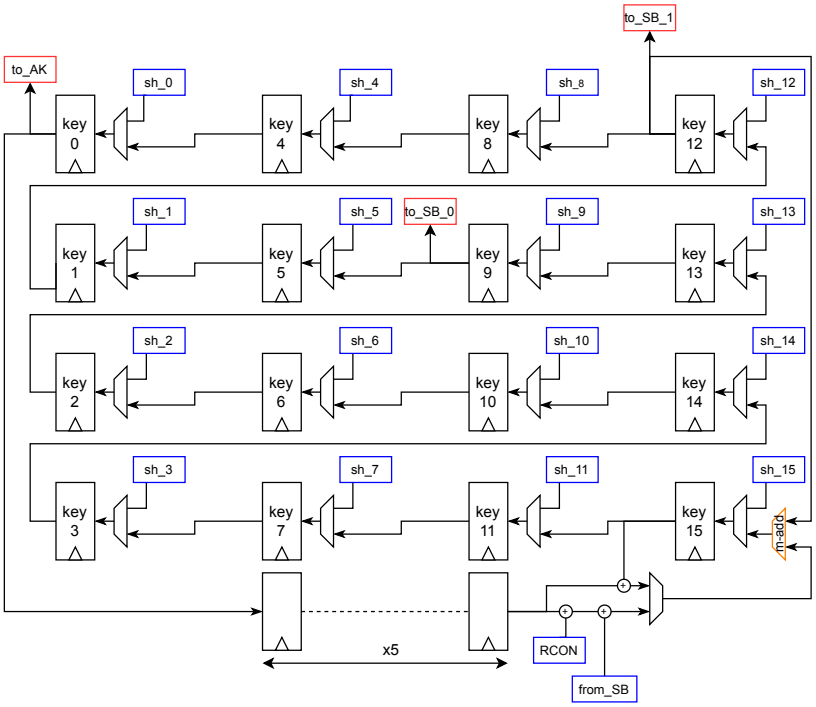

Fig. 5. 8-bit implementation: architecture of the key holder.

3.3 32-bit serial implementation

The 32-bit serial architecture is also organized around two main blocks containing the values of the state and the key. As for the 8-bit serial architecture, the data is stored in 16 register blocks, each of them holding a shared byte value. A round is computed serially, 32 bits per 32 bits. To do so, four S-boxes are instantiated in the architecture. The latter are fed either with the result of the `AddRoundKey` layer (represented in Figure 7) or by the `KeyHolder` (in order to perform the key schedule), as controlled by the `toSBi` MUXes (see Figure 6). Finally, a dedicated combinatorial logic block is used to compute the masked `MixColumns` operation on a full column.

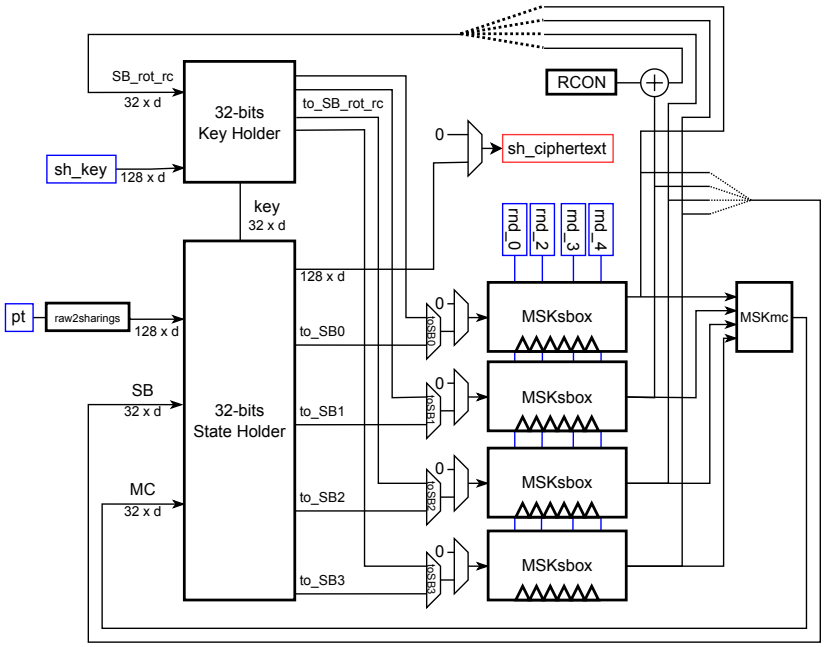

Fig. 6. Global Architecture of the 32-bit serial implementation.

