Automated Generation of Fault-Resistant Circuits

Nicolai Müller[1] and Amir Moradi[2]

[1] Ruhr University Bochum, Horst Görtz Institute for IT Security, Bochum, Germany
firstname.lastname@rub.de
[2] Technische Universität Darmstadt, Darmstadt, Germany
firstname.lastname@tu-darmstadt.de

Abstract. Fault Injection (FI) attacks, which involve intentionally introducing faults into a system to cause it to behave in an unintended manner, are widely recognized and pose a significant threat to the security of cryptographic primitives implemented in hardware, making fault tolerance an increasingly critical concern. However, protecting cryptographic hardware primitives securely and efficiently, even with well-established and documented methods such as redundant computation, can be a time-consuming, error-prone, and expertise-demanding task. In this research, we present a comprehensive and fully-automated software solution for the Automated Generation of Fault-Resistant Circuits (AGEFA). Our application employs a generic and extensively researched methodology for the secure integration of countermeasures based on Error-Correcting Codes (ECCs) into cryptographic hardware circuits. Our software tool allows designers without hardware security expertise to develop fault-tolerant hardware circuits with pre-defined correction capabilities under a comprehensive fault adversary model. Moreover, our tool applies to masked designs without violating the masking security requirements, in particular to designs generated by the tool AGEMA. We evaluate the effectiveness of our approach through experiments on various block ciphers and demonstrate its ability to produce fault-tolerant circuits. Additionally, we assess the security of examples generated by AGEFA against Side-Channel Analysis (SCA) and FI using state-of-the-art leakage and fault evaluation tools.

Keywords: Fault Analysis · Impeccable Circuits · SIFA · Hardware · Masking

1 Introduction

The already well-advanced integration of embedded systems into our daily lives highlights the critical need for reliable guarantees with respect to the confidentiality of sensitive data processed by these systems. Cryptographic primitives, such as block ciphers, are well-established methods that ensure the confidentiality of data both at rest and in transit. However, the implementation of cryptographic primitives in hardware presents significant challenges that are yet to be fully resolved. Physical attacks, such as passive Side-Channel Analysis (SCA) attacks [Koc96] and active Fault Injection (FI) attacks [BDL97], have emerged as severe attack vectors due to the physical accessibility that embedded devices offer to potential adversaries.

In the context of SCA, an adversary records a certain physical characteristic of the device during the execution of a cryptographic primitive and subsequently exploits the relation between the measurements and the processed data to recover secret information. Examples include but are not restricted to [Koc96, KJJ99, GMO01, HS13, GST14]. Masking [CJRR99], as a well-studied countermeasure against SCA based on secret sharing [Sha79], has gained a considerable amount of attention from the scientific community due to its simple security assumptions and adversary models. The d-probing model [ISW03]

and its extension to cover physical defaults, known as the robust probing model [FGP+18], are the commonly used adversary models and form the basis for security notions such as probing security and composability, which enable to prove the formal security of masked implementations. However, designing and implementing a properly masked cryptographic primitive is complex, error-prone, and tedious. Many examples of insecure masking schemes can be found in the literature [MMSS19], highlighting the need for automated solutions that reduce manual interaction and the potential for human error. One such solution is the Automated Generation of Masked Hardware (AGEMA) [KMMS22], which automates the generation of masked hardware circuits by translating unprotected gate-level netlists into masked hardware circuits albeit with an associated increase in circuit size and latency compared to manually crafted masked designs. To narrow the performance gap in terms of area and latency between automatically-generated masked designs and manual approaches, various optimizations have been introduced and implemented into new software tools like the Automated Generation of Masked Nonlinear Components (AGMNC) [WFP+23] or Compress [CGM+23].

In addition to passive SCA attacks, active adversaries employ techniques that deliberately introduce faults into the operation of the device to compromise its security [BDL97]. Such attacks can be carried out through various fault injection methods, i.e. by shortening the period of particular clock cycles (clock-glitching) [ADN+10], by altering the power supply (voltage-glitching) [SGD08], by electromagnetic pulses (as Electromagnetic Fault Injection (EMFI)) [SSAQ02], or by focused laser beams [SA02]. The injected faults can be exploited by a various set of concrete techniques including Differential Fault Analysis (DFA) [BS97], Differential Fault Intensity Analysis (DFIA) [GYTS14], Statistical Fault Attack (SFA) [FJLT13], Fault Sensitivity Analysis (FSA) [LSG+10], Ineffective Fault Attack (IFA) [Cla07], and Statistical Ineffective Fault Attack (SIFA) [DEK+18]. The first generic and comprehensive hardware fault adversary model in [RSG23] abstracts any fault, regardless of its location, type, or timing, by replacing a gate whose output is faulty with another gate realizing the functionality of the faulty gate. The faulty circuit is then compared with a fault-free circuit to determine whether the fault is propagated to the primary output. Incorporating redundancy into the system can be an effective strategy for countering FI attacks. This redundancy allows faulty states to be compared with fault-free states, thereby detecting or correcting faults introduced during the attack. One way of incorporating redundancy is to use binary linear codes, which can be implemented in a variety of ways. Importantly, the fault tolerance of an implementation depends on the specification of the underlying code used.

- Schemes based on Error-Detecting Codes (EDCs) are used to detect a pre-defined number of faults. If a fault is detected, the device discards the faulty output and may eliminate the key if a certain threshold is reached [AMR+20].
- Schemes based on Error-Correcting Codes (ECCs) are used to correct a pre-defined number of faults. Hence, an adversary will always observe fault-free outputs as long as the injected fault is covered by the underlying fault model, defined by the employed code [SRM20].

Since EDC-based schemes do not protect against SIFA [DEK+18], this work focuses on the application of ECC-based schemes to block ciphers, as exemplarily and manually done in [SRM20].

Our Contributions

To the best of our knowledge, there is no AGEMA-equivalent tool for protecting a given design against FI attacks, i.e. there is no tool for automatically generating fault-tolerant circuits. We fill this gap by introducing an extension to the current security-aware

hardware design flow, namely Automated Generation of Fault-Resistant Circuits (AGEFA). In particular, our tool, which is publicly available via GitHub[1], has the following key features.

- AGEFA enables the fully automated translation of unprotected hardware designs, i.e. designs without any countermeasures against FI, into provably fault-tolerant hardware circuits. In particular, the fault-tolerance of AGEFA's outputs, even against SIFA, can be exhaustively verified by cryptographic fault-analysis tools, e.g., VerFI [AWMN20]. For this work, we have chosen to exclusively focus on symmetric ciphers, aiming to ensure a comprehensive verification process. While there are no inherent limitations to applying AGEFA to, e.g., post-quantum cryptography, the potentially extensive outcomes would pose significant challenges for verification. As a result, AGEFA enables even inexperienced engineers without deep knowledge in the field of hardware security to reliably protect any design against FI attacks. Concretely, AGEFA provides automated solutions for the following problems:
  - Searching for an efficient ECC that is not only correct but also tailored precisely to the designer's specific requirements can be a time-consuming task when approached manually. In Section 3.2, we present an automated procedure aimed at expediting the process of finding an efficient ECC tailored to the individual requirements of the user.
  - Implementing a fault-resistant design, which relies on the generated ECC, is a task fraught with a high risk of implementation flaws. Even minor oversights can compromise the security of the entire design. In Section 3.3, we outline a procedure that automates the implementation process, thereby eliminating human errors.
  - Furthermore, AGEFA can incorporate several non-trivial optimizations, typically performed manually. This capability enables the generated designs to achieve an efficiency level comparable to that of manually crafted designs.
- When AGEFA processes a masked design, e.g. an SCA-secure design generated by AGEMA, it preserves all the security guarantees provided by the masking countermeasure that satisfy Probe-Isolating Non-Interference (PINI) requirements, while adding protection against FI attacks. In practice, this means that an engineer can give an unprotected design without any countermeasures to AGEMA first, and receive an SCA-secure design based on PINI gadgets. Further processing of the resulting design with AGEFA will produce a solution that is SCA- and FI-secure.

2 Background

2.1 Notations

As summarized in Table 1, we use lower-case characters to denote atomic elements, e.g. Boolean variables, and sans-serif fonts to denote Boolean functions.Additionally, we use upper-case bold characters, such as $\mathbf{X}$, to represent a set of elements with cardinality $|\mathbf{X}|$. We refer to the individual elements of a set using their index, such that $x_i \in \mathbf{X}$ represents the i-th element of $\mathbf{X}$. Exceptionally, we use upper-case characters to denote multi-bit variables such as $X \in \mathbb{F}_2^n$.

[1] https://github.com/Chair-for-Security-Engineering/AGEMA

Table 1: Notations used in this work.

Notation	Description
$\mathsf{f}: \mathbb{F}_2^n \to \mathbb{F}_2$	n-ary Boolean function
$\mathsf{F}: \mathbb{F}_2^n \to \mathbb{F}_2^m$	n-ary vectorial Boolean function
$\mathsf{f}_i: \mathbb{F}_2^n \to \mathbb{F}_2$	n-ary i-th coordinate function of F
$x \in \mathbb{F}_2$	Boolean variable
$X \in \mathbb{F}_2^n$	Vector of n Boolean variables
$x_i \in \mathbb{F}_2$	i-th Boolean variable of X
$\mathbf{X}$	Set
$\underline{P} \in \mathbb{F}_2^{n \times m}$	$(n \times m)$-matrix
$\underline{I}_n \in \mathbb{F}_2^{n \times n}$	Identity matrix of size n
$\mathcal{M}$	Module

Variable	Description
d	Security order
g	Glitch coverage
t	Transition coverage
c	Coupling coverage
f	Fault cardinality
s	Fault style
l	Fault location
k	Message size
n	Codeword size
δ	Minimum distance

Table 2: Parameters associated with a single wire.

Notation	Description
`attribute`(w)	Returns the attribute associated with w. It holds that `attribute`$(w) \in$ {`clock`, `control`, `layer`, `reset`, `secure`} introduced in Section 3.
`share_domain`(w)	If w carries the share of a sensitive variable, this parameter stores the share domain of w. This is introduced in Section 3.1.
`linear_index`(w)	If w is the input or output of a binary matrix with elements in $\mathbb{F}_2^k$ we enumerate all k-bit input and output chunks and store the number related to w as its linear index. This is introduced in Section 3.1.

2.2 Circuit Model

We formalize the operations carried out by a hardware circuit through a Boolean function $\mathbb{F}_2^i \to \mathbb{F}_2^p$ with i inputs and o outputs. The circuit acquires every input through a signal driven by a physical wire. Subsequently, the corresponding outputs are retrieved from the circuit by reading the signals transmitted through the output wires. Each wire w in a circuit is uniquely defined by its name while we refer to the signal currently transmitted by a wire as its state. Formally, we associate multiple parameters with each wire and define functions to access them as given in Table 2.

Further, review a circuit as a composition of separate building blocks called modules.

Definition 1 (Module)**.** A module $\mathcal{M}$ defines a quadruple $(\mathbf{IN}_\mathcal{M}, \mathbf{T}_\mathcal{M}, \mathbf{OUT}_\mathcal{M}, \mathbf{INST}_\mathcal{M})$ with the following elements:

- $\mathbf{IN}_\mathcal{M}$ (resp. $\mathbf{OUT}_\mathcal{M}$) defines a set of primary input wires (resp. primary output wires) belonging to $\mathcal{M}$.
- $\mathbf{T}_\mathcal{M}$ defines the set of intermediate (internal) wires of $\mathcal{M}$.
- $\mathbf{INST}_\mathcal{M}$ defines a set of instructions representing the functionality of the $\mathcal{M}$. We define an instruction as a Boolean function based on a restricted set of operands $\{\mathsf{not}, \mathsf{and}, \mathsf{nand}, \mathsf{or}, \mathsf{nor}, \mathsf{xor}, \mathsf{xnor}, \mathsf{reg}\}$.

We refer to an atomic module $\mathcal{G}$ as a gate (either combinational or sequential). Every gate evaluates a single output ($|\mathbf{OUT}_\mathcal{G}| = 1$[2]) by applying a single operation from $\{\mathsf{not}, \mathsf{and}, \mathsf{nand}, \mathsf{or}, \mathsf{nor}, \mathsf{xor}, \mathsf{xnor}, \mathsf{reg}\}$ to the signals in $\mathbf{IN}_\mathcal{G}$, i.e. $\mathbf{T}_\mathcal{G} = \emptyset$.

Example 1. Let $\mathcal{M}$ be a module that abstracts multiple gates computing the Boolean function $z = ab \oplus c$. Then, it holds that $\mathbf{IN}_\mathcal{M} = \{a, b, c\}$ and $\mathbf{OUT}_\mathcal{M} = \{z\}$. There

[2]Note that, for this work, we exclude gates with multiple outputs and do not use the inverted output of a register. This restriction is related to the use of AGEMA.

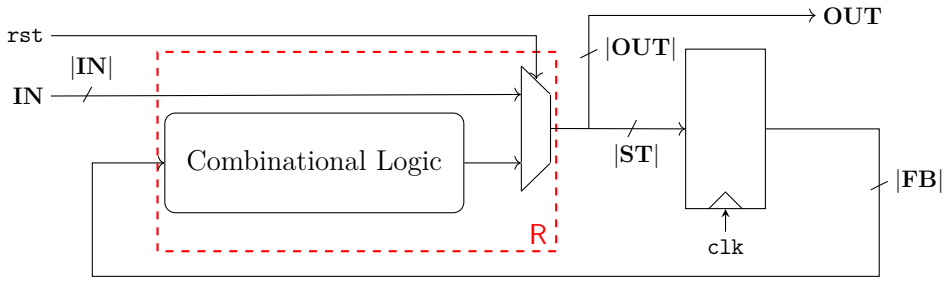


Figure 1: General Mealy model of a synchronous sequential circuit.

are multiple possibilities to construct $\mathbf{INST}_{\mathcal{M}}$ such as $\mathbf{INST}_{\mathcal{M}} = \{z = a \text{ and } b \text{ xor } c\}$ with $\mathbf{T}_{\mathcal{M}} = \emptyset$ or $\mathbf{INST}_{\mathcal{M}} = \{q = a \text{ and } b, z = q \text{ xor } c\}$ with $\mathbf{T}_{\mathcal{M}} = \{q\}$. Further, we can represent $\mathcal{M}$ as a composition of $\mathcal{G}_{\mathsf{and}} = (\{a, b\}, \emptyset, \{z\}, \{z = a \text{ and } b\})$ and $\mathcal{G}_{\mathsf{xor}} = (\{a, b\}, \emptyset, \{z\}, \{z = a \text{ xor } b\})$.

Definition 2 (Circuit)**.** We model a circuit as a directed graph $(\mathbf{G}, \mathbf{W})$ while each vertex $\mathcal{G} \in \{\mathcal{G}_{\mathsf{not}}, \mathcal{G}_{\mathsf{and}}, \mathcal{G}_{\mathsf{nand}}, \mathcal{G}_{\mathsf{or}}, \mathcal{G}_{\mathsf{nor}}, \mathcal{G}_{\mathsf{xor}}, \mathcal{G}_{\mathsf{xnor}}, \mathcal{G}_{\mathsf{reg}}\}$ represents an atomic module, e.g. a sequential or combinational gate, and each edge $w \in \mathbf{W}$ represents a wire.

We focus on sequential circuits, where a clock signal `clk` synchronizes the processed data by storing the state of wires in **ST** in registers. Any synchronous sequential circuit (which has no combinational loops) can be modeled as a Mealy machine [Mea55]. We show the schematic model of a generic sequential circuit in Figure 1.

The model incorporates a single register stage created by merging all synchronization elements, such as registers from an arbitrary number of stages. Initially, an active reset signal `rst` forces the register stage to load the state of the primary inputs (data and control signals) carried by wires in **IN**. Subsequently, the circuit repeatedly processes the state of **ST** by feeding the feedback signal state carried by wires in **FB** back to the combinational logic. The register stage synchronizes the state of all wires in **ST** using a clock signal `clk` while the combinational logic computes the subsequent register inputs based on signals from **FB**. We refer to the combinational logic together with its subsequent multiplexer as the round function R.

2.3 Security Models

The discussion on security models in our work is twofold, encompassing both the SCA and FI adversary models. We remark, that SCA adversary models allow us to prove the secure application of masking [CJRR99] which stands as the predominant concept for protecting a circuit against SCA. Therefore, masking is the only SCA countermeasure considered in this work. According to secret sharing [Sha79], masking involves randomizing every sensitive variable $X \in \mathbb{F}_2^n$ with $d + 1$ uniformly and randomly distributed shares $X^0, \ldots, X^d \in (\mathbb{F}_2^n)^{d+1}$ satisfying $X = \bigoplus_{i=0}^{d} X^i$. In addition, the circuit must undergo transformation into a masked variant, executing every operation on a subset of shares. However, while we briefly revisit SCA models to provide necessary background information, we focus more extensively on the FI adversary models.

