Automated Generation of Masked Hardware

David Knichel*, Amir Moradi*, Nicolai Müller* and Pascal Sasdrich*

Ruhr University Bochum, Horst Görtz Institute for IT Security, Bochum, Germany
firstname.lastname@rub.de

Abstract. Masking has been recognized as a sound and secure countermeasure for cryptographic implementations, protecting against physical side-channel attacks. Even though many different masking schemes have been presented over time, design and implementation of protected cryptographic Integrated Circuits (ICs) remains a challenging task. More specifically, correct and efficient implementation usually requires manual interactions accompanied by longstanding experience in hardware design and physical security. To this end, design and implementation of masked hardware often proves to be an error-prone task for engineers and practitioners.
As a result, our novel tool for *automated generation of masked hardware* (AGEMA) allows even inexperienced engineers and hardware designers to create secure and efficient masked cryptograhic circuits originating from an unprotected design. More precisely, exploiting the concepts of *Probe-Isolating Non-Interference (PINI)* for secure composition of masked circuits, our tool provides various processing techniques to transform an unprotected design into a secure one, eventually accelerating and safeguarding the process of masking cryptographic hardware. Ultimately, we evaluate our tool in several case studies, emphasizing different trade-offs for the transformation techniques with respect to common performance metrics, such as *latency*, *area*, and *randomness*.

Keywords: Side-Channel Analysis · Masking · Hardware · Composable Gadget

1 Introduction

Side-Channel Analysis (SCA) has not lost any of its topicality and remains a major threat to security-critical implementations, even after more than two decades of intensive research since its seminal description. In the wake of this lasting discovery [Koc96, KJJ99], it has been admittedly recognized that secure implementation of cryptographic algorithms is a challenging tasks, given that an adversary can observe and measure physical effects, such as timing [Koc96], power consumption [Koc96, KJJ99], electromagnetic (EM) radiations [GMO01], or temperature and heat dissipation [HS13], in order to infer sensitive information during execution. However, in the course of time, different classes of counteractive measures have emerged amongst which *masking* [CJRR99], based on concepts of *secret sharing*, prevails due to its formal and sound security foundation.

Over the last years, many different hardware masking variants and schemes have been proposed [ISW03, NRR06, RBN+15, GMK17, GM18], constantly improving efficiency and security. Unfortunately, experience has shown that new schemes often have a short retention time, mostly due to inaccuracies and design flaws [MMSS19]. However, even for schemes that stand the test of time, correct and secure implementation remains an enormous engineering challenge. As a matter of fact, even with longstanding experience

*Authors list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf

and expertise in hardware security and design of masked hardware, correct physical instantiation of masking schemes is a delicate and error-prone task. Evidently, unconsidered and unintentional physical effects, e.g., *glitches* [MPG05], *transitions* [CGP+12, BGG+14], or *coupling* [CBG+17], and implementation defects due to architectural conditions, e.g., *parallelism* [BDF+17] or *pipelining* [CGD18], can render a theoretically secure scheme practically insecure.

As a consequence, a new line of research emerged, investigating the masking of atomic and reusable components, often considered as *gadgets* in literature, to limit the engineering complexity and error susceptibility. In this regard, a great deal of attention has been devoted to the construction of secure gadgets for basic non-linear operations (e.g., AND), allowing to efficiently mask any digital logic circuit given its AND-XOR representation. However, the continuous progress in this domain is inevitably associated with fundamental research and advancements in the realm of formal security definitions and adversary models. More specifically, the formal and abstract Ishai-Sahai-Wagner (ISW) d-probing model [ISW03] is consulted prevalently to reason about security of masked circuits in the presence of side-channel adversaries.

Unfortunately, research has shown that security in this simple model does not imply secure composition of gadgets [CPRR13]. As a consequence, advanced security notions and properties are essential to reason about the *composability* of masked gadgets. In a first attempt, Barthe et al. [BBD+15] introduced the notion of Non-Interference (NI), allowing to verify the composability of gadgets based on the concept of *perfect simulation* of joint probability distributions. However, due to disregarded effects which later became know as *probe propagation* [CS20], the notion of NI is deficient and has been complemented by the notion of Strong Non-Interference (SNI) shortly afterwards [BBD+16]. Most recently, Cassiers and Standaert [CS20] introduced the notion of Probe-Isolating Non-Interference (PINI) enabling more efficient compositions with respect to *multi-input, multi-output* gadgets and *trivially secure* linear operations.

Now, provided with such sound and formal security and composability notions, hardware designers are able to construct secure circuits more easily. However, transforming an entirely unprotected design into a secure circuits remains a complicated and mostly manual process, even when endowed with an adequate set of secure and composable gadgets.

Contribution. In this work, we present our novel, and open-source, software tool for *automated generation of masked hardware* (AGEMA), enabling engineers and hardware designers of any level of expertise to easily generate masked hardware circuits starting from a simple but unprotected design. Utilizing different methods for processing the given netlist of a design and supporting arbitrary masked gagdets, AGEMA offers high flexibility with respect to the security level, required randomness, latency, and area overhead and consequently gives designers the ability to configure AGEMA to their particular needs. Exploiting the essential security notions for secure composability, the final designs are provably secure and assuredly free of any heuristics, implementation defects, and design mistakes. As a consequence, our tool facilitates and accelerates the process of masking digital logic circuits while in the same vein, the quality and security of the resulting designs are increased.

We should highlight that, up to now, various tools have been developed to automatically generate masked software implementations. The examples include [BRB+11, BRN+15] and the recently-introduced ones Tornado [BDM+20] and Rosita [SSB+21]. To the best of our knowledge AGEMA is the only one which is dedicated to hardware implementations and the security of its generated circuits are based on the PINI security notion which guarantees to maintain the security through composition.

Outline. We start by summarizing our notations and all necessary theoretical concepts in Section 2, before elaborating the fundamental methodology of AGEMA in Section 3. This includes a brief summary of existing types of Hardware Private Circuit (HPC) and a detailed explanation for the supported processing and transformation methods of the original netlist. In Section 4, we examine AGEMA with an extensive list of case studies on a broad variety of hardware implementations of different block ciphers and compare the results, i.e., the protected designs, with respect to common performance metrics such as required fresh randomness, latency degradation, and area overhead. Before we conclude our work, we provide reasoning behind the security of HPC circuits constructed by AGEMA in Section 5, and give result of experimental analyses, i.e., Test Vector Leakage Assessment (TVLA), on some exemplary designs as the outcome of application of AGEMA.

2 Basics

2.1 Notations

Let us denote functions using sans-serif fonts, e.g., f for Boolean functions and F for vectorial Boolean functions. Next, we denote single-bit random variables in $\mathbb{F}_2$ by lower case letters like x, and vectors by uppercase letters X while sets of random variables are given in bold $\mathbf{X}$. Further, we use subscripts like x_i to indicate elements within a vector or a set while superscripts are used to denote (randomized) shares of random variables, e.g., X^j. As a special case, the set of all shares of each random variable in $\mathbf{X}$ is denoted as $Sh(\mathbf{X})$.

2.2 Boolean Masking

Masking is based on secret sharing and has proven to be well suited for hardware implementations as a countermeasure against side-channel attacks. In *Boolean masking*, a sensitive variable $X \in \mathbb{F}_n$ is split into $s \geq 2$ randomized shares $(X^0, X^1, \ldots, X^{s-1}) \in \mathbb{F}_n^s$, such that $X = \bigoplus_{i=0}^{s-1} X^i$. Usually this sharing is initially achieved by sampling $X^i \overset{\$}{\leftarrow} \mathbb{F}_n$ for all $0 \leq i < s-1$ and calculating $X^{s-1} = \left(\bigoplus_{i=0}^{s-2} X^i\right) \oplus X$. Eventually, instead of performing logic operations on the sensitive value X, they will be performed on the (randomized) shared representation of X, i.e., $X^0, X^1, \ldots, X^{s-1}$.

2.3 Probing Security

In order to abstract and formalize the behavior of a masked circuit and the adversarial capabilities to extract information from the underlying circuit, several models have been introduced over time, aiming to achieve different trade-offs between simplicity and accuracy. In the d-probing model, firstly introduced by Ishai et. al in [ISW03], an adversary is granted the ability to observe the distribution over up to d wires of a given circuit. To achieve d-probing security for a masked circuit, any adversary in conformity with this model should not be able to learn anything about the processed sensitive value X.

Definition 1. A masked circuit C is said to achieve d-probing security iff every (joint) distribution over up to d wires is statistically independent of any sensitive value X.

Definition 1 directly implies that splitting any sensitive variable into at least $d+1$ shares is necessary to achieve d-probing security. In the context of masking, d is also referred to as the *security order* of a given masked circuit.

Robust Probing Model and Glitch-Extended Probes Since the traditional probing model is limited to software implementations due to its inability to capture physical defaults occurring in hardware implementations, e.g., *transitions* (memory recombinations), *glitches* (combinational recombinations), or *coupling* (routing recombinations), the *robust* probing model was introduced in [FGP+18], aiming to consider and model these defaults accurately while being sufficiently simple in order to enable efficient verification of masked designs. In contrast to the traditional probing model, where the value of a wire is always assumed stable during evaluation and where no dedicated synchronization elements, i.e., registers, exist, the robust probing model loosens this assumption by introducing registers and so-called extended probes. Here, a single probe can be extended to additionally capture leakage caused by physical defaults like data transitions at registers, glitches in combinational logic, and coupling of adjacent wires.

In particular, *glitches* are switching activities of wires caused by different delays of signals contributing to their intended values. These glitches enable a probe on a single wire to observe not only the field element of its driving gate, but possibly a recombination of signals contributing to its combinatorial value. Hence, in order to capture these effects, *glitch-extended* probes were introduced. Here, a single probe on a wire is assumed to capture the leakage of the joint distribution over every stable signal contributing to the calculation of the probed wire. As a result, given the glitch-extended probing model, every probe on a wire is replaced by the set of probes placed on the register outputs and primary inputs that contribute to the observed wire, i.e., there exists a path from a stable source to the current probe position.

2.4 Composable Masking Schemes

Especially for higher security orders d and more complex functions, it is hard to find efficient masked representations for circuits to become provably d-probing secure, as the number of possible probe combinations increases with the security order and the complexity of a circuit.

Following a divide-and-conquer approach, composable gadgets were introduced as a remedy to directly derive masked representations of large functions. Composable gadgets are masked circuits realizing small and atomic logic functions, like a simple AND or OR gate. Fundamentally, these gadgets fulfill certain properties that imply probing security when composed to a larger circuit. This way, the problem of finding secure masked realizations of large functions is reduced to the task of finding gadgets realizing small functions with certain properties.

Probe Propagation and Composability Notions To understand favorable properties for gadgets in order to achieve secure composability, we explain the concept of *probe propagation*, which was firstly introduced in [CS20] and defines the information a probe can access and how this access to information is propagated throughout the circuit. Generally, a (glitch-extended) probe is said to *propagate* into a wire if this wire is needed to perfectly simulate each observation of the probe, i.e., in order to compute the underlying probability distribution. Now, to achieve composability of a gadget, propagation of internal probes and output probes needs to be restricted to a subset of the input wires of the gadgets. These constraints on gadget level have to guarantee that all possible probes in a composed circuit only propagate in a subset of the initial sharing of an input value and not into all of them. After Non-Interference (NI) was proven to be insufficient to offer composability, Strong Non-Interference (SNI) was proposed which further restricts probe propagation and was originally restricted to single-output gadgets. In [CS20], Cassier and Standaert showed that the scope of the original definition can be extended to cover multiple-output gadgets as well, but at the same time unveiled issues of SNI with respect to the extent of required entropy and circuit area. Eventually, Probe-Isolating Non-Interference (PINI) was

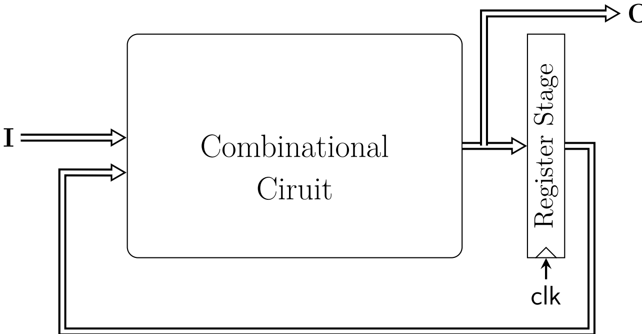


Figure 1: General schematic of a sequential circuit.

introduced in the same work as an elegant way to guarantee composability at any security order. Similar to Domain Oriented Masking (DOM), share domains were introduced and any probe was restricted to only propagate within its own share domain, enabling trivial implementation of linear functions on the one hand and direct composition of gadgets on the other.

2.5 Formal Verification

Several tools have been published for the purpose of formally verifying the security and composability characteristics of masked hardware circuits [BGI+18a, BGI+18b, CGLS21, BBC+19, KSM20]. They all support different varieties of security and composability notions while working on different abstraction levels. We choose SILVER [KSM20] for performing all verification in this work, due to its unique support of checking composability under the PINI notion in the glitch-extended probing model.

2.6 Combinational and Sequential Circuits

Combinational circuits are digital circuits where the output is a pure function of the primary inputs and where no synchronization elements and clock signal exist. In contrast, in a *sequential circuit*, a sequence of data, synchronized by a clock signal, is processed. A sequential circuit may contain a feedback loop through registers, such that the output at any given time is not only a function of the primary input but of previous outputs as well. A schematic overview is depicted in Figure 1. We want to stress that this structure offers a unique representation of any given logical circuit without combinational loops that possibly contains multiple register stages. More precisely, every given circuit without a combinational loop can be represented as a *sequential circuit* that follows the structure shown in Figure 1, where all synchronization elements are packed into the main register stage, the combinational circuit receives the primary input and the outputs of the main register, and the primary output is taken from the combinational circuit. Note that, this illustrates a Mealy machine, which covers Moore machines as well [Mea55].