Both the `StateHolder` and the `KeyHolder` are organized as four shift registers (see Figures 7 and 8). The `AddRoundKey` operation is performed with combinatorial logic before sending the signals `to_SBi` to the S-boxes. Under this mode of computation, the MUXes `loopi` (depicted in green in Figure 7) are routing the data in a loop manner over the shift registers. In such a way, 4 cycles are sufficient to feed the S-boxes with the full state. The positions of the `to_SBi` signals (i.e., on byte indexes 0, 5, 10 and 15) have been carefully chosen to perform the `ShiftRows` operation at the same time as feeding the S-boxes without using a dedicated clock cycle. For full round computations (i.e., all except the last), the output of the S-boxes is directly routed to the `MSKmc` block before entering back

the `StateHolder`. In the last round, the MUXes `mc-i` are used to bypass the `MixColumns` layer. The last `AddRoundKey` operation is performed by enabling the `loopi` MUXes as well as the key addition.

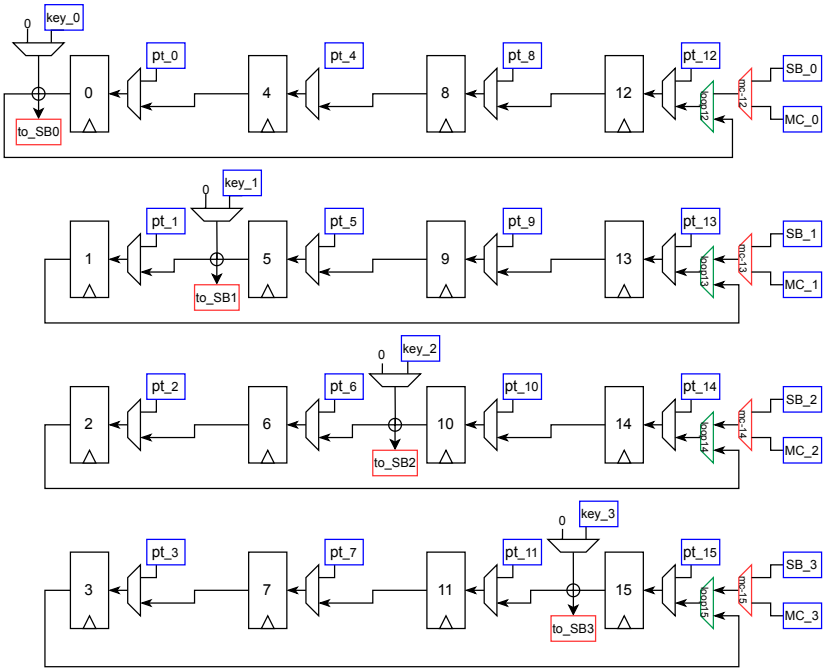

Fig. 7. 32-bit implementation: architecture of the state holder.

The key schedule is performed in parallel to the round computation, by interleaving appropriately the feeding of the S-boxes with key bytes. More precisely, the fourth column of the key is sent to the S-boxes during the last cycle of the `MixColumns` computation of the previous round. We take advantage of the S-box latency to perform the key schedule algorithm in the time lap required to compute the `MixColumns` operation. For this mechanism to work properly, the encryption execution starts by feeding the S-boxes with key material during the first cycle. The rotation operation is performed via direct routing and does not require dedicated clock cycle as shown in Figure 6. The addition of the round constant is performed in parallel to the rotation. Once the `SubByte` operation is performed, the new key value is computed column per column during four cycles. To this end, the dedicated MUXes `addi` (depicted in red in Figure 8) are configured in such a way that `SB_rot_rc` (the output of the S-box) is added to the first column of the round key. Configuring the `addi` MUXes, the other columns of the new round key are then computed by serially adding them to the new column. The latency of a full encryption is therefore $1 + 10 \cdot (4 + 6) + 4 = 105$ cycles.