2.3.1 Side-Channel Analysis (SCA) Adversary Model

The d-probing adversary model [ISW03] restricts the adversary's capabilities to place a maximum of d probes on the wires of an ideal circuit. Each probe provides access to the intermediate value of a specific wire at a particular point in time, corresponding to a clock cycle.

Definition 3 (d-probing security)**.** A circuit is d-probing secure iff it does not disclose any sensitive information to any d-probing adversary.

Further, the (g, t, c)-robust d-probing model [FGP+18] extends the d-probing model with the coverage of physical defaults, such as glitches (g), transitions (t), and couplings (c). A (g, t, c)-robust d-probing adversary can place up to d extended probes, each capable of recording all intermediate values on a specific wire that could potentially be leaked due to the physical defaults.

Definition 4 ((g, t, c)-robust d-probing security)**.** A circuit is (g, t, c)-robust d-probing secure iff it does not disclose any sensitive information to any (g, t, c)-robust d-probing adversary.

We remark that the most prominent instances of the robust probing model are the $(1, 0, 0)$-robust d-probing model, i.e. the glitch-extended probing model, $(0, 1, 0)$, i.e. the transition-extended probing model, and $(1, 1, 0)$, i.e. the glitch- and transition-extended probing model. Except [CBG+17, CEM18], couplings are not extensively considered particularly from a theoretical point of view due to the necessity to have access to the detailed information about the physical realization of the target device, e.g., place and routing details.

2.3.2 Composability

As designs have continued to increase in complexity, it has become challenging to design and evaluate circuits that are provably robust-probing secure, especially those that require higher-order security. To address this issue, researchers have introduced *composable gadgets* [CGLS21], which are small and provably secure building blocks that can be composed to create circuits of any order that are also provably secure. In practice, tools such as AGEMA replace unprotected cells with their protected counterparts, i.e. gadgets.

Definition 5 (Composability)**.** A robust-probing secure gadget is composable iff its arbitrary composition with other robust-probing secure and composable gadgets also results in a robust-probing secure circuit.

For a set of gadgets to be composable, each gadget must individually satisfy particular security requirements, and all possible combinations of the gadgets must also be in conformity with the underlying security model. As explained in detail below, PINI [CS20] is known as one of such composable security notions.

Definition 6 (Perfect Probe Simulation)**.** Let $\mathbf{P}$ be a set of d (extended) probes on a gadget. $\mathbf{P}$ can be perfectly simulated with a set $\mathbf{S}$ of input shares if a probabilistic polynomial-time simulator can be found that computes a joint probability distribution based on $\mathbf{S}$, which is identical to the distribution of $\mathbf{P}$.

Definition 7 (d-Probe-Isolating Non-Interference (PINI))**.** Let $\mathbf{P}$ be a set of d_0 (extended) probes on a gadget, and $\mathbf{S}$ a set of d_1 (extended) probes on the gadget's primary outputs, while $\mathbf{O}$ contains all output share indices probed by the probes in $\mathbf{S}$. A gadget is d-PINI iff for every $\mathbf{P} \cup \mathbf{S}$ with $d_0 + d_1 \leq d$, there exists a set $\mathbf{I}$ of d_0 share indices such that the wires probed by probes in $\mathbf{P} \cup \mathbf{S}$ can be perfectly simulated from input shares with indices in $\mathbf{I} \cup \mathbf{O}$ [CS20].

2.3.3 Fault Injection (FI) Adversary Model

Based on [RSG23], we consider a formal model for the adversary whose abilities are denoted by $\zeta(f, s, l)$, where $f \in \{1, 2, \ldots, |\mathbf{G}|\}$ denotes the maximum number of faults that can be simultaneously injected into different gates, $s \in \{\tau_{sr}, \tau_s, \tau_r, \tau_{bf}, \tau_{fm}\}$ the fault type

i.e. stuck-at ($\tau_{sr}, \tau_s, \tau_r$), bit-flip ($\tau_{bf}$), or custom ($\tau_{fm}$) faults, and $l \in \{c_i, m, mc_i\}$ the possible fault locations, which are usually a restricted set of gates, defined by $l = c_i \subseteq c_\infty$, and registers, defined by $l = m$. In this work, we focus on a conservative (worst-case) adversary model that allows toggle faults ($s = \tau_{bf}$), including stuck-at faults as well, at arbitrary locations of the circuit, such as gates and registers ($l = mc_\infty$),but excluding primary signals[3]. Each fault is modeled by replacing the faulty cell with a predefined cell that implements another functionality, i.e. for $s = \tau_{bf}$ a faulty gate gets replaced by its negated variant. Hence, for certain given inputs X, the faulty circuit may produce a faulty intermediate state $\tilde{Q}$ instead of Q while for other inputs the fault gets suppressed. Then, the model compares the primary output of the faulty circuit $\tilde{Z}$ with that of an equivalent circuit without any fault Z to determine whether the injected fault was effective (cf. Definition 8) or ineffective (cf. Definition 9).

Definition 8 (Effective fault)**.** We consider an injected fault as effective, w.r.t to a primary input X, iff $\tilde{Z} \neq Z$, i.e. if the injected fault affects the primary output when X is processed.

Definition 9 (Ineffective fault)**.** We consider an injected fault as ineffective, w.r.t to a primary input X, iff $\tilde{Z} = Z$, i.e. if the injected fault does not affect the primary output when X is processed.

As our focus is solely on error correction, we can disregard separate definitions of detected and undetected faults. We consider a circuit to be secure under a fault model $\zeta(f, t, l)$ if all the considered faults will be corrected, i.e. are ineffective.

Definition 10 ((f, s, l)-fault security)**.** A circuit is (f, s, l)-fault secure if all considered faults are ineffective.

We remark that $(f, \tau_{bf}, mc_\infty)$-fault security, which we consider in this work, is equivalent to security under the multivariate adversary model defined in [SRM20] and that security according to Definition 10 also implies security against SIFA.

2.3.4 Fault Propagation and Independence

Whenever an adversary injects faults in f intermediate gates, it is possible that more than f output signals of the circuit can become faulty [AMR+20]. To illustrate, let us assume that the adversary injects a fault in one intermediate gate inside the circuit. The faulty output of the gate may become the input of multiple subsequent gates, which are driven by the faulty gate propagating to multiple primary outputs. This phenomenon is known as fault propagation. To prevent fault propagation, every module must satisfy the independence property [AMR+20], meaning that every intermediate wire in a module should contribute to at most one primary output of the circuit.

Definition 11 (Independence)**.** The implementation of a module $\mathcal{M}$ inside a circuit fulfills the independence property iff $|\mathbf{OUT}_{\mathcal{M}}| = 1$, i.e. if $\mathcal{M}$ computes a single output.

If the independence property is satisfied, every introduced fault can make at most one output signal faulty.

2.4 Countermeasures

2.4.1 Error-Correcting Codes (ECCs)

To achieve (f, s, l)-fault security an error-correction mechanism is required. Usually, error-correction is based on redundancy and binary linear codes.

[3] Faults on primary inputs without any redundancy cannot be detected or corrected.

Definition 12 (Binary linear code)**.** A binary linear code $\mathbf{C}$ is a k-dimensional subset of $\mathbb{F}_2^n$ representing a linear and injective mapping function $\mathsf{C} : \mathbb{F}_2^k \mapsto \mathbb{F}_2^n$, i.e. C applies $\mathbf{C}$ to *encode* a *message* $X \in \mathbb{F}_2^k$ to a *codeword* $Y \in \mathbf{C}$.

As C is linear, we can formalize the mapping by a $(k \times n)$-matrix, refered to as *generator matrix* $\underline{G}$, i.e. it holds that $X \cdot \underline{G} = Y$. Furthermore, we refer to a code as *systematic* iff it embeds the message into the first k bits of the codeword, i.e. $\underline{G}$ is of the form $[\underline{I_k}|\underline{P}]$, with $\underline{I_k}$ being the $k \times k$ identity matrix. In this context, where we are dealing with binary linear systematic codes (denoted as $[n, k]$-codes in the following), we refer to the codeword associated with the message $X \in \mathbb{F}_2^k$ as $Y = \langle X|X' \rangle \in \mathbf{C}$ while X' denotes the *parity* of X. Formally, the parity is computed as $X' = X \cdot \underline{P}$, while $\underline{P}$ again is the matrix representation of a linear and injective mapping $\mathsf{P} : \mathbb{F}_2^k \mapsto \mathbb{F}_2^{n-k}$. To prove the injectivity of a linear function, we refer to Lemma 1.

Lemma 1. *A linear function C is injective iff it holds that $\mathsf{C}(X) = \{0\}^n \implies X = \{0\}^k$.*

When transmitting a codeword $Y \in \mathbf{C}$ over an unreliable communication channel, the receiver can obtain a faulty codeword $\tilde{Y} = \langle X \oplus E|X' \oplus E' \rangle$, where $\langle E|E' \rangle$ is the *error vector* that affected the transmission. The number of faults to correct within a single codeword depends on the code's *minimum distance* δ. Therefore, we denote an $[n, k]$-code satisfying a minimum distance of δ as $[n, k, \delta]$-code.

Definition 13 (Minimum Distance)**.** The minimum distance δ of an $[n, k]$-code $\mathbf{C}$ is defined as $\min_{\forall Y,Z \in \mathbf{C}, Y \neq Z} \mathsf{HD}(Y, Z) = \min_{\forall Y,Z \in \mathbf{C}, Y \neq Z} \mathsf{HW}(Y \oplus Z) = \min_{\forall Y \in \mathbf{C}} \mathsf{HW}(Y)$.

In the above definition, $\mathsf{HD}(Y, Z)$ denotes the Hamming Distance (HD) of two different codewords Y and Z, i.e. the number bits which differ between Y and Z, while $\mathsf{HW}(.)$ denotes the Hamming Weight (HW) of a codeword.

To correct the faults, $\tilde{Y}$ is replaced by its nearest valid codeword, i.e. the codeword with minimal distance to $\tilde{Y}$. Accordingly, $\tilde{Y}$ will only be correctly replaced by Y if the distance between Y and $\tilde{Y}$ is still smaller than the distance between $\tilde{Y}$ and other codewords. This directly leads to Lemma 2. Hence, an $[n, k, \delta]$-code can correct at most $f = \frac{\delta}{2} - 1$ faulty bits in a single codeword.

Lemma 2. *An $[n, k, \delta]$-code can correct all faulty codewords $\tilde{Y} = \langle X \oplus E|X' \oplus E' \rangle$ with $\mathsf{HW}(E) + \mathsf{HW}(E') < \frac{\delta}{2}$.*

The translation from a faulty codeword to its nearest valid codeword can be done by means of *syndrome decoding*. For a valid codeword $Y = \langle X|X' \rangle$ it holds that $X' = X \cdot \underline{P}$ and therefore $X' \oplus X \cdot \underline{P} = 0$. For a faulty codeword $\tilde{Y} = \langle X \oplus E|X' \oplus E' \rangle$ with the error vector $\langle E|E' \rangle$, we denote $(X' \oplus E') \oplus (X \oplus E) \cdot \underline{P} = E' \oplus E \cdot \underline{P}$ as a *syndrome*. Once the syndrome is calculated, the *Syndrome Decoder* (SD) compares it to a pre-calculated lookup table of syndromes and their corresponding error vectors $\mathsf{S} : E' \oplus E \cdot P \mapsto \langle E|E' \rangle$. If the syndrome matches one of the entries in the table, it means that an error has occurred and the decoder can use the associated error vector to correct the error. Based on syndrome decoding, we can correct faulty codewords as shown in Figure 2. We call the place where such a correction is made a *correction point*.

We remark that S is split into two functions namely $\mathsf{S_0} : E' \oplus E \cdot \underline{P} \mapsto E$ and $\mathsf{S_1} : E' \oplus E \cdot \underline{P} \mapsto E'$, meaning that both receive the syndrome. Syndrome decoders $\mathsf{S_0}$ and $\mathsf{S_1}$ predict E and E' correctly as long as the fault can be corrected by the underlying $\mathbf{C}$. Hence, it must hold that $\mathsf{HW}(E) + \mathsf{HW}(E') < \frac{\delta}{2}$ to correct $\tilde{Y}$ with an $[n, k, \delta]$-code $\mathbf{C}$.

2.4.2 Majority Voting

A simple yet effective correction point derived from an $[n, 1, \delta]$-code with $\mathsf{C} : \mathbb{F}_2 \mapsto \mathbb{F}_2^n$ and $\mathsf{C}(x) = \{x\}^n$ can correct up to $f = \frac{\delta}{2} - 1$ faults in a codeword with an odd codeword

size of $n = \delta$ through the application of a technique known as Majority Voting (MV). In essence, MV involves examining each bit within the codeword and identifying the majority value among them. By aligning the bits with the majority decision, the correction point effectively corrects up to f faults in the codeword. Typically, the correction points are designed to receive the redundantly generated results of a system, which often involves obtaining the same output from n independent and parallel instances. For example, consider a scenario where three cipher cores simultaneously process identical inputs. In this setup, the correction points are responsible for evaluating these redundant outputs in order to correct one fault per output bit. Since the independent instances operate without sharing any signals, the occurrence of a fault in one instance does not impact the outputs of the other instances. Additionally, given that each output bit is corrected individually, there is no necessity for the instances to adhere to the independence property which reduces the area overhead associated with MV to a factor of n due to the presence of these n parallel instances plus an additional MV circuit. If the correction is applied exclusively to the system's output, the effectiveness of MV is constrained as it can only correct a total of f faults throughout the entire execution of the system, as opposed to f faults per clock cycle. Consequently, to attain the desired level of fault security denoted as $(f, \tau_{bf}, mc_{\infty})$, the correction logic must be capable of addressing the cumulative sum of faults that an adversary could potentially inject during the system's execution. This necessitates correcting a much larger number of faults, rendering the corresponding overhead arising from redundancy impractical for our needs.

2.4.3 Impeccable Circuits

Aghaie et al. apply the aforementioned $[n, k, \delta]$-codes in Concurrent Error Detection (CED) schemes for hardware circuits [AMR+20]. The presented strategy allows to limit the fault propagation in a way that the circuit guarantees the detection of up to f faults at arbitrary positions when employing an $[n, k, \delta]$-code. Again f refers to the number of simultaneously injected faults as introduced in Section 2.3.3 and applied in Definition 10. As detection of faults is not enough to prevent SIFA, a follow-up work shows a scheme for correcting faults at arbitrary positions by applying an $[n, k, \delta]$-code [SRM20]. To implement the error correction scheme, we commence with a general sequential circuit, as illustrated in Figure 1. To simplify matters, we assume $|\mathbf{IN}| \bmod k = 0 \wedge |\mathbf{FB}| \bmod k = 0$ meaning that the input state of the round function and the feedback state are both multiples of k bits. In accordance with the $[n, k, \delta]$-code, each k-bit message X, carried by a dedicated k-bit segment of **IN** or **FB**, can be encoded into its respective codeword $Y = \langle X|X'\rangle$ by applying the parity mapping function P, as defined in Section 2.4.1. For example, if $|\mathbf{IN}| = q \cdot k$ (or $|\mathbf{FB}| = q \cdot k$), we define $\mathsf{F} : \mathbb{F}_2^{q \cdot k} \mapsto \mathbb{F}_2^{q \cdot (n-k)}$. This function maps a collection of q messages, each composed of k bits, to their corresponding q parities, each comprised of $n - k$ bits. Specifically, F applies P individually to each of the q messages. While the straightforward procedure mentioned above applies to data signals, it is essential to carefully consider control signals, which are typically processed by a finite-state machine and play a critical role in the round function. Changes in the control flow can potentially enable an adversary

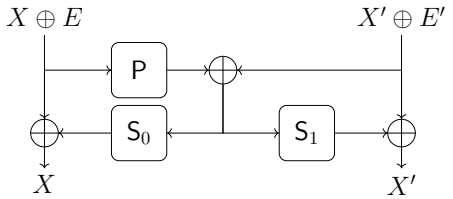


Figure 2: Correction point correcting a faulty codeword $\langle X \oplus E|X' \oplus E'\rangle$ back to $\langle X|X'\rangle$.

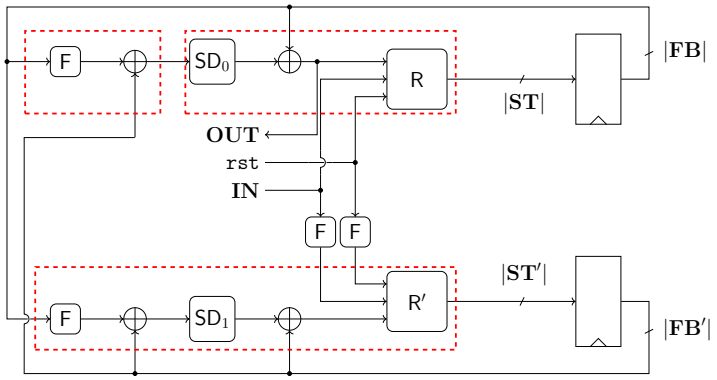


Figure 3: CED based on Impeccable Circuits II with injective F.

to propagate intermediate states to the output. To prevent any alterations in the control flow, every control signal in **IN** must be encoded separately, distinct from the data signals or other control signals [SRM20]. In practical terms, this means that each single-bit control signal is padded with zeros to create an individual k-bit chunk while data signals are only padded if the whole state encompasses no multiple of k bits.

