Security Aspects of Masking on FPGAs

Barbara Gigerl
Graz University of Technology
Graz, Austria
barbara.gigerl@iaik.tugraz.at

Kevin Pretterhofer
Graz University of Technology
Graz, Austria
kevin.pretterhofer@gmail.com

Stefan Mangard
Graz University of Technology
Graz, Austria
stefan.mangard@iaik.tugraz.at

***Abstract*—Many IoT and automotive use cases employ cryptographic hardware implementations, which are susceptible to physical attacks such as power analysis. Masking is a popular approach to protect against these attacks on algorithmic level. In general, a masked hardware implementation must be secure in theory, but also in practice. The practical security of masked ASIC designs has been analyzed thoroughly in literature, especially regarding physical defaults such as glitches and transitions, and optimizations performed during synthesis. Besides ASICs, FPGAs are often used to implement masked hardware designs, which utilize reconfigurable look-up tables (LUTs) instead of binary logic gates to implement arbitrary logic functions. Due to their different structure, FPGAs apply different synthesis flows and optimizations, whose effects on the security of masked implementations have not been investigated yet.**

In this work, we present a case study of leakage sources in masked hardware implementations on FPGAs. We investigate several implementations that are formally proven to be secure even in the presence of glitches when implemented as an ASIC but show leakage when running them on an FPGA. We demonstrate in several practical experiments that this leakage is caused by optimizations performed during synthesis, such as LUT combining or register retiming. In order to identify such leaks, we present FENIX, the first tool to formally verify any-order masked hardware implementations directly on FPGA netlists. FENIX takes glitches into account and automatically localizes the leaking wire in case of an insecure design. We demonstrate the practicality of our tool using several masked hardware implementations, including masked multiplication gadgets, a 2nd-order Keccak S-box, and a full Ascon round.

***Index Terms*—Side-Channels, Masking, Formal Verification, FPGAs**

I. INTRODUCTION

Embedded devices have become an essential part of many IoT, automotive, and industrial applications, where they unavoidably get in touch with sensitive information. One key aspect is therefore privacy, which raises the need for strong cryptographic primitives that are able to withstand both theoretical (mathematical) and physical attacks. Physical attacks assume that the adversary has direct access to the device and can observe information about the device during the computation. A famous example is Differential Power Analysis (DPA) [23], which works by monitoring the power consumption of a cryptographic device during computation. The power consumption correlates with the processed sensitive data, such as the encryption key, which can then be extracted using statistical tools. To defeat such attacks, implementations employ the masking countermeasure [8], [16], [21], which splits each sensitive variable into $d + 1$ random shares and performs the cryptographic operations on these shares instead. During the computation, the power consumption of the device at a given point in time does not correlate with the unshared sensitive value, but with at most one share.

The masking countermeasure can be applied to cryptographic hardware implementations, which are realized using an ASIC (Application-Specific Integrated Circuit) or an FPGA (Field-Programmable Gate Array). While ASICs are custom circuits designed for a specific purpose, FPGAs can be configured many times with a design using configurable logic blocks (look-up tables). In recent years, FPGAs have become more and more important in the area of cryptography due to their short time-to-market and reconfigurability [19], [31].

The security of a masked cipher can formally be proven on algorithmic level by analyzing an abstract description of the protected cipher implementation. However, even if the masking scheme is theoretically secure, a concrete implementation of the scheme, such as a hardware implementation running on an FPGA, might still be insecure. Hardware-related physical side-effects such as glitches and transitions, which are not part of the theoretical model, can still lead to the unintentional combination of shares and therefore leak the sensitive value [6], [7], [17], [21], [29], [32], [36]. Additionally, the synthesis process, which transforms an HDL design into a gate-level netlist, may introduce leaks by performing optimizations such as reordering operations [5], [33], [34].

Consequently, the verification of masked implementations has gained a lot of attention recently. In general, there are two approaches to determine whether a masked implementation is secure in practice. The first option is to perform empirical verification by fabricating the design as an ASIC or porting it to an FPGA, recording power traces, and manually analyzing these regarding leakage using statistical tools. Empirical verification is not only an error-prone and laborious task but does also not guarantee leakage-freeness on other platforms if no leak is found. As an alternative, formal verification tools like SILVER [22], REBECCA [5], COCO [13], [20] and `maskVerif` [1] have been proposed which aim at proving the absence of leakage by verifying the respective post-synthesis gate-level netlist. However, these verification tools are tailored to ASIC netlists consisting of logic gates with two inputs and one output. Currently, there does not exist a tool that can handle the structure of FPGA netlists, which consist

of LUTs with multiple inputs and up to two outputs. The effects of the FPGA synthesis process on the security of the design are also not understood yet. Therefore, in practice, to ensure security properties are preserved during synthesis, optimizations are usually disabled globally, for example, by selecting the *keep_hierarchy* option [10], [27], [28], [37]. While this serves the purpose of ensuring that security-critical design partitioning is kept, it also prevents optimizations in parts of the design that are not security-critical.

Our contribution: It is still an open question to which extent the FPGA synthesis process introduces leakage into a masked design. Since existing formal verification tools are built for ASIC netlists, no formal tool exists to detect these leaks on FPGAs. We close this gap by providing the following contributions:

- We present a case study to investigate whether FPGA-specific synthesis flows introduce leakage to masked designs. We investigate two masked multiplication gadgets, for which we formally prove their security on RTL level with an existing ASIC verification tool. Using two different FPGA synthesis tools, we show that the produced netlist is insecure due to glitches introduced by optimization measures such as LUT combining or register retiming. We confirm that the leakage is observable in practice using empirical measurements. (Section III)
- We present FENIX, a tool that can formally verify the security of (any-order) masked FPGA implementations. FENIX operates directly on the FPGA netlist which allows to detect leakage introduced by optimization measures. Instead of turning off optimizations globally by default, which leads to inefficient designs, designers can apply more specific optimizations and check the security of the resulting design using the tool. (Section IV)
- We show the practicality of FENIX by verifying various masked FPGA implementations, including masked multiplication gadgets, a 2nd-order Keccak S-box, and a full Ascon round. We demonstrate the two operation modes, which allow verification with and without considering glitches. If a leak is found, the exact wire and cycle are automatically localized. (Section V)
- We publish FENIX on GitHub[1].

II. BACKGROUND AND RELATED WORK

In this section, we cover necessary background on masking, formal verification tools for masked ASIC designs, and describe formal verification using the Fourier expansion of Boolean functions in more detail.

A. Masking

The power consumption of a cryptographic device depends mostly on the performed operations and the processed data [9], [23]. Using techniques such as DPA [23], this dependency can be exploited to recover the secret key used in an encryption. The masking countermeasure aims at breaking this dependency by randomizing sensitive values before starting the cryptographic operation [8], [16]. Randomization in a dth-order Boolean masking scheme is achieved by splitting each sensitive value s into $d+1$ shares such that $s = s_0 \oplus ... \oplus s_d$, where $\oplus$ denotes the exclusive OR (XOR) operation. $s_0 ... s_{d-1}$ are chosen uniformly at random and statistically independent of s, while $s_d = s \oplus s_0 \oplus ... s_{d-1}$. Consequently, an attacker observing up to d shares cannot infer any information about the sensitive value. In order to mask a cryptographic algorithm, masked descriptions for the linear and non-linear parts need to be found. Masking linear functions is simple since they be computed individually for each share. Masking non-linear functions is more complex because it requires operations on all shares and usually additional insertion of fresh randomness to avoid unintended direct combinations of shares.