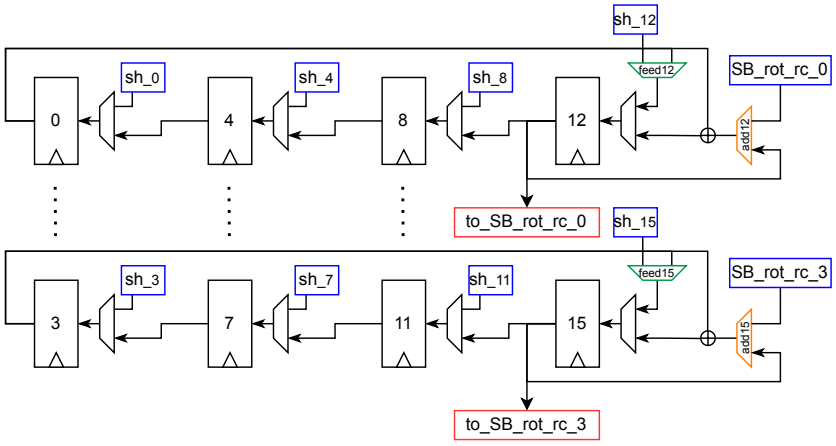

Fig. 8. 32-bit implementation: architecture of the key holder.

3.4 128-bit serial implementation

Compared to the 8-bit serial and 32-bit serial implementations, the organization of the architecture of the 128-bit serial implementation is more straightforward. As depicted in Figure 9, it contains all the logic necessary to operate on a full 128-bit state, as well as the logic required to perform the key scheduling in parallel to the round computation. In more details, the state and the key sharing `sh_key` are stored in a register. After the `AddRoundKey` layer, the state in directly routed to the `SubByte` logic composed of 16 S-box instances. Directly following the latter, the `ShiftRows` operation is performed at the routing level before entering the logic for the `MixColumns` operation, itself composed of four independent blocks, each operating on one individual column of the shared state. Depending on the round counter, MUXes are used to feed back to the state register the results either after the `ShiftRows` operation, or after the `MixColumns` operation.

For the key scheduling, four S-box instances are used order to process the 4 bytes of the fourth column of the key in parallel. As for the `ShiftRows` layer, the rotation occurring is performed as the routing level, requiring no additional logic. Finally, combinatorial XOR gadgets are placed at the output of the S-box in order to finalize the round key. The resulting round key is fed back to the key holder register. For this 128-bit architecture, it therefore follows that the latency of a full encryption is $10 \cdot (6 + 1) = 70$ cycles.

4 Implementation results

In this section, we present the post-synthesis ASIC implementation results obtained for the different architectures and compare them with results obtained with AGEMA generated implementation. All the syntheses have been performed using Genus Synthesis Solution (Cadence) using the TSMC-N65 design kit. We

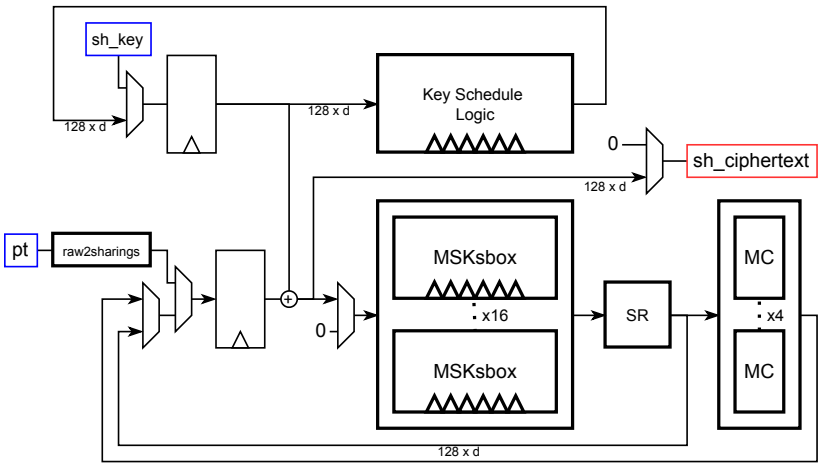

Fig. 9. Global Architecture of the 128-bit serial implementation

complete these results with a brief discussion of physical security guarantees based on both formal verification and experimental leakage assessment.