Further, we assume that F is an injective function[4]. Therefore, the redundancy size must be at least as large as the message size, implying that $n \geq 2k$. Additionally, since F is an injective function, the redundant counterpart of the round function R can exclusively operate on the parities of the input signals. This results in a redundant round function denoted as $\mathsf{R}' = \mathsf{F} \circ \mathsf{R} \circ \mathsf{F}^{-1}$, operating on the set of parities belonging to the input signals of the round function. In particular, we denote the set of wires driving the redundancy state of the feedback signals as $\mathbf{FB}'$. To construct a full correction point for **FB** we employ the same principle as for defining F to establish $\mathsf{SD}_0 : \mathbb{F}_2^{q\cdot(n-k)} \mapsto \mathbb{F}_2^{q\cdot k}$ (and $\mathsf{SD}_1 : \mathbb{F}_2^{q\cdot(n-k)} \mapsto \mathbb{F}_2^{q\cdot(n-k)}$). These functions are responsible for decoding the syndromes of q parity chunks. As a result, we can apply q correction points in parallel to correct the input state of the round function. Figure 3 shows the schematic of the circuit from Figure 1 after the application of such a CED.

It is important to note that contrary to MV as explained in Section 2.4.2, the round functions R and R', together with the correction logic, must be implemented in a way that the propagation of a single intermediate fault to multiple faults within **ST** or $\mathbf{ST}'$ is prevented. This necessitates their implementation in a manner that ensures the independence property. In the case of an ECC with $\delta = 2f + 1$, the output of SD_0 remains unchanged even when up to f faults are introduced at its input. It means that SD_0 does not propagate any fault E with $\mathsf{HW}(E) < f$. Therefore, it is feasible to instantiate F and the corresponding XOR operation separately. However, this does not hold true for SD_1 [SRM20]. Consequently, all sub-circuits enclosed by dashed red lines in Figure 3 must adhere to the independence property. Unfortunately, this may lead to the necessity of incorporating multiple instances of the same correction logic, potentially increasing the area requirements. To mitigate the increase in area overhead, the authors of [SRM20] suggested the insertion of multiple correction points while maintaining the circuit's latency, provided that the round function R can be decomposed. Specifically, they suggest to split R into two sub-functions $\mathsf{R} = \mathsf{R}_1 \circ \mathsf{R}_0$ in a way that R_1 becomes linear. Further, if R_1 can be effectively represented as a binary matrix composed of elements from the finite field $\mathbb{F}_2^k$, it has been established in [AMR+20] that R_1 exhibits a property of fault non-propagation. Consequently, when implemented in accordance with the aforementioned decomposition,

[4]Indeed, another scheme based on non-injective F is also presented in [SRM20]. However, according to the results presented in [BBM+22], we excluded such cases in this work.

there is no necessity for an additional correction point before the computation of R_1. However, if R_0 and R_1 are both non-linear, this would necessitate additional circuitry to implement multiple correction points. In this case, two individual correction points are required to correct the inputs of R_0 and R_1 respectively. It is worth noting that the round function can be decomposed into multiple sub-functions, as opposed to having just one linear and one non-linear sub-function. An extreme example of this approach is illustrated in [BKHL20], where a correction point is introduced for each non-linear gate. While this strategy simplifies the achievement of the independence property, it is important to consider that the substantial increase in the number of correction points could potentially result in an impractical area overhead, depending on the specific target. Since we allow an adversary to introduce up to f faults per clock cycle, it is important to note that the number of faults to be corrected between two correction points is effectively doubled, as observed in [SRM20]. To illustrate this, consider an adversary with the capability to inject a single fault per clock cycle. Exemplary, the adversary can fault the input of a specific register in one clock cycle and fault the output of another register in the subsequent clock cycle. In this scenario, the fault introduced at the register's input is propagated to its output, resulting in two simultaneous faults on register outputs. Consequently, the underlying ECC must be designed to correct for a total of $2f$ faults to achieve $(f, \tau_{bf}, mc_{\infty})$-fault security.

3 Technique

In this section, we present AGEFA's procedure (cf. Figure 4) for converting an unprotected implementation into a fault-resistant design, utilizing the technique outlined in Impeccable Circuits II [SRM20]. We have chosen to adopt the scheme due to its assurance of security within our designated fault adversary model, a feat that, practically, cannot be achieved through the application of, for instance, cipher-level MV. Additionally, Impeccable Circuits II provides a high degree of flexibility in its implementation, primarily through the ability to decompose the round function as needed. This flexibility permits us to optimize protection in two vital dimensions: latency and area. On one hand, we can opt for no decomposition to minimize the latency, while on the other hand, we have the option to decompose the round function into multiple sub-functions to reduce the overall area footprint at the cost of higher latency. Moreover, this flexibility allows us to experiment with various configurations, enabling us to identify the most suitable one. This advantage is a direct result of automation and would be unattainable through manual methods.

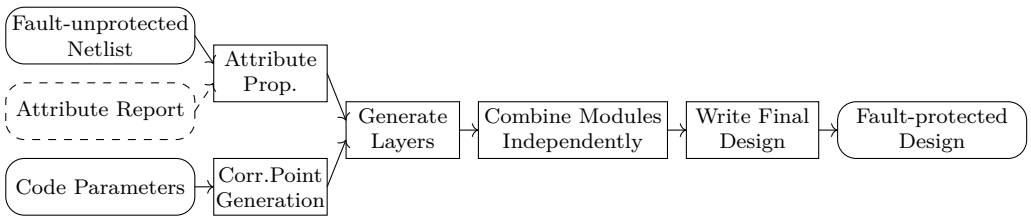


Figure 4: Procedure of AGEFA to protect a circuit against FI attacks.

AGEFA follows the process outlined in Figure 4. The process begins with receiving a gate-level netlist written in Verilog. The given netlist implements no protection against FI attacks but can be a masked implementation to be protected against SCA. To prepare the netlist to be processed by AGEFA, the designer must first synthesize the behavioral-level description of the design using a synthesizer such as Design Compiler (DC) [Inc] or Yosys [Wol]. Additionally, for every primary input or output wire w, the designer has to set `attribute`(w) through textual annotations in the netlist in the following manner:

- If wire w should be identified as the clock signal, it must hold that $\texttt{attribute}(w) = \texttt{clock}$. In this work, we denote the clock signal as `clk`.
- If wire w should be identified as the reset signal, it must hold that $\texttt{attribute}(w) = \texttt{reset}$. In this work, we denote the reset signal as `rst`.
- For every primary input or output wire w which carries a data signal (including plaintext, ciphertext, and key), it must hold that $\texttt{attribute}(w) = \texttt{secure}$. Likewise, if w carries a control signal, e.g. enable or done signals, it must hold that $\texttt{attribute}(w) = \texttt{control}$. Further, for every primary input wire (resp. output wire) w, it holds that $w \in \mathbf{IN}$ (resp. $w \in \mathbf{OUT}$) iff $\texttt{attribute}(w) = \texttt{secure} \vee \texttt{attribute}(w) = \texttt{control}$.
- If we assume that $\mathsf{R} = \mathsf{R}_{h-1} \circ \ldots \mathsf{R}_0$, i.e. if the round function is a decomposition of h sub-functions, and we abstract every sub-function with a module $\mathcal{R}_{h-1}, \ldots, \mathcal{R}_0$, it holds that $\texttt{attribute}(w) = \texttt{layer}$ for every intermediate wire w with $w \in \mathbf{IN}_{\mathcal{R}_i} \wedge w \in \mathbf{OUT}_{\mathcal{R}_{i-1}} \wedge 0 < i \leq h$.

Once the annotated netlist is received, AGEFA applies a slightly modified version of the AGEMA's parser, resulting in a graph $(\mathbf{G}, \mathbf{W})$ that represents the netlist based on the circuit model described in Section 2.2. We straightforwardly render R into a separate set of modules $\mathbf{R}$. Again, $\mathcal{R} \in \mathbf{R}$ abstracts one coordinate function of R. If R is decomposed, we create additional coordinate functions for every intermediate wire x with $\texttt{attribute}(x) = \texttt{layer}$. More concretely, if $\texttt{attribute}(x) = \texttt{layer}$, we add a new module $\mathcal{X}$ to $\mathbf{R}$ while $\mathbf{OUT}_{\mathcal{X}} = \{x\}$ and $\mathbf{IN}_{\mathcal{X}}$ can contain primary input wires or other intermediate wires w with $\texttt{attribute}(w) = \texttt{layer}$. Further, we render the register stage into another set of modules $\mathbf{FF}$ while $\mathcal{FF} \in \mathbf{FF}$ abstracts a single register. Based on this representation of the circuit, AGEFA performs the following high-level steps.

- To generate a secure and efficient design, AGEFA sets all parameters mentioned in Table 2 for every signal of the circuit.
- Based on the given message size k and fault cardinality $2f$, AGEFA automatically finds an efficient ECC including all operations required to construct a correction point. For every operation required for the correction, AGEFA generates a module that satisfies the independence property.
- AGEFA transforms all coordinate functions of the (decomposed) combinational logic in a way that they fulfill the independence property.
- Subsequently, it proceeds to construct the three functions, as indicated by the dashed red lines in Figure 3. In this process, AGEFA connects the generated functions, each of which independently satisfies the independence property. Importantly, the compositions are crucially designed to uphold the independence property.
- Lastly, AGEFA finalizes the protected design by connecting the cascaded functions.

3.1 Attribute Propagation

We repeat that the protection mechanism presented in Impeccable Circuits II [SRM20] takes special care of the Finite State Machine (FSM) as injecting faults on the FSM may change the control flow, i.e. enables an adversary to observe intermediate states of the design. Therefore, AGEFA must distinguish between the protection of control signals and data signals. While all $w \in \mathbf{IN}$ are already annotated, AGEFA automatically determines all $\texttt{attribute}(x)$ for every $x \in \mathbf{ST}$ and for every $x \in \mathbf{FB}$ (see Figure 1). To propagate the attributes of the primary inputs through the circuit, we can apply Algorithm 1 of [KMMS22] on $\mathbf{R}$ and $\mathbf{FF}$. Hence, the following two rules apply.

- For every $\mathcal{R} \in \mathbf{R}$, it holds for the output wire $w \in \mathbf{OUT}_{\mathcal{R}}$ that $\texttt{attribute}(w) = \texttt{control}$ if $\nexists x \in \mathbf{IN}_{\mathcal{R}} : \texttt{attribute}(x) = \texttt{secure}$. Conversely, $\texttt{attribute}(w) = \texttt{secure}$ if $\exists x \in \mathbf{IN}_{\mathcal{R}} : \texttt{attribute}(x) = \texttt{secure}$.
- For every $\mathcal{FF} \in \mathbf{FF}$, it holds for the output wire $w \in \mathbf{OUT}_{\mathcal{FF}}$ that $\texttt{attribute}(w) = \texttt{attribute}(x)$ if $x \in \mathbf{IN}_{\mathcal{FF}}$ denotes the input wire with $\texttt{attribute}(x) \neq \texttt{clock}$.

If AGEFA processes a masked design operating on multiple shares, we make sure that the error-correction logic does not violate the SCA-security assumptions of the original design. Specifically, if a masked design adheres to probing security (resp. composability) under the robust probing model, the final masked and fault-tolerant design must similarly guarantee probing security (resp. composability) under the same model. Implementing the error-correction logic SCA-securely can be done share-wise, allowing each share domain to be processed independently, thereby satisfying the PINI-notion. However, it is crucial to ensure that the error-correction logic does not mistakenly combine shares from different domains. To avoid this, each message (for encoding) must only contain shares from the same share domain, which can be achieved by labeling every $w \in \mathbf{ST}$ with a corresponding share domain, i.e. $\texttt{share\_domain}(w)$. The designer has to provide an additional report assigning all $w \in \mathbf{ST}$ that carry shared variables to their corresponding share domain $\texttt{share\_domain}(w)$[5]. If a handcrafted masked implementation is given to AGEFA, the designer must manually generate and provide the wires with their respective share domain, i.e. in the synthesis script or the behavioral design, which might be a challenging task. On the other hand, if the design is entirely made by secure and composable gadgets, such as a result of AGEMA, we can annotate the required wires in each gadget separately and propagate the annotation during the synthesis procedure[6]. However, if $\texttt{share\_domain}(w)$ for a $w \in \mathbf{ST}$ is not specified in the report, e.g. in case of $\texttt{attribute}(w) = \texttt{control}$, we set $\texttt{share\_domain}(w) = 0$ while it holds that $\texttt{share\_domain}(w) > 0$ for all w carrying a shared variable. This annotation ensures that subsequent stages, such as Algorithm 5, encode signals exclusively from the same share domain within a given message. We acknowledge that annotating a masked circuit in this manner can be a tedious task. Nonetheless, if we were to limit the annotation to only primary inputs of the circuit, we would need to exercise greater caution when consolidating signals into the same share domain. This cautious approach would, in turn, lead to a further increase in the area overhead of the final result.

3.1.1 Optimizations

If the round function R can be expressed as the composition of two sub-functions, $\mathsf{R} = \mathsf{R}_1 \circ \mathsf{R}_0$, and R_1 can be represented as a binary matrix with elements in $\mathbb{F}_2^k$, there is no need to correct the inputs of R_1 [AMR+20, SRM20]. Hence, avoiding an additional correction point leads to a more efficient design w.r.t circuit size and latency. Therefore, AGEFA searches in the module-based representation of R, denoted as $\mathbf{R}$ for a subset of modules $\mathbf{R}_1 \subset \mathbf{R}$ where the instructions can be represented as the aforementioned matrix multiplication. This procedure is twofold. First, AGEFA extracts all $\mathcal{R} \in \mathbf{R}$ representing linear functions. Then, AGEFA considers all linear $\mathcal{R} \in \mathbf{R}$ as a linear layer and checks if the correction at inputs can be safely removed.

To identify every $\mathcal{R} \in \mathbf{R}$ representing a linear function, we create its Algebraic Normal Forms (ANFs) based on $\mathbf{INST}_{\mathcal{R}}$ and examine whether it exclusively comprises

[5] The `report_attribute` command ensures that Synopsys DC creates such a report.

[6] Technically, Yosys automatically propagates all user-defined attributes in the RTL code to the netlist, unless the `-noattr` flag is set. For Synopsys DC, the designer must define an attribute for each gadget type through `define_user_attribute` and propagate it via `propagate_user_attributes` command. This procedure results in a separate attribute report that contains all nets with their corresponding attribute, which then can be given to AGEFA.

Algorithm 1 Mark inputs of linear functions

Input: $\mathbf{R}$ ▷ The module-based abstractions of the round function.
Input: $\mathbf{FF}$ ▷ The module-based abstraction of the register stage.
Input: $\mathbf{IN}$ ▷ The set of wires carrying primary input signals
Output: $\mathbf{R}$ ▷ The round function with marked input and output wires.