B. Leakage sources in masked designs

A masked implementation, even though proven theoretically secure on algorithmic level, is not necessarily secure when implemented on an FPGA. Possible reasons for this are transient timing effects such as glitches and transitions [25], [26], which unintentionally combine the shares of a sensitive value. Glitches are counteracted by inserting registers, which serve as synchronization points to balance out different signal propagation times caused by physical circumstances like different wire lengths. Goddard et al. [14] analyze a masked PRESENT S-box implementation without synchronization and show that leakage can be observed when implemented on an FPGA. Roy et al. [33] study an implementation of a first-order masked SIMON without synchronization and show that glitches and reordering of logic operations lead to leakage when implemented on an FPGA. Finally, Li et al. [24] suggests a simulation-based tool to automatically identify locations where synchronization registers need to be inserted in an FPGA design. Currently, there exists no work which shows that even after inserting synchronization registers, the optimizations performed by EDA tools may again lead to glitches, which in turn cause unintentional share combinations.

C. Formal verification of masking

In order to prove that a masked circuit is algorithmically secure and secure in the presence of glitches, formal verification tools consider a specific attacker model. The probing model introduced by Ishai et al. [21] involves an attacker with d probing needles, which can placed on arbitrary wires/gates in a masked circuit. A probe allows capturing the value from a wire for an infinite amount of clock cycles. A masked hardware implementation is dth-order secure in the probing model if the attacker cannot infer any information about the sensitive value by combining the d probed values. The robust probing model [12] was introduced as an extension to the original probing model, which allows a probe to observe temporary wire values caused by glitches and transitions.

Several automated tools to check the security of a masked circuit have been proposed. REBECCA [5] and COCO [13], [20] apply Fourier-based techniques to approximate correlation

[1]https://github.com/kevPretterhofer/FENIX

sets (leakage sets) for every gate in the circuit. The leakage set contains all values that can be probed by the attacker by observing the gate output. Other tools like `maskVerif` [1], SILVER [22], and IronMask [3] compute these correlation sets exactly, achieving higher accuracy at the cost of efficiency. All these tools first synthesize a masked hardware design with a synthesis tool such as Yosys [38] and then perform verification directly on the gate-level ASIC netlist.

D. Fourier-based verification

REBECCA formally verifies the security of masked ASIC circuits based on Fourier expansions of Boolean functions [5], [30]. Given that every gate represents a Boolean function, the correlation of the gate with respect to a sensitive value can be determined using the Fourier expansion (Walsh expansion) [4], [5], which represents the function as a multilinear polynomial. Given the input variables $X = (x_1, x_2, ...x_n)$ with $x_i \in \{-1, 1\}$, the Fourier expansion of the function $f : \{-1, 1\}^n \to \{-1, 1\}$ with $\{-1, 1\} = \{\top, \bot\}$, is defined as:

$$f(X) = \sum_{T \subseteq X} \widehat{f}(T) \prod_{x_i \in T} x_i \tag{1}$$

The real number $\widehat{f}(T)$ is called the Fourier coefficient of f on T, which directly describes the statistical dependence (correlation) of a Boolean function with regard to its inputs. That is, it does not correlate with $T \subseteq X$ iff $\forall T' \subseteq T$ it holds that $\widehat{f}(T') = 0$ [39]. In a masked circuit, the inputs of a gate are either shares of a sensitive variable, a fresh random value or any unrelated value such as a control signal. To verify that no 1st-order leakage is exhibited by a gate, we must ensure that the output does not directly correlate with a linear combination of shares, i.e., without fresh randomness. This can be done by computing the Fourier coefficients, and checking that the coefficients of linear share combinations are zero. For dth-order security, REBECCA checks the nonlinear combination of any tuple of d gates.

From a verification perspective it is sufficient to know whether a Fourier coefficient is non-zero or not, while computing the exact value is unnecessary. Therefore, REBECCA groups together all linear combinations of inputs a gate correlates to into correlation sets. A correlation set $\mathcal{C}$ of a gate g computing a function f satisfies the following condition:

$$\text{For } T \subseteq X : \prod_{x_i \in T} x_i \in \mathcal{C}(g) \text{ if } \widehat{f}(T) \neq 0 \tag{2}$$

Correlation sets are sound but incomplete approximations, i.e., linear combinations with zero Fourier coefficients might end up in the correlation set. In order to determine the correlation set of a gate, REBECCA applies propagation rules, starting with the correlation sets of circuit inputs which consist only of the respective input variable. For every gate $g = a * b$, $\mathcal{C}(g)$ is computed by applying the propagation rule for the operator *, considering the correlation sets $\mathcal{C}(a)$ and $\mathcal{C}(b)$.

For example, considering a simple circuit with inputs $X = (p, q, r)$ computing $g(p, q, r) = (p \oplus q) \wedge r$. The Fourier expansion of g is $0.5 + 0.5pq + 0.5r + 0.5pqr$. The correlation set of the first gate $g_1(p, q, r) = p \oplus q$ is $\mathcal{C}(g_1) = \{\{pq\}\}$. The correlation set of g, computed by propagating $\mathcal{C}(g_1)$ and $\mathcal{C}(r)$, is $\mathcal{C}(g) = \{\{\}, \{p, q\}, \{r\}, \{p, q, r\}\}$.

III. LEAKAGE SOURCES ON FPGAS

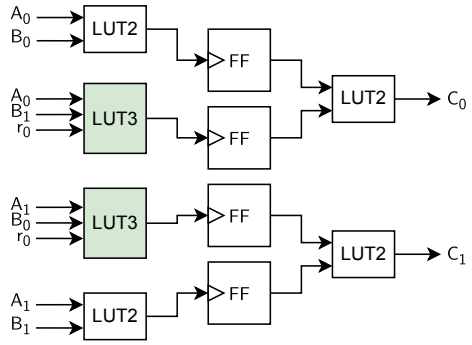
In this section, we present a case study of leakages introduced to masked hardware implementations when synthesized to an FPGA. We show that two masked multiplication gadgets, which are secure in the presence of glitches, become insecure when implemented on an FPGA. We identify glitches introduced by LUT combining and register retiming, optimization measures applied by the FPGA synthesis process, as the main reason for the observed leakage. We give empirical evidence using physical measurements that the leakage is observable in practice. In the following, we first describe our evaluation setup (Section III-A), followed by the experiments in which we observe leakage due to LUT combining (Section III-B) and register retiming (Section III-C). A LUT with n inputs will be denoted as LUTn.

A. Experiment setup

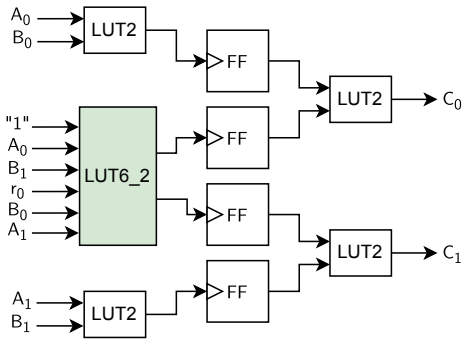
In our case study, we investigate the security of masked multiplication gadgets. Given two shared field elements in $GF(2^n)$, A and B, a masked multiplication gadget computes $C = A \wedge B$ in a secure way. Many masked multiplication gadgets have been introduced in literature [17], [21], [29], [36]. We focus on 1st-order DOM (Domain-Oriented Masking) [17] and 2nd-order ISW (Ishai-Sahai-Wagner) [21].