4.1 Masked S-box Implementations

Starting with the S-box implementation results depicted in Table 1, our implementation performs a computation in 6 cycles, which is 25% faster than the 8 cycles obtained by AGEMA for both pipeline and clock-gating synchronization strategies. This is a direct effect of the input ports switching for AND2-HPC2 gadgets described in Section 3.1. Compared to the S-box with pipelined synchronization, our implementation is approximately 20% smaller (see Table 2). The first reason for this is the reduced number of synchronization registers thanks to the lower latency. A second reason for this difference lies in the the implementation of the AND2-HPC2 gadget used by AGEMA: they require more registers than what is required, i.e., $4d(d-1)+3d$ per multiplication gadget, while our implementation (which comes from fullVerif's library) only instantiates $2d+7/2\times d(d-1)$ registers per multiplication. Since both implementations are fully pipelined, they reach a throughput of one S-box evaluation per cycle.

The comparison with the S-box generated with the clock-gating synchronization strategy is interesting. On the one hand, it avoids all synchronization registers, but on the other hand, clock gating logic is added, and the multiplication gadgets are more expensive, as discussed above. The increased cost of the the multiplication gadgets dominates for $d \geq 3$, while for $d = 2$ the AGEMA implementation is a bit smaller than ours. In addition to having a higher latency, the AGEMA clock-gated S-box has a much lower throughput.

Instance	Share [count]	Seq. area [GE]	Area [GE]	Latency [cycle]	Throughput [exec/cycle]
AGEMA c.g.*	2	2009	2972	8	0.125
AGEMA c.g.*	3	4625	6822	8	0.125
AGEMA c.g.*	4	8329	12 290	8	0.125
AGEMA pipe.†	2	3024	3981	8	1
AGEMA pipe.†	3	6168	8360	8	1
AGEMA pipe.†	4	10 400	14 356	8	1
New	2	2273	3213	6	1
New	3	4831	6705	6	1
New	4	8354	11 515	6	1

* Clock-gating synchronization mechanism.
† Pipeline synchronization mechanism.

Table 1. ASIC TSMC-N65 S-box implementation results (post-synthesis).

Instance	Shares	Area [%]*	Latency [%]*	Area [%]†	Throughput [%]†
New	2	+8	−25	−19	0
New	3	−2	−25	−20	0
New	4	−6	−25	−20	0

* Compared to AGEMA with clock-gating synchronization.
† Compared to AGEMA with pipeline synchronization.

Table 2. ASIC TSMC-N65 S-box implementation results comparison (post-synthesis).

4.2 Masked AES Implementations

The previous differences are amplified when considering a full encryption core, as shown in Tables 3 and 4. Starting with the 8-bit serial architecture, our new implementation has a 6.4 times lower latency than the ones generated with AGEMA for both synchronization mechanisms. This is due to the fact that the hand-crafted control mechanism takes full advantage of the pipeline architecture of the S-box without adding superficial pipeline levels, speeding up the computation of the `SubByte` operation by processing up to 6 bytes of the same AES encryption in parallel instead of 1. Regarding the throughput, the 8-bit pipelined AGEMA implementation is better then ours: it is able to optimally use the S-box pipeline while our implementation dedicates cycles to other operations.

As for the area, our new implementation is roughly 2.6 times smaller than the one generated with pipeline synchronization. This difference is mainly due to the very large number of registers needed to achieve a complete round pipeline for the full AES state and round keys in the AGEMA pipelined implementation. Compared to the implementations synchronized with clock-gating, the area of

Instance	Share [count]	Seq. area [GE]	Area [GE]	Latency [cycle]	Throughput [exec/cycle]
AGEMA 8-bit c.g.*	2	4068	9356		
AGEMA 8-bit c.g.*	3	7678	16 319	2043	0.000 49
AGEMA 8-bit c.g.*	4	12 375	24 919		
AGEMA 8-bit pipe.†	2	25 055	30 338		
AGEMA 8-bit pipe.†	3	38 667	47 302	2043	0.0044
AGEMA 8-bit pipe.†	4	53 382	65 921		
New 8-bit	2	4790	10 634		
New 8-bit	3	8571	17 591	322	0.0031
New 8-bit	4	13 315	25 915		
New 32-bit	2	11 139	19 598		
New 32-bit	3	22 399	36 776	105	0.0095
New 32-bit	4	37 511	59 217		
AGEMA 128-bit c.g.*	2	40 198	63 613		
AGEMA 128-bit c.g.*	3	92 909	143 100	99	0.010
AGEMA 128-bit c.g.*	4	167 376	254 948		
AGEMA 128-bit pipe.†	2	86 317	109 725		
AGEMA 128-bit pipe.†	3	161 789	211 969	99	0.091
AGEMA 128-bit pipe.†	4	259 307	346 880		
New 128-bit	2	47 597	73 699		
New 128-bit	3	99 859	148 129	70	0.10
New 128-bit	4	171 274	249 011		

* Clock-gating synchronization mechanism.
† Pipeline synchronization mechanism.