3 Technique

In this section, we gradually present the technical details of the procedure which AGEMA follows to generate a secure masked implementation from the given unprotected implementation. To this end, we first review the masking schemes which are currently supported by AGEMA.

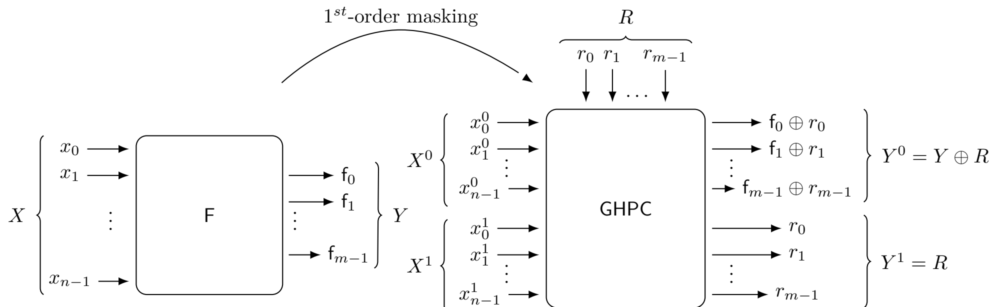


Figure 2: The GHPC concept of transforming any vectorial Boolean function into a first-order secure and composable gadget.

3.1 PINI Hardware Gadgets

As PINI offers trivial composition of hardware gadgets in the robust probing model, we restrict our examples to known gadgets fulfilling this security notion. However, we like to stress that AGEMA offers a generic framework to dynamically substitute circuit parts with masked gadgets and is not restricted to any specific type of gadgets. Recently, several gadgets have been proposed that fullfill the PINI notion in the robust probing model. As they differ with respect to the logic function they realize, the fresh randomness they require and their latency, we will describe and compare them in the following, where the number of required fresh randomness is denoted by r and the number of added register stages (i.e., the latency) by l.

3.1.1 HPC1

HPCs – proposed by Cassier et al. in [CGLS21] – realize 2-input AND gadgets composable under the PINI notion in the robust probing model and are generic for arbitrary security orders. The authors introduced HPC1, which simply consists of a DOM-AND where the sharing of one input is refreshed. Hence, the added number of register stages is $l = 2$. The DOM-AND needs $d(d + 1)/2$ bits of fresh masks for any given security order d. However, since the mask refreshing is expected to be SNI, the required number of additional fresh masks is identified through the table $[1, 2, 4, 5, 7, 9, 11, 12, 15, 17]$ for security order $d \leq 10$.

3.1.2 HPC2

Cassier et al. further proposed another construction for an AND gadget, HPC2, requiring $r = d(d + 1)/2$ fresh randomness and $l = 2$ added register stages for any security order d.

3.1.3 GHPC

Generic Hardware Private Circuits (GHPCs), introduced in [KSM21], allow the construction of gadgets realizing any (vectorial) Boolean function but are currently limited to first-order security. Here, $l = 2$ is the number of added register stages, and the required number of fresh masks is $r = 1$ per output bit, regardless of the Boolean function the gadget realizes.

The concept of GHPC is depicted in Figure 2. Every input is split into two shares while the result of the gadget is simply the coordinate function f_i blinded by fresh randomness r_i for every $0 \leq i < n$.

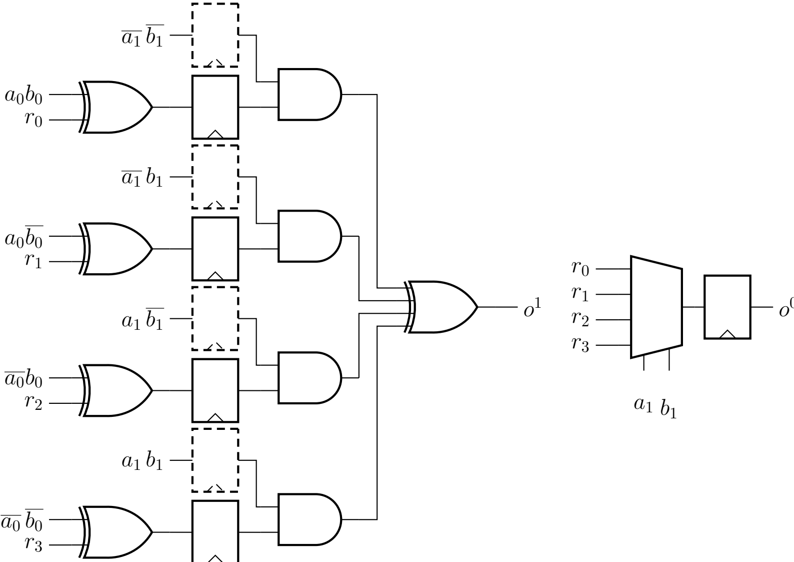


Figure 3: First-order $\mathsf{GHPC_{LL}}$-AND realizing $o = ab$, dashed registers are optional for a pipeline design.

3.1.4 GHPC$_{\mathsf{LL}}$

In [KSM21], the authors further introduced $\mathsf{GHPC_{LL}}$, a low-latency variant of GHPC which requires only one register stage to compute any vectorial Boolean function but requires 2^n fresh random bits per output for a Boolean function with n inputs.

A simple 2-input AND gadget realized as a $\mathsf{GHPC_{LL}}$ is shown in Figure 3. This is the only known composable AND gadget with a latency of a single clock cycle.

3.1.5 HPC-MUX

Designing an efficient shared version of a 2-input multiplexer offering security under the PINI notion is straightforward as it can be directly derived by using a single HPC gadget realizing a 2-input AND. The Boolean function f, describing a multiplexer, where one of two inputs $a \in \mathbb{F}_2$, $b \in \mathbb{F}_2$ is selected by $s \in \mathbb{F}_2$ can be rewritten as $f = sa \oplus \bar{s}b = s(a \oplus b) \oplus b$. Realizing a composable, shared multiplexer for arbitrary security orders is hence possible by using two trivial XOR operations and a single HPC-AND gadget. As a result, the randomness requirements and the latency is inherited from the HPC-AND gadget initiation, i.e., whether an $\mathsf{HPC1}$, $\mathsf{HPC2}$, GHPC, or $\mathsf{GHPC_{LL}}$- AND is instantiated, where the two latter cases are restricted to constructing first-order secure designs only.

3.2 Procedure

As the main goal is the conversion of an unprotected implementation to a masked one, we first have to analyze the netlist of the unprotected implementation. In other words, the unprotected implementation should be first synthesized by a synthesizer, e.g., Design Compiler [Inc] or Yosys [Wol]. The resulting Verilog netlist[1] is then given to AGEMA. Note that AGEMA has a custom library file, where the user should specify the details of each cell, e.g., their input and output ports. Therefore, the synthesizer should also be

[1] This can be set in a script executed by the synthesizer to generate a Verilog netlist as the result of the synthesis.

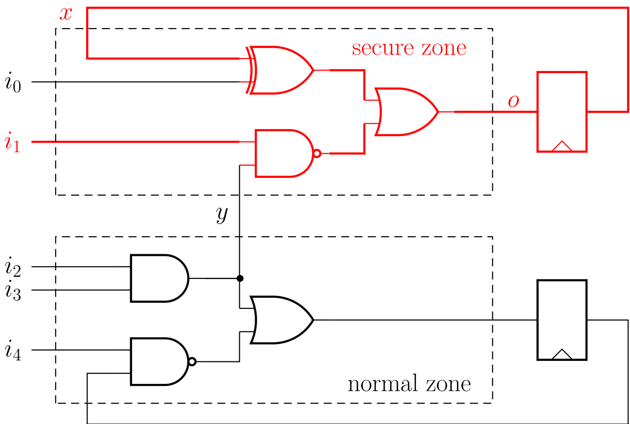


Figure 4: Exemplary circuit after the propagation of secure signals. i_1 is the only primary input marked as secure. The red signals and cells indicate the parts which should be masked.

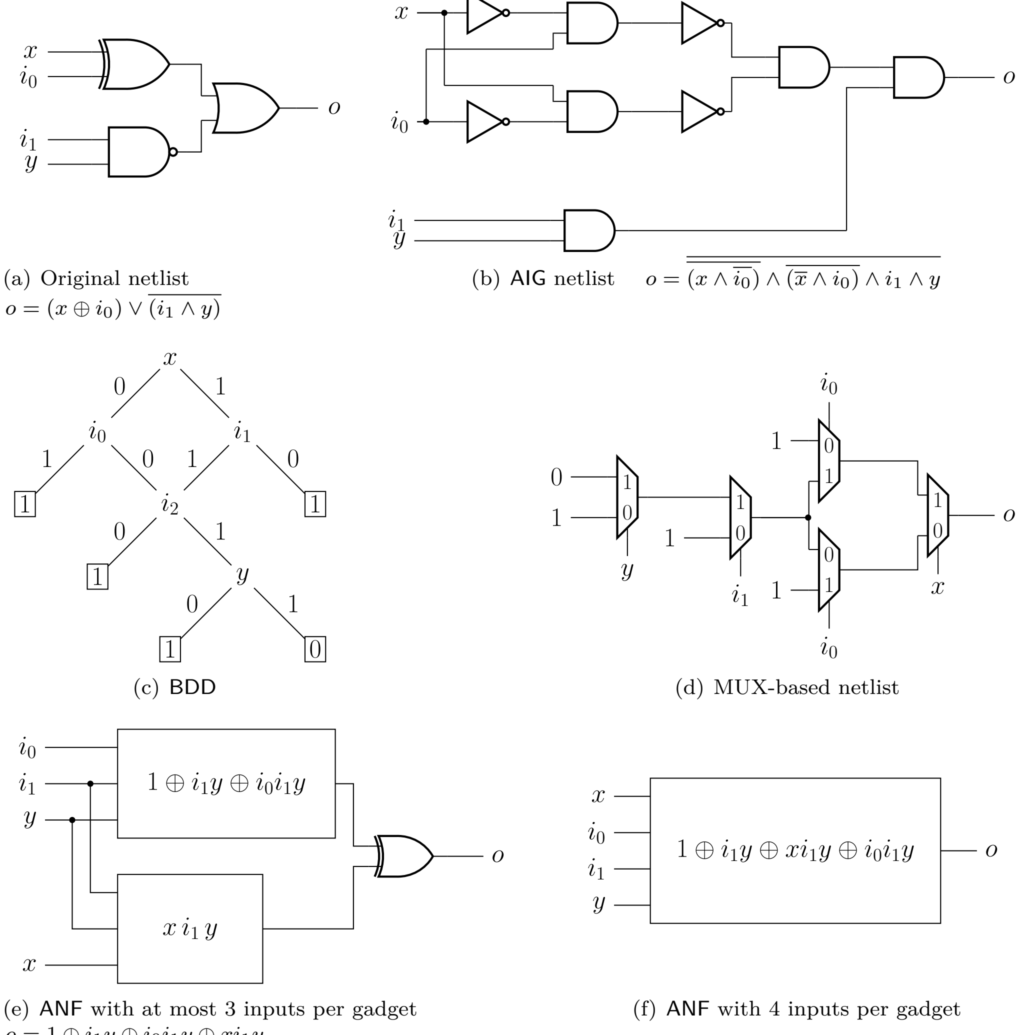
set to just use a resisted list of cells to generate the Verilog netlist, typically only NOT, 2-input AND, NAND, OR, NOR, XOR, XNOR, MUX, and D flip-flops.

Then, as a first step, AGEMA builds a graph based on the given Verilog netlist, and represents the circuit following the concept given in Section 2.6, i.e., a combinational circuit and a single main register stage. Naturally, not necessarily all parts of the given design should be masked, e.g., the control logic should be excluded. Further, the designer may desire to not mask the key and the key schedule, for example if protection against profiling attacks targeting the key schedule is excluded (for such cases, see [MS16, PMK+11, UHA17, SM20], where no key masking is applied). In AGEMA, this is supported by setting the attribute of the primary input signals. If a signal is annotated as *secure*, in the resulting masked circuit, it should be provided in a masked form with $d+1$ shares, while d is also defined by the user. This also helps to identify the control and handshaking signals which should not be masked.

Hence, the next step of the process is to identify which parts of the given circuit should be masked. If an input of a cell is marked as secure, its output should also be marked secure. Therefore, in a recursive manner, we propagate the secure signals through all cells of the circuit until no new signal is marked as secure. Note that this includes the main registers and their role as the input to the combinational circuit. Afterwards, we split the circuit into two parts: the *secure zone* and the *normal zone*. Figure 4 shows a simple example.

3.3 Processing Methods

The next step is to construct the masked variant of the secure zone. Apart from the fact that different masking schemes are supported (see Section 3.1), we can process the secure zone and build a more optimized netlist in favor of the selected masking scheme. To this end, AGEMA supports five different processing methods explained below with an example for each method in Figure 5 which is based on the secure zone identified in Figure 4. As a side note, all processing methods can be freely combined with all supported masking schemes, except ANF which is dedicated to GHPC and $\mathsf{GHPC_{LL}}$. An overview of the possible combinations is given in Table 1.



(a) Original netlist $o = (x \oplus i_0) \vee \overline{(i_1 \wedge y)}$

(b) AIG netlist $o = \overline{\overline{(x \wedge \overline{i_0})} \wedge \overline{(\overline{x} \wedge i_0)} \wedge i_1 \wedge y}$

(c) BDD

(d) MUX-based netlist

(e) ANF with at most 3 inputs per gadget $o = 1 \oplus i_1 y \oplus i_0 i_1 y \oplus x i_1 y$

(f) ANF with 4 inputs per gadget

Figure 5: Different processing methods for the secure zone of the exemplary circuit in Figure 4.