We use Xilinx Vivado 2022.2 and Xilinx ISE 14.7 to process our designs. Both tools first synthesize the RTL (register-transfer level) design into a netlist consisting of registers, multiplexers and LUTs, and then perform place-and-route, where they assign the netlist components to concrete locations on the FPGA and establish the wiring. Every LUT is physically implemented as a six-input LUT with two outputs (LUT6_2) [40]. For example, if the post-synthesis netlist contains a LUT4, it is mapped to a LUT6_2 during place-and-route, leaving the unused inputs unconnected. The post-place-and-route netlist is eventually used to generate the FPGA configuration file (bitstream file).

For the empirical measurements, we use the NewAE CW305 Artix-7 FPGA evaluation board, connected to a PicoScope 6404C at $1.25\,\mathrm{Gs/s}$ sampling rate ($800\,\mathrm{ps}$ sampling interval). The hardware designs operate at a clock frequency of $1.5625\,\mathrm{MHz}$, a submultiple of the sampling rate. We synchronize the clocks between the FPGA and the oscilloscope to reduce the noise level. We apply Welch's t-test following the guidelines of Goodwill et al. [15] to investigate whether 1st-order leakage is present. For this purpose, a random and fixed set of traces is created. The random set is constructed by assigning fresh random values to the shares of A and B for every trace. For the fixed set, both A and B are set to zero, and fresh values are generated for the shares for every trace. The null hypothesis is that both trace sets have equal means, which can be rejected with a confidence greater than 99.999%

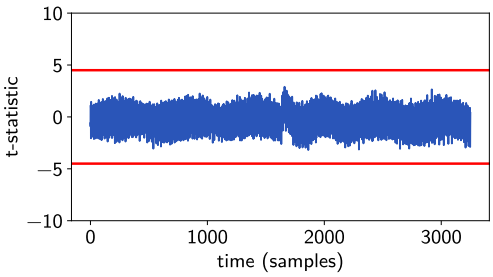


(a) Post-synthesis (without LUT combining)

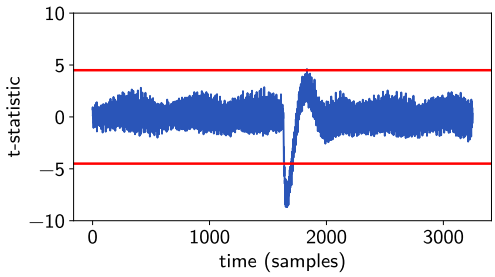


(b) Post-place-and-route (with LUT combining)

Fig. 1: Netlist of 1st-order DOM multiplier with input shares (A_0, A_1), (B_0, B_1) and fresh randomness r_0, and output shares (C_0, C_1). LUT combining is performed during place-and-route.



(a) Without LUT combining



(b) With LUT combining

Fig. 2: Univariate fixed-vs.-random t-test results for the 1st-order DOM multiplier using 6 million traces. No leakage is visible without LUT combining (a), but the design leaks with LUT combining enabled (b).

if the t-score exceeds -4.5 and 4.5. The RNG is enabled for our measurements.

B. LUT combining

LUT combining is an optimization measure that merges several smaller LUTs into a single bigger LUT to save area and reduce the length of the critical path [42]. LUT combining is either applied during synthesis or during place-and-route. LUT combining during place-and-route means that the post-synthesis netlist contains multiple individual LUTs that get merged when mapped to physical components on the FPGA. In the following case study, we show that for masked hardware designs, LUT combining may merge functions, which are supposed to be computed individually, into a single LUT.

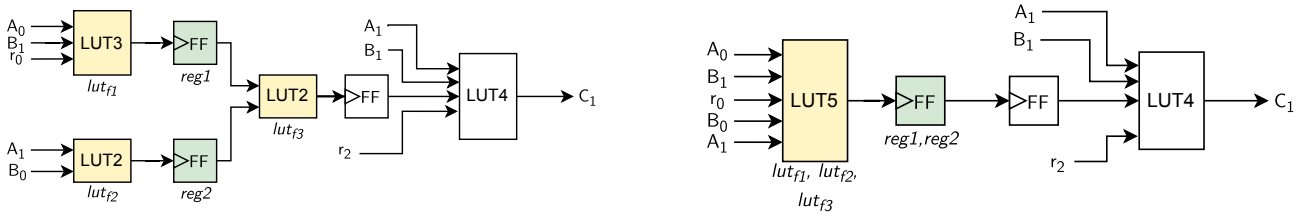
LUT combining during place-and-route: In our first experiment, we investigate a 1st-order masked DOM multiplication gadget [17]. Given the input sharings (A_0, A_1) and (B_0, B_1) and the fresh random variable r_0, the multiplier computes the output sharing (C_0, C_1) as follows, where registers are indicated by parenthesis:

$$C_0 = (A_0 \times B_0) \oplus \big((A_0 \times B_1) \oplus r_0\big) \quad (3)$$

$$C_1 = \big((A_1 \times B_0) \oplus r_0\big) \oplus (A_1 \times B_1) \quad (4)$$

When synthesized to an ASIC netlist with Yosys, as sketched in Appendix A, the design is 1st-order probing secure, i.e., secure also in the presence of glitches, as we confirm by verification with Coco [13], [20].

We synthesize the design with Vivado 2022.2 for an arbitrary chosen 7-series FPGA (xa7s25csga225-2I) using the default settings for synthesis and place-and-route. Figure 1a shows the resulting FPGA netlist of the design after synthesis. The partial multiplication terms $\big((A_0 \times B_1) \oplus r_0\big)$ and $\big((A_1 \times B_0) \oplus r_0\big)$ have been realized using two LUT3s each. The place-and-route process then merges the two LUT3 into a single LUT6_2, as shown in Figure 1b. The first output of the LUT6_2 refers to the first partial multiplication term, and the second output of the LUT6_2 refers to the second partial multiplication term. Internally, a LUT6_2 realizes the two functions using two LUT5s and then uses a multiplexer to realize one output while the other output is directly tied to one of the LUT5s [36], as sketched in Appendix B. Each of the two LUT5s however gets as an input *both* shares of A (A_0, A_1) and both shares of B (B_0, B_1). The functional configurations of the LUT5s are such that the partial multiplication terms are computed, and the unused inputs do not influence the function result once all inputs have stabilized. Still, before all inputs have stabilized, the LUT outputs allow to probe a combination of (in the worst case) all inputs temporarily. The exact combinations that can be observed depend on several things, including the arrival time and ordering of inputs.



(a) Post-synthesis without retiming

(b) Post-synthesis with retiming

Fig. 3: Netlist of 2nd-order ISW multiplier with input shares (A_0, A_1, A_2), (B_0, B_1, B_2) and fresh randomness r_0, r_1, r_2, and output shares (C_0, C_1, C_2). Only the part of the circuit computing one output share C_1 is shown.

Using an empirical evaluation over 6 million traces, we confirm that this leakage can be seen in practical measurements, as shown in Figure 2. Figure 2a presents the t-test results when LUT combining during place-and-route is disabled. As expected, the t-test reveals no significant peaks, which indicates the absence of 1st-order leakage. Figure 2b presents the t-test results when LUT combining is enabled, i.e., following the structure of Figure 1b, which shows significant peaks over the 4.5 border, and therefore, indicates 1st-order leakage. Our evaluation spans four clock cycles, capturing the leaking LUT6_2 computation in the third cycle.

The leak introduced by LUT combining during place-and-route can hardly be detected manually by the designer on RTL level or by looking at the post-synthesis netlist. Manual detection becomes infeasible for larger designs due to the growing complexity, and a possible solution is to disable LUT combining globally. With a formal tool capable of verification on FPGA netlist level, it would be possible to identify single LUTs for which LUT combining must be turned off instead.