Table 3. ASIC TSMC-N65 AES encryption implementation results (post-synthesis).

our new architecture is slightly higher, but are of the same order of magnitude. In more details, the overheads observed for the 8-bit implementations vary from 14 % to 4 % (for 2, 3 and 4 shares) and are caused by the synchronization registers used in our implementations. In particular, besides the registers introduced in the S-box, $48d$ registers are used in the global architecture ($40d$ in the key schedule, as shown in Figure 5, and $8d$ in the global datapath, as depicted in Figure 3).

For the 128-bit implementations, we also achieve a latency reduction and a throughput increase compared to the AGEMA implementations, but the gain is much smaller than in the 8-bit case. Our implementation is slightly larger than the AGEMA clock-gated one, while the pipelined one is larger than ours.

Finally, we also report the result of a 32-bit implementation (an architecture that was not given in [KMMS22]). It performs a full encryption in 105 cycles, which is similar to what is achieved by the round-based implementation gener-

Instance	Shares	Area [%]*	Latency [%]*	Area [%]†	Throughput [%]†
New 8-bit	2	+14	−84	−65	−30
New 8-bit	3	+8	−84	−63	−30
New 8-bit	4	+4	−84	−60	−30
New 128-bit	2	+16	−30	−33	+10
New 128-bit	3	+4	−30	−30	+10
New 128-bit	4	−2	−30	−28	+10

* Compared to AGEMA with clock-gating synchronization.
† Compared to AGEMA with pipeline synchronization.

Table 4. ASIC TSMC-N65 AES enc. implem. results comparison (post-synthesis).

ated by AGEMA. On top of that, its area turns out to be significantly lower than what is achieved for round-based architectures, making it an interesting alternative when the area vs. latency trade-off is considered.

4.3 Physical Security

We use a two-step methodology to validate the $d-1$-th order security.

As a first step, we use the fullVerif tool to validate that the implementation satisfies the HPC conditions [CGLS21]. Those conditions guarantee glitch-robust probing security, but give no assurance against transition leakage. To also take into account the latter, we relied on the "Optimized composition approach" presented in [CS21], which ensures security against both glitches and transitions. In our context, this approach requires the insertion of a pipeline bubble in the S-box between each AES rounds. In such pipeline bubbles, the data processed by the S-boxes should not depend on any sensitive input.

We extended fullVerif to check this property. Concretely, using the identification of non-sensitive pipeline bubbles, the new verification algorithm builds groups of executions that are not separated by such bubbles for each S-box (in full genericity, for each PINI but non-affine gadget). Then for each execution in each group, it checks that none of the input sharings is computed using an output sharing of a gadget in the same group, which is implemented as a path existence check in the computation graph.

The second step leverages the TVLA testing methodology in order to validate practical security and rule-out issues that cannot be caught by fullVerif such as those due to (post-)synthesis optimizations. Namely, we used fixed vs. random T-tests as a preliminary leakage assessment [GGJR+11]. The measurements were performed on a Sakura-G board running at 6 MHz. We conducted the acquisitions with a PicoScope 5244D sampling at 500 MS/s with 12-bit resolution. As in [KMMS22], each randomness bit was generated on-the-fly by a randomly seeded independent instance of the 31-bit maximum length LFSR presented in [Alf98]. As shown in Figure 10 for the 8-bit implementation, leakage

can be observed starting at second order with 2 shares (1 M traces for both test orders) and at the third order with 3 shares (10 M traces for all test orders).