1: $\mathbf{R}_0 \leftarrow \{\mathcal{R} \in \mathbf{R} | \exists w \in \mathbf{OUT}_{\mathcal{R}} : \texttt{linear\_index}(w) = 0\}$
2: $\mathbf{R}_1 \leftarrow \mathbf{R} \backslash \mathbf{R}_0$
3: $\mathbf{X} \leftarrow \emptyset$
4: **for** $\forall \mathcal{R} \in \mathbf{R}_0$ **do**
5: $\quad \mathbf{X} \leftarrow \mathbf{X} \cup \mathbf{OUT}_{\mathcal{R}_0}$
6: **end for**
7: $\mathbf{Y} \leftarrow \mathbf{X}$
8: **for** $\forall x \in \mathbf{X}$ **do** ▷ Check if the non-linear output is an input of only linear functions.
9: $\quad \mathbf{Z} \leftarrow \{\mathbf{OUT}_{\mathcal{FF}} | \mathcal{FF} \in \mathbf{FF} : x \in \mathbf{IN}_{\mathcal{FF}}\}$ ▷ Get the register output wire.
10: $\quad$ **if** $\nexists z \in \mathbf{IN}_{\mathcal{R}} | \mathcal{R} \in \mathbf{R}_0 : z \in \mathbf{Z}$ **then**
11: $\quad\quad$ **for** $\forall z \in \mathbf{Z}$ **do** ▷ Mark register outputs that are no inputs of linear functions.
12: $\quad\quad\quad \texttt{linear\_index}(z) \leftarrow 1$
13: $\quad\quad$ **end for**
14: $\quad$ **end if**
15: $\quad$ **for** $\forall \mathcal{R} \in \mathbf{R}_1$ **do** ▷ Consider linear layer outputs as inputs of another linear layer.
16: $\quad\quad$ **if** $\nexists w \in \mathbf{IN}_{\mathcal{R}} : \texttt{linear\_index}(w) = 0 \wedge w \notin \mathbf{IN} \wedge w \notin \mathbf{Y}$ **then**
17: $\quad\quad\quad \mathbf{X} \leftarrow \mathbf{X} \cup \mathbf{OUT}_{\mathcal{R}}$
18: $\quad\quad\quad \mathbf{Y} \leftarrow \mathbf{Y} \cup \mathbf{OUT}_{\mathcal{R}}$
19: $\quad\quad$ **end if**
20: $\quad$ **end for**
21: $\quad \mathbf{X} \leftarrow \mathbf{X} \backslash \{x\}$
22: **end for**

monomials with an algebraic degree of one. If $\mathcal{R}$ is confirmed to be abstract a linear function, we mark $w \in \mathbf{OUT}_{\mathcal{R}}$ by setting $\texttt{linear\_index}(w) = 1$. Otherwise, we set $\texttt{linear\_index}(w) = 0$. Additionally, it has been demonstrated in [AMR+20] that the outputs of a multiplexer stage controlled by `rst` can be directly integrated into the following linear layer without an additional correction. Thus, we set $\texttt{linear\_index}(w) = 1$ not only when $\mathcal{R}$ abstracts a linear function but also in cases where $\mathcal{R}$ represents a function with non-linear components limited to multiplexers using `rst` as the select signal. Hence, it holds that $\texttt{linear\_index}(w) = 1$ if the ANF of $\mathcal{R}$ exclusively contains monomials with an algebraic degree of one or monomials with an algebraic degree of two with `rst` as one of the variables. This approach enables the identification of such multiplexers, even if they are not explicitly represented by dedicated multiplexer modules in the netlist but are expressed in their underlying algebraic form. After this step, the output wire w of every $\mathcal{R} \in \mathbf{R}$ is either marked as linear ($\texttt{linear\_index}(w) = 1$) or non-linear ($\texttt{linear\_index}(w) = 0$) and we denote the list of linear modules as $\mathbf{R}_1 = \{\mathcal{R} \in \mathbf{R} | \exists w \in \mathbf{OUT}_{\mathcal{R}} : \texttt{linear\_index}(w) = 1\}$. Similarly, we denote the non-linear modules as $\mathbf{R}_0 = \mathbf{R} \backslash \mathbf{R}_1$. To identify if $\mathbf{R}_1$ abstracts one or multiple layer(s) which do not require additional correction, we apply Algorithm 1. Initially, in Lines 3-7, Algorithm 1 creates two sets, $\mathbf{X}$ and $\mathbf{Y}$, encompassing all output wires of $\mathbf{R}_0$. In the following Lines 9-14, Algorithm 1 systematically examines each output wire $x \in \mathbf{X}$ produced by a non-linear function, determining whether its signal is exclusively propagated (though a register, as checked in Line 9) to the inputs of linear coordinate functions. This criterion is satisfied iff the signal carried by x is not propagated to the input of any $\mathcal{R} \in \mathbf{R}_0$. To identify such wires, Algorithm 1 marks every input wire of $\mathbf{R}_1$, denoted as z in Line 12, meeting this condition by setting $\texttt{linear\_index}(z) = 1$.

Algorithm 2 Remove the correction point

Input: $\mathbf{L}$ ▷ The sorted linear layer with marked linear outputs
Input: $\mathbf{IN_L}$, $\mathbf{OUT_L}$ ▷ A sorted list of input and output wires of $\mathbf{L}$
Input: k ▷ The message size
Output: $\mathbf{L}$ ▷ The linear layer with updated linear outputs and inputs

1: $q \leftarrow 0$
2: **for** $\forall w \in \mathbf{IN_L}$ **do**
3: $\texttt{linear\_index}(w) \leftarrow \lfloor \frac{q}{k} \rfloor + 1$
4: $q \leftarrow q + 1$
5: **end for**
6: **for** $\forall w \in \mathbf{OUT_L}$ **do**
7: $\texttt{linear\_index}(w) \leftarrow \lfloor \frac{q}{k} \rfloor + 1$
8: $q \leftarrow q + 1$
9: **end for**
10: **for** $\forall q \in \{0, k, 2k, \ldots, |\mathbf{OUT_L}| - k\}$ **do**
11: $\mathbf{M} \leftarrow \{\mathcal{L}_q, \ldots, \mathcal{L}_{q+k-1}\} \subset \mathbf{L}$
12: **for** $\forall (\mathcal{M}, \mathcal{M}^*) \in \mathbf{M} \times \mathbf{M} : \mathcal{M} \neq \mathcal{M}^*$ **do**
13: **if** $\exists w \in \mathbf{IN}_{\mathcal{M}} \cap \mathbf{IN}_{\mathcal{M}^*} : \texttt{attribute}(w) \neq \texttt{reset}$ **then**
14: **for** $\forall w \in \mathbf{IN_L} \cup \mathbf{OUT_L}$ **do**
15: $\texttt{linear\_index}(w) \leftarrow 0$
16: **end for**
17: **end if**
18: **if** $|\mathbf{IN}_{\mathcal{M}}| \neq |\mathbf{IN}_{\mathcal{M}^*}| \vee (\exists w \in \mathbf{IN}_{\mathcal{M}} \wedge \nexists q \in \mathbf{IN}_{\mathcal{M}^*} : \texttt{linear\_index}(w) = \texttt{linear\_index}(q))$ **then**
19: **for** $\forall w \in \mathbf{IN_L} \cup \mathbf{OUT_L}$ **do**
20: $\texttt{linear\_index}(w) \leftarrow 0$
21: **end for**
22: **end if**
23: **end for**
24: **end for**

Additionally, in Lines 15-20 the algorithm considers all output wires of linear functions and assesses whether their signals are exclusively propagated to the inputs of other linear coordinate functions. This step enables the detection of multiple cascaded linear layers. Ultimately, all elements $\mathcal{R} \in \mathbf{R}$ possessing solely linear inputs and outputs are identified as constituting a linear layer. Formally, we denote the linear layer as $\mathbf{L} = \{\mathcal{R} \in \mathbf{R} | \nexists w \in \mathbf{OUT}_{\mathcal{R}} \cup \mathbf{IN}_{\mathcal{R}} : \texttt{linear\_index}(w) = 0\}$ with a joint set of primary input wires $\mathbf{IN_L}$ and primary output wires $\mathbf{OUT_L}$. We remark, that both sets $\mathbf{IN_L}$ and $\mathbf{OUT_L}$ must be sorted based on the signal names. Utilizing the information from $\mathbf{L}$, $\mathbf{IN_L}$, and $\mathbf{OUT_L}$, the procedure presented in Algorithm 2 determines whether the correction point can be removed from all wires in $\mathbf{IN_L}$. Specifically, it examines if the implementation of $\mathbf{L}$ can be represented as a binary matrix with elements in $\mathbb{F}_2^k$. A crucial condition for such a representation is that both $|\mathbf{IN_L}|$ and $|\mathbf{OUT_L}|$ are multiples of k, except `rst`, which is encoded into a separate message. Consequently, `rst` within $|\mathbf{IN_L}|$ is counted as k signals. If $\mathbf{L}$ satisfies this condition, indicating that $|\mathbf{IN_L}|$ and $|\mathbf{OUT_L}|$ are multiples of k, Algorithm 2 is executed. Otherwise, we set $\texttt{linear\_index}(w) = 0$ for all $w \in \mathbf{IN_L} \cup \mathbf{OUT_L}$.

In Lines 2-9 of Algorithm 2, each k-bit message derived from the sorted primary wires of $\mathbf{L}$ is assigned a distinct linear index greater than 0. This annotation signifies that no correction is necessary. However, Algorithm 2 is responsible for verifying whether the conditions for removing the correction are indeed met. If not, Algorithm 2 resets all linear

Algorithm 3 Generation of a binary, linear, systematic, and injective $[n, k, \delta]$-code.

Input: k, δ ▷ Message size k and minimum distance δ
Output: $\mathbf{C}$ ▷ A binary, linear, systematic, and injective $[n, k, d]$-code

1: $\mathbf{C} \leftarrow \emptyset$
2: **for** $X = 0$ **to** $2^k - 1$ **do** ▷ Iterate through all possible messages
3: $X' \leftarrow 0$ ▷ X' stores the parity of X
4: $Y \leftarrow \langle X | X' \rangle$
5: **while** $\exists Z : \langle T | T' \rangle \in \mathbf{C}$ s.t. $\mathsf{HD}(Y, Z) < \delta \ \vee \ X' = T'$ **do** ▷ Check codeword Y
6: $X' \leftarrow X' + 1$
7: $Y \leftarrow \langle X | X' \rangle$
8: **end while**
9: $\mathbf{C} \leftarrow \mathbf{C} \cup \{Y\}$ ▷ Add the new codeword to the code
10: **end for**

indices to 0 (cf. Lines 15 and 20). This reset implies that correction is required for all signals. To assess this, AGEFA iterates through all chunks consisting of k modules and stores both their inputs and linear indices. Two conditions must be satisfied to warrant the removal of the correction point.

1. The same input signal (except the signal carried by `rst`) must not be distributed across multiple coordinate functions forming a k-bit message. This condition is examined in Line 13.

2. Each coordinate function forming a k-bit message should receive the same number of inputs with identical linear indices. This condition is examined in Line 18.

It is important to note that the question whether the correction logic can be removed or not depends on the order of the wires in $\mathbf{IN_L}$ and $\mathbf{OUT_L}$ We specifically examine only the case where $\mathbf{IN_L}$ and $\mathbf{OUT_L}$ are sorted based on the wire names while also unsorted $\mathbf{IN_L}$ and $\mathbf{OUT_L}$ can lead to a removed correction point. However, validating all these diverse orderings is computationally infeasible. Instead, we choose this sorted representation based on the belief that it is likely to be implemented, given that it results in a k-bit message for k subsequent bits of a larger state. To illustrate, in the context of a cipher with a 64-bit round state $\{x_0, \ldots, x_{63}\}$, and $k = 4$, we assume that a matrix representation in $\mathbb{F}_2^4$ is implemented by considering sets such as $\langle x_0, x_1, x_2, x_3 \rangle, \ldots, \langle x_{60}, x_{61}, x_{62}, x_{63} \rangle$ as 4-bit messages. In other words, we assume that a designer would interpret the 64-bit state as 16 subsequent 4-bit words. However, if the designer chooses another representation, AGEFA cannot remove the correction point resulting in a design that is still secure but not as efficient as possible.

3.2 Correction Point Generation

The designer specifies the code parameters, i.e. the message size k and the maximum number of faults to correct within one clock cycle, usually set to $2f$ to achieve $(f, \tau_{bf}, mc_\infty)$-fault security. Utilizing these parameters, AGEFA estimates the appropriate ECC parameters $[n, k, \delta]$. As outlined in Lemma 2 and Section 2.4.3, it holds that $n \geq 2k$. Further, the underlying ECC must correct $2f$ faults, hence $\delta = 4f + 1$.

3.2.1 Error-Correcting Code (ECC) Generation

The procedure for finding a binary, linear, systematic, and injective $[n, k, \delta]$-code, denoted as $\mathbf{C}$ which is a vector subspace of $\mathbb{F}_2^n$ is outlined in Algorithm 3. It processes every message X by assigning a parity X' and generating the codeword $Y = \langle X | X' \rangle$. In Line 5,

it is checked if Y can be added to $\mathbf{C}$ without violating the code's minimum distance and injectivity, i.e. by checking if the minimum distance of Y and all other codewords of $\mathbf{C}$ is sufficient and by ensuring that no other codeword in $\mathbf{C}$ shares X' as parity. As long as Y is not a suited codeword for $\mathbf{C}$, we increment its parity X' until the codeword satisfies all the requirements. Algorithm 3 repeats this procedure until all $X \in \mathbb{F}_2^k$ are associated with a corresponding X' in $\mathbf{C}$. The presented procedure follows a Greedy approach for constructing a, so-called, lexicographic code [CS86, Con90]. We remark that (1) such a code exists for all possible k and d [BP93], (2) the resulting codes are provably linear, systematic, and injective [Lev60, CS86, BP93], and (3) n is usually minimal [CS86]. The injectivity and systematicity of the code generated by Algorithm 3 are easily provable due to the algorithm's nature. However, the linearity of the code is not immediately apparent. A lexicographic code, where the codewords are arranged and iterated in lexicographic order, was first proven to be linear by Levenshtein [Lev60]. Brualdi et al. later generalized this proof to lexicographic codes generated using arbitrarily ordered bases of $\mathbb{F}_2^n$, resulting in a lexicographic ordering on the coefficient vectors [BP93]. As Algorithm 3 iterates through the codewords in lexicographic order, the generated codes are also linear by extension of the aforementioned proofs. In summary, Algorithm 3 leads to codes with small parities and thus an efficient encoding. However, the designer is free to force AGEFA to employ a certain ECC that better fits particular needs. From $\mathbf{C}$, we derive the mapping $\mathsf{P} : \mathbb{F}_2^k \to \mathbb{F}_2^{n-k}$ between messages $X \in \mathbb{F}_2^k$ and their corresponding parities $X' \in \mathbb{F}_2^{n-k}$ in form of a lookup table. We compute the table lookup for an arbitrary message X as $X \cdot \underline{P} = X'$. Further, we generate the mapping $\mathsf{P}^{-1}(.)$, again as a lookup table to map arbitrary parities back to their corresponding messages.

3.2.2 Syndrome Decoder (SD) Generation

Algorithm 4 constructs the corresponding SD of the previously generated $[n, k, \delta]$-code $\mathbf{C}$. As shown in Figure 2, we split the SD into two separate functions, namely $\mathsf{S}_0 : E' \oplus E \cdot \underline{P} \mapsto E$ and $\mathsf{S}_1 : E' \oplus E \cdot \underline{P} \mapsto E'$. As previously mentioned, we generate the mappings S_0 and S_1 as lookup tables. Algorithm 4 generates mappings for all possible error vectors that $\mathbf{C}$ can correct, specifically for error vectors with less than $\frac{\delta-1}{2}$ faulty bits. In Line 3, the syndrome is computed based on the error vector. The syndrome is subsequently mapped to the corresponding parts of the error vector and stored as a mapping (lookup table) in S_0 and S_1. These lookup tables are not complete, meaning that S_0 and S_1 do not cover all $\langle E|E'\rangle \in \mathbb{F}_2^n$. We deal with such cases in the following section.

3.2.3 Algebraic Representation

To integrate P, P^{-1}, S_0, and S_1 into a hardware design and to facilitate further steps, we convert the lookup tables into a set of modules in accordance with Definition 1. We repeat that a module can abstract complex operations consisting of multiple gates and inputs. For every module $\mathcal{M}$ with $|\mathbf{IN}_\mathcal{M}| = n$ inputs and $|\mathbf{OUT}_\mathcal{M}| = m$ outputs, it holds that

Algorithm 4 Generation of the Syndrome Decoder (SD).

Input: P, n, k, δ ▷ Resulting ECC parameters from Algorithm 3
Output: $\mathsf{S}_0, \mathsf{S}_1$ ▷ The corresponding SD with two mappings
1: $\mathsf{S}_0 \leftarrow \emptyset$, $\mathsf{S}_1 \leftarrow \emptyset$
2: **for** $\forall\langle E|E'\rangle \in \mathbb{F}_2^n$ s.t. $\mathsf{HW}(\langle E|E'\rangle) < \frac{\delta-1}{2}$ **do** ▷ Generate error vectors to correct
3: $T \leftarrow \mathsf{P}(E) \oplus E'$ ▷ Compute the syndrome T
4: $\mathsf{S}_0 \leftarrow \mathsf{S}_0 \cup (T, E)$ ▷ Add $\mathsf{S}_0(T) = E$ to the mapping
5: $\mathsf{S}_1 \leftarrow \mathsf{S}_1 \cup (T, E')$ ▷ Add $\mathsf{S}_1(T) = E'$ to the mapping
6: **end for**

it can be formalized by an n-ary vectorial Boolean function with m coordinate functions. However, we can also decompose the Boolean functions in a way that multiple vectorial Boolean functions compute the module's intermediates in $\mathbf{T}_{\mathcal{M}}$ while other functions process the intermediates to compute the primary outputs $\mathbf{OUT}$. We denote the resulting sets of modules as $\mathbf{P}$, $\mathbf{P}^{-1}$, $\mathbf{S}_0$, and $\mathbf{S}_1$ while every module in the set abstracts one coordinate function of the respective operation, e.g. $\mathcal{P}_i$ denotes the i-th module in $\mathbf{P}$ abstracting the i-th coordinate function of P. Further, it holds that none of the modules stores intermediates, meaning that every module implements a Boolean function processing all input signals to compute a single output.