LUT combining during synthesis: Designs containing many multiplexers, e.g., a multiplexer tree, are realized by combining them into LUTs during synthesis, which potentially causes additional leakage in masked designs. For example, Shahverdi et al. [35] introduce an area-optimized TI-Simon implementation using three shares, which splits the round function into three identical component functions. The component function is instantiated exactly once, and another set of shares is sent to the input in each cycle. In order to determine which share is being processed, a single LUT is used that has the shares and the select signals as an input. When the multiplexer select signals glitch, a combination of multiple shares might be visible on the LUT output, resulting in leakage. In an ASIC implementation, this leakage is more challenging to observe because the multiplexers are still individual physical components that take at most one share as an input, and there are many other influencing factors, such as the concrete wiring. By contrast, on an FPGA, a glitch on one of the input select signals might directly allow probing a combination of all shares on the LUT output.

C. Register retiming

Register retiming, also known as register balancing, allows improving the delay on the critical path of a design by moving the location of registers across combinatorial logic [11], [41], [42]. The original input/output behavior and latency in terms of cycle count remain unchanged, but the possible clock frequency of the resulting design is increased. Retiming is enabled per default when performing synthesis with ISE 14.7.

In masked designs, the insertion and correct location of registers is crucial for security against glitches. Retiming changes the location of registers and, therefore, introduces glitches into the design. In our case study, we investigate a 2nd-order masked ISW multiplier [21] in which retiming changes the location of registers and share combination happens due to glitches. Again, when synthesized to an ASIC netlist with Yosys, as sketched in Appendix C, we confirm the 2nd-order probing security with COCO. The multiplier computes $C = A \wedge B$ using the random variables r_0, r_1, r_2, resulting in the output sharing (C_0, C_1, C_2), where registers are indicated by parenthesis:

$$C_0 = (A_0 \wedge B_0) \oplus r_0 \oplus r_1 \tag{5}$$

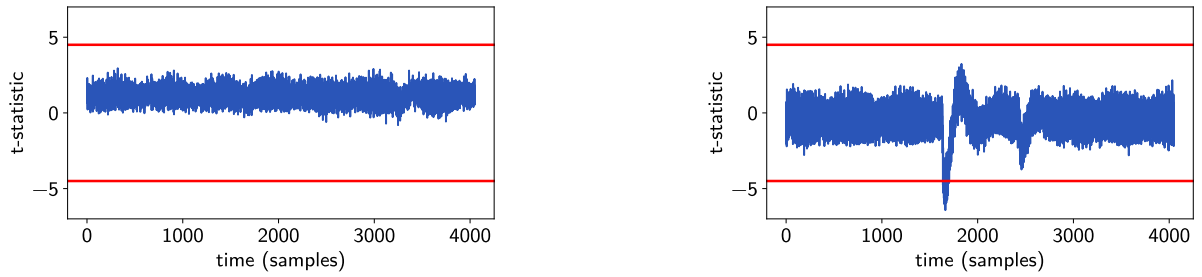
$$C_1 = (A_1 \wedge B_1)\big((r_0 \oplus (A_0 \wedge B_1)) \oplus (A_1 \wedge B_0)\big) \oplus r_2 \tag{6}$$

$$C_2 = (A_2 \wedge B_2) \oplus \big((r_1 \oplus A_0 \wedge B_2) \oplus A_2 \wedge B_0\big) \tag{7}$$

$$\oplus \big((r_2 \oplus A_1 \wedge B_2) \oplus A_2 \wedge B_1)\big) \tag{8}$$

As shown in Figure 3a, when computing C_1 two register stages are required. The intermediate result $(r_0 \oplus A_0 \wedge B_1)$ is computed by a LUT3 (lut_{f1}), while $(A_1 \wedge B_0)$ is computed by a LUT2 (lut_{f2}).

The result of lut_{f1} needs to be stored to a register *reg1* to ensure the refreshing with r_0 is finished before combining it with the result of lut_{f2}. Figure 3b shows the netlist of the circuit after synthesis with ISE 14.7 where register retiming is applied. The registers *reg1* and *reg2* are moved forward, i.e., they are swapped with lut_{f3}. Then, the synthesizer merges lut_{f1}, lut_{f2} and lut_{f3} into a single LUT5, and registers *reg1* and *reg2* can be merged. The area of the design is reduced, since the resulting circuit only requires two LUTs instead of four, and two registers instead of three. However, *reg1* ensuring proper refreshing is removed and consequently, an attacker probing the output of the LUT5 can learn a function of two shares of A and B respectively due to glitches, for example, if the randomness r_0 arrives later at the LUT5 input. Note that in this example, LUT combining happens besides register retiming, but the leakage would also be present if LUT combining was



(a) Without register retiming

(b) With register retiming

Fig. 4: Univariate fixed-vs.-random t-test results for the 2nd-order ISW multiplier using 30 million traces. No leakage is visible without register retiming (a), but the design leaks with retiming enabled (b).

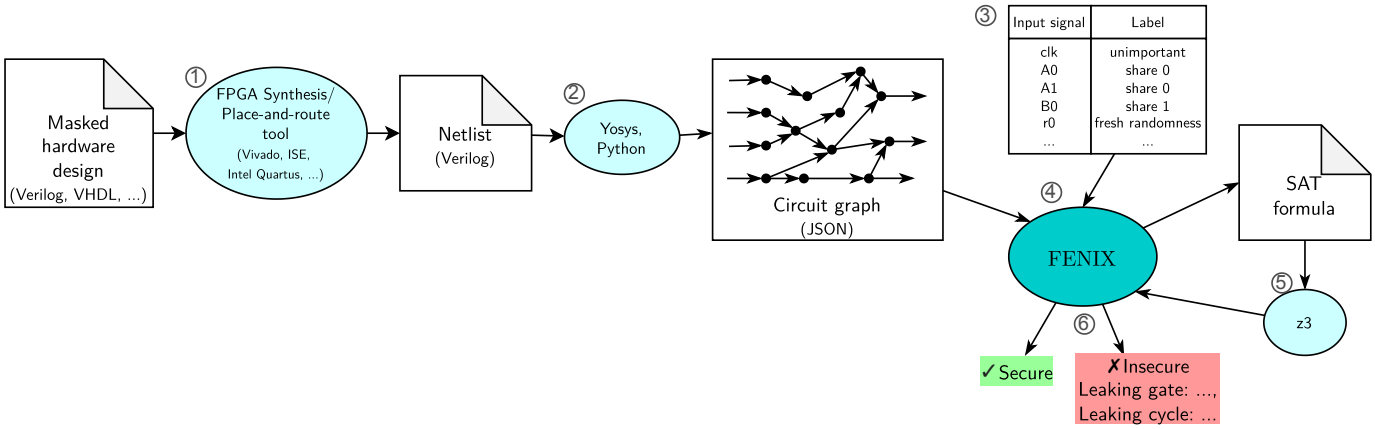


Fig. 5: Verification flow of FENIX, consisting of ①-③ three preprocessing steps, ④ the actual verification step, ⑤ the SAT solving step and ⑥ the interpretation step.

disabled.

Intuitively, the computations of C_0, C_1, and C_2 individually must be 1st-order secure such that the whole gadget can be 2nd-order secure. Therefore, in the empirical measurements, we assess the 1st-order security of the part of the circuit computing C_1. Figure 4 shows the results of our empirical leakage evaluation using 30 million traces, which confirms that register retiming introduces 1st-order leakage to the design. More specifically, we compare the security of the design without register retiming (Figure 4a) to the security of the design when retiming is enabled (Figure 4b). In the latter case, we perceive peaks in the t-score over the 4.5 border as expected, indicating 1st-order leakage.