Table 1: Supported masking schemes and processing methods.

Processing Method	**Masking Scheme**		
	HPC1/HPC2 ($d \geq 1$)	GHPC ($d = 1$)	GHPC_{LL} ($d = 1$)
Naive	✓	✓	✓
AIG	✓	✓	✓
$\text{BDD}_{\text{SYLVAN}}$	✓	✓	✓
BDD_{CUDD}	✓	✓	✓
ANF		✓	✓

3.3.1 Naive

Every cell in the netlist of the secure zone can be naturally exchanged with its masked variant depending on the selected masking scheme. Since this is just a translation of one netlist into another while keeping the original structure (i.e., the number of cells and how they are connected), the efficiency of the resulting masked circuit depends on how the original circuit has been synthesized. For example, the non-linear gadgets need fresh randomness and introduce register stages into the gates (see Section 3.1). Therefore, the number of non-linear gates and how they are composed (i.e., the logical depth of the circuit) has a direct effect on the number of required fresh masks and latency overhead of the resulting masked circuit.

We should also highlight that every signal in the secure zone is transformed into a masked form with $d + 1$ shares. However, the signals which are not marked as secure but involved in the secure zone are padded with 0 to form $d + 1$ shares. For example, the primary input i_0 and the internal signal y in the example shown in Figure 4. Note that it should be carefully examined to make sure that this does not pose any security issue in the used gadgets. We have verified this in HPC1, HPC2, GHPC, and $\mathsf{GHPC_{LL}}$ gadgets.

3.3.2 AIG

Generally, AND-Inverter Graphs (AIGs) are promising candidates for a unified representation of Boolean functions suitable for logic synthesis, simulation, and verification.

Representation. An AIG is a Directed Acyclic Graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with nodes in $\mathcal{V}$ and edges in $\mathcal{E}$. More precisely, any Boolean function $\mathbb{F}_2^n \mapsto \mathbb{F}_2^m$, over inputs $\mathcal{X} = \{x_i | 1 \leq i \leq n\}$ and outputs $\mathcal{Y} = \{y_i | 1 \leq i \leq m\}$, can be modeled as AIG which is syntactically and semantically defined as follows.

Definition 2 (Syntax of AIGs)**.** Given a finite DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V}$ and edges $\mathcal{E}$, the syntax of an m-rooted AIG is defined as follows:

(1) There are exactly n terminal nodes v_t, each labeled with a unique $x_i \in \mathcal{X}$.

(2) There are exactly m root nodes v_r, each labeled with a unique $y_i \in \mathcal{Y}$.

(3) Each non-terminal, non-root node $v \in \mathcal{V}$ has exactly two incoming edges while each edge $e \in \mathcal{E}$ is either labeled as *regular* or *complement* edge.

Given the syntactical, graph-based representation of an AIG, the semantic definition of AIGs, based on *DeMorgan's theorem*, can be provided as follows.

Definition 3 (Semantic of AIGs)**.** The representation of a Boolean function $\mathsf{f} : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ in terms of AIGs is given according to the following specification:

(1) Each non-terminal, non-root node represents a Boolean conjunction $\wedge$ (AND) of the two nodes represented by the incoming edges.

(2) Each *regular* edge represents the original function of the source node, whereas each *complement* edge represents the inverted function of the source node.

Transformation. As an alternative to Naive, for the AIG processing method, the netlist of the secure zone is converted into an AIG based on *DeMorgan's rules*. In particular, any standard Boolean gate and operation (e.g., OR, XOR, etc.) is replaced by a representation purely relying on AND and inverter gates only. More precisely, the converted netlist of the secure zone eventually consist of only 2-input AND and single-input NOT gates while

keeping the same functionality as the original netlist (see an example in Figure 5(b)). In a final step, the AND and NOT gates of the AIG-netlist are replaced with their masked counterparts, based on the selected masking scheme.

However, in order to process the original secure zone netlist and construct the corresponding AIG, our tool heavily relies on the public-domain ABC system and library for synthesis and formal verification of digital logic circuits.

ABC[2] is a system for digital logic synthesis and verification, particularly focusing on synchronous binary logic circuits. Internally, the system combines logic representations and optimizations based on AIG. In the context of this work, we particularly leverage the innovative AIG functionalities provided in terms of a C-library that was included and integrated into our tool flow.

Limitations. In contrast to the Naive processing method, the AIG method only requires a single (optimized) masked non-linear gadget (AND) while the inversion generally is for free (in case of Boolean masking). However, since such a reduced representation, purely based on conjunction and inversion operations, is not unique, the AIG method leaves room for optimizations and improvements with respect to logical depth and latency. However, as the creation of the masked secure zone originates from the synthesized netlist while all non-linear gates are represented in terms of conjunction and inversion gates (based on De Morgan's laws), we do not expect to reduce the number of non-linear gates and their associated number of random bits.

3.3.3 BDD

Besides AIGs, in discrete mathematics and computer science, Binary Decision Diagrams (BDDs) are often used as basic data structure to represent and manipulate Boolean functions. The seminal concept of BDDs has been introduced by Akers [Ake78] and refined by Bryant [Bry86], improving efficiency and conciseness through variable ordering. Nowadays, many applications in logic synthesis and formal verification of digital Integrated Circuits (ICs) rely on (reduced and ordered) BDDs[3]. This also holds for SILVER [KSM20] introduced in Section 2.5.

Representation. Multi-root BDDs provide a unique, concise, and canonical representation of Boolean functions $\mathbb{F}_2^n \mapsto \mathbb{F}_2^m$. In particular, any multi-root BDD can be represented as a DAG with m root nodes and two terminal nodes $\{\mathbf{0}, \mathbf{1}\}$. More precisely, BDDs can be defined syntactically and semantically as follows.

Definition 4 (Syntax of BDDs)**.** Given a pair $(\pi, \mathcal{G})$, where π denotes a variable ordering and $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a finite DAG with vertices $\mathcal{V}$ and edges $\mathcal{E}$, the syntax of an m-rooted Reduced Ordered Binary Decision Diagram is defined as follows:

(1) There are exactly m root nodes and each node $v \in \mathcal{V}$ is either a non-terminal or one of the two terminal nodes $\{\mathbf{0}, \mathbf{1}\}$.

(2) Each non-terminal node $v \in \mathcal{V}$ is labeled with a variable, denoted as $\mathsf{var}(v)$ and has exactly two distinct child nodes in $\mathcal{V}$, which are denoted as $\mathsf{then}(v)$ and $\mathsf{else}(v)$. More precisely, there is no non-terminal node v such that $\mathsf{then}(v) = \mathsf{else}(v)$.

(3) There are no duplicate nodes, i.e., for each pair of nodes $\{v, v'\} \in \mathcal{V}^2$ at least one of the following conditions holds:

    (i) The variable label is different, i.e., $\mathsf{var}(v) \neq \mathsf{var}(v')$.

[2] https://github.com/berkeley-abc/abc

[3] For the sake of simplicity, we refer to Reduced Ordered Binary Decision Diagrams as BDDs.

(ii) The child nodes are different, i.e., $\mathsf{then}(v) \neq \mathsf{then}(v')$ or $\mathsf{else}(v) \neq \mathsf{else}(v')$.

(4) For each path from root nodes to terminal nodes, the variable labels are encountered at most once and in the same order, defined by the variable ordering π.

Using the principle of *Shannon decompositions*, each multi-rooted BDD recursively defines a Boolean function $\mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ and arbitrary Boolean operations.

Definition 5 (Semantic of BDDs)**.** The representation of a Boolean function $\mathsf{f} : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$, defined over the input variables $\mathcal{X} = \{x_i | 1 \leq i \leq n\}$, is defined recursively according to the following specification:

(1) Given a terminal node v, then $\mathsf{f}_v|_x = 0$ if v is the terminal node $\mathbf{0}$, and $\mathsf{f}_v|_x = 1$ otherwise.

(2) Given a non-terminal node v and $\mathsf{var}(v) = x_i$, then f_v is defined recursively by the Shannon decomposition: $\mathsf{f}_v = x_i \cdot \mathsf{f}_{\mathsf{then}(v)}|_{x_i=1} + \overline{x_i} \cdot \mathsf{f}_{\mathsf{else}(v)}|_{x_i=0}$.

(3) Given two root nodes v_{r_1} and v_{r_2} and any binary Boolean operation $\circ$, such that $\mathsf{f} = \mathsf{f}_{v_{r_1}} \circ \mathsf{f}_{v_{r_2}}$, then f can be derived recursively as:

$$\begin{aligned}
\mathsf{f} &= x_i \cdot \mathsf{f}|_{x_i=1} + \overline{x_i} \cdot \mathsf{f}|_{x_i=0} \\
&= x_i \cdot (\mathsf{f}_{v_{r_1}} \circ \mathsf{f}_{v_{r_2}})|_{x_i=1} + \overline{x_i} \cdot (\mathsf{f}_{v_{r_1}} \circ \mathsf{f}_{v_{r_2}})|_{x_i=0} \\
&= x_i \cdot (\mathsf{f}_{v_{r_1}}|_{x_i=1} \circ \mathsf{f}_{v_{r_2}}|_{x_i=1}) + \overline{x_i} \cdot (\mathsf{f}_{v_{r_1}}|_{x_i=0} \circ \mathsf{f}_{v_{r_2}}|_{x_i=0})
\end{aligned}$$

Transformation. In contrast to the AIG processing method, the BDD processing method attempts to create BDD for the secure zone netlist. More precisely, the Boolean function of the secure zone netlist is transformed into a multi-root BDD, whereas each node in the BDD corresponds to a multiplexer (MUX). Note, however, that in the context of BDDs, each select signal is connected to a primary input, while the data signals are connected to the constants **0** or **1** or any other multiplexer corresponding to a BDD node. Therefore, our tool extracts an equivalent netlist of the secure zone, purely based on 2-to-1 multiplexers, which afterwards are exchanged and replaced by their masked counterpart (see an example in Figure 5(c) and Figure 5(d)).

Similar to the application of an external library for creation of AIGs, our tool also employs two different C/C++ libraries for the construction and manipulation of BDDs.

SYLVAN[4] is a state-of-the-art BDD high-performance, multi-core decision diagram package implemented in C/C++. In particular, manipulation and processing of BDDs and binary operations has been extensively optimized and implemented for multi-core support, outperforming existing, but single-core BDD packages.

CUDD[5] (Colorado University Decision Diagram) is a package for manipulation and processing of BDDs, Algebraic Decision Diagrams (ADDs), and Zero-suppressed Binary Decision Diagrams (ZDDs) implemented in C. In contrast to SYLVAN, CUDD provides an extensive set of features and operations that can be performed on BDDs, including automatic and dynamic reordering of the variables. Hence, although CUDD has been mostly designed for single-core processors, it can outperfom SYLVAN in certain applications, mostly due to reduced memory requirements and BDD sizes (due to more optimal variable orderings).

[4] https://github.com/utwente-fmt/sylvan.git
[5] https://github.com/ivmai/cudd.git

Limitations. In contrast to the Naive and AIG processing techniques, the BDD transformation results in a unique representation (under a given variable ordering). As a result, the BDD representation is independent of the original netlist representation but solely depends on the underlying Boolean function, hence reducing the effort of optimizing the original and unprotected design. Further, since each BDD can be represented as multiplexer-cascade in digital logic, creation and optimization of a single masked multiplexer-gadget is sufficient to convert unprotected designs into protected designs. However, in contrast to common approaches, the primary inputs of the secure zone serve as selection signals of the multiplexers (instead of being connected to the multiplexer data inputs). As a consequence, the logical depth of the multiplexer-cascade is solely limited by the number of primary inputs of the secure zone, hence, determining the resulting latency of the masked circuit.

Besides, since BDDs are only canonical under a given variable ordering, we employ two different state-of-the-art BDD libraries. While SYLVAN is a high-performance library captivating through multi-core algorithms and operations, in particular with respect to BDD generation and recombination, CUDD also supports automated and dynamic variable re-ordering. In fact, using some pre-defined and global thresholds, the library automatically performs variable re-orderings once the thresholds are exceeded, in order to find better (i.e., smaller) BDD representations through changing the ordering of the variables (i.e., primary inputs). As a smaller BDD directly translates to smaller masked circuits using fewer multiplexers, we decided to support and evaluate both BDD libraries for their various benefits and limitations.

3.3.4 ANF

As stated in Section 3.1, GHPC and its low-latency variant $\mathsf{GHPC_{LL}}$ allow to construct first-order secure composable gadgets from arbitrary functions. More specifically, Boolean functions in arbitrary number of variables can be translated into single gadgets. However, with increasing variable dependencies, area overhead of the gadgets gradually becomes more obstructive. To this end, the ANF processing method tries to find trade-offs between single gadget size and overall circuit size.

Representation. In general, any Boolean function can be expressed canonically using several normal forms, such as Conjunctive Normal Form (CNF), Disjunctive Normal Form (DNF), or Algebraic Normal Form (ANF). In particular, the ANF representation is often considered for masking purposes due to the trivial masking of exclusive-or operations.

Definition 6 (Algebraic Normal Form)**.** For any Boolean function $\mathsf{f} : \mathbb{F}_2^n \mapsto \mathbb{F}_2$ there exists a unique AND-XOR representation, called the ANF of f:

$$\mathsf{f}(\mathbf{x}) = \bigoplus_{u \in \mathbb{F}_2^n} a_u x^u \text{ with } a_u \in \mathbb{F}_2$$

The summands of f are called *monomials*. Each monomial forms a conjunction of a unique subset of $\mathbf{x}$ defined by its corresponding index u. The *degree* of a monomial is defined as the size of its input set. Furthermore, the *algebraic degree* of f is defined as the highest degree of all functions monomials.