Since all functions are injective, their corresponding lookup tables may contain *don't care* values. For example, S_0 and S_1 certainly have such cases as explained above. To find the most efficient logic function that represents the lookup table even if it contains *don't cares*, we apply the Quine-McCluskey algorithm [Qui52, McC56] on every coordinate function of the respective mappings. Hence, every mapping is translated from a lookup table into its minimal sum-of-products form becoming efficient in terms of circuit size. Finally, we store the generated Boolean function, i.e. the minimal sum-of-products form, using a single instruction, in accordance with Definition 1, into the corresponding module.

3.2.4 Optimizations

Depending on the complexity of the combinational logic, choosing $\mathbf{C}$ with minimum n, as shown in Section 3.2.1 may not always be the optimal approach. As an example, consider an arbitrary linear, injective, and systematic $[8, 4, 3]$-code $\mathbf{C}$ which is capable of correcting one fault in an 8-bit codeword with 4-bit message. It can be shown that it is possible to implement the mapping $\mathsf{P} : \mathbb{F}_2^4 \to \mathbb{F}_2^4$, where a message $X : \langle x_3, x_2, x_1, x_0 \rangle$ is mapped to its corresponding parity $X' : \langle x'_3, x'_2, x'_1, x'_0 \rangle$, using four separate coordinate functions $\mathsf{p}_0, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 : \mathbb{F}_2^4 \to \mathbb{F}_2$ as follows.

$$x'_3 = \mathsf{p}_3(x_3), \qquad x'_2 = \mathsf{p}_2(x_2, x_1), \qquad x'_1 = \mathsf{p}_1(x_2, x_0), \qquad x'_0 = \mathsf{p}_0(x_3, x_2, x_1, x_0)$$

According to Figure 3 and the explanations given in Section 2.4.3, the redundant part of the round function, i.e. R', is derived as $\mathsf{F} \circ \mathsf{R} \circ \mathsf{F}^{-1}$. We remark that each coordinate function of R' should satisfy the independence property. Therefore, the output of sub-functions of $\mathsf{R} \circ \mathsf{F}^{-1}$ are the inputs of F, i.e. X in the above equations. Due to the independence property, coordinate functions p_0 to p_3 cannot share any inputs. As an example, the circuit which computes x_2 must be instantiated three times as p_0, p_1, and p_2 receive x_2. In other words, each input of the coordinate functions p_0 to p_3 should be individually generated. This means that these coordinate functions need in total 9 individual inputs. As a side note, there is no other $[8, 4, 3]$-code with a smaller number of individual inputs.

If we, artificially, increase the parity size by one bit, we achieve a linear, injective, and systematic $[9, 4, 3]$-code with message-to-parity mapping $\mathsf{Q} : \mathbb{F}_2^4 \to \mathbb{F}_2^5$ which maps a message $X : \langle x_3, x_2, x_1, x_0 \rangle$ to its corresponding parity $X' : \langle x'_4, x'_3, x'_2, x'_1, x'_0 \rangle$ with the following five coordinate functions $\mathsf{q}_0, \mathsf{q}_1, \mathsf{q}_2, \mathsf{q}_3, \mathsf{q}_4 : \mathbb{F}_2^4 \to \mathbb{F}_2$.

$$x'_4 = \mathsf{q}_4(x_3), \quad x'_3 = \mathsf{q}_3(x_2), \quad x'_2 = \mathsf{q}_2(x_1), \quad x'_1 = \mathsf{q}_1(x_0), \quad x'_0 = \mathsf{q}_0(x_3, x_2, x_1, x_0)$$

Despite the larger parity size, the coordinate functions of Q need in total only 8 individual inputs. We remark that the number of individual inputs of P (resp. Q) has a decisive influence on the complexity of the design, i.e. circuit size, as R' needs to employ these message-to-parity mappings. Consequently, a mapping with a minimal number of inputs per coordinate function can avoid multiple instances of the same combinational sub-circuit, potentially saving more area than that of the additional parity. The specific size of the circuit depends on the round function and the syndrome decoders, which generally become more complex as the parity size increases. Therefore, the designer has to balance

between fewer instances of individual coordinate functions belonging to the round function and an increasingly complex SD.

To determine the optimal parity size, AGEFA automatically finds the parity size resulting in the smallest number of individual inputs for the message-to-parity mapping. This is done by incrementally increasing the parity size as long as the number of individual inputs decreases and continuing with the code that results in the smallest number of individual inputs. However, it is recommended to apply AGEFA on every design twice, once with these optimizations and once without, to ensure that the optimizations truly result in a decrease in the circuit size. If optimizing the parity size does not lead to a smaller circuit, it can be concluded that the SD becomes too complex and that the smallest possible parity size, i.e. AGEFA without optimizations, is a better choice.

3.3 Layer Generation

In contrast to the round function, P, P^{-1}, S_0, and S_1, are designed to process a single message (or parity) rather than the entire state **ST**. To correct **ST**, we have to decide which signals of **ST** can be securely corrected within a single message and which signals need to be processed separately. To address this limitation, we extend the modules to operate on **ST**, as given below.

3.3.1 Generating Redundant State

To accurately encode every signal of the circuit state, it is necessary to construct the redundant counterpart belonging to **ST**. Due to the importance of control signals, it holds that, every $x \in \mathbf{ST}$ with $\texttt{attribute}(x) = \texttt{control}$ builds its own message while k data signals can be processed within the same message. In particular, we pad every control signal x with $k-1$ leading zeros, resulting in an $(n-k)$-bit parity, denoted as $X' = \mathsf{P}(\langle\{0\}^{k-1}, x\rangle)$. Contrary, data signals are encoded as k-bit chunks, each of which is denoted as $X = \langle x_{k-1}, \ldots, x_0\rangle$ and results in $(n-k)$-bit parity, derived as $X' = \mathsf{P}(X)$. X is only padded with i leading zeros if only $k-i$ data signals are available, i.e. if the number of data signals is not a multiple of k bits. Furthermore, special care must be taken when the given design is masked since the combination of multiple share domains of the same variable within a k-bit message violates the security of masking. To transfer **ST** into a padded state, denoted as **SP**, we apply Algorithm 5.

Before executing Algorithm 5, we cluster the unpadded state **ST** so that all signals with the same attribute, the same share domain, and the same linear index appear in groups one after the other while control signals make a distinct group. In Line 5 of Algorithm 5, a separate message is generated for each control signal by padding the message with $(k-1)$ zeros. To indicate leading zeros, we insert dummy signals driving constant zero into **SP** which we denote as $0 \in \mathbf{SP}$ (see Line 5). As control signals are usually unmasked, and are associated to separate messages, their share domain can be neglected. In contrast, data signals (processed in Lines 7-16) cannot be combined in the same message if their share domains or linear indices are different. Therefore, we only combine data signals in a message with the same parameters and start a new message as soon as we reach a data signal with a different share domain or linear index. Finally, if the number of signals in a message is smaller than k, we fill the remaining space with zeros, see Line 19. Note that we process the feedback state **FB** in the same way to receive a padded feedback state **FP**. Further, we remark that **ST** and **FB** are clustered in the same way meaning that the signals of a certain register have the same index in both states. For example, $u_i \in \mathbf{SP}$ denotes the input signal of the i-th register while $v_i \in \mathbf{FP}$ denotes its corresponding output signal. Now, based on the padded data states **SP** and **FP**, we can create $q \cdot (n-k)$ signals representing the corresponding parity states $\mathbf{SP}'$ and $\mathbf{FP}'$.

Algorithm 5 Secure padding of a state

Input: ST ▷ An unpadded but clustered state, i.e. a set of signals
Input: k ▷ The size of one message
Output: SP ▷ A padded state, i.e. a set of signals

1: $q \leftarrow 0$
2: $l \leftarrow 0$
3: **for** $\forall w \in \mathbf{ST}$ **do**
4: **if** `attribute`$(w) =$ `control` **then**
5: $\mathbf{SP} \leftarrow \mathbf{SP} \cup \{\langle \{0\}^{k-1}, w\rangle\}$
6: **else**
7: **if** `share_domain`$(w) \neq q$ **then**
8: $\mathbf{SP} \leftarrow \mathbf{SP} \cup \{\langle\{0\}^{k-(|\mathbf{SP}| \mod k)}\rangle\}$
9: $q \leftarrow$ `share_domain`(w)
10: **else**
11: **if** `linear_index`$(w) \neq l$ **then**
12: $\mathbf{SP} \leftarrow \mathbf{SP} \cup \{\langle\{0\}^{k-(|\mathbf{SP}| \mod k)}\rangle\}$
13: $q \leftarrow$ `linear_index`(w)
14: **end if**
15: **end if**
16: $\mathbf{SP} \leftarrow \mathbf{SP} \cup \{w\}$
17: **end if**
18: **end for**
19: $\mathbf{SP} \leftarrow \mathbf{SP} \cup \{\langle\{0\}^{k-(|\mathbf{SP}| \mod k)}\rangle\}$

3.3.2 Extending ECC Modules

As depicted in Figure 3, **FB** must be given as an input state to the functions F of the correction logic while both F compute the parity state depending on **FB**. Here, we refer to the sets of outputs of the functions F as $\mathbf{U}'$ and $\mathbf{V}'$. We highlight that both states indicate parity states as $|\mathbf{U}'| = |\mathbf{V}'| = |\mathbf{FP}'|$ and that we derive the wire names of $\mathbf{U}'$ and $\mathbf{V}'$ from $\mathbf{FP}'$ by extending every wire name of $\mathbf{FP}'$ with an unique extension. Both functions $\mathsf{F} : \mathbb{F}_2^{q \cdot k} \rightarrow \mathbb{F}_2^{q \cdot (n-k)}$ map **FP** as a state of q concatenated k-bit messages to a state of q corresponding $(n-k)$-bit parities ($\mathbf{U}'$ and $\mathbf{V}'$). This functionality is achieved through the parallel application of q instances of the modules in $\mathbf{P}$, with each instance processing a single message. Therefore, we start with creating an additional set of modules $\mathbf{F} = \{\mathcal{F}_0, \ldots, \mathcal{F}_{q \cdot (n-k)-1}\}$ abstracting $q \cdot (n-k)$ coordinate functions $\mathsf{f}_i : \mathbb{F}_2^k \rightarrow \mathbb{F}_2$. We remark that every $\mathcal{F}_i \in \mathbf{F}$ computes exactly one signal of the parity state $\mathbf{U}'$. To instantiate every $\mathcal{F}_i \in \mathbf{F}$ we apply Algorithm 6.

Algorithm 6 receives the input state **FP**, the output state $\mathbf{U}'$, the set of modules $\mathbf{P}$, the size of one message k, and generates the set of modules $\mathbf{F}$. It investigates $|\mathbf{P}|$ outputs for every k-bit chunk of the input state **FP**. If the investigated output signal is not driving constant zero (as checked in Line 4), we create a new module that computes the output signal based on the given k-bit input chunk in Line 5. The applied instructions are taken from a particular module of $\mathbf{P}$. As these instructions depend on the input and output signals of $\mathbf{P}$, we replace all signal names in the instructions with the input and output signals of the new module.

More precisely, Algorithm 6 initially takes the first k signals from **FP** to compute the first $(n-k)$ signals from $\mathbf{U}'$. Hence, it creates the first $(n-k)$ modules $\{\mathcal{F}_0, \ldots, \mathcal{F}_{n-k-1}\}$ while every module receives k signals from **FP** as input. As $\mathbf{F}$ is just the extension of $\mathbf{P}$ to a full state, $\mathcal{F}_i \in \{\mathcal{F}_0, \ldots, \mathcal{F}_{n-k-1}\}$ implements the same function as $\mathcal{P}_i \in \mathbf{P}$ but on different signals. Therefore, we can copy the instructions from $\mathcal{P}_i$ to $\mathcal{F}_i$ if we update the

Algorithm 6 Extending ECC Modules

Input: $\mathbf{FP}$, $\mathbf{U}'$ ▷ Input and output states
Input: $\mathbf{P}$ ▷ Set of modules for one codeword
Input: k ▷ The message size
Output: $\mathbf{F}$ ▷ Sets of modules for the entire state

$\mathbf{Z} \leftarrow \emptyset$
for $i \in \{0, \ldots, \frac{|\mathbf{FP}|}{b} - 1\}$ **do** ▷ Iterate through each message
  for $j \in \{0, \ldots, |\mathbf{P}| - 1\}$ **do** ▷ Iterate through each signal of a message
    if $w^{\mathbf{U}'}_{i \cdot |\mathbf{P}| + j} \neq 0$ **then**
      $\mathbf{F} \leftarrow \mathbf{F} \cup \{(\{w^{\mathbf{FP}}_{(i+1) \cdot k - 1}, \ldots, w^{\mathbf{FP}}_{i \cdot k}\}, \emptyset, \{w^{\mathbf{U}'}_{i \cdot |\mathbf{P}| + j}\}, \mathbf{INST}_{\mathcal{P}_j})\}$
    end if
  end for
end for

signal names in the instructions. Afterwards, Algorithm 6 continues with the next k input signals until all messages are processed.

We repeat Algorithm 3 with $\mathbf{V}'$ instead of $\mathbf{U}'$ to receive the modules for the second instance of the function F. The same can be done to produce $\mathbf{F}^{-1}$ but with $\mathbf{P}^{-1}$ instead of $\mathbf{P}$ and appropriate input and output states. Further, the same strategy with $\mathbf{S}_0$ (resp. $\mathbf{S}_1$) and appropriate input and output states can be applied to generate the SD modules $\mathbf{SD}_0$ (resp. $\mathbf{SD}_1$). We remark that we do not apply Algorithm 6 to generate modules for the linear layers as these modules can be straightforwardly generated. For example, if the linear layer receives $\mathbf{U}'$ and $\mathbf{FP}'$ as inputs, the i-th module processes the i-th signal of $\mathbf{U}'$ and $\mathbf{FP}'$ to generate the corresponding single output signal.

3.3.3 Satisfying Independence Property

To establish a functional and correct design, it would be enough to implement a circuit architecture in which all modules are connected based on their input and output wires, as depicted in Figure 3. However, this design would lack robustness against potential faults, if multiple signals, e.g. the outputs of the round function, fail to satisfy the independence property.

Example 2 (One Advanced Encryption Standard (AES) round)**.** In the context of a single round of the AES cipher, it is notable that all coordinate functions of the S-box share the same 8-bit input in a standard implementation, while the MixColumns operation combines the 8-bit outputs of four different S-boxes to produce 32 output bits. As a result, if a single-bit input of any S-box is faulty, the fault can propagate to every S-box output bit and subsequently to every 32 output bits of MixColumns. This example highlights how a single faulty intermediate in an AES round can potentially lead to several faulty bits in the output state of the same cipher round.

To reduce the propagation of faults to a single output bit, such that every faulty intermediate signal results in at most one faulty output bit, AGEFA connects all components in a manner to ensure that no signal serves as a shared input for multiple modules. This is accomplished by allowing only one output signal per module, such that no fault within one module can propagate to multiple outputs. Although all generated modules are solely independent, they must also be connected to other modules to form the desired functions. This connection must be established without sharing intermediate signals between multiple coordinate functions. To accomplish this, we follow a specific composing procedure (cf. Algorithm 7), composing two sets of modules $\mathbf{X}$ and $\mathbf{Y}$. Further, we can iteratively compose the result of Algorithm 7, which is also a set of modules $\mathbf{Z}$, with further sets of modules.