The leak introduced by register retiming was not detected by COCO because the ASIC netlist did not contain the retimed registers. To spot the leakage, the post-synthesis FPGA netlist needs to be considered.

IV. FENIX

In this section, we describe how we built FENIX, the first tool for the formal verification of masked FPGA implementations. FENIX operates directly on post-place-and-route netlist level and therefore easily detects leaks introduced by synthesis and place-and-route. Similar to REBECCA/COCO, our tool utilizes Fourier expansions and approximates correlation sets for LUTs, registers, and multiplexers, which are part of the FPGA netlist. The correlation sets are encoded and then checked with a SAT solver. First, we describe the high-level verification flow of FENIX, give a precise description of the input netlist produced by ①, and describe the verification step (④) in detail.

A. Verification Flow

The verification flow implemented by FENIX consists of six steps, as shown in Figure 5, divided into three preprocessing steps ①-③, the verification step ④, the solving step ⑤, and the interpretation step ⑥:

① The masked hardware design written in an HDL such as Verilog or VHDL is processed by the FPGA design flow, which typically consists of synthesis and place-and-route. In our experiments, we focus on Xilinx tools, although the verification flow could, in principle, be used with tools from other vendors such as Intel or Lattice as well. The result of this process is the post-place-and-route netlist in Verilog format. LUTs that are combined into a LUT6_2 are marked as pairs using dedicated annotations.

② Using Yosys [38], we parse the netlist and transform it into a circuit graph with gates as nodes and wires as edges.

TABLE I: Overview of propagation rules

Gate type	Function	Propagation rule	
		Stable $\mathcal{S}^t(g)$	**Transient** $\mathcal{T}^t(g)$
Input	$x \in X$	$\{x\}$	$\{x\}$
Register	Copy	$\mathcal{S}^{t-1}(g)$	$\mathcal{S}^{t-1}(g)$
Multiplexer	$g = g_3?g_1 : g_2$	$\mathcal{S}^t(g_1) \cup \mathcal{S}^t(g_2) \cup (\mathcal{S}^t(g_3) \triangle \mathcal{S}^t(g_1)) \cup (\mathcal{S}^t(g_3) \triangle \mathcal{S}^t(g_2))$	$\triangle_i(\emptyset \cup \mathcal{T}^t(g_i))$ for $1 \leq i \leq 3$
LUTn	$g = f(g_1, ...g_n)$	Derived from INIT	$\triangle_i(\emptyset \cup \mathcal{T}^t(g_i))$ for $1 \leq i \leq n$
LUT6_2	$g = f_1(A), A \subseteq (g_1, ...g_5)$	Derived from INIT_g	$\triangle_i(\emptyset \cup \mathcal{T}^t(g_i))$ for $1 \leq i \leq 5$
	$h = f_2(B), B \subseteq (g_1, ...g_5)$	Derived from INIT_h	$\triangle_i(\emptyset \cup \mathcal{T}^t(g_i))$ for $1 \leq i \leq 5$

The graph is stored in JSON format. Using a dedicated preprocessing script, we merge the LUTs, which are marked as pairs in the netlist, into a single LUT6_2 element. Additionally, we remove cycles in the circuit by unrolling and then topologically sort the graph such that the root nodes are registers and circuit inputs.

③ A label is assigned to every circuit input bit, which expresses its purpose in the masking scheme. An input bit can either be a *share*, *random* or *unimportant*.

④ For each node in the circuit graph, FENIX computes the respective correlation set according to the propagation rules and encodes them into a SAT formula with regard to the masking order. For LUTs, FENIX determines the propagation rules in a precomputation step. This process is repeated for a specific number of clock cycles for a specific masking order chosen by the user. Similar to REBECCA/COCO, FENIX supports both stable and transient verification.

⑤, ⑥ The resulting SAT formula is given to the Z3 Theorem Prover, which searches for leaks in the correlation sets over all cycles. If a leak is found, the SAT formula is satisfiable, and a possible variable assignment is given to the FENIX tool. The tool interprets the model and reports the leaking gate and cycle. If no leak is found, the SAT formula is unsatisfiable, and the tool reports that the circuit is secure.

B. Input netlist

FENIX operates on the post-place-and-route netlist in Verilog format. In Vivado, it can be extracted using the `write_verilog` TCL command. The netlist describes the design as a graph where nodes are either circuit inputs, outputs, LUTs, registers, or multiplexer cells, and edges are wires. For every LUT, the netlist contains the initialization (INIT) string, which represents the output values of the LUT for every possible assignment of the inputs and eventually allows to determine the logic function.

On the FPGA, LUTs are grouped into slices, and slices are grouped into CLBs (Configurable Logic Blocks), which are connected to a switch matrix for routing [40]. Slices are categorized into SLICEL and SLICEM. SLICEL consist of four LUT6_2, eight registers, several cascade multiplexers, and logic elements to realize carry logic efficiently. SLICEM additionally support storing data using distributed RAM and contain shift registers. Which features are used depends on the configuration of the slice. For most cryptographic implementations, SLICEL are configured to process an input using the LUTs and storing either the result to the register or forwarding it to another LUT. In this case, the post-place-and-route netlist verified by FENIX will contain only the LUTs and registers if used because the post-place-and-route netlist replicates the exact configuration of slices. The cascade multiplexers and carry logic are not included in the netlist, and therefore not considered by the verification, even though they are physically present. However, it is valid to exclude these logic elements from the verification because they are inactive, i.e., do not actively compute anything, and can therefore not cause any share-dependent combinations leading to leakage in the design. For example, as sketched in Appendix D, the three cascade multiplexers each connect two LUT6_2 outputs, and the third multiplexer connects the output of the two multiplexers before. If the multiplexer select signals glitch, the attacker could, in the worst case, probe a combination of the output of all four LUTs in the slice. However, in practice, the select signals are stable and never change their value, which allows to probe only the output of a single LUT and therefore does not give the attacker any advantage. In case any implementation ever requires the use of the cascade multiplexers and carry logic, which is very rare for cryptographic implementations, the slice configuration is changed, and the respective elements are added to the netlist.

C. Verification of LUTs

Before starting verification with FENIX, the user needs to provide a label file indicating the purpose of each input signal. We incorporate the labeling system of REBECCA/COCO, that is, a *share* represents a share of a sensitive bit, *random* means a fresh and uniformly-distributed random bit, and *unimportant* means that it is not relevant for the masked implementation, e.g., a control signal or the clock input. These labels are directly translated into correlation sets for the input signals of the circuits. Then, FENIX propagates these correlation sets through the circuit according to the propagation rules. Which propagation rule is used depends on the gate type (input, register, LUT) and the verification mode (stable or transient). The stable verification mode refers to checking the security in the classic probing model [21], while the transient verification mode allows glitch-extended probes. In this work, we propose propagation rules for LUTs for both stable and transient veri-

fication for the first time . The propagation rules for registers and multiplexers are carried over from REBECCA/COCO.

Like COCO, FENIX generally performs verification in the time-constrained probing model. The time-constrained probing model [13] restricts the temporal scope of a probe to one clock cycle, i.e., a probe is used to observe information in one specific clock cycle at one specific location. By that, FENIX can be used to verify non-pipelined circuits, although it is restricted to verifying circuits without state machines and control signals. Therefore, we formulate propagation rules (correlation sets) implemented by FENIX in a cycle-aware fashion, which allows a more intuitive interpretation of results and several optimizations. In Table I we give an overview of the propagation rules used by FENIX. $\triangle$ refers to the symmetric set difference.