Tranformation. We first construct the ANF of the secure zone. The construction mechanism represents each gate (e.g., AND, OR, XOR) of the original netlist with its corresponding ANF. More precisely, our tool computes the ANF of any gate based on the primary inputs of the secure zone. To this end, the inputs of each gate are expressed in terms of an ANF given in the primary inputs.

However, for the construction of gadgets, we are only interested in the ANF of the outputs of the entire secure zone. Unfortunately, as stated above, constructing an individual

gadget per secure zone output may lead to a very large circuit if the corresponding ANF depends on a very large number of primary inputs. As a consequence, to reduce the circuit size, we apply two optimization techniques introduced in the following. For the sake of simplicity and better understanding, we analyze an exemplary Boolean function $\mathsf{f} : \mathbb{F}_2^4 \mapsto \mathbb{F}_2$ represented by the following ANF.

$$\mathsf{f} = x_0x_2 \oplus x_0x_3 \oplus x_0x_2x_3 \oplus x_1x_2 \oplus x_1x_3 \oplus x_1x_2x_3 \tag{1}$$

Now, the trivial approach is to construct the entire function as a single gadget. As f depends on four different input values, the corresponding gadget also requires four inputs. Nevertheless, f can be rewritten in a way that the linear combination $x_0 \oplus x_1$ is processed instead of x_0 and x_1, i.e.:

$$\mathsf{f} = (x_0 \oplus x_1)(x_2 \oplus x_3 \oplus x_2x_3) \tag{2}$$

Note that finding suited linear combinations is not trivial. We explain our methodology in the following. Providing the linear combination $(x_0 \oplus x_1)$ to the gadget instead of x_0 and x_1 reduces the input size of the gadget by one. In practice, such linear combinations occur in many modern block ciphers including a key addition operation. Hence, detection of such operations and extraction of linear combinations often allows to typically halve the number of inputs per gadget. However, the minimization of gadgets due to finding linear combinations is not only restricted to the gadgets inputs. For instance, considering $\mathsf{h} : \mathbb{F}_2^4 \mapsto \mathbb{F}_2$ as

$$\mathsf{h} = x_0 \oplus x_1 \oplus x_0x_1 \oplus x_2 \oplus x_3 \oplus x_2x_3,$$

Again, computing the entire function in a single gadget is inefficient since the algebraic degree of h is smaller than the number of inputs. In particular, only x_0 and x_1 as well as x_2 and x_3 are combined non-linearly. In addition, only two different monomials of degree two (x_0x_1 and x_2x_3) occur, i.e., not all inputs are combined in conjunctions. Hence, splitting h into two functions $\mathsf{h} = \mathsf{h}_0 \oplus \mathsf{h}_1$ such that $\mathsf{h}_0 = x_0 \oplus x_1 \oplus x_0x_1$ and $\mathsf{h}_1 = x_2 \oplus x_3 \oplus x_2x_3$ results in two gadgets with only two inputs each. Compared to a single gadget with four inputs the area footprint is reduced. Again, linear output combinations, as described here, occur in many modern block ciphers where linear diffusion layers permute the outputs of multiple non-linear S-boxes which typically operate on a small set of inputs (e.g., 4 or 8 bits).

Now, given an arbitrary Boolean function representing a secure zone output in ANF, similar to the examples shown above, we can split the entire ANF into multiple sub-functions of independent inputs. For instance, considering a full round of a block cipher, this step is beneficial as different S-boxes are usually computed on non-colliding sets of input variables. Therefore, we should be able to construct gadgets that operate only on a small set of input values (depending on the S-box input size). More precisely, in order to find suitable sub-functions, we first extract all monomials with maximal algebraic degree and place them into sub-functions. Specifically, all monomials that share no input are placed in different sub-functions, while monomials sharing at least one input are placed in the same sub-function. In the next step, we extract all smaller monomials of the ANF and place them in one of the existing sub-functions such that each monomial is placed in a sub-function if it shares inputs with the largest monomial of this sub-function. Eventually, we repeat this procedure for every output ANF while adding new outputs to the gadgets. Hence, each gadget may be used to compute sub-funtions of multiple output ANFs if they depend on the same inputs. As a result, each output ANF can be computed as the sum of different sub-functions while each sub-function receives a different set of inputs. At this point, the gadgets are independent of each other and we can optimize them individually.

For some lightweight block ciphers, such as PRINCE [BCG+12], PRESENT [BKL+07], Midori [BBI+15], Skinny [BJK+16], and CRAFT [BLMR19], the diffusion layer only

combines the output of different S-boxes. In other words, the diffusion layer never combines different outputs of the same S-box. As the gadgets will be optimized at most up to the S-box sizes, the sub-functions and the outputs of the S-boxes become equivalent. Prominent exceptions of this rule are the AES [DR02] and LED [GPPR11], where the MixColumns also linearly combines the outputs computed by the same S-box (through Galois-field multiplication with constants in the MixColumns matrix). Note that since we are analyzing the netlist of the secure zone, such a linear combination is not trivially visible in the output ANFs. This translates to gadgets with a large number of outputs as the gadgets compute all linear combined outputs separately and not the small set of S-box outputs resulting in a high area and fresh-randomness overhead.

In order to detect such linear combinations and reduce the number of gadget outputs (ideally) up to the number of S-box outputs, we search for a minimal set of different functions whose linear combinations compute all required sub-functions. We perform such a search with simulated annealing [KGV83], a discrete optimization technique that searches for a minimal solution by evaluating solutions that are similar to the current solution (so-called neighbors). The advantages of simulated annealing compared to other optimization techniques such as constraint programming [Apt03] are the great performance and the ability to escape local minima. This is achieved since the acceptance of a neighbor is partially probabilistic. Hence, sometimes also bad neighbors are accepted to escape local minima. We start with an input solution A (a set of functions) that computes all sub-functions separately. Therefore, each sub-function is given as a single output and with a single summand, what translates to a list with a single element. During the simulated annealing, we split our initial solution into multiple summands that compute the outputs with a minimal number of different summands. We show our method in Algorithm 1. For the neighboring function $Neighbor(C)$, we randomly modify input solution C by selecting one summand from a random sub-function (at the beginning, each function is given as a single summand), XOR it to every occurrence of another randomly-chosen summand, and insert it to every modified sub-function. Moreover, we define our objective costs $Cost(C)$ of the solution C as the number of different summands required to compute all sub-functions. It turns out that a very small number of iterations and a very low cooling factor are enough to recover the minimal set of gadget outputs. In our experiments, we used a cooling factor *cool* of 0.9 and start with $n = 100$ iterations per cooling step which increases by 100 after every cooling step. The initial temperature is $T = 1$ and cooled down until it reaches $T_{min} = 0.5$. Note that the set of gadget outputs not necessarily encompasses the outputs of the underlying S-boxes but linear combinations which also result in a minimal number of gadget outputs.

After optimizing the gadgets outputs we investigate each gadgets inputs. As shown above in Equation (2), a gadget can depend on a linear combination of its inputs. In order to detect such cases for a given gadget, for each input (e.g., x_0) we first make a set $\mathcal{L}_{f,x_0}$ of all monomials that occur in output function f and include x_0. As only monomials including x_0 are in $\mathcal{L}_{f,x_0}$, we erase x_0 from the monomial before storing it. Then, we search for input combinations by iterating over all pairs of inputs, e.g., (x_0, x_1), and examining the corresponding sets $\mathcal{L}_{f,x_0}$ and $\mathcal{L}_{f,x_1}$. Since we erase x_0 from all monomials in $\mathcal{L}_{f,x_0}$ and x_1 from all monomials in $\mathcal{L}_{f,x_1}$ both sets are exactly equivalent iff they differ only in (x_0, x_1). Hence, equivalence shows that we can replace (x_0, x_1) by its linear combination. Naturally, two inputs x_i and x_j, such that $i \neq j$, can be replaced by their linear combination $x_i \oplus x_j$ in an ANF, if both x_i and x_j are similarly combined with other inputs in all monomials. This is given if both sets of monomials are equal and non-empty for every output function of the gadget. If both conditions are met, x_i and x_j can be replaced with their linear combination $x_i \oplus x_j$ in the entire gadget. As a short example, we verify the linear combination in f (cf. Equation 1 and 2). For the input pair (x_0, x_1) it holds that $\mathcal{L}_{f,x_0} = \{x_2, x_3, x_2x_3\}$ and $\mathcal{L}_{f,x_1} = \{x_2, x_3, x_2x_3\}$. Since it hold that $\mathcal{L}_{f,x_0} = \mathcal{L}_{f,x_1}$, the linear input combination

Algorithm 1 Search for optimal gadget outputs

Input: $T, T_{min}, cool, n, A$
Output: C ▷ Outputs an optimized solution
1: $C \leftarrow A$
2: **while** $T > T_{min}$ **do**
3: **for** $i = 0, 1, ..., n-1$ **do**
4: $D \leftarrow Neighbor(C)$
5: $\delta \leftarrow Cost(D) - Cost(C)$
6: $r \leftarrow rand()$ ▷ Random value in range [0,1]
7: **if** $\delta \leq 0$ **or** $\exp(-(\delta/T)) > r$ **then**
8: $C \leftarrow D$
9: **end if**
10: $i \leftarrow i + 1$
11: **end for**
12: $T \leftarrow T \cdot cool$
13: **end while**

(x_0, x_1) can be applied. This reduces the number of inputs of the gadget and the complexity of the computed function. We formalize our technique in Algorithm 2. Internally, we represent each monomial as a set of its inputs and each function as a set of monomials. Hence, we can represent a gadget ($\mathcal{L}_{G_i}$ and $\mathcal{L}_{G_o}$ in the algorithm) as a set of its output functions.

Finally, the result of the ANF processing method is a combination of gadgets and linear layers. A general structure is depicted in 6. Initially, input layer L_{in} computes all linear input combinations which are fed to the gadgets computing all non-linear components. The different outputs of the i-th gadget are then linearly combined (through L_i). Finally, the output layer L_{out} linearly combines different gadgets outputs.

Limitations. As already stated, although the other previously discussed processing methods can be combined with different masking schemes, ANF is purely dedicated to GHPC and $\mathsf{GHPC_{LL}}$. Hence, ANF can only generate first-order secure circuits. Similar to BDD, ANF generates a unique ANF of each output independent of the underlying netlist. Nevertheless, the optimizations are not unique due to the probabilistic characteristic of simulated annealing. Hence, improvements in terms of area are possible. In particular, the optimization generates ineffective gadgets if the complexity of the secure zone grows. The given parameters for simulated annealing are suited for the optimization of typical S-boxes (up to eight-bit input and output). Adjusting the parameters of simulated annealing could be helpful for more complex secure zones but also increases the runtime. Up to now, all gadgets are instantiated in parallel leading to a fixed latency of two clock cycles for GHPC. On the other hand, the largest gadget can not be smaller than the algebraic degree of the largest output function.

3.4 Optimization

Up to this point, we have explained how the secure zone is extracted from the netlist and how it can be translated to a masked circuit. Depending on the chosen processing method and the masking scheme and more importantly the initial netlist of the secure zone, the resulting masked circuit might introduce additional latency (more clock cycles) to the circuit and demand for a high or low number of fresh masks, depending on the architecture of the selected gadgets. An important part, which heavily affects the performance of the resulting circuit, are the multiplexers of the secure zone. We have already given an efficient way to realize an HPC-MUX in Section 3.1.5. However, it is commonly seen that the secure

Algorithm 2 Search for linear input combinations

Input: $\mathcal{L}_{G_i}$, $\mathcal{L}_x$ ▷ List of the gadgets output functions and inputs
Output: $\mathcal{L}_{G_o}$ ▷ List of the gadgets substituted output functions

1: $\mathcal{L}_{G_o} \leftarrow \mathcal{L}_{G_i}$
2: **for** $\forall(x_0, x_1) \in \mathcal{L}_x \times \mathcal{L}_x$ **do**
3:   **if** $x_0 \neq x_1$ **then** ▷ Get two different inputs of the gadget
4:     **for** $\forall f \in \mathcal{L}_{G_o}$ **do**
5:       $\mathcal{L}_{f,x_0} \leftarrow \emptyset$
6:       $\mathcal{L}_{f,x_1} \leftarrow \emptyset$
7:       **for** $\forall m \in f$ **do** ▷ Get the monomials of the output function
8:         **if** $x_0 \in m$ **then** $\mathcal{L}_{f,x_0} \leftarrow \mathcal{L}_{f,x_0} \cup (\{m \backslash \{x_0\}\})$
9:         **end if**
10:         **if** $x_1 \in m$ **then** $\mathcal{L}_{f,x_1} \leftarrow \mathcal{L}_{f,x_1} \cup (\{m \backslash \{x_1\}\})$
11:         **end if**
12:       **end for**
13:     **end for**
14:     **if** $\mathcal{L}_{f,x_0} = \mathcal{L}_{f,x_1} \forall f \in \mathcal{L}_{G_o}$ **then**
15:       $n \leftarrow \{x_0, x_1\}$ ▷ Create the new linear combination
16:       $\mathcal{L}_x \leftarrow (\mathcal{L}_x \backslash \{x_0\}, \{x_1\}) \cup n$ ▷ Update the gadgets inputs
17:       **for** $\forall f \in \mathcal{L}_{G_o}$ **do**
18:         **for** $\forall m \in f$ **do** ▷ Substitute the inputs with their linear combination
19:           **if** $x_0 \in m$ **then** $m \leftarrow (m \backslash \{x_0\}) \cup n$
20:           **end if**
21:           **if** $x_1 \in m$ **then** $f \leftarrow (f \backslash \{m\})$
22:           **end if**
23:         **end for**
24:       **end for**
25:     **end if**
26:   **end if**
27: **end for**

zone additionally contains multiplexers whose select signal is not marked as secure. For example, a plaintext which is given as the primary input is loaded under certain conditions, e.g., when the reset signal is high (or low). As another example, different computations are performed in different clock cycles, e.g., last round of the cipher is different to the other rounds (e.g., MixColumns is missing in the last round of AES), or in a serialized architecture during some clock cycles the output of the Sbox is taken, and in some other clock cycles that of the diffusion layer. In such cases, these is no need to mask and translate the multiplexer with an HPC-MUX. Similar to the XOR, which is secure under PINI notion, such a multiplexer can be straightforwardly instantiated $d + 1$ times (for security order d). Note that security under the PINI notion requires every signal to have an independent sharing [CS20]. Hence, connecting corresponding shares of two masked signal to an ordinary multiplexer controlled by an insecure signal would not violate any security requirements. This would greatly improve the efficiency of the resulting masked circuit. As a side note, the synthesize should be directed to make use of multiplexers in such cases. If the same functionality is realized by Boolean gates (AND, OR, XOR, etc.) and the secure zone is optimized (e.g., for area, latency, or power), it would not be straightforward to detect the multiplexers in the secure zone, and most likely the resulting circuit would suffer from a high number of added register stages and a high demand for fresh randomness.