Algorithm 7 Compose two subsequent sets of modules fulfilling independence property

Input: $\mathbf{X} = \{\mathcal{X}_0, \ldots, \mathcal{X}_{|\mathbf{X}|-1}\}$, $\mathbf{Y} = \{\mathcal{Y}_0, \ldots, \mathcal{Y}_{|\mathbf{Y}|-1}\}$ ▷ Two subsequent sets of modules
Input: $\mathbf{OUT} = \{o_0, \ldots, o_{|\mathbf{O}|-1}\}$ ▷ Set of primary output signals
Output: $\mathbf{Z} = \{\mathcal{Z}_0, \ldots, \mathcal{Z}_{|\mathbf{Z}|-1}\}$ ▷ Composed module set computing $\mathsf{Z} = \mathsf{Y} \circ \mathsf{X}$

1: $\mathbf{Z} \leftarrow \mathbf{Y}$
2: **for** $\forall \mathcal{Z} \in \mathbf{Z}$ **do** ▷ Iterate though all subsequent modules
3: $\mathbf{W} \leftarrow \emptyset$
4: **for** $\forall i \in \mathbf{IN}_{\mathcal{Z}}$ **do** ▷ Iterate though all inputs of $\mathcal{Z}$
5: **if** $\exists \mathcal{X} \in \mathbf{X} : i \in \mathbf{OUT}_{\mathcal{X}}$ **then** ▷ Check if a signal is an output of $\mathcal{X}$ and input $\mathcal{Z}$
6: $\mathbf{W} \leftarrow \mathbf{W} \cup \mathbf{IN}_{\mathcal{X}}$ ▷ All inputs $\mathcal{X}$ will become inputs of $\mathcal{Z}$
7: $\mathbf{T}_{\mathcal{Z}} \leftarrow \mathbf{T}_{\mathcal{Z}} \cup \mathbf{OUT}_{\mathcal{X}}$ ▷ All outputs of $\mathcal{X}$ become intermediates of $\mathcal{Z}$
8: $\mathbf{INST}_{\mathcal{Z}} \leftarrow \mathbf{INST}_{\mathcal{Z}} \cup \mathbf{INST}_{\mathcal{X}}$ ▷ $\mathcal{Z}$ computes $\mathbf{T}_{\mathcal{Z}}$ based on $\mathbf{INST}_{\mathcal{X}}$
9: $\mathbf{T}_{\mathcal{Z}} \leftarrow \mathbf{T}_{\mathcal{Z}} \cup \mathbf{T}_{\mathcal{X}}$ ▷ All intermediates of $\mathcal{X}$ become intermediates of $\mathcal{Z}$
10: **end if** ▷ As i is processed, we can continue with the next input
11: **if** $\nexists \mathcal{X} \in \mathbf{X} : i \in \mathbf{OUT}_{\mathcal{X}} \wedge i \neq 0$ **then**
12: $\mathbf{W} \leftarrow \mathbf{W} \cup \{i\}$ ▷ If i is no connection, we consider i as input signal
13: **end if**
14: **end for**
15: $\mathbf{IN}_{\mathcal{Z}} \leftarrow \mathbf{W}$
16: **end for**
17: **for** $\forall \mathcal{X} \in \mathbf{X}$ **do** ▷ Iterate though all primary outputs computed by $\mathbf{X}$
18: **for** $\forall o \in \mathbf{OUT}_{\mathcal{X}} : o \in \mathbf{OUT}$ **do**
19: $\mathbf{Z} \leftarrow \mathbf{Z} \cup \{\mathcal{X}\}$
20: **end for**
21: **end for**

The algorithm takes as input two sets of modules $\mathbf{X}$ and $\mathbf{Y}$, each representing a set of coordinate functions corresponding to X and Y. The resulting output is a third set of modules $\mathbf{Z}$, with each element $\mathcal{Z} \in \mathbf{Z}$ representing a coordinate function of the composition of $\mathbf{X}$ and $\mathbf{Y}$, i.e. $\mathsf{Z} = \mathsf{Y} \circ \mathsf{X}$. In other words, the goal is to construct coordinate functions of $\mathsf{Z} = \mathsf{Y} \circ \mathsf{X}$ while satisfying the independence property.

Algorithm 7 begins by building each coordinate function $\mathcal{Z} \in \mathbf{Z}$ independently, starting with the corresponding $\mathcal{Y} \in \mathbf{Y}$ (cf. Line 2). In the subsequent lines 4-10, the algorithm compares all inputs of $\mathcal{Y}$ with the outputs of every $\mathcal{X} \in \mathbf{X}$. If a match is found, i.e. a signal that is the output of $\mathcal{X}$ and input of $\mathcal{Y}$, the algorithm proceeds to connect $\mathcal{Y}$ independently with $\mathcal{X}$ to $\mathcal{Z}$. The specific steps involved in this connection process are detailed in Lines 6-8.

- In Line 6, all input signals of $\mathcal{X}$ become input signals of $\mathcal{Z}$. Further, in Line 12, all inputs of $\mathcal{Y}$ which are not computed by any $\mathcal{X} \in \mathbf{X}$ become inputs of $\mathcal{Z}$.
- In Line 7, the output signal of $\mathcal{X}$, with connection to $\mathcal{Y}$ becomes an intermediate signal of $\mathcal{Z}$.
- In Line 8, all instructions of $\mathcal{X}$ become instructions of $\mathcal{Z}$ and all intermediates of $\mathcal{X}$ become intermediates of $\mathcal{Z}$ (cf. Line 9).

We acknowledge that Line 8 of Algorithm 7 does not further investigate the instructions of $\mathcal{X}$ before adding them to $\mathcal{Z}$. However, this may result in multiple instructions computing the same intermediate signal, which could lead to multi-driven signals. For instance, suppose that $\mathcal{Y}$ has two distinct input signals that are output signals of modules $\mathcal{X}_0$ and $\mathcal{X}_1$. It is possible that $\mathcal{X}_0$ and $\mathcal{X}_1$ both compute the same intermediate signal $w \in \mathbf{T}_{\mathcal{X}_0} \wedge w \in \mathbf{T}_{\mathcal{X}_1}$. If this is the case, $\mathcal{Z}$ will contain two instructions that compute w, which is redundant. To

address this issue, we include a post-processing step after executing Algorithm 7 where we examine each $\mathbf{I}_{\mathcal{Z}}$ to ensure that each intermediate signal is computed only once. In other words, we remove any instructions that compute an intermediate signal that has already been computed. Moreover, particular attention must be given to primary outputs computed by any module in $\mathbf{X}$. These signals may be either directly wired out without reaching a module in $\mathbf{Y}$, resulting in no connection between the modules, or wired out and also given as inputs to modules in $\mathbf{Y}$, thus connecting $\mathbf{X}$ and $\mathbf{Y}$ while also being primary outputs. To ensure the independence property is maintained, primary outputs are automatically handled as outputs of $\mathcal{Z}$. Therefore, if $\mathcal{X}$ computes a primary output, it is added as an additional module to $\mathbf{Z}$. This ensures that primary outputs are computed by independent modules, even if they serve as intermediates in another module. Algorithm 7 handles primary outputs between two layers in Line 19.

3.4 Finalize

At this juncture, we have arrived at three sets of modules, each representing the sub-circuits demarcated by red dashed boxes in Figure 3. Since these modules do not require any further extensions, we can proceed to write them to the final design. Moving on, we can create the modified register stage by utilizing $\mathbf{ST}$, $\mathbf{ST}'$, $\mathbf{FB}$, $\mathbf{FB}'$ along with the annotated clock signal, and finally we can connect all the modules together in a top module, which can then be printed to complete the final design.

If the circuit encompasses a signal indicating the termination of the cipher computation (commonly identified as a `done` signal), we incorporate the multiplexer-based construction from [SRM20] to avoid that faults injected on such a signal reveal intermediate states to an adversary. Hence, we connect all primary outputs of the final circuit depicted in Figure 3 together with the redundancy of the done signal to a multiplexer tree that only forwards the primary output if there are less than $\frac{\delta}{2}$ bits of the done codeword faulty.

4 Case Studies

To demonstrate the flexibility of AGEFA and the performance of the designs it generates, we applied it to a wide range of publicly available unprotected cipher designs.

4.1 Design Sources

We target the complete cipher cores given in Table 2 of [KMMS22]. We took all the designs from GitHub[7] including both behavioral-level descriptions and Verilog netlists. All given netlists were generated by Synopsys DC and the NanGate 45 nm standard cell library. Their specifications are listed in Table 3.

For the unmasked cipher cores, we utilized the available Verilog netlists from GitHub and annotated the primary inputs and intermediates according to the description given in Section 3, which are then given to AGEFA. Where circuit decomposition was desired, we decomposed **CRAFT**, **Skinny64**, and **Midori** into two sub-circuits, separating the non-linear part and the linear part (without input correction). This procedure is in line with the decomposition strategy outlined in [SRM20]. For the other cores, where the correction of the linear part cannot be removed, we decided to further decompose the linear part based on the diffusion properties of the cipher if it tends to reduce the area footprint. On the other hand, if the designs are masked by AGEMA, we first synthesized the behavioral-level result of AGEMA by Synopsys DC. Beforehand, we annotated every register input or primary output of each respective gadget with its corresponding share domain. Technically, this is done by introducing a new attribute `share_domain` in the

[7] https://github.com/Chair-for-Security-Engineering/AGEMA

Table 3: Full cipher implementation case studies.

Cipher	Implementation	Area	Delay	Latency
		[kGE]	[ns]	[cycle]
AES-128 (serial) [DR02]	Byte-serial enc.	3.3	0.83	227
AES-128 [DR02]	Round-based enc.	9.9	1.85	11
Skinny64 [BJK+16]	Round-based enc. w. 64-bit key	1.5	0.52	33
CRAFT [BLMR19]	Round-based enc. wo. tweak	1.2	0.68	32
PRESENT-80 [BKL+07]	Nibble-serial enc.	1.6	0.59	543
LED-64 [GPPR11]	Round-based enc.	2.1	1.22	33
Midori-64 [BBI+15]	Round-based enc./dec.	2.0	0.97	17

synthesis script. We also kept the annotation during synthesis and included register inputs of the netlist in the attribute report along with their share domain.

4.2 Verification Setup

Since AGEFA produces behavioral-level designs, we proceeded to synthesize the output of AGEFA and verify the resiliency of the resulting netlists against potential faults using VerFI, an open-source tool for cryptographic fault diagnosis [AWMN20]. Available on GitHub[8], VerFI simulates faults on the netlist based on user-defined specifications and determines whether the injected faults are detected and/or corrected. To provide a thorough security analysis in our fault adversary model, we configured VerFI to exhaustively verify whether all possible combinations of $\frac{\delta-1}{2}$ toggle faults injected on arbitrary cells of the netlist during every clock cycle are corrected. Hence, VerFI considers every possible fault during its analysis. Further, if AGEFA processes a masked design, we additionally evaluated the robust probing security of the resulting netlist with PROLEAD [MM22]. PROLEAD, like VerFI, makes use of a simulator to determine the independence of distributions for each possible robust probing adversary. We followed the evaluation guidelines provided on the PROLEAD GitHub[9] repository and conducted two separate evaluations for each design. Specifically, we evaluated each design in compact mode[10] using up to 100 million simulations and in normal mode with an effect size of 0.1. We confirmed the robust probing security of a design only if PROLEAD detected no leakage during both evaluations. We remark that PROLEAD does not perform an exhaustive evaluation, i.e. not all input vectors are checked. Hence, PROLEAD cannot provide a security proof as usually given by formal verification tools, such as SILVER [KSM20], and ends up with a false-negative probability of 10^{-5} when reporting the absence of leakage. However, since full cipher designs protected against SCA and FI are quite large, exhaustive evaluations (e.g. by SILVER) become infeasible while PROLEAD claims efficiency even for large circuits.

Finally, as a sanity check, we performed experimental SCA evaluations on selected designs using an Field Programmable Gate Array (FPGA)-based setup. We measured power consumption traces using a SAKURA-X board encompassing a Kintex-7 target FPGA. We monitored the power consumption, recorded by a digital oscilloscope at a sampling rate of 500 MS/s, at a shunt resistor inserted in the target FPGA's Vdd path while the target design received a stable 6 MHz clock. Using 100 million traces obtained by encrypting either a fixed or a random plaintext for a constant key, we conducted a non-specific (fixed vs. random) t-test to gain an impression about the first-order information leakage of the design under test. This test compares the statistical properties of two groups of traces and detects the presence of information leakage by estimating t-statistic values

[8] https://github.com/emsec/VerFI
[9] https://github.com/ChairImpSec/PROLEAD
[10] Compact and normal mode refer to the techniques of how probability distributions are estimated.

Table 4: Fault-tolerant cipher cores with $\delta = 3$ and smallest circuit size.

Implementation	**Area**		**Delay**		**Latency**
	[kGE]	[overhead]	[ns]	[overhead]	[cycle]
AES-128 (serial)	47.87	×14.51	4.53	×5.45	227
AES-128	141.64	×15.74	7.76	×4.19	11
Skinny64	9.45	×6.30	1.17	×2.25	33
CRAFT	6.63	×5.53	1.36	×2.00	32
PRESENT-80	15.71	×9.82	3.18	×5.39	543
LED-64	28.69	×13.66	4.55	×3.73	33
Midori-64	22.64	×11.32	2.52	×2.60	17

Table 5: Fault-tolerant cipher cores with $\delta = 3$ and smallest critical path delay.

Implementation	**Area**		**Delay**		**Latency**
	[kGE]	[overhead]	[ns]	[overhead]	[cycle]
AES-128 (serial)	58.37	×17.69	2.92	×3.52	227
AES-128	475.05	×47.98	3.28	×1.77	11
Skinny64	9.45	×6.30	1.17	×2.25	33
CRAFT	9.38	×7.82	1.12	×1.65	32
PRESENT-80	21.16	×13.23	1.39	×2.36	543
LED-64	44.40	×21.14	2.19	×1.80	33
Midori-64	32.85	×16.43	1.67	×1.72	17

for every single sample point based on student-t distribution [SM15]. In this work, we depict the t-values for the first- and second-order statistical moments.

4.3 Unmasked Designs

We start with the unmasked designs summarized in Table 3. For every cipher core, we investigated eight different message sizes k, from 1 to 8, and two different minimum distances $\delta = 3$ and $\delta = 5$. Technically, a code that satisfies $\delta = 3$ can correct a single fault while $\delta = 5$ enables the correction of two faults. We remark that the underlying code of the presented scheme must satisfy the correction of two faults per cycle ($\delta = 5$) in order to be $(1, \tau_{bf}, mc_{\infty})$-fault secure. The detailed results including the circuit sizes and the critical path delays for every experiment are given in Appendix A while we summarize the results leading to the best performance in terms of circuit size and critical path delay in Table 4 and Table 5. We remark that AGEFA does not add additional latency to any design. Hence, the number of required clock cycles stays the same as in Table 3.

The results depicted in Table 4 and Table 5 demonstrate the significant impact of the message size k and the applied decomposition on the generated designs. As a result, we can provide practical recommendations for AGEFA's settings based on the specific requirements of the designer.

Recommended message sizes. Our observations indicate that the most area-efficient designs are generated by using $k = 2$ or $k = 4$. These message sizes offer a good trade-off between the number of bits per message, i.e., the number of parallel messages processed, and the area requirements to process one message. However, using $k = 1$ leads to a slightly larger, but still small, circuit size compared to the designs generated with $k = 2$ and $k = 4$ due to the high number of messages and required parallel instances of correction logic. Additionally, larger codes with $k > 4$ become inefficient in terms of area and latency as they require increasingly complex correction logic. Moreover, most cipher cores process

states with sizes that are multiples of 4, leading to less padding. This explains why designs with $k = 8$ perform better than, e.g. $k = 7$, even though the code becomes more complex. Therefore, to optimize the area overhead, we recommend using codes with $2 \leq k \leq 4$. For designers seeking optimization for a short critical path delay, $k = 1$ appears to be the best choice. In this case, AGEFA applies the same correction logic as for MV but to correct faults during every round. In our experiments, setting the message size $k = 1$ consistently yielded the shortest critical path delay for almost every design. This outcome is because the MV code simply duplicates the data signal to build redundancy. Consequently, the mapping between 1-bit messages and their redundant counterparts and vice versa is achieved through wiring, i.e., without any additional computation. Hence, only the SDs contributes to increasing the critical path delay. However, MV codes are usually large, as they cannot be realized with a code size of $2k$. Therefore, they may not be the best choice when seeking area-optimized results. It is important to note that these recommendations should not be considered as strict rules and may not always apply. Therefore, we suggest exploring multiple designs generated with different message sizes before finalizing the design choice.

Recommended decomposition strategy. The impact of decomposition becomes particularly evident when applied to complex functions, such as the **AES-128**, and in scenarios where no correction is needed for the linear layer such as **CRAFT** and **Skinny64**. Therefore, we recommend decomposing the circuit into a linear and a non-linear part whenever it is feasible to remove the correction for the linear component. However, in the case of lightweight ciphers, where removing the correction of the linear layer is not possible, decomposition seems to hurt the area and latency overheads. This is because restricting fault propagation does not justify the costs of additional correction logic. This observation holds since the round function of lightweight ciphers – in contrast to the round-based AES – are simple and the faults cannot propagate extensively. For more complex functions, even the incorporation of multiple additional correction layers can result in a smaller area overhead while the results without decomposition become impractical. Therefore, we recommend decomposing complex functions in general to avoid obtaining impractically large results.

4.3.1 Verification

Our verification with VerFI demonstrated that all injected faults, except for the gate responsible for computing the done signal, become ineffective. However, such a fault is detected and reveals no information to the adversary since no intermediate state is forwarded to the primary output.