Propagation rules for LUTn: In the stable case, the correlation set computed according to the propagation rule reflects the correlation at the LUT output, assuming no glitches on the input. Before verification starts, we determine a stable propagation rule for each LUT in the netlist by computing the Fourier representation of the LUT's function. In order to determine these propagation rules for a LUT representing the function $g = f(g_1, ...g_n)$, we apply a three-step process inspired by [30]:

1) Every LUTn in the circuit graph is associated to a hexadecimal initialization string which represents the output values of the LUT for every possible assignment of inputs in ascending order. We convert the initialization string to a 2^n-digit binary number $\mathsf{INIT} = \{-1, 1\}^n$, and then compute the truth table Y:
$$\forall a \in \{-1, 1\}^n : Y[a] = \mathsf{INIT}[pos(a)] \tag{9}$$
The function $pos(a)$ converts the n-bit binary vector a into the index such that $f(a) = \mathsf{INIT}[pos(a)]$.
2) We compute the indicator polynomials $\mathbb{1}_a$. Intuitively, $\mathbb{1}_a(g_1, ...g_n) = 1$ if $a = (g_1, ...g_n)$, else it is 0:
$$\forall a \in \{-1, 1\}^n : \mathbb{1}_a(g_1, ...g_n) = \prod_{i=1}^{n} \frac{1 + a_i g_i}{2} \tag{10}$$
3) Using the indicator polynomials we arrive at the Fourier representation of f by computing and summarizing the following expression:
$$f(g_1, ..g_n) = \sum_{a \in \{-1,1\}^n} Y[a] \cdot \mathbb{1}_a(g_1, ...g_n) \tag{11}$$

From that expression, we extract the linear combinations of the inputs and the Fourier coefficients, which enables the construction of the correlation set $\mathcal{S}^t(g)$ according to Equation (2). The propagation rule is then formed by replacing the abstract LUT inputs by the respective input correlation sets $\mathcal{S}^t(g_1), ...\mathcal{S}^t(g_n)$.

In the transient case, FENIX makes the worst case assumption and extends the attacker's abilities by assuming any arbitrary Boolean function from the LUT's original inputs may be probed, independent of the INIT string or the concrete ordering of inputs. This is reflected by the respective propagation rule as shown in Table I.

Example: Consider a LUT3 with initialization string 0x78 = $(01111000)_2$, representing the function $f(g_1, g_2, g_3) = (g_1 \wedge g_2) \oplus g_3$. To compute the propagation rule for the stable mode we first convert 0x78 to INIT = (1,-1,-1,-1,-1,1,1,1). By computing the truth table and indicator polynomials we arrive at the Fourier expansion $f(g_1, g_2, g_3) = 0.5g_3 + 0.5g_1g_3 + 0.5g_2g_3 - 0.5g_1g_2g_3$, and the correlation set with regard to abstract inputs $\mathcal{S}(g) = \{\{g_3\}, \{g_1, g_3\}, \{g_2, g_3\}, \{g_1, g_2, g_3\}\}$. This yields the propagation rule $\mathcal{S}^t(g) = \mathcal{S}^t(g_3) \triangle (\mathcal{S}^t(g_1) \cup \mathcal{S}^t(g_3)) \triangle (\mathcal{S}^t(g_2) \cup \mathcal{S}^t(g_3)) \triangle (\mathcal{S}^t(g_1) \cup \mathcal{S}^t(g_2) \cup \mathcal{S}^t(g_3))$. In the transient case, the propagation rule says to propagate all possible combinations of the transient correlation sets, i.e., $\mathcal{T}^t(g) = (\{\emptyset\} \cup \mathcal{T}^t(g_1)) \triangle (\{\emptyset\} \cup \mathcal{T}^t(g_2)) \triangle (\{\emptyset\} \cup \mathcal{T}^t(g_3))$.

Propagation rules for LUT6_2: A LUT6_2 consists of two LUTs connected by a multiplexer (cf. Appendix B), that have the common inputs $(g_1, ...g_5)$. Each LUT operates on a subset of inputs, i.e., the first LUT computes $g = f_1(A), A \subseteq (g_1, ...g_5)$ and the second LUT computes $h = f_2(B), B \subseteq (g_1, ...g_5)$. The sixth input is driven high, which ensures that the output of the first LUT is forwarded to the first output port, and the output of the second LUT is forwarded to the second output port. In the stable case, we compute the propagation rules for the two individual LUTs, and use the rules to assign the correlation sets to the respective output. Conceptually, we treat the two LUT5s independently from each other. In the transient case, we however need to take into account that all five inputs are forwarded to each of the two LUTs, and an attacker might probe an arbitrary combination of all five inputs even though not all inputs are processed functionally. Therefore, we assign the same correlation set to both outputs to capture the fact that all inputs $(g_1, ...g_5)$ enter both LUTs. Similar to the regular LUTn, we assume the attacker can probe an arbitrary function of input signals.

V. EVALUATION

In this section, we first describe the evaluation setup and then discuss and interpret the results obtained from verifying different masked implementations. We compare our tool to others, including REBECCA [5], COCO [13], `maskVerif` [1], SILVER [22], and discuss possible optimizations to improve the tool for the future.

A. Setup

For evaluating FENIX in terms of verification runtime, we use a notebook with an AMD Ryzen 5 4500U CPU with 16 GB of RAM. To synthesize the designs, we use Vivado 2021.1 and a Xilinx 7 Series FPGA as the target device (XC7S75-FGGA676-1). For our experiments, we use the default synthesis options of Vivado with a few exceptions. First, we set the Verilog attribute `extract_reset` to the reset signals in our designs. This ensures that the reset signal is directly connected to the respective register instead of being used as an input to the LUT before the register. Unless stated otherwise, we set the options `-no_lc` to true to prevent

TABLE II: Verification of masked implementations using FENIX. ✖ indicates intentionally broken implementations. Nodes include inputs, outputs, registers, multiplexers and LUTs.

Name	Nodes	Cycles	Input shares	Fresh randomness	Runtime	
					Stable	Transient
1st-order						
DOM-AND [17]	107	2	4 × 8 bit	8 bit	0.1 s	0.1 s
DOM-AND [17] with LUT combining	20	2	4 × 1 bit	1 bit	0.1 s	< 0.1 s ✖
DOM-AND [17] without registers [17]	74	1	4 × 8 bit	8 bit	<0.1 s	<0.1 s ✖
TI-AND [29]	96	1	6 × 8 bit	-	<0.1 s	<0.1 s
ISW-AND [21]	155	3	4 × 8 bit	8 bit	0.3 s	0.6 s
Trichina-AND without registers [36]	64	1	4 × 8 bit	8 bit	<0.1 s	0.6 s ✖
DOM Keccak S-box [18]	62	2	10 × 1 bit	5 bit	<0.1 s	0.6 s
DOM Ascon Round	5136	3	10 × 64 bit	320 bit	9 min	9.5 h
DOM AES S-box [17]	409	5	2 × 8 bit	28 bit	4.5 min	-
2nd-order						
DOM-AND [17]	219	2	6 × 8 bit	24 bit	2.1 s	19 s
ISW-AND [21]	315	3	6 × 8 bit	24 bit	42 s	1.8 m
ISW-AND [21] with retiming	29	4	6 × 1 bit	3 bit	0.3 s	< SI0.1s
DOM Keccak S-box [18]	62	2	15 × 1 bit	15 bit	0.3 s	1.7 s
3rd-order						
DOM-AND [17]	49	2	8 × 1 bit	6 bit	<0.1 s	0.1 s
DOM-AND [17]	371	2	8 × 8 bit	48 bit	1.3 min	1.9 h

LUT combining (cf. Section III-B), and `-retiming` to false to prevent register retiming (cf. Section III-C) to construct secure FPGA designs. Instead, we could also have placed `dont_touch` attributes on selected wires.