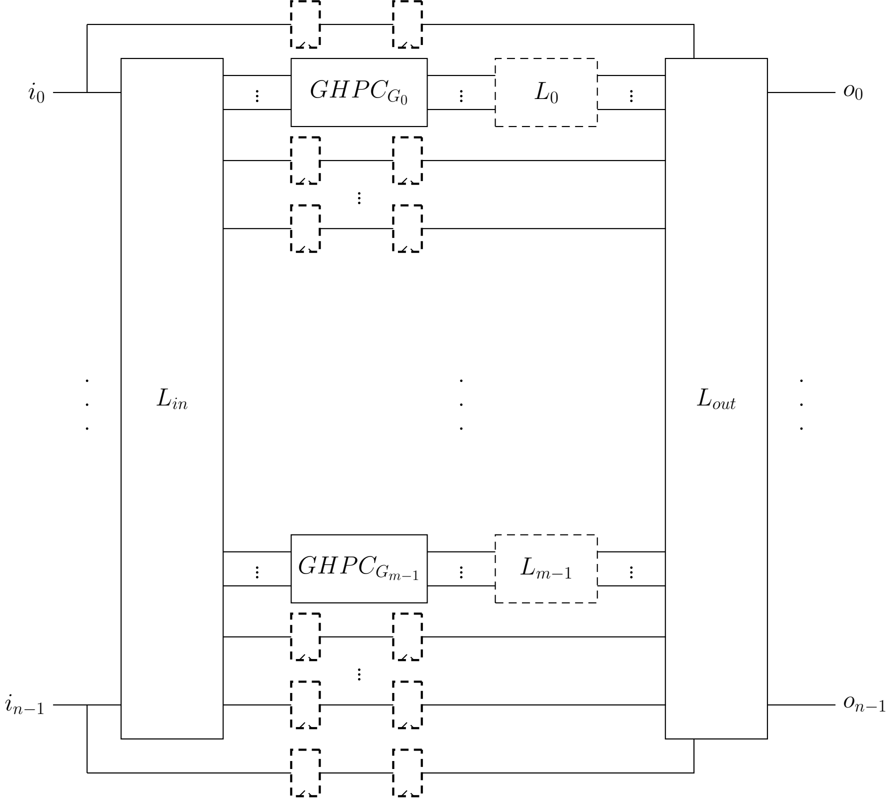


Figure 6: Generic structure of a masked circuit after the ANF processing.

3.5 Control Logic

Since masked gadgets often have internal register stage(s), the combinational circuit of the secure zone, after applying the selected masking scheme, is not fully combinational anymore. Therefore, the circuit (control logic, etc.) would not necessarily work properly. Hence, the circuit should be adjusted to keep its correct functionality. We achieve this by two different techniques explained below.

Pipelining. We can add extra registers to synchronize all inputs of every gadget as well as all inputs of the main register stage. An example is shown in Figure 7(a), which is based on the circuit depicted in Figure 4. Each HPC2 gadget introduces two register stages. Hence, in order to synchronize the inputs of the HPC2 OR gate in Figure 7(a) we need to place two cascaded registers at its first input. This procedure is done by synchronizing all inputs of each gadget processed in order of their logic depth. At the end, all inputs of the main register stage are also synchronized. For the example shown in Figure 7(a), four registers are placed in the normal zone to synchronize it with the output of the secure zone. This way, the circuit keeps its correct functionality while realizing a pipelined design with $p+1$ stages if p register stages are added to the circuit (in the shown example, $p=4$). Hence, the circuit can process $p+1$ consecutive and independent inputs. We should highlight that the area overhead of the resulting circuit is relatively high, but it constructs a circuit with a high throughput due to its underlying pipeline architecture.

Clock Gating. In order to mitigate the area overhead of the former technique, we can make use of clock gating. More precisely, we need to make sure that the main register

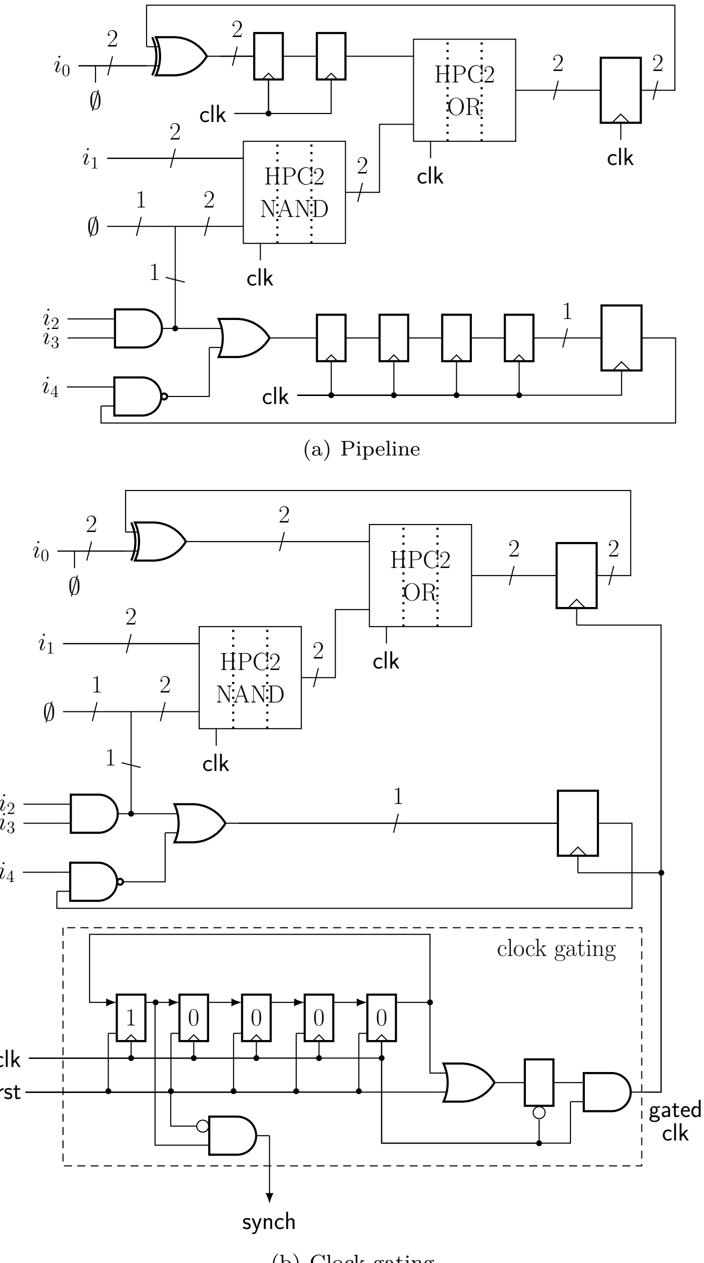


Figure 7: Different architectures: pipeline versus clock gating for the secure zone of the exemplary circuit in Figure 4 processed by the Naive method using HPC2 masking scheme.

stage keeps its value until the computation of the secure zone is terminated. The same holds for the primary inputs. Hence, we do not add any extra registers to the circuit, but change the clock of the main register stage. Hence, all internal registers of the gadgets are controlled by the main clock, but the main register stage by an added gated clock enabled once per evaluation of the secure zone. The circuit equivalent to the former example is shown in Figure 7(b). In order to keep the generality, a clock gating module is added to the design which can be adjusted based on the latency of the masked secure zone, i.e., p. To this end, the clock gating module instantiates a rotating shift register with $p + 1$ bits

Table 2: Full cipher implementation case studies.

Cipher	Implementation	Reference
Skinny64	Round-based encryption with 64-bit key	[BJK+16]
AES-128	Byte-serial and round-based encryption	[DR02]
CRAFT	Round-based encryption without tweak	[BLMR19]
PRESENT-80	Nibble-serial encryption	[BKL+07]
LED-64	Round-based encryption	[GPPR11]
Midori-64	Round-based encryption/decryption	[BBI+15]

initialized by $1\{0\}^p$ using an added control signal `rst`. Hence, every $p+1$ cycles the main register stage is clocked to proceed with the next round of the calculation of the secure zone. As a result, the latency of the clock-gated circuit is the same as the pipeline one, but it has a lower throughput as well as lower area overhead. Note that since the primary inputs as well as the fresh masks (used by the gadgets) are only allowed to change once per evaluation cycle right after the main register stage is clocked, the clock gating module generates an additional output signal `synch` to let the outer modules synchronize. More princely, a positive edge is seen on the `synch` signal which can be used to trigger the clock of a random-number generator to update the fresh masks.

4 Case Studies

In order to examine the efficiency and performance of circuits constructed by AGEMA, we evaluated several designs including different S-boxes and full cipher implementations under different settings, i.e., various processing methods and different masking schemes. We start with the 4-bit S-box of Skinny [BJK+16] and provide two different representations. In the first one, we straightforwardly implemented the S-box by a lookup table. The synthesizer translates such a behavior representation to a netlist, which is then given to AGEMA for further processing. For the second one, we followed the optimized representation provided in [CGLS21]. The corresponding results are given in Table 3 and Table 4 respectively. As the results are extensive and many tables are presented, all performance results are given in Appendix A. Note that all syntheses have been done using Synopsis Design Compiler and a NanGate 45 nm standard cell library. For these analyses, we covered all processing methods Naive, AIG, $\mathsf{BDD}_{\mathsf{SYLVAN}}$, and $\mathsf{BDD}_{\mathsf{CUDD}}$ for masking scheme HPC2 at different security orders. For the sake of comparison, we covered HPC1 only for Naive method. ANF is also covered as a preprocessing step where transformation into a secure design is only possible in combination with GHPC or $\mathsf{GHPC}_{\mathsf{LL}}$ (see Table 1). The effect of the given netlist on the performance of the masked circuit can be easily seen in Naive and AIG processing methods. The (HPC2, Naive)-approach for the lookup table based S-box adds 10 clock cycles to the latency compared to 4 clock cycles for the optimized S-box. ANF and BDD methods are actually not affected by the optimality of the given netlist as they reconstruct the netlist. Further, it can be seen that HPC1 leads to a lower area overhead while it certainly demands for more fresh randomness.

We repeated this procedure for the AES S-box. In addition to a lookup table based representation, we took the Canright version [Can05] and the optimized design presented in [BP12], which – in addition to the linear layers (isomorphisms) – has at most 4 cascaded 2-input AND gates, making it suitable for a masked design. Performance results are given in Table 5, Table 6 and Table 7. The effect of the optimality of the given netlist on the performance (area and latency) is even clearer compared to the former case study.

For the full cipher implementations, we cover the list given in Table 2. The performance results are shown in tables in Appendix A. For all such case studies, we considered the

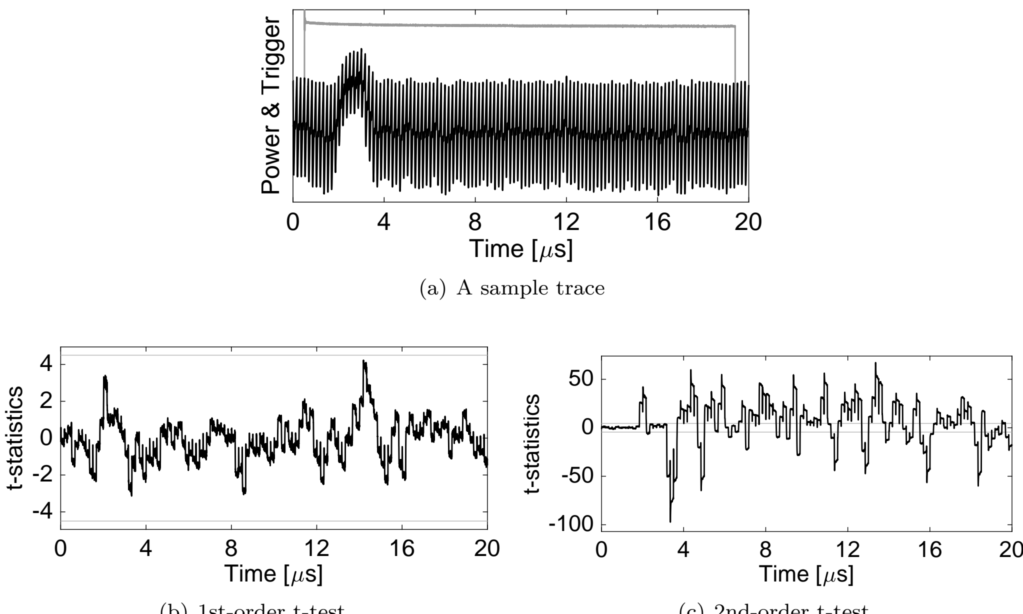
following facts.