4.4 Masked Designs

We selected **CRAFT** to examine the application of AGEFA on masked implementations. This selection is justified by its comparably small area footprint and moderate latency, making it easier to verify using both tool-based and experimental analyses. Due to the same reason, we restrict our experiment to a first-order masked version of **CRAFT** protected by a code with minimum distance $\delta = 3$. AGEFA's generated fault-tolerant version of masked **CRAFT** is still small enough to be verified with PROLEAD in a feasible time and using a realistic amount of memory. Additionally, the design is still compact enough to fit on our FPGA-based setup for experimental analysis, and the length of the power consumption traces remains practical. The above arguments highlight that the decision to use **CRAFT** was purely based on verification considerations. However, AGEFA itself is capable of handling, even higher-order masked versions of all ciphers discussed above in a matter of minutes. Below, we describe our design flow in detail.

- We started with the behavioral-level description of **CRAFT** and synthesized it with Synopsys DC and NanGate 45 nm library to receive a Verilog netlist. The synthesized design has an area footprint of 1.21 kGE and a critical path delay of 0.68 ns.

- To process the netlist with AGEMA, we annotated the primary inputs directly in the Verilog file by marking all signals related to the plaintext or key as `secure`. This was done to ensure that AGEMA produces a design with masked plaintext and key. Additionally, we adjusted AGEMA to use first-order ($d = 1$) protection and to apply HPC2 gadgets [CGLS21] to protect the given netlist. Further, we adjusted AGEMA to operate in the naive mode, which involves replacing every cell that needs to be masked with its HPC2 variant and to produce a non-pipeline design. Compared to a pipelined version of the same design, this approach significantly reduces the overall circuit size by avoiding additional pipeline registers, but at the cost of being able to encrypt only one plaintext in each cipher run. As a result, AGEMA produced a masked behavioral-level design of the given netlist which is provably secure under the $(1, 1, 0)$-robust 1-probing model, i.e. first-order glitch- and transition-extended probing secure.

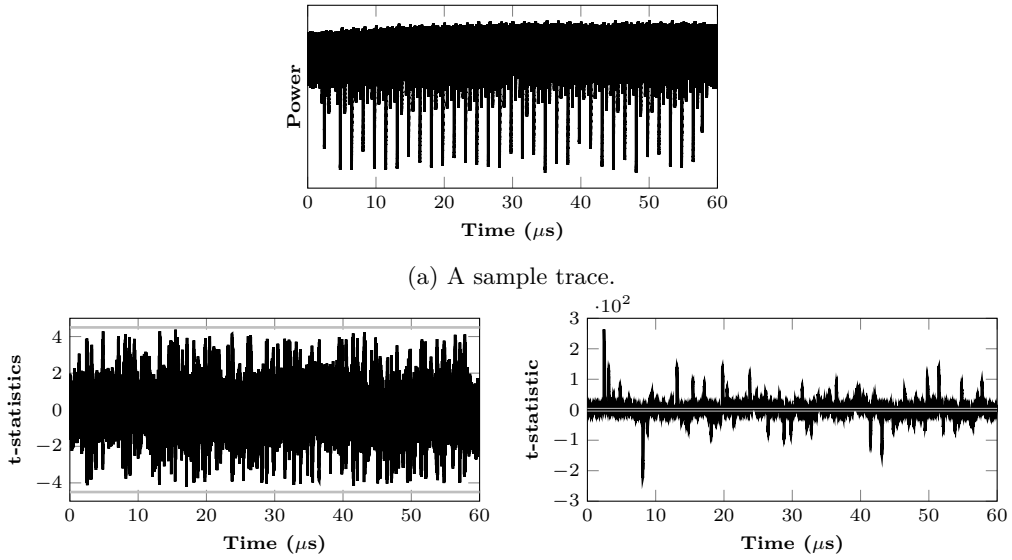
- We synthesized the masked behavioral-level design along with the provided HPC2 gadgets obtained from AGEMA's GitHub. Beforehand, we extended our synthesis script to define the `share_domain` attribute for every masked gadget and set the `share_domain` attribute for every register input for each gadget separately. Finally, we adjusted our synthesis script to generate the attribute report. The masked design provided by AGEMA has an area footprint of 10.84 kGE and a critical path delay of 1.12 ns.

- The synthesized netlist of the masked design serves as the input of AGEFA. We annotated all data inputs, i.e. the masked plaintext and key signals as `secure` while we annotated the register enable and done signal as `control`. Next, we applied AGEFA to generate a fault-tolerant design using ECCs capable of correcting a single fault ($\delta = 3$). It is important to note that $\delta = 3$ is not sufficient for security under the $(1, \tau_{bf}, mc_{\infty})$-fault model, but it facilitates the verification process. By setting $k = 1$, every single bit is seen as a separate message. As a side note, the resulting design does not require the attribute report to maintain probing security if all registers are realized with separate modules. Each message contains only $k = 1$ bit which avoids any two signals from different share domains being combined in a syndrome decoder. This is not the case for any other implementation with $k > 1$, and the attribute report generated by the synthesis script is strictly required.

- To receive the final netlist allowing us to use PROLEAD and VerFI to evaluate the results, we synthesized both behavioral-level designs generated by AGEFA with Synopsys DC using NanGate 45 nm library. The final masked and fault-tolerant design has an area footprint of 112.46 kGE and a critical path delay of 1.47 ns.

4.4.1 Verification

Similar to the unmasked designs, our verification with VerFI led to the conclusion that all considered faults are corrected. We also verified the security of the given design under the $(1, 1, 0)$-robust 1-probing model using PROLEAD. Using 100 million simulations, PROLEAD reported the highest probability for detectable leakage as $-\log_{10}(p) = 5.88$. Since PROLEAD assumes a false-positive probability of 10^{-5}, the original authors claim that exceeding the 5.0 threshold is not a strict criterion for insecure designs if the number of considered probing sets is quite high. Since in our case study, there are 150 560 possible robust probing sets, the probability of at least one probing set surpassing the threshold is



(a) A sample trace.

(b) 1st-order t-test.

(c) 2nd-order univariate t-test.

Figure 5: FPGA-based fixed versus random t-test using 100 million traces, masked **CRAFT**, first-order HPC2 with $[3, 1, 3]$-ECC.

$1 - (1 - 10^{-5})^{150560} = 77.81\%$. Additionally, we observed that the probabilities did not increase when we considered more simulations. Therefore, we conclude that the surpassed threshold is due to a false positive, i.e., the underlying design maintains first-order security.

To verify the absence of leakage, we conducted experimental analyses using the setup detailed in Section 4.2 that yielded the results depicted in Figure 5. Given that the design is anticipated to possess only first-order security (with 2 shares), we expected the absence of leakage in the first order, as confirmed by Figure 5b, as well as the presence of higher-order detected leakages, which is evident from Figure 5c.

4.5 Benchmark

In addition to evaluating the security of the hardware designs produced by AGEFA, it is crucial to analyze the performance of the tool itself, specifically in terms of runtime. To this end, we conducted runtime measurements of AGEFA while generating each of the individual case studies. Our measurements were performed on a standard laptop with an i7-10610U CPU running at a clock frequency of 1.80 GHz, and 16 GB of RAM. We used the Ubuntu 20.04 subsystem running on Windows 10 as the environment for executing AGEFA. The runtime measurements are depicted in Figure 6.

For $\delta = 3$, there are significant differences in the runtime of AGEFA when optimizations are turned on versus when they are turned off. Notably, artificially increasing the code size becomes the primary bottleneck if the message size increases while the optimized variant is usually faster than without optimizations if $k < 5$. When the message size is sufficiently small, the time saved by finding minimal Boolean functions can outweigh the time required to perform the code optimization. This explains the lower runtime observed when AGEFA optimizes the code. Upon analyzing Figure 6, we can conclude that finding an optimized code with $k = 9$ would take hours, and further increasing the message size or the minimum distance would make this optimization computationally infeasible. However, it is essential to emphasize the following points:

- The process of finding an optimized ECC is independent of the specific netlist being

processed. Thus, it is sufficient to find a code with specific k and δ once and reuse it for subsequent designs. This approach allows for the precomputation of ECCs for arbitrary parameters and the use of these precomputations as a database for AGEFA. If a code with specific parameters already exists in the database, AGEFA can simply load the code from the database and skip the code generation process. This strategy can significantly reduce the runtime of AGEFA and make the optimization of ECC computationally feasible for larger message sizes.

- Iteratively checking a large number of codes for their applicability, as they are required to find an optimized code, can be efficiently performed in parallel. Therefore, the aforementioned database of precomputed ECCs can be created on a more powerful machine using multiple threads.

- All case studies imply that large message sizes lead to inefficient designs. Hence, generating large codes should generally be avoided.

If the code optimization is disabled, optimizing Boolean functions with the Quine-McCluskey algorithm can become time-consuming when correcting multiple faults, i.e. as δ grows. However, optimizing Boolean functions is optional and can be avoided by replacing don't cares with concrete results, albeit at the cost of performance. Additionally, this optimization is mostly problematic for large message sizes, which lead to inefficient designs. For instance, generating designs with a code that satisfies $k = 6$ was the most time-consuming case in our experiments with $\delta = 5$, taking just around an hour.

4.6 Comparison

When evaluating the performance of our constructions, it is important to compare them to hand-crafted designs, in which countermeasures are manually implemented by the designer rather than through automated tools. This comparison involves two key aspects. Firstly, we assess how AGEFA processes masked designs generated by AGEMA in comparison to manually masked designs. Secondly, we evaluate how arbitrary designs generated by AGEFA compare to designs where error correction is manually implemented.

4.6.1 Hand-Made Masked Designs vs. AGEMA-Generated Designs

In the context of SCA, an extensive discussion on the advantages and disadvantages of using composable gadgets, as automatically instantiated by AGEMA, versus handmade masking is given in [KMMS22]. Hand-crafted masked circuits are often more efficient under some performance metrics, such as area, latency, or demand for fresh randomness, but can

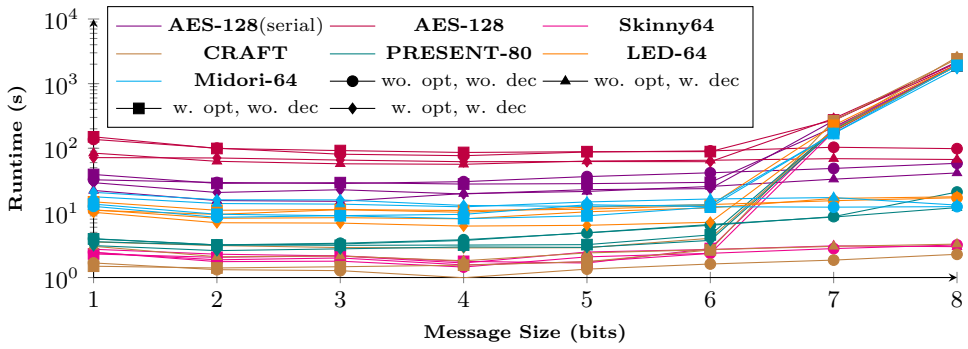


Figure 6: Runtimes of AGEFA for all case studies with $\delta = 3$.

Table 6: Comparison of different **CRAFT** designs.

Implementation	**Area**		**Delay**		**Latency**
	[kGE]	[overhead]	[ns]	[overhead]	[cycle]
Unprotected [SRM20]	1.1	-	0.55	-	32
Cipher-level MV ($\delta = 3$) [SRM20]	4.5	$\times$4.1	1.0	$\times$1.8	32
Cipher-level MV ($\delta = 5$) [SRM20]	7.7	$\times$7.0	1.0	$\times$1.8	32
Impeccable Circuits II ($\delta = 3$) [SRM20]	5.2	$\times$4.5	0.87	$\times$1.6	32
Impeccable Circuits II ($\delta = 5$) [SRM20]	21.6	$\times$19.6	1.1	$\times$2.0	32
Unprotected [*this work*]	1.21	-	0.68	-	32
AGEFA ($\delta = 3$) [*this work*]	6.63	$\times$5.5	1.12	$\times$1.7	32
AGEFA ($\delta = 5$) [*this work*]	29.88	$\times$24.7	1.56	$\times$2.3	32

be difficult to verify and evaluate. Although PROLEAD can evaluate full masked cipher cores, evaluating higher-order designs may be infeasible due to the high demand on runtime and memory. Composable gadgets achieve higher-order provable security by design but at the cost of some overhead. If AGEFA processes a design made out of composable gadgets, the overhead compared to a hand-crafted design is multiplied by at least a factor of two due to the addition of redundancy, i.e. the second instantiation of the round function. However, it is important to note that this overhead applies to all target circuits and is not unique to masked designs generated by AGEMA. Furthermore, error correction typically involves redundant computation, which cannot be avoided. Therefore, the overhead is not specific to AGEFA and would also arise if the designer manually integrates an error-correction facility. As demonstrated above, preserving the robust probing security of a hand-crafted or automatically generated masked circuit when applying AGEFA is straightforward when it is adjusted to set $k = 1$. However, if the user applies another code with $k > 1$, annotating every register input and primary output with its corresponding share domain can be complicated and error-prone. If we pre-annotate the gadgets of AGEMA, the designs can be processed by AGEFA together with an attribute report (generated by the synthesizer like DC) to integrate arbitrary ECCs without violating the robust probing security.

4.6.2 Hand-Made Fault-Resistant Designs vs. AGEFA-Generated Designs

Impeccable Circuits II [SRM20]. Initially, we compare the manually protected designs of **CRAFT** from [SRM20] with the outcomes of AGEFA. Given that the synthesis script utilized to generate the outcomes in [SRM20] is not publicly accessible, we have opted to provide not only the absolute performance metrics but also the relative area and delay overheads when compared to the unprotected design. Specifically, we present the overheads for the synthesized AGEFA-generated designs in relation to the unprotected input design of AGEFA, which we synthesized by ourselves. The performance metrics are detailed in Table 6.

While we acknowledge that cipher-level MV leads to more area-efficient designs compared to the application of Impeccable Circuits II, we must emphasize once more that cipher-level MV falls short of guaranteeing the desired security level (see. Section 2.4.2). Further, the hand-crafted Impeccable Circuits II design with $\delta = 3$ incorporates a non-injective code which complicates the fair comparison to the design generated using AGEFA. To make a truly equitable comparison, we focus on the design for $\delta = 5$, where both the hand-crafted version and the AGEFA result employ injective codes, and the circuit decomposition is consistent. Upon examination, we observe that the AGEFA-generated design exhibits an approximately 26% increase in relative area overhead and an approximately 15% increase w.r.t the critical path delay compared to the hand-crafted design. However, it is essential to note that manual protection of the design offers opportunities for optimization of the target cipher itself, specifically tailored to minimize overhead. Such

optimizations are not feasible with AGEFA since it operates on a pre-synthesized netlist where the target structure is at least partially removed. Consequently, we posit that optimizing the netlist before applying AGEFA may yield more area-efficient designs.

A Countermeasure Against SIFA [BKHL20]. The authors of [BKHL20] investigated a $[3, 1, 3]$-code to protect **Skinny** against one-bit FI attacks. While they did not report the relative area overhead of a real hardware design, they estimated the relative overhead based on the number of basic logic gates. It is worth noting that such an estimated overhead may tend to be overly optimistic, as it does not take into account additional logic elements needed for signal distribution, such as gates with higher drive strength or buffers instantiated to allow higher fan out. However, when considering the smallest design of **Skinny** generated by AGEFA, we end up with a relative area overhead of 6.3 times which is 12.5% higher compared to the relative overhead of 5.6 times reported in [BKHL20] while the manual protection, again, allows a wider range of optimizations on the target cipher. Again, we estimated the relative overhead of the AGEFA-generated design by comparing it to our unprotected, synthesized **Skinny** design (see. Table 3).

4.7 Summary

We observe that the designs generated by AGEFA tend to be quite large, particularly when considering the round-based implementation of the AES without any form of decomposition. This characteristic makes these particular designs more suitable for academic exploration, where our primary objective is to demonstrate the limitations and capabilities of AGEFA. However, our findings indicate that when dealing with lightweight ciphers and/or employing decomposition techniques, the results become more acceptable. In this context, we must emphasize that we operate within a strong adversary model, wherein security cannot be assured through significantly more cost-effective methods like cipher-level MV. Furthermore, it is worth noting that AGEFA produces smaller designs when the unprotected design is serialized (involving multiple cycles per cipher round) as opposed to the round-based approach (with one cycle per cipher round). This introduces a trade-off between area and latency. When we primarily provided round-based designs to AGEFA, we obtained designs of substantial size but with relatively low latency.