B. Results

The verification results of the masked hardware implementations and the verification runtime are shown in Table II. We evaluate several 1st-, 2nd- and 3rd-order masked hardware implementations. Table II shows the name of the masked design, the number of nodes (inputs, outputs, registers, multiplexers, LUTs) in the circuit graph, the number of verified cycles, and the number of labels provided by the user (shares and fresh randomness). In terms of runtime, we report the total verification runtime for the stable and transient mode, which includes the steps ④-⑥.

The selection of masked circuits covers various 1st-order masked AND gadgets [17], [21], [29], [36] which can all be verified in less than a second. We include a DOM-AND implementation without a register stage, which is secure in the stable case but exhibits leakage in the transient case due to glitches, which is correctly detected by FENIX. During our analysis, we make an interesting observation about the security of the Trichina-AND gadgets, which we implement without a register stage. In the transient case, the implementation is therefore insecure due to glitches, as confirmed by our tool. However, the Trichina-AND gadget passes the stable verification on the FPGA netlist with FENIX, but fails stable verification with REBECCA when implemented on an ASIC as shown in [5]. The reason for this is that in the case of REBECCA, the ASIC synthesis tool reorders the individual binary gates, but the specific order of gates is crucial for stable security. By contrast, when implemented on an FPGA, the binary gates are merged into a single LUT, leaving less possibilities for reordering.

FENIX can also be applied to bigger designs, like the Keccak S-box. We successfully verify a complete Ascon round consisting of more than 5000 nodes using 64-bit input shares in less than 10 hours in the transient mode. By verifying 2nd- and 3rd-order DOM-AND gadgets, and a 2nd-order ISW-AND gadget, we show that FENIX can be applied to higher orders. We successfully verify a 1st-order masked AES S-box protected by DOM with a latency of five cycles in 5 minutes for the stable case. In the transient case, the solver returned no result after one week, although FENIX managed to build the formula in 3 seconds. This is mainly due to the complex structure of an AES S-box, and to choosing Z3 as a solver. Also, in the original publication of REBECCA [5], the authors mention that they checked each sensitive bit separately while treating the others as constants and starting the verification eight times instead of labeling all sensitive bits at once, which significantly reduces the complexity of the SAT problem. The reduction of the complexity can, for example, be seen very well when comparing the verification runtimes of FENIX when verifying a 3rd-order DOM-AND gadget. For 1 bit, the verification finishes in 0.1 s, while for 8 bits, it takes 1.9 h.

C. Comparison and Future Work

Currently, no verification tool operating on FPGA netlists exists, which does not allow for a direct and fair comparison with another state-of-the-art tool. Also, comparisons with other ASIC verification tools need to be made with caution, as these tools often use parallelization instead of only one CPU core. Compared to REBECCA, FENIX provides similar and, in most cases, even better verification runtimes, given the fact that REBECCA uses multithreading and focuses on 1-bit implementations. COCO, which was built on top of REBECCA for verifying masked software implementations on CPU netlists, was recently extended for hardware implementations [20]. COCO is able to provide very low verification runtimes due to extensive simplification of correlation sets based on the behavior of ASIC gates. For example, it tracks whether the input of an AND gate is zero and glitch-free, allowing to safely

assume that the output will also be zero and glitch-free. Such assumptions are not possible in the case of LUTs since the exact internal structure of LUTs, and therefore, their glitch behavior, is unknown. Additionally, COCO considers control signals by reading simulation traces of the respective designs, which also helps to simplify stable correlation sets. A similar technique could be integrated into FENIX' stable verification mode and would require to re-compute the propagation rules for LUTs in every cycle depending on the value of a concrete control signal.

The verification runtimes of basic gadgets are also comparable to the ones reported by `maskVerif` and SILVER. For future work, it would be very interesting to check other properties such as composability, e.g., strong non-interference (SNI) [2] with FENIX as it is already done by `maskVerif` and SILVER.

The total verification runtime of FENIX is largely determined by the time it takes to solve the generated SAT formula. For example, for verifying the ASCON Round, it took only 15 s to create the SAT formula (step ④), while it took 9 min to solve it. The evaluation of different solvers is therefore also an open point for future work, as we believe a solver better suited for our problem would further decrease the verification runtimes drastically.

VI. CONCLUSION

In this paper, we investigated the effect of the FPGA synthesis process on the security of masked hardware designs. We showed that optimization measures such as LUT combining and register retiming could introduce additional leakage into a design that was formally verified to be secure even in the presence of glitches. Using empirical measurements, we demonstrated that the leakage is also observable in practice. These leaks can hardly be detected manually by the designer on RTL level and usually require a close inspection of the post-synthesis or post-place-and-route netlist, which becomes infeasible for larger designs due to the growing complexity. Based on this case study, we therefore presented FENIX, the first formal verification tool to verify masked FPGA designs directly on netlist level. More concretely, we show how the Fourier analysis of Boolean functions can be used to determine propagation rules for LUTs in a circuit, which can then be used to estimate correlation sets during the verification. We evaluated the tool using several masked designs, including multiplication gadgets and a full Ascon round.

ACKNOWLEDGEMENTS

This research is supported by the Austrian Science Fund (FWF SFB project SPyCoDe F8504). We thank the anonymous reviewers for their valuable feedback.

APPENDIX A
1ST-ORDER DOM MULTIPLIER

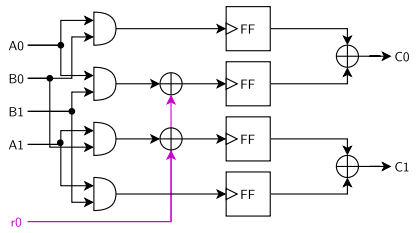


Fig. 6: 1st-order DOM multiplication gadget [17] represented as an ASIC netlist.

APPENDIX B
INTERNAL STRUCTURE OF LUT6_2 PRIMITIVE

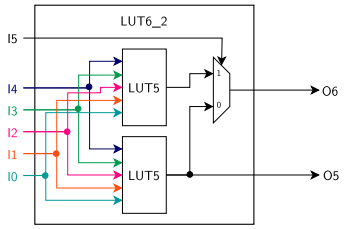


Fig. 7: Internal structure of a LUT6_2, which consists of two LUT5 and a multiplexer [43].

APPENDIX C
2ND-ORDER ISW MULTIPLIER

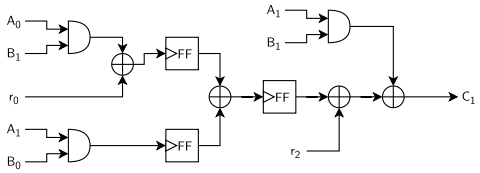


Fig. 8: 2nd-order ISW multiplication gadget [21] represented as an ASIC netlist.

APPENDIX D
SLICEL WITH CASCADE MULTIPLEXERS

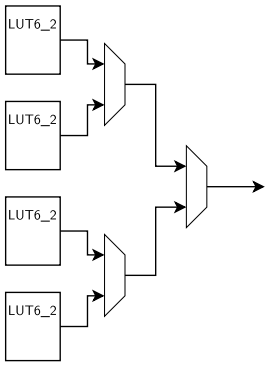


Fig. 9: Simplified sketch of SLICEL connecting four LUT6_2s and cascade multiplexers.