- For all designs, we marked the plaintext/cipheretxt and the key as secure for AGEMA. In other words, the resulting masked circuit receives all inputs (except the control signals) in a masked form with $d + 1$ shares and provide the output also with $d + 1$.
- If possible and available, we provided an optimized representation of the S-box. Above, we have given the source of such optimized designs for the the Skinny and AES S-boxes. For PRESENT and LED, which share the same S-box, we took the optimized S-box representation from [CGLS21]. However, for Midori and CRAFT, which also share the same S-box, such representations are not available. Therefore, we represented the S-box by a lookup table. It can be seen in the performance results of CRAFT and Midori that the added latency (for Naive method) is higher compared to the other ciphers with an optimized 4-bit S-box.
- We hard-coded the multiplexers (controlled by Finite State Machine (FSM) or primary input control signals like RST) and directed the synthesizer to not optimize them (see Section 3.4). The same holds for XORs. If the XORs are also merged in other combinational circuits, the synthesizer may optimize in other directions leading to a netlist with more (cascaded) non-linear gates.
- As explained in Section 3.3.3, BDD processing methods are not necessarily efficient for large combinational circuits when an optimized representation is available. This can be seen in for Midori and CRAFT, where the S-box is based on look up tables and BDD methods have the same latency overhead, while this does not hold true for the other cases, where an optimized representation of the S-box is given. Further, in AES round-based implementation, the round function, including 16 S-boxes followed by the MixColumns and 4 S-boxes of the KeySchedule, is too large to be processed by BDD methods.
- The latency reported in the tables of Appendix A should be read as the added latency to each clock cycle of the unprotected implementation. For example, the unprotected AES round-based encryption needs $11 \times 1 = 11$ clock cycles to accomplish the encryption, and based on Table 10, the HPC2 Naive implementation adds 8 cycles latency. this results in the clock-gating implementation needing $11 \times (1 + 8) = 99$ clock cycles for an encryption. The pipeline implementation has the same latency, but processes $8 + 1$ plaintexts consecutively in those 99 clock cycles. Hence, its throughput (ignoring the delay) stays the same as the unprotected implementation, but certainly has a considerably higher area overhead compared to the clock-gating implementation.

The tool and all case studies are provided in the GitHub: https://github.com/Chair-for-Security-Engineering/AGMEA.

5 Experimental Analyses

As the first analysis step, we have examined our implementations of HPC1 and HPC2 gadgets with SILVER [KSM20] to verify if they are PINI secure under robust (glitch-extended) probing model. The gadgets include 2-input AND, NAND, OR, NOR, XOR, XNOR and 2-to-1 MUX. We made VHDL/Verilog implementation of all gadgets parametric, i.e., the security order and whether a pipeline design is desired are easily set when instantiating such modules. Although constructing the circuit with PINI gadgets would result in a PINI-secure circuit, we further verified this on some masked Skinny S-box implementations listed in Table 3 and Table 4.



(a) A sample trace

(b) 1st-order t-test

(c) 2nd-order t-test

Figure 8: FPGA-based fixed versus random t-test using 100 million traces, Skinny-64-64 round-based encryption, first-order GHPC ANF pipeline.

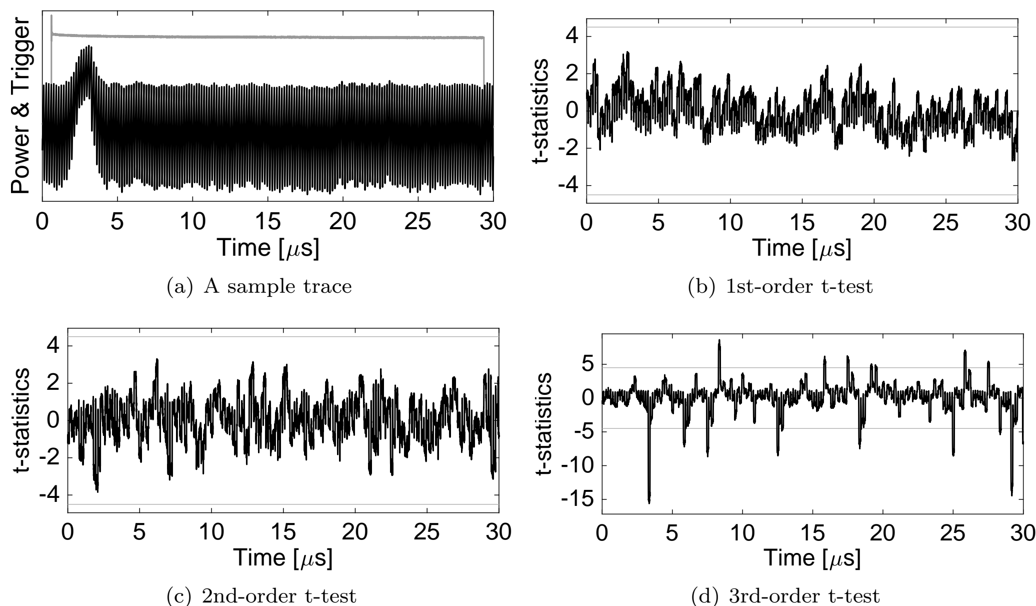
This however is an impossible task when considering full cipher implementations. Hence, for the remaining options, we performed Field Programmable Gate Array (FPGA)-based experimental analyses. Naturally, it is not possible to examine all designs reported as case study. However, since the security of designs constructed by AGEMA is based on composability of PINI gadgets, we contented ourselves with two exemplary designs of our masked Skinny round-based designs. The first design is first-order GHPC ANF pipeline and the second design is second-order HPC2 Naive pipeline. We made use of SAKURA-G [SAK] and implemented the selected designs on the target FPGA to monitor their power consumption by a digital sampling oscilloscope at a sampling frequency of 500 MS/s. During the measurements the target design was driven by a 6 MHz stable clock. The fresh masks have also been generated internally (inside the target FPGA). For each required fresh mask bit we instantiated a 31-bit Linear Feedback Shift Register (LFSR) initialized randomly[6].

The conduced analyses are based on the common and well-known TVLA [GJJR11], where the SCA leakages associated to a fixed input are compared to those associated to random inputs, i.e., fixed versus random t-test. Conducting such analyses at first three orders using 100 million traces led to the results shown in Figure 8 and Figure 9 for two aforementioned design, respectively. As the first design is first order ($d = 1$), higher-order detected leakage is expected, as it can also be seen in the corresponding figures. The second design is second order ($d = 2$) and, as shown, no first- and no second-order leakage is detected.

6 Conclusions

In this work we developed a comprehensive framework for *automated generation of masked hardware* (AGEMA), allowing engineers and hardware designers of all levels of experience to easily create securely masked cryptographic hardware circuits. Based on the security and composability notion of PINI, our tool explores different processing techniques to

[6]We have taken the FPGA-optimized LFSR design presented in [DMW18].



(a) A sample trace

(b) 1st-order t-test

(c) 2nd-order t-test

(d) 3rd-order t-test

Figure 9: FPGA-based fixed versus random t-test using 100 million traces, Skinny-64-64 round-based encryption, second-order HPC2 Naive pipeline.

transform any unprotected cryptographic design into a securely masked circuit using different masked gadgets as fundamental building blocks.

Demonstrating the benefits and limitations of our proposed tool, we provide several case studies for well-established symmetric block ciphers, showing different performance trade-offs in terms of area overhead, latency increase, and fresh entropy demands based on our proposed transformation methodologies. Eventually, verifying the viability of our tool and the security of the resulting masked circuits, we perform practical experiments and evaluations that confirm our claims. For this, AGEMA is an important building block towards security-aware Electronic Design Automation (EDA), assisting in the automation process of creating secure ICs.

Apart from unique benefits and facilities that AGEMA offers, the intensive case studies, which we have provided in this article, highlight the importance of the employed gadgets with respect to their performance. The demands for fresh randomness and the latency of the constructed masked circuits heavily depend on the gadget types and their requirements. In terms of latency, $\mathsf{GHPC}_{\mathsf{LL}}$ gadgets are the only known constructions with only a single additional register stage, but they are limited to only first-order security. In contrast, HPC2 gadgets, which can arbitrarily be adjusted to any security order, add two register stages to the circuit. This might be seen as just one more clock cycle, but as shown by our case studies, the latency of the resulting masked circuit is doubled compared to that with $\mathsf{GHPC}_{\mathsf{LL}}$. This difference is seen more clearly when considering ciphers which employ S-boxes with a high algebraic degree (i.e., a high depth of non-linear gadgets). Naturally, more research in this area is required to fill the gap.

Acknowledgments

The work described in this paper has been supported in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and through the projects 393207943 GreenSec and 435264177 SAUBER.

References

[Ake78] Sheldon B. Akers. Binary Decision Diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[Apt03] Krzysztof R. Apt. *Principles of constraint programming.* Cambridge University Press, 2003.

[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 2019*, 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 2015*, 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 2016*, pages 116–129. ACM, 2016.

[BBI+15] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A Block Cipher for Low Energy. In *ASIACRYPT 2015*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436. Springer, 2015.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In *ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model. In *EUROCRYPT 2017*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, 2017.

[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 2020*, volume 12107 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.

[BGG+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the Cost of Lazy Engineering for Masked Software Implementations. In *CARDIS 2014*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014.

[BGI+18a] 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 2018*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

[BGI+18b] Roderick Bloem, Hannes Groß, Rinat Iusupov, Martin Krenn, and Stefan Mangard. Sharing Independence & Relabeling: Efficient Formal Verification of Higher-Order Masking. *IACR Cryptol. ePrint Arch.*, 2018:1031, 2018.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In *CRYPTO 2016*, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 2016.

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BLMR19] Christof Beierle, Gregor Leander, Amir Moradi, and Shahram Rasoolzadeh. CRAFT: Lightweight Tweakable Block Cipher with Efficient Protection Against DFA Attacks. *IACR Trans. Symmetric Cryptol.*, 2019(1):5–45, 2019.

[BP12] Joan Boyar and René Peralta. A Small Depth-16 Circuit for the AES S-Box. In *Information Security and Privacy Conference, SEC 2012*, volume 376 of *IFIP*, pages 287–298. Springer, 2012.

[BRB+11] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A first step towards automatic application of power analysis countermeasures. In *DAC 2011*, pages 230–235. ACM, 2011.

[BRN+15] Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. Automatic Application of Power Analysis Countermeasures. *IEEE Trans. Computers*, 64(2):329–341, 2015.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[Can05] David Canright. A Very Compact S-Box for AES. In Josyula R. Rao and Berk Sunar, editors, *CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.

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

[CGD18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors. In *COSADE 2018*, volume 10815 of *Lecture Notes in Computer Science*, pages 82–98. Springer, 2018.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware Private Circuits: From Trivial Composition to Full Verification. *IEEE Trans. on Computers*, 2021.

[CGP+12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. Conversion of Security Proofs from One Leakage Model to Another: A New Issue. In *COSADE 2012*, volume 7275 of *Lecture Notes in Computer Science*, pages 69–81. Springer, 2012.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *CRYPTO '99*, 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 2013*, 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. Information Forensics and Security*, 15:2542–2555, 2020.

[DMW18] Lauren De Meyer, Amir Moradi, and Felix Wegener. Spin Me Right Round Rotational Symmetry for FPGA-Specific AES. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):596–626, 2018.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[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.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for sidechannel resistance validation. In *NIST non-invasive attack testing workshop*, 2011.

[GM18] Hannes Groß and Stefan Mangard. A unified masking approach. *J. Cryptogr. Eng.*, 8(2):109–124, 2018.

[GMK17] Hannes Groß, Stefan Mangard, and Thomas Korak. An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order. In *CT-RSA 2017*, volume 10159 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2017.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In *CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED Block Cipher. In *CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011.

[HS13] Michael Hutter and Jörn-Marc Schmidt. The Temperature Side Channel and Heating Fault Attacks. In *CARDIS 2013*, volume 8419 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2013.

[Inc] Synopsys Inc. Design compiler graphical. https://www.synopsys.com/.

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

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - Statistical Independence and Leakage Verification. In *ASIACRYPT 2020*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[KSM21] David Knichel, Pascal Sasdrich, and Amir Moradi. Generic Hardware Private Circuits - Towards Automated Generation of Composable Secure Gadgets. Cryptology ePrint Archive, Report 2021/247, 2021. https://eprint.iacr.org/2021/247.

[Mea55] George H Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[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 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.

[MS16] Amir Moradi and Tobias Schneider. Side-Channel Analysis Protection and Low-Latency in Action - - Case Study of PRINCE and Midori -. In *ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*, pages 517–547, 2016.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In *ICICS 2006*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[PMK+11] Axel Poschmann, Amir Moradi, Khoongming Khoo, Chu-Wee Lim, Huaxiong Wang, and San Ling. Side-Channel Resistant Crypto for Less than 2, 300 GE. *J. Cryptology*, 24(2):322–345, 2011.

[RBN+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating Masking Schemes. In *CRYPTO 2015*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

[SAK] SAKURA. Side-channel Attack User Reference Architecture. http://satoh.cs.uec.ac.jp/SAKURA/index.html.

[SM20] Aein Rezaei Shahmirzadi and Amir Moradi. Re-Consolidating First-Order Masking Schemes - Nullifying Fresh Randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):305–342, 2020.

[SSB+21] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. In *NDSS 2021*. The Internet Society, 2021.

[UHA17] Rei Ueno, Naofumi Homma, and Takafumi Aoki. Toward More Efficient DPA-Resistant AES Hardware Architecture Based on Threshold Implementation. In *COSADE 2017*, Lecture Notes in Computer Science, pages 50–64. Springer, 2017.

[Wol] Clifford Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`.

A Performance Results

Table 3: Synthesis results, Skinny Sbox lookup-table representation.