5 Conclusions

In conclusion, this paper introduces AGEFA, a fully-automated and open-source framework designed to enable engineers and hardware designers to generate fault-tolerant cryptographic hardware circuits with ease and reliability. The framework leverages various optimization techniques to implement the ECC-based procedure presented in Impeccable circuits II [SRM20] on arbitrary hardware circuits, ensuring the improved performance of the resulting designs. Our tool is effectively demonstrated through a series of case studies that feature well-known symmetric block ciphers, providing insight into the diverse performance trade-offs that arise based on the message size and number of faults to correct, particularly in terms of area overhead and critical path delay. Our case studies yield specific recommendations for creating area-optimized designs (by selecting $k = 2$ or $k = 4$ with decomposition) or delay-optimized designs (by selecting $k = 1$ without decomposition), depending on the desired outcome. Furthermore, we demonstrated that our tool naturally extends the existing security-aware hardware design flow. AGEFA can add fault-tolerance to masked circuits generated with AGEMA, without violating the given security proofs under the robust probing security model by simply annotating internal signals of each gadget. Hence, the combination of AGEMA and AGEFA allows for the generation of fault- and SCA-secure hardware circuits from an unprotected netlist. We conducted

practical experiments and tool-based evaluations to demonstrate the fault-tolerance and SCA resistance of the designs generated by our framework, further affirming our claims. However, while our evaluation shows that the generated designs are secure against SCA and FI adversaries individually, we cannot guarantee their security against combined adversaries who apply both types of attacks simultaneously. Hence, further research in extending AGEFA or AGEMA to ensure security against combined adversaries would be a promising avenue to explore. Additionally, the substantial overhead attributed to error correction can be mitigated in scenarios where assured error detection is deemed sufficient, such as in Root-of-Trust (RoT) modules. Integrating error detection support into AGEFA necessitates an additional output signal, signaling the injection of a fault while employing an underlying EDC with a smaller minimum distance than that of an ECC. Hence, we highlight the exploration of a fault-detection variant of AGEFA as a compelling avenue for future research.

Acknowledgments

The work described in this paper has been supported in part by the German Research Foundation (DFG) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and through the projects 435264177 (SAUBER), by the Federal Ministry of Education and Research of Germany through the Project KOSEF (16KIS1597).

References

[ADN+10] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: On critical paths and clock faults. In *CARDIS 2010*, volume 6035 of *LNCS*, pages 182–193. Springer, 2010.

[AMR+20] Anita Aghaie, Amir Moradi, Shahram Rasoolzadeh, Aein Rezaei Shahmirzadi, Falk Schellenberg, and Tobias Schneider. Impeccable circuits. *IEEE Trans. Computers*, 69(3):361–376, 2020.

[AWMN20] Victor Arribas, Felix Wegener, Amir Moradi, and Svetla Nikova. Cryptographic Fault Diagnosis using VerFI. In *HOST 2020*, pages 229–240. IEEE, 2020.

[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*, volume 9453 of *LNCS*, pages 411–436. Springer, 2015.

[BBM+22] Timo Bartkewitz, Sven Bettendorf, Thorben Moos, Amir Moradi, and Falk Schellenberg. Beware of Insufficient Redundancy An Experimental Evaluation of Code-based FI Countermeasures. *IACR TCHES*, 2022(3):438–462, 2022.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In *EUROCRYPT 1997*, volume 1233 of *LNCS*, pages 37–51. Springer, 1997.

[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*, volume 9815 of *LNCS*, pages 123–153. Springer, 2016.

[BKHL20] Jakub Breier, Mustafa Khairallah, Xiaolu Hou, and Yang Liu. A countermeasure against statistical ineffective fault analysis. *IEEE Trans. Circuits Syst.*, 67-II(12):3322–3326, 2020.

[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 *LNCS*, 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.

[BP93] Richard A. Brualdi and Vera Pless. Greedy codes. *J. Comb. Theory, Ser. A*, 64(1):10–30, 1993.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *CRYPTO 1997*, volume 1294 of *LNCS*, pages 513–525. Springer, 1997.

[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 *LNCS*, pages 1–18. Springer, 2017.

[CEM18] Thomas De Cnudde, Maik Ender, and Amir Moradi. Hardware Masking, Revisited. *IACR TCHES*, 2018(2):123–148, 2018.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware Private Circuits: From Trivial Composition to Full Verification. *IEEE Trans. Comp.*, 70(10):1677–1690, 2021.

[CGM+23] Gaëtan Cassiers, Barbara Gigerl, Stefan Mangard, Charles Momin, and Rishub Nagpal. Compress: Reducing area and latency of masked pipelined circuits. *IACR Cryptol. ePrint Arch.*, page 1600, 2023.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *CRYPTO 1999*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.

[Cla07] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In *CHES 2007*, volume 4727 of *LNCS*, pages 181–194. Springer, 2007.

[Con90] John H. Conway. Integral lexicographic codes. *DM*, 83(2-3):219–235, 1990.

[CS86] John H. Conway and Neil J. A. Sloane. Lexicographic codes: Error-correcting codes from game theory. *IEEE TIT*, 32(3):337–348, 1986.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference. *IEEE TIFS*, 15:2542–2555, 2020.

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: exploiting ineffective fault inductions on symmetric cryptography. *IACR TCHES*, 2018(3):547–572, 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 TCHES*, 2018(3):89–120, 2018.

[FJLT13] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on AES with faulty ciphertexts only. In *FDTC 2013*, pages 108–118. IEEE Computer Society, 2013.

[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*, volume 6917 of *LNCS*, pages 326–341. Springer, 2011.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO 2014*, volume 8616 of *LNCS*, pages 444–461. Springer, 2014.

[GYTS14] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa M. I. Taha, and Patrick Schaumont. Differential fault intensity analysis. In *FDTC 2014*, pages 49–58. IEEE Computer Society, 2014.

[HS13] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *CARDIS 2013*, volume 8419 of *LNCS*, 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 *LNCS*, pages 463–481. Springer, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO 1999*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

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

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

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - Statistical Independence and Leakage Verification. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020*, volume 12491 of *LNCS*, pages 787–816. Springer, 2020.

[Lev60] Vladimir I. Levenshtein. A class of systematic codes. volume 131, pages 1011–1014, 1960.

[LSG+10] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault sensitivity analysis. In *CHES 2010*, volume 6225 of *LNCS*, pages 320–334. Springer, 2010.

[McC56] E. J. McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956.

[Mea55] George H Mealy. A method for synthesizing sequential circuits. *Bell Syst. tech. j. 1955*, 34(5):1045–1079, 1955.

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

[MMSS19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-Resistant Masking Revisited or Why Proofs in the Robust Probing Model are Needed. *IACR TCHES*, 2019(2):256–292, 2019.

[Qui52] W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[RSG23] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. Revisiting Fault Adversary Models - Hardware Faults in Theory and Practice. *IEEE Trans. Computers*, 72(2):572–585, 2023.

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In *CHES 2002*, volume 2523 of *LNCS*, pages 2–12. Springer, 2002.

[SGD08] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. Practical setup time violation attacks on AES. In *EDCC-7 2008*, pages 91–96. IEEE Computer Society, 2008.

[Sha79] Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.

[SM15] Tobias Schneider and Amir Moradi. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In *CHES 2015*, volume 9293 of *LNCS*, pages 495–513. Springer, 2015.

[SRM20] Aein Rezaei Shahmirzadi, Shahram Rasoolzadeh, and Amir Moradi. Impeccable circuits II. In *57th ACM/IEEE DAC 2020*, pages 1–6. IEEE, 2020.

[SSAQ02] David Samyde, Sergei P. Skorobogatov, Ross J. Anderson, and Jean-Jacques Quisquater. On a new way to read data from memory. In *IEEE SISW 2002*, pages 65–69. IEEE Computer Society, 2002.

[WFP+23] Lixuan Wu, Yanhong Fan, Bart Preneel, Weijia Wang, and Meiqin Wang. Automated generation of masked nonlinear components: From lookup tables to private circuits, 2023.

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

A Performance Results of Unmasked Designs

Table 7: Synthesis results, $\delta = 3$, without optimized code and without round function decomposition, circuit size in kGE.

Cipher	Message size k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	58.37	55.28	61.74	50.97	69.40	68.78	71.52	57.39
AES-128	475.00	430.04	439.15	372.47	449.42	451.73	459.85	379.43
Skinny64	12.52	13.05	19.17	13.21	20.67	21.48	21.82	17.30
CRAFT	9.38	9.73	13.35	10.38	14.13	14.12	14.48	12.57
PRESENT-80	21.16	20.90	25.84	23.75	26.59	29.04	29.25	28.88
LED-64	41.71	39.97	44.40	36.10	44.52	44.80	45.65	41.35
Midori-64	30.93	32.90	43.80	36.93	44.65	44.66	45.07	39.79

Table 8: Synthesis results, $\delta = 3$, without optimized code and without round function decomposition, critical path delay in ns.

Cipher	Message size k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	2.92	3.12	3.31	3.32	3.32	3.65	3.38	3.57
AES-128	3.49	3.66	3.78	3.72	4.00	4.11	4.10	4.06
Skinny64	1.22	1.38	1.84	1.64	2.06	2.12	2.12	2.09
CRAFT	1.12	1.29	1.75	1.60	1.94	2.03	1.88	1.77
PRESENT-80	1.39	1.48	1.41	1.73	1.90	1.89	1.94	1.92
LED-64	2.37	2.48	2.19	2.22	2.37	2.40	2.57	2.55
Midori-64	1.70	1.69	2.05	1.99	2.42	2.52	2.58	2.85

Table 9: Synthesis results, $\delta = 3$, with optimized code and without round function decomposition, circuit size in kGE.

Cipher	Message size k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	58.37	55.28	56.72	55.17	58.21	58.95	60.51	59.25
AES-128	475.05	430.55	408.17	385.05	392.23	386.55	389.42	362.82
Skinny64	12.52	13.05	13.84	12.48	14.56	14.72	15.71	14.70
CRAFT	9.38	9.73	10.01	9.05	10.18	10.04	10.67	9.96
PRESENT-80	21.16	20.90	22.42	22.17	22.76	23.31	23.55	23.35
LED-64	41.71	39.97	39.69	36.22	38.54	38.13	38.97	37.86
Midori-64	30.93	32.85	33.38	30.63	33.14	32.81	33.30	31.51

Table 10: Synthesis results, $\delta = 3$, with optimized code and without round function decomposition, critical path delay in ns.

Cipher	Message size k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	2.92	3.12	3.18	3.22	3.23	3.17	3.23	3.39
AES-128	3.28	3.64	3.59	3.59	3.72	3.86	3.82	4.02
Skinny64	1.22	1.38	1.48	1.52	1.63	1.75	1.74	1.80
CRAFT	1.12	1.29	1.38	1.55	1.62	1.58	1.78	1.65
PRESENT-80	1.39	1.48	2.07	1.87	1.83	1.96	1.93	2.07
LED-64	2.37	2.46	2.35	2.50	2.56	2.60	2.53	2.64
Midori-64	1.70	1.67	1.76	1.82	1.83	2.00	2.02	2.29

Table 11: Synthesis results, $\delta = 3$, without optimized code and with round function decomposition, circuit size in kGE.

Cipher	**Message size** k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	53.85	51.31	57.53	47.87	65.38	64.93	68.06	54.64
AES-128	142.51	142.59	179.36	166.28	199.95	191.84	199.59	176.17
Skinny64	9.45	10.06	19.30	10.04	21.76	22.05	23.19	13.34
CRAFT	7.20	7.25	12.92	7.28	14.78	14.88	15.83	8.63
PRESENT-80	15.71	16.74	22.66	20.62	23.75	26.00	26.02	25.79
LED-64	30.73	29.58	33.87	28.69	36.43	37.44	38.78	36.51
Midori-64	22.64	24.91	35.07	26.17	39.05	39.61	41.16	36.36

Table 12: Synthesis results, $\delta = 3$, without optimized code and with round function decomposition, critical path delay in ns.

Cipher	**Message size** k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	4.19	4.11	4.46	4.53	5.19	5.10	5.43	5.07
AES-128	4.41	5.77	7.71	7.28	8.02	8.52	7.84	9.47
Skinny64	1.17	1.37	2.30	1.47	2.61	2.65	2.71	2.05
CRAFT	1.35	1.37	2.08	1.35	2.39	2.26	2.65	1.45
PRESENT-80	3.18	3.27	3.17	3.35	3.40	3.31	3.57	3.58
LED-64	3.73	4.03	4.59	4.55	5.13	5.18	5.53	5.53
Midori-64	2.52	2.88	3.71	3.42	4.53	4.71	5.07	5.05

Table 13: Synthesis results, $\delta = 3$, with optimized code and with round function decomposition, circuit size in kGE.

Cipher	**Message size** k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	53.84	51.31	52.93	52.19	54.94	56.08	57.63	56.94
AES-128	142.72	142.59	142.48	141.64	148.39	147.48	152.02	146.25
Skinny64	9.45	10.06	13.38	9.84	14.61	14.86	16.06	11.53
CRAFT	7.20	7.25	11.02	6.63	11.66	11.83	12.64	7.27
PRESENT-80	15.71	16.74	18.47	18.43	19.17	19.78	20.04	19.94
LED-64	30.73	29.58	30.97	29.04	32.02	32.43	33.99	33.83
Midori-64	22.64	24.91	27.92	27.82	29.83	29.98	31.47	30.82

Table 14: Synthesis results, $\delta = 3$, with optimized code and with round function decomposition, critical path delay in ns.

Cipher	**Message size** k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	4.19	4.11	4.24	4.68	4.72	5.28	5.22	5.26
AES-128	4.40	5.77	6.52	7.76	8.50	7.96	8.01	8.81
Skinny64	1.17	1.37	1.96	1.41	2.26	2.49	2.56	1.38
CRAFT	1.35	1.37	2.16	1.36	2.28	2.35	2.37	1.36
PRESENT-80	3.18	3.27	3.45	3.35	3.57	3.52	3.67	3.69
LED-64	3.73	4.03	4.26	4.58	4.78	4.97	4.80	5.20
Midori-64	2.52	2.88	3.36	3.76	4.07	4.55	4.57	4.61

Table 15: Synthesis results, $\delta = 5$, without optimized code and without round function decomposition, circuit size in kGE.

Cipher	**Message size** k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	308.94	274.75	274.43	335.72	437.70	455.73	463.46	211.67
AES-128	1410.83	1173.13	1109.49	1233.04	1489.54	1454.25	1467.83	890.17
Skinny64	55.32	55.22	66.42	79.84	141.51	151.26	172.08	98.86
CRAFT	45.47	39.54	44.96	45.75	78.88	78.46	99.98	82.65
PRESENT-80	104.83	103.09	107.29	153.85	171.62	193.23	201.02	156.94
LED-64	180.20	144.65	148.35	145.94	207.74	208.62	220.16	184.81
Midori-64	160.61	140.31	150.79	169.08	245.90	251.23	256.48	200.35

Table 16: Synthesis results, $\delta = 5$, without optimized code and without round function decomposition, critical path delay in ns.

Cipher	**Message size** k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	3.25	3.42	3.57	3.73	4.13	4.02	4.64	3.98
AES-128	3.99	4.16	3.97	4.27	4.56	4.48	4.77	4.55
Skinny64	1.56	1.90	2.00	2.20	2.44	2.55	2.99	2.80
CRAFT	1.56	1.68	1.98	2.33	2.39	2.45	2.94	2.91
PRESENT-80	1.85	2.10	1.78	2.07	2.38	2.33	2.82	2.78
LED-64	2.59	3.07	2.90	2.67	2.92	2.90	3.13	3.26
Midori-64	2.02	2.33	2.31	2.43	2.79	2.98	3.15	3.15

Table 17: Synthesis results, $\delta = 5$, without optimized code and with round function decomposition, circuit size in kGE.

Cipher	**Message size** k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	297.87	268.37	268.30	336.97	437.50	459.78	466.19	209.97
AES-128	476.57	485.53	512.93	812.75	1045.05	1140.49	1247.28	832.30
Skinny64	39.87	40.88	67.35	60.46	155.06	170.23	193.60	76.23
CRAFT	30.02	26.86	51.82	29.88	105.45	110.61	126.23	50.33
PRESENT-80	82.02	87.46	95.65	144.44	167.61	190.93	200.32	155.64
LED-64	150.05	142.39	150.93	191.31	272.09	290.63	308.55	203.58
Midori-64	114.30	120.68	140.73	206.74	286.75	308.00	329.97	199.36

Table 18: Synthesis results, $\delta = 5$, without optimized code and with round function decomposition, critical path delay in ns.

Cipher	**Message size** k							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AES-128 (serial)	4.51	5.16	5.49	6.05	6.86	7.02	7.74	6.29
AES-128	5.88	7.87	8.58	10.87	12.02	12.99	13.44	12.39
Skinny64	1.40	1.68	2.59	2.04	3.73	4.04	4.52	2.93
CRAFT	1.79	2.05	2.62	2.14	3.71	3.72	4.48	2.79
PRESENT-80	4.14	4.48	4.17	4.60	4.75	4.66	4.49	4.64
LED-64	4.93	5.15	6.09	6.60	7.05	7.72	8.46	8.11
Midori-64	3.56	4.13	4.79	5.95	6.81	7.09	8.01	7.56