References

[1] Barthe, G., Belaïd, S., Cassiers, G., Fouque, P., Grégoire, B., Standaert, F.: maskverif: Automated verification of higher-order masking in presence of physical defaults. In: Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11735, pp. 300–318. Springer (2019)

[2] Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P., Grégoire, B., Strub, P., Zucchini, R.: Strong non-interference and type-directed higher-order masking. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 116–129. ACM (2016)

[3] Belaïd, S., Mercadier, D., Rivain, M., Taleb, A.R.: Ironmask: Versatile verification of masking security. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022. pp. 142–160. IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833600, https://doi.org/10.1109/SP46214.2022.9833600

[4] Bhasin, S., Carlet, C., Guilley, S.: Theory of masking with codewords in hardware: low-weight *d*th-order correlation-immune boolean functions. IACR Cryptol. ePrint Arch. p. 303 (2013), http://eprint.iacr.org/2013/303

[5] Bloem, R., Groß, H., Iusupov, R., Könighofer, B., Mangard, S., Winter, J.: Formal verification of masked hardware implementations in the presence of glitches. In: Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II. Lecture Notes in Computer Science, vol. 10821, pp. 321–353. Springer (2018)

[6] Cassiers, G., Grégoire, B., Levi, I., Standaert, F.: Hardware private circuits: From trivial composition to full verification. IEEE Trans. Computers **70**(10), 1677–1690 (2021)

[7] Cassiers, G., Standaert, F.: Trivially and efficiently composing masked gadgets with probe isolating non-interference. IEEE Trans. Inf. Forensics Secur. **15**, 2542–2555 (2020)

[8] Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M.J. (ed.) Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1666, pp. 398–412. Springer (1999)

[9] Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: CHES. Lecture Notes in Computer Science, vol. 2523, pp. 13–28. Springer (2002)

[10] Cnudde, T.D., Reparaz, O., Bilgin, B., Nikova, S., Nikov, V., Rijmen, V.: Masking AES with d+1 shares in hardware. In: Gierlichs, B., Poschmann, A.Y. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9813, pp. 194–212. Springer (2016)

[11] Dudha, C.: Retiming in vivado synthesis (2023), https://support.xilinx.com/s/article/934201?language=en_US, https://support.xilinx.com/s/article/934201?language=en_US. Retrieved on 23/05/2023

[12] Faust, S., Grosso, V., Pozo, S.M.D., Paglialonga, C., Standaert, F.: Composable masking schemes in the presence of physical defaults & the robust probing model. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 89–120 (2018)

[13] Gigerl, B., Hadzic, V., Primas, R., Mangard, S., Bloem, R.: Coco: Co-design and co-verification of masked software implementations on cpus. In: Bailey, M., Greenstadt, R. (eds.) 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. pp. 1469–1468. USENIX Association (2021)

[14] Goddard, Z.N., LaJeunesse, N., Eisenbarth, T.: Power analysis of the t-private logic style for fpgas. In: IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015. pp. 68–71. IEEE Computer Society (2015)

[15] Goodwill, G., Jun, B., Jaffe, J., Rohatgi, P.: A testing methodology for side-channel resistance validation. In: NIST Non-Invasive Attack Testing Workshop (2011)

[16] Goubin, L., Patarin, J.: DES and differential power analysis (the "duplication" method). In: Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1717, pp. 158–172. Springer (1999)

[17] Groß, H., Mangard, S., Korak, T.: Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In: Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016. p. 3. ACM (2016)

[18] Groß, H., Schaffenrath, D., Mangard, S.: Higher-order side-channel protected implementations of KECCAK. In: Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017. pp. 205–212. IEEE Computer Society (2017)

[19] Güneysu, T.: Fpgas in cryptography. In: van Tilborg, H.C.A., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, 2nd Ed, pp. 499–501. Springer (2011)

[20] Hadzic, V., Bloem, R.: COCOALMA: A versatile masking verifier. In: Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021. pp. 1–10. IEEE (2021). https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_9, https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_9

[21] Ishai, Y., Sahai, A., Wagner, D.A.: Private circuits: Securing hardware against probing attacks. In: Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2729, pp. 463–481. Springer (2003)

[22] Knichel, D., Sasdrich, P., Moradi, A.: SILVER - statistical independence and leakage verification. In: Moriai, S., Wang, H. (eds.) Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12491, pp. 787–816. Springer (2020)

[23] Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: CRYPTO. Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer (1999)

[24] Li, Y., Tang, M., Li, Y., Zhang, H.: A pre-silicon logic level security verification flow for higher-order masking schemes against glitches on fpgas. Integr. **70**, 60–69 (2020)

[25] Mangard, S., Popp, T., Gammel, B.M.: Side-channel leakage of masked CMOS gates. In: Menezes, A. (ed.) Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3376, pp. 351–365. Springer (2005)

[26] Mangard, S., Pramstaller, N., Oswald, E.: Successfully attacking masked AES hardware implementations. In: Rao, J.R., Sunar, B. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3659, pp. 157–171. Springer (2005)

[27] Masure, L., Méaux, P., Moos, T., Standaert, F.: Effective and efficient masking with low noise using small-mersenne-prime ciphers. In: Hazay, C., Stam, M. (eds.) Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 14007, pp. 596–627. Springer (2023)

[28] Meyer, L.D., Reparaz, O., Bilgin, B.: Multiplicative masking for AES in hardware. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 431–468 (2018)

[29] Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4307, pp. 529–545. Springer (2006)

[30] O'Donnell, R.: Analysis of Boolean Functions. Cambridge University Press (2014)

[31] Pramstaller, N., Mangard, S., Dominikus, S., Wolkerstorfer, J.: Efficient AES implementations on asics and fpgas. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 3373, pp. 98–112. Springer (2004)

[32] Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., Verbauwhede, I.: Consolidating masking schemes. In: Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9215, pp. 764–783. Springer (2015)

[33] Roy, D.B., Bhasin, S., Guilley, S., Danger, J., Mukhopadhyay, D.: From theory to practice of private circuit: A cautionary note. In: 33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015. pp. 296–303. IEEE Computer Society (2015). https://doi.org/10.1109/ICCD.2015.7357117, https://doi.org/10.1109/ICCD.2015.7357117
[34] Sadhukhan, R., Saha, S., Mukhopadhyay, D.: Shortest path to secured hardware: Domain oriented masking with high-level-synthesis. In: 58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021. pp. 223–228. IEEE (2021)
[35] Shahverdi, A., Taha, M., Eisenbarth, T.: Silent simon: A threshold implementation under 100 slices. In: IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015. pp. 1–6. IEEE Computer Society (2015)
[36] Trichina, E.: Combinational logic design for AES subbyte transformation on masked data. IACR Cryptol. ePrint Arch. **2003**, 236 (2003)
[37] Wei, Y., Yao, F., Pasalic, E., Wang, A.: New second-order threshold implementation of AES. IET Inf. Secur. **13**(2), 117–124 (2019)
[38] Wolf, C.: Yosys open synthesis suite. https://yosyshq.net/yosys/
[39] Xiao, G., Massey, J.L.: A spectral characterization of correlation-immune combining functions. IEEE Trans. Inf. Theory **34**(3), 569–571 (1988)
[40] Xilinx: 7 series fpgas configurable logic block user guide (ug474) (2016), https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB, https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB. Retrieved on 21/05/2023
[41] Xilinx: Vivado design suite user guide implementation (ug904) (2016), https://docs.xilinx.com/v/u/2016.2-English/ug904-vivado-implementation, https://docs.xilinx.com/v/u/2016.2-English/ug904-vivado-implementation. Retrieved on 23/05/2023
[42] Xilinx: Ultrafast design methodology guide for fpgas and socs (ug949) (2022), https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology, https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology. Retrieved on 23/05/2023
[43] Xilinx: Vivado design suite 7 series fpga and zynq 7000 soc libraries guide (ug953) (2022), https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries/LUT6_2, https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries/LUT6_2. Retrieved on 23/05/2023