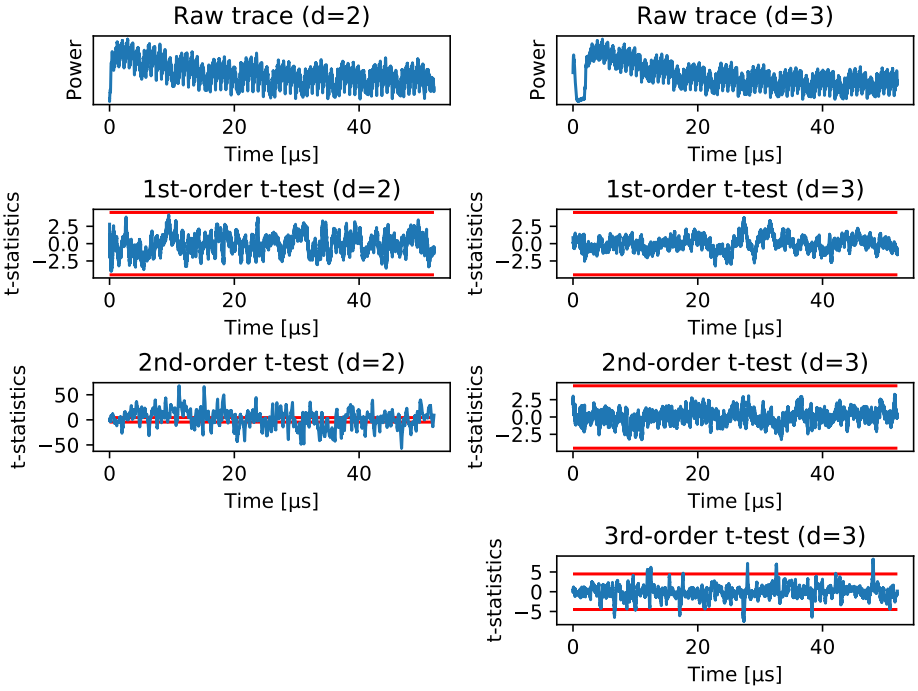

Fig. 10. Fixed vs random T-test results (AES-128, 8-bit serial architecture).

5 Conclusion

While the automated generation of masked hardware performed by AGEMA is a significant advance towards improving the usability of masking for non-expert designers, our results show that there remains room for further performance optimizations and therefore raise the question whether our handcrafted improvements could be integrated in AGEMA. For example, when dealing with asymmetric gadgets like the ones developed in the HPC2 scheme, a first approach could be to implement the optimization method based on the input switching described in Section 3.1. We believe that such a feature should be easily implemented in AGEMA by working with high-level netlists containing information about the timings of signals' propagation. Such timings may be either provided in the HDL by the use of annotations (as done for the fullVerif tool) or hard-coded in the tool. As another easy-to-integrate option, relying on the AND2

gadget implementation proposed in [CGLS21] could also be considered, as the latter requires less registers and enables asymmetric optimizations.

By contrast, which strategy to follow in order to automatically take advantage of the pipeline nature of the masked architectures is less clear. Indeed, our architecture optimizations are based on a deep understanding of the operations to be performed (e.g., how the S-box can be re-ordered). Automating them would *a minima* require to analyze the precise control logic of the core.

Finally, we note that besides the aforementioned performance optimizations, implementing the automated generation of annotations compliant with fullVerif may be an interesting addition to AGEMA, in order to facilitate the verification of compositional properties that generated masked implementations exploit.

Acknowledgments

Gaëtan Cassiers and François-Xavier Standaert are respectively Research Fellow and Senior Associate Researcher of the Belgian Fund for Scientific Research (FNRS-F.R.S.). This work has been funded in part by the ERC project number 724725 (acronym SWORD) and by the Walloon Region through the Win2Wal project PIRATE (convention number 1910082).

References

Alf98. Peter Alfke. Efficient shift registers, lfsr counters, and long pseudo-random sequence generators. *http://www. xilinx. com/bvdocs/appnotes/xapp052. pdf*, 1998.

BBC+19. Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *ESORICS (1)*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

BBD+15. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.

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.

BDM+20. Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In *EUROCRYPT (3)*, volume 12107 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.

BGI+18. Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

BMP13. Joan Boyar, Philip Matthews, and René Peralta. Logic minimization techniques with applications to cryptology. *J. Cryptol.*, 26(2):280–312, 2013.

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.

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.

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

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

CRB+16. Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 shares in hardware. In *CHES*, volume 9813 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2016.

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

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

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

GGJR+11. Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

GMK17. Hannes Groß, Stefan Mangard, and Thomas Korak. An efficient side-channel protected AES implementation with arbitrary protection order. In *CT-RSA*, volume 10159 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2017.

KMMS22. David Knichel, Amir Moradi, Nicolas Mullër, and Pascal Sasdrich. Automated generation of masked hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022.

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.

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

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

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