Masking Scheme	Processing Method	Pipeline	Order	Area GE	Delay ns	Random bits	Latency cycles
unprotected	-	-	0	42	0.20		
$GHPC_{LL}$	ANF		1	962	0.39	64	1
$GHPC_{LL}$	ANF	✓	1	1093	0.39	64	1
$GHPC_{LL}$	Naive		1	809	0.42	52	5
$GHPC_{LL}$	Naive	✓	1	1172	0.41	52	5
GHPC	ANF		1	1305	0.53	4	2
GHPC	ANF	✓	1	1270	0.53	4	2
GHPC	Naive		1	1137	0.30	13	10
GHPC	Naive	✓	1	1870	0.29	13	10
HPC1	Naive		1	898	0.30	26	10
HPC1	Naive		2	1854	0.34	65	10
HPC1	Naive		3	3065	0.34	130	10
HPC1	Naive		4	4501	0.39	195	10
HPC1	Naive	✓	1	1488	0.29	26	10
HPC1	Naive	✓	2	2778	0.33	65	10
HPC1	Naive	✓	3	4323	0.33	130	10
HPC1	Naive	✓	4	6094	0.38	195	10
HPC2	AIG		1	1065	0.29	17	12
HPC2	AIG		2	2856	0.36	51	12
HPC2	AIG		3	5520	0.41	102	12
HPC2	AIG		4	9006	0.49	170	12
HPC2	AIG	✓	1	2390	0.28	17	12
HPC2	AIG	✓	2	5178	0.35	51	12
HPC2	AIG	✓	3	9032	0.41	102	12
HPC2	AIG	✓	4	13898	0.49	170	12
HPC2	BDD_{CUDD}		1	1083	0.34	17	8
HPC2	BDD_{CUDD}		2	2879	0.43	51	8
HPC2	BDD_{CUDD}		3	5535	0.52	102	8
HPC2	BDD_{CUDD}		4	9009	0.57	170	8
HPC2	BDD_{CUDD}	✓	1	2306	0.34	17	8
HPC2	BDD_{CUDD}	✓	2	5025	0.42	51	8
HPC2	BDD_{CUDD}	✓	3	8792	0.51	102	8
HPC2	BDD_{CUDD}	✓	4	13567	0.57	170	8
HPC2	BDD_{SYLVAN}		1	1307	0.36	21	8
HPC2	BDD_{SYLVAN}		2	3517	0.44	63	8
HPC2	BDD_{SYLVAN}		3	6789	0.52	126	8
HPC2	BDD_{SYLVAN}		4	11069	0.61	210	8
HPC2	BDD_{SYLVAN}	✓	1	2748	0.35	21	8
HPC2	BDD_{SYLVAN}	✓	2	6047	0.44	63	8
HPC2	BDD_{SYLVAN}	✓	3	10650	0.51	126	8
HPC2	BDD_{SYLVAN}	✓	4	16493	0.60	210	8
HPC2	Naive		1	847	0.35	13	10
HPC2	Naive		2	2236	0.42	39	10
HPC2	Naive		3	4287	0.47	78	10
HPC2	Naive		4	6968	0.56	130	10
HPC2	Naive	✓	1	1890	0.34	13	10
HPC2	Naive	✓	2	4055	0.41	39	10
HPC2	Naive	✓	3	7034	0.47	78	10
HPC2	Naive	✓	4	10790	0.55	130	10

Table 4: Synthesis results, Skinny Sbox optimized representation.

Masking Scheme	Processing Method	Pipeline	Order	Area GE	Delay ns	Random bits	Latency cycles
unprotected	-	-	0	40	0.20		
$GHPC_{LL}$	ANF		1	962	0.39	64	1
$GHPC_{LL}$	ANF	✓	1	1093	0.39	64	1
$GHPC_{LL}$	Naive		1	335	0.40	16	2
$GHPC_{LL}$	Naive	✓	1	480	0.39	16	2
GHPC	ANF		1	1305	0.53	4	2
GHPC	ANF	✓	1	1270	0.53	4	2
GHPC	Naive		1	438	0.30	4	4
GHPC	Naive	✓	1	739	0.29	4	4
HPC1	Naive		1	365	0.26	8	4
HPC1	Naive		2	698	0.31	20	4
HPC1	Naive		3	1110	0.30	40	4
HPC1	Naive		4	1591	0.35	60	4
HPC1	Naive	✓	1	621	0.26	8	4
HPC1	Naive	✓	2	1101	0.30	20	4
HPC1	Naive	✓	3	1661	0.29	40	4
HPC1	Naive	✓	4	2289	0.34	60	4
HPC2	AIG		1	350	0.33	4	4
HPC2	AIG		2	813	0.40	12	4
HPC2	AIG		3	1482	0.44	24	4
HPC2	AIG		4	2343	0.53	40	4
HPC2	AIG	✓	1	745	0.33	4	4
HPC2	AIG	✓	2	1493	0.39	12	4
HPC2	AIG	✓	3	2491	0.44	24	4
HPC2	AIG	✓	4	3727	0.52	40	4
HPC2	BDD_{CUDD}		1	1072	0.36	17	8
HPC2	BDD_{CUDD}		2	2862	0.43	51	8
HPC2	BDD_{CUDD}		3	5515	0.51	102	8
HPC2	BDD_{CUDD}		4	8979	0.59	170	8
HPC2	BDD_{CUDD}	✓	1	2225	0.35	17	8
HPC2	BDD_{CUDD}	✓	2	4906	0.42	51	8
HPC2	BDD_{CUDD}	✓	3	8633	0.50	102	8
HPC2	BDD_{CUDD}	✓	4	13366	0.59	170	8
HPC2	BDD_{SYLVAN}		1	1072	0.36	17	8
HPC2	BDD_{SYLVAN}		2	2862	0.42	51	8
HPC2	BDD_{SYLVAN}		3	5515	0.51	102	8
HPC2	BDD_{SYLVAN}		4	8979	0.60	170	8
HPC2	BDD_{SYLVAN}	✓	1	2225	0.35	17	8
HPC2	BDD_{SYLVAN}	✓	2	4906	0.42	51	8
HPC2	BDD_{SYLVAN}	✓	3	8633	0.50	102	8
HPC2	BDD_{SYLVAN}	✓	4	13366	0.59	170	8
HPC2	Naive		1	353	0.32	4	4
HPC2	Naive		2	818	0.39	12	4
HPC2	Naive		3	1489	0.44	24	4
HPC2	Naive		4	2351	0.52	40	4
HPC2	Naive	✓	1	747	0.31	4	4
HPC2	Naive	✓	2	1497	0.38	12	4
HPC2	Naive	✓	3	2498	0.43	24	4
HPC2	Naive	✓	4	3736	0.52	40	4

Table 5: Synthesis results, AES Sbox lookup-table representation.

Masking Scheme	Processing Method	Pipeline	Order	Area	Delay	Random	Latency
				GE	ns	bits	cycles
unprotected	-	-	0	664	0.35		
GHPC_{LL}	ANF		1	150409	0.57	2048	1
GHPC_{LL}	ANF	✓	1	157377	1.11	2048	1
GHPC_{LL}	Naive		1	46476	0.50	3472	17
GHPC_{LL}	Naive	✓	1	65708	0.49	3472	17
GHPC	ANF		1	157808	0.92	8	2
GHPC	ANF	✓	1	135228	0.82	8	2
GHPC	Naive		1	66226	0.40	868	34
GHPC	Naive	✓	1	104448	0.39	868	34
HPC1	Naive		1	50267	0.33	1736	34
HPC1	Naive		2	111763	0.37	4340	34
HPC1	Naive		3	190377	0.38	8680	34
HPC1	Naive		4	284043	0.42	13020	34
HPC1	Naive	✓	1	79087	0.32	1736	34
HPC1	Naive	✓	2	155064	0.36	4340	34
HPC1	Naive	✓	3	248114	0.36	8680	34
HPC1	Naive	✓	4	356210	0.41	13020	34
HPC2	AIG		1	47745	0.46	879	34
HPC2	AIG		2	139100	0.53	2637	34
HPC2	AIG		3	275707	0.55	5274	34
HPC2	AIG		4	455262	4.20	8790	34
HPC2	AIG	✓	1	106178	0.45	879	34
HPC2	AIG	✓	2	241813	0.52	2637	34
HPC2	AIG	✓	3	432582	0.67	5274	34
HPC2	AIG	✓	4	675987	0.76	8790	34
HPC2	BDD_{CUDD}		1	24841	0.55	406	16
HPC2	BDD_{CUDD}		2	68416	0.56	1218	16
HPC2	BDD_{CUDD}		3	132834	2.72	2436	16
HPC2	BDD_{CUDD}		4	216733	3.23	4060	16
HPC2	BDD_{CUDD}	✓	1	53471	0.54	406	16
HPC2	BDD_{CUDD}	✓	2	118318	0.55	1218	16
HPC2	BDD_{CUDD}	✓	3	208765	0.69	2436	16
HPC2	BDD_{CUDD}	✓	4	323122	0.78	4060	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$		1	25077	0.52	410	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$		2	69065	0.57	1230	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$		3	134122	2.65	2460	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$		4	218861	3.20	4100	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$	✓	1	53753	0.51	410	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$	✓	2	119134	0.55	1230	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$	✓	3	210328	0.69	2460	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$	✓	4	325669	0.79	4100	16
HPC2	Naive		1	46854	0.48	868	34
HPC2	Naive		2	137437	0.60	2604	34
HPC2	Naive		3	272327	0.82	5208	34
HPC2	Naive		4	449461	4.33	8680	34
HPC2	Naive	✓	1	105908	0.44	868	34
HPC2	Naive	✓	2	240470	0.58	2604	34
HPC2	Naive	✓	3	429483	0.67	5208	34
HPC2	Naive	✓	4	670365	0.76	8680	34

Table 6: Synthesis results, AES Sbox Canright representation.

Masking Scheme	Processing Method	Pipeline	Order	Area	Delay	Random	Latency
				GE	ns	bits	cycles
unprotected	-	-	0	246	0.39		
GHPC_{LL}	ANF		1	150409	0.57	2048	1
GHPC_{LL}	ANF	✓	1	157377	1.11	2048	1
GHPC_{LL}	Naive		1	46476	0.50	3472	17
GHPC_{LL}	Naive	✓	1	65708	0.49	3472	17
GHPC	ANF		1	157808	0.92	8	2
GHPC	ANF	✓	1	135228	0.82	8	2
GHPC	Naive		1	66226	0.40	868	34
GHPC	Naive	✓	1	104448	0.39	868	34
HPC1	Naive		1	50267	0.33	1736	34
HPC1	Naive		2	111763	0.37	4340	34
HPC1	Naive		3	190377	0.38	8680	34
HPC1	Naive		4	284043	0.42	13020	34
HPC1	Naive	✓	1	79087	0.32	1736	34
HPC1	Naive	✓	2	155064	0.36	4340	34
HPC1	Naive	✓	3	248114	0.36	8680	34
HPC1	Naive	✓	4	356210	0.41	13020	34
HPC2	AIG		1	47745	0.46	879	34
HPC2	AIG		2	139100	0.53	2637	34
HPC2	AIG		3	275707	0.55	5274	34
HPC2	AIG		4	455262	4.20	8790	34
HPC2	AIG	✓	1	106178	0.45	879	34
HPC2	AIG	✓	2	241813	0.52	2637	34
HPC2	AIG	✓	3	432582	0.67	5274	34
HPC2	AIG	✓	4	675987	.76	8790	34
HPC2	BDD_{CUDD}		1	24841	0.55	406	16
HPC2	BDD_{CUDD}		2	68416	0.56	1218	16
HPC2	BDD_{CUDD}		3	132834	2.72	2436	16
HPC2	BDD_{CUDD}		4	216733	3.23	4060	16
HPC2	BDD_{CUDD}	✓	1	53471	0.54	406	16
HPC2	BDD_{CUDD}	✓	2	118318	0.55	1218	16
HPC2	BDD_{CUDD}	✓	3	208765	0.69	2436	16
HPC2	BDD_{CUDD}	✓	4	323122	.78	4060	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$		1	25077	0.52	410	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$		2	69065	0.57	1230	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$		3	134122	2.65	2460	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$		4	218861	3.20	4100	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$	✓	1	53753	0.51	410	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$	✓	2	119134	0.55	1230	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$	✓	3	210328	0.69	2460	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$	✓	4	325669	0.79	4100	16
HPC2	Naive		1	46854	0.48	868	34
HPC2	Naive		2	137437	0.60	2604	34
HPC2	Naive		3	272327	0.82	5208	34
HPC2	Naive		4	449461	4.33	8680	34
HPC2	Naive	✓	1	105908	0.44	868	34
HPC2	Naive	✓	2	240470	0.58	2604	34
HPC2	Naive	✓	3	429483	0.67	5208	34
HPC2	Naive	✓	4	670365	0.76	8680	34

Table 7: Synthesis results, AES Sbox optimized representation.

Masking Scheme	Processing Method	Pipeline	Order	Area GE	Delay ns	Random bits	Latency cycles
unprotected	-	-	0	299	150		
$\mathsf{GHPC}_{\mathsf{LL}}$	ANF		1	150409	0.57	2048	1
$\mathsf{GHPC}_{\mathsf{LL}}$	ANF	✓	1	157377	1.11	2048	1
$\mathsf{GHPC}_{\mathsf{LL}}$	Naive		1	2311	1.41	136	4
$\mathsf{GHPC}_{\mathsf{LL}}$	Naive	✓	1	3382	0.64	136	4
GHPC	ANF		1	157808	0.92	8	2
GHPC	ANF	✓	1	135228	0.82	8	2
GHPC	Naive		1	3074	1.09	34	8
GHPC	Naive	✓	1	5271	0.52	34	8
HPC1	Naive		1	2448	0.92	68	8
HPC1	Naive		2	5084	1.02	170	8
HPC1	Naive		3	8388	1.07	340	8
HPC1	Naive		4	12307	1.17	510	8
HPC1	Naive	✓	1	4263	0.40	68	8
HPC1	Naive	✓	2	7839	0.44	170	8
HPC1	Naive	✓	3	12085	0.44	340	8
HPC1	Naive	✓	4	16919	0.51	510	8
HPC2	AIG		1	2348	1.32	34	8
HPC2	AIG		2	6120	1.61	102	8
HPC2	AIG		3	11718	1.66	204	8
HPC2	AIG		4	18897	1.93	340	8
HPC2	AIG	✓	1	5342	0.51	34	8
HPC2	AIG	✓	2	11208	0.64	102	8
HPC2	AIG	✓	3	19233	0.72	204	8
HPC2	AIG	✓	4	29267	0.80	340	8
HPC2	$\mathsf{BDD}_{\mathsf{CUDD}}$		1	25161	2.08	411	16
HPC2	$\mathsf{BDD}_{\mathsf{CUDD}}$		2	69291	2.48	1233	16
HPC2	$\mathsf{BDD}_{\mathsf{CUDD}}$		3	134490	2.70	2466	16
HPC2	$\mathsf{BDD}_{\mathsf{CUDD}}$		4	219462	3.24	4110	16
HPC2	$\mathsf{BDD}_{\mathsf{CUDD}}$	✓	1	54076	0.57	411	16
HPC2	$\mathsf{BDD}_{\mathsf{CUDD}}$	✓	2	119704	0.63	1233	16
HPC2	$\mathsf{BDD}_{\mathsf{CUDD}}$	✓	3	211169	0.69	2466	16
HPC2	$\mathsf{BDD}_{\mathsf{CUDD}}$	✓	4	326936	0.77	4110	16
HPC2	$\mathsf{BDD}_{\mathsf{SYLVAN}}$		1	25072	2.08	410	16
HPC2	$\mathsf{BDD}_{\mathsf{SYLVAN}}$		2	69081	2.55	1230	16
HPC2	$\mathsf{BDD}_{\mathsf{SYLVAN}}$		3	134122	2.65	2460	16
HPC2	$\mathsf{BDD}_{\mathsf{SYLVAN}}$		4	218861	3.20	4100	16
HPC2	$\mathsf{BDD}_{\mathsf{SYLVAN}}$	✓	1	53764	0.54	410	16
HPC2	$\mathsf{BDD}_{\mathsf{SYLVAN}}$	✓	2	119135	0.63	1230	16
HPC2	$\mathsf{BDD}_{\mathsf{SYLVAN}}$	✓	3	210328	0.69	2460	16
HPC2	$\mathsf{BDD}_{\mathsf{SYLVAN}}$	✓	4	325669	0.79	4100	16
HPC2	Naive		1	2346	1.32	34	8
HPC2	Naive		2	6126	1.61	102	8
HPC2	Naive		3	11716	1.68	204	8
HPC2	Naive		4	18894	1.99	340	8
HPC2	Naive	✓	1	5339	0.51	34	8
HPC2	Naive	✓	2	11205	0.61	102	8
HPC2	Naive	✓	3	19217	0.68	204	8
HPC2	Naive	✓	4	29267	0.74	340	8

Table 8: Synthesis results, Skinny-64-64 round-based encryption function.

Masking Scheme	Processing Method	Pipeline	Order	Area	Delay	Random	Latency
				GE	ns	bits	cycles
unprotected	-	-	0	1494	0.52		(33)
GHPC_{LL}	ANF		1	18705	0.85	1024	1
GHPC_{LL}	ANF	✓	1	18789	0.85	1024	1
GHPC_{LL}	Naive		1	6817	0.48	256	2
GHPC_{LL}	Naive	✓	1	12725	0.46	256	2
GHPC	ANF		1	22850	0.80	64	2
GHPC	ANF	✓	1	28850	0.80	64	2
GHPC	Naive		1	8260	0.46	64	4
GHPC	Naive	✓	1	20082	0.45	64	4
HPC2	AIG		1	6919	0.86	64	4
HPC2	AIG	✓	1	20234	0.54	64	4
HPC2	BDD_{CUDD}		1	18832	1.95	280	16
HPC2	BDD_{CUDD}	✓	1	68410	0.52	280	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$		1	17969	1.96	262	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$	✓	1	66933	0.52	262	16
HPC2	Naive		1	6895	0.55	64	4
HPC2	Naive		2	15193	0.61	192	4
HPC2	Naive		3	26777	0.65	384	4
HPC2	Naive	✓	1	20210	0.53	64	4
HPC2	Naive	✓	2	36147	0.59	192	4
HPC2	Naive	✓	3	56096	0.63	384	4

Table 9: Synthesis results, AES byte-serial encryption function.

Masking Scheme	Processing Method	Pipeline	Order	Area	Delay	Random	Latency
				GE	ns	bits	cycles
unprotected	-	-	0	3263	0.83		(227)
GHPC_{LL}	ANF		1	161922	2.67	2048	1
GHPC_{LL}	ANF	✓	1	143778	2.67	2048	1
GHPC_{LL}	Naive		1	10056	2.34	136	4
GHPC_{LL}	Naive	✓	1	25656	0.91	136	4
GHPC	ANF		1	146894	2.67	8	2
GHPC	ANF	✓	1	176509	2.83	8	2
GHPC	Naive		1	10818	1.73	34	8
GHPC	Naive	✓	1	42078	0.91	34	8
HPC2	AIG		1	10097	2.19	34	8
HPC2	AIG	✓	1	42148	1.23	34	8
HPC2	BDD_{CUDD}		1	33124	2.56	414	16
HPC2	BDD_{CUDD}	✓	1	120293	1.16	414	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$		1	34173	2.64	431	16
HPC2	$\text{BDD}_{\text{SYLVAN}}$	✓	1	122566	0.97	431	16
HPC2	Naive		1	10090	2.11	34	8
HPC2	Naive		2	17649	2.66	102	8
HPC2	Naive		3	27026	2.71	204	8
HPC2	Naive	✓	1	42146	0.98	34	8
HPC2	Naive	✓	2	65583	1.41	102	8
HPC2	Naive	✓	3	91149	1.01	204	8

Table 10: Synthesis results, AES round-based encryption function.

Masking Scheme	Processing Method	Pipeline	Order	Area	Delay	Random	Latency
				GE	ns	bits	cycles
unprotected	-	-	0	9906	1.85		(11)
$GHPC_{LL}$	Naive		1	52450	2.28	2720	4
$GHPC_{LL}$	Naive	✓	1	98448	0.84	2720	4
GHPC	Naive		1	67193	1.48	680	8
GHPC	Naive	✓	1	160080	0.83	680	8
HPC2	AIG		1	52726	2.07	680	8
HPC2	AIG	✓	1	161538	0.87	680	8
HPC2	Naive		1	52597	2.04	680	8
HPC2	Naive		2	131631	2.39	2040	8
HPC2	Naive		3	246924	2.53	4080	8
HPC2	Naive	✓	1	161440	0.82	680	8
HPC2	Naive	✓	2	305274	0.89	2040	8
HPC2	Naive	✓	3	492077	0.93	4080	8

Table 11: Synthesis results, CRAFT round-based encryption function.

Masking Scheme	Processing Method	Pipeline	Order	Area	Delay	Random	Latency
				GE	ns	bits	cycles
unprotected	-	-	0	1066	0.58		(32)
GHPC	ANF		1	22106	0.75	64	2
GHPC	ANF	✓	1	27214	0.66	64	2
GHPC	Naive		1	21365	0.63	256	8
$GHPC_{LL}$	ANF		1	15748	0.81	1024	1
$GHPC_{LL}$	ANF	✓	1	17605	0.63	1024	1
$GHPC_{LL}$	Naive		1	15568	1.01	1024	4
$GHPC_{LL}$	Naive	✓	1	25852	0.54	1024	4
GHPC	Naive	✓	1	41951	0.54	256	8
HPC2	BDD_{CUDD}		1	14927	1.13	229	8
HPC2	BDD_{CUDD}	✓	1	42451	0.55	229	8
HPC2	BDD_{SYLVAN}		1	17509	1.16	272	8
HPC2	BDD_{SYLVAN}	✓	1	47785	0.55	272	8
HPC2	Naive		1	15680	0.94	256	8
HPC2	Naive		2	43172	1.03	768	8
HPC2	Naive		3	84024	1.12	1536	8
HPC2	Naive	✓	1	42367	0.55	256	8
HPC2	Naive	✓	2	87291	0.57	768	8
HPC2	Naive	✓	3	148316	0.50	1536	8

Table 12: Synthesis results, PRESENT nibble-serial encryption function.

Masking Scheme	Processing Method	Pipeline	Order	Area GE	Delay ns	Random bits	Latency cycles
unprotected	-	-	0	1613	0.59		(543)
$\mathsf{GHPC_{LL}}$	ANF		1	4727	0.89	64	1
$\mathsf{GHPC_{LL}}$	ANF	✓	1	6734	0.68	64	1
$\mathsf{GHPC_{LL}}$	Naive		1	4143	1.03	16	2
$\mathsf{GHPC_{LL}}$	Naive	✓	1	8061	0.59	16	2
GHPC	ANF		1	5177	1.04	4	2
GHPC	ANF	✓	1	8945	0.70	4	2
GHPC	Naive		1	4245	0.68	4	4
GHPC	Naive	✓	1	12095	0.59	4	4
HPC2	AIG		1	4162	0.98	4	4
HPC2	AIG	✓	1	12104	0.59	4	4
HPC2	$\mathsf{BDD_{CUDD}}$		1	5180	1.26	22	8
HPC2	$\mathsf{BDD_{CUDD}}$	✓	1	21966	0.59	22	8
HPC2	$\mathsf{BDD_{SYLVAN}}$		1	5245	1.30	23	8
HPC2	$\mathsf{BDD_{SYLVAN}}$	✓	1	22064	0.59	23	8
HPC2	Naive		1	4160	0.99	4	4
HPC2	Naive		2	6478	1.13	12	4
HPC2	Naive		3	8977	1.18	24	4
HPC2	Naive	✓	1	12103	0.59	4	4
HPC2	Naive	✓	2	18270	0.55	12	4
HPC2	Naive	✓	3	24692	0.67	24	4

Table 13: Synthesis results, LED-64 round-based encryption function.

Masking Scheme	Processing Method	Pipeline	Order	Area GE	Delay ns	Random bits	Latency cycles
unprotected	-	-	0	2056	150		(33)
$\mathsf{GHPC_{LL}}$	ANF		1	17382	150	1024	1
$\mathsf{GHPC_{LL}}$	ANF	✓	1	19893	150	1024	1
$\mathsf{GHPC_{LL}}$	Naive		1	7611	150	256	2
$\mathsf{GHPC_{LL}}$	Naive	✓	1	13383	150	256	2
GHPC	ANF		1	22904	150	64	2
GHPC	ANF	✓	1	27309	150	64	2
GHPC	Naive		1	9056	150	64	4
GHPC	Naive	✓	1	20615	150	64	4
HPC2	AIG		1	7714	150	64	4
HPC2	AIG	✓	1	20767	150	64	4
HPC2	$\mathsf{BDD_{CUDD}}$		1	31416	150	469	16
HPC2	$\mathsf{BDD_{CUDD}}$	✓	1	96238	150	469	16
HPC2	$\mathsf{BDD_{SYLVAN}}$		1	38243	150	598	16
HPC2	$\mathsf{BDD_{SYLVAN}}$	✓	1	110725	150	598	16
HPC2	Naive		1	7691	150	64	4
HPC2	Naive		2	16375	150	192	4
HPC2	Naive		3	28322	150	384	4
HPC2	Naive	✓	1	20743	150	64	4
HPC2	Naive	✓	2	36890	150	192	4
HPC2	Naive	✓	3	57021	150	384	4

Table 14: Synthesis results, Midori-64 round-based encryption/decryption function.

Masking Scheme	Processing Method	Pipeline	Order	Area GE	Delay ns	Random bits	Latency cycles
unprotected	-	-	0	2035	0.97		(17)
$\mathsf{GHPC}_{\mathsf{LL}}$	ANF		1	19493	1.08	1024	1
$\mathsf{GHPC}_{\mathsf{LL}}$	ANF	✓	1	21986	0.94	1024	1
$\mathsf{GHPC}_{\mathsf{LL}}$	Naive		1	17679	1.10	1024	4
$\mathsf{GHPC}_{\mathsf{LL}}$	Naive	✓	1	32898	0.95	1024	4
GHPC	ANF		1	23901	1.05	64	2
GHPC	ANF	✓	1	30539	0.85	64	2
GHPC	Naive		1	23508	0.96	256	8
GHPC	Naive	✓	1	53893	0.95	256	8
HPC2	AIG		1	17971	1.02	256	8
HPC2	AIG	✓	1	54824	0.95	256	8
HPC2	$\mathsf{BDD}_{\mathsf{CUDD}}$		1	17162	1.29	231	8
HPC2	$\mathsf{BDD}_{\mathsf{CUDD}}$	✓	1	53478	0.95	231	8
HPC2	$\mathsf{BDD}_{\mathsf{SYLVAN}}$		1	21123	1.27	304	8
HPC2	$\mathsf{BDD}_{\mathsf{SYLVAN}}$	✓	1	61576	0.95	304	8
HPC2	Naive		1	17801	1.10	256	8
HPC2	Naive		2	46371	1.21	768	8
HPC2	Naive		3	88246	1.27	1536	8
HPC2	Naive	✓	1	54309	0.95	256	8
HPC2	Naive	✓	2	105198	0.67	768	8
HPC2	Naive	✓	3	172179	0.69	1536	8