A Look at the Dark Side of Hardware Reverse Engineering – A Case Study

Sebastian Wallat*, Marc Fyrbiak†, Moritz Schlögel†, Christof Paar*†,
*University of Massachusetts Amherst, USA
†Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
swallat@umass.edu, *prename.surname*@rub.de,

***Abstract*—A massive threat to the modern and complex IC production chain is the use of untrusted off-shore foundries which are able to infringe valuable hardware design IP or to inject hardware Trojans causing severe loss of safety and security. Similarly, market dominating SRAM-based FPGAs are vulnerable to both attacks since the crucial gate-level netlist can be retrieved even in field for the majority of deployed device series. In order to perform IP infringement or Trojan injection, reverse engineering (parts of) the hardware design is necessary to understand its internal workings. Even though IP protection and obfuscation techniques exist to hinder both attacks, the security of most techniques is doubtful since realistic capabilities of reverse engineering are often neglected.**

The contribution of our work is twofold: first, we carefully review an IP watermarking scheme tailored to FPGAs and improve its security by using opaque predicates. In addition, we show novel reverse engineering strategies on proposed opaque predicate implementations that again enables to automatically detect and alter watermarks. Second, we demonstrate automatic injection of hardware Trojans specifically tailored for third-party cryptographic IP gate-level netlists. More precisely, we extend our understanding of adversary's capabilities by presenting how block and stream cipher implementations can be surreptitiously weakened.

***Keywords*—Hardware Reverse Engineering, Hardware Trojans, FPGAs, Crypto Trojans, IP Infringement, IP watermarking**

I. Introduction

To assure security and safety of any modern application, it is of crucial importance that the underlying cryptographic primitive is not compromised. Given that most crypto algorithms in use such as the Suite B ciphers [1] are robust against traditional attacks, i.e. brute-force and cryptanalysis, adversaries are often forced to exploit implementation attacks. The most prominent examples of implementation attacks are Side-Channel Analysis (SCA) and Fault Injection (FI). Both attack families have been investigated in great detail in the Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) context over the last two decades, in the scientific community as well as in industry, cf. [2], [3], [4], [5], [6]. Even though SCA and FI countermeasures are certainly not solved problems, there is a sound understanding of attacks and countermeasures.

Another prominent class of attacks attempt to weaken the security by manipulating the underlying hardware, often referred to as Trojans. In spite of extensive research [7], [8], there are still open questions regarding the practicability of many proposed Trojans since either adversarial access to the source code is assumed [9] or crucial reverse engineering steps are neglected [10]. However, in real-world scenarios the adversary (e.g., a malicious foundry) faces daunting tasks of: (1) reverse engineering high-level information from a gate-level netlist, (2) overcoming possible Intellectual Property (IP) protection mechanisms, (3) identifying the security-critical modules, (4) followed by the actual Trojan insertion in the target design. Understanding the feasibility and complexity of these steps is crucial for a sound estimation of the threat posed by hardware manipulations and more importantly, aid with developing sound countermeasures against Trojans.

Goals and Contributions. In this paper, we focus on destructive aspects of reverse engineering. Our goal is to demonstrate the practicability of IP infringement and hardware Trojan injection for third-party gate-level netlists of market-dominating Static Random Access Memory (SRAM)-based FPGAs. To this end, we first review the security of a constraint-based IP watermarking scheme specifically tailored for FPGAs and subsequently we show general improvements to increase the security against reverse engineering. We then demonstrate the automation of hardware Trojan injection in third-party gate-level netlists of cryptographic designs. To highlight devastating consequences of hardware Trojans for both block ciphers and stream ciphers, we selected the standardized and widely-used Present block cipher and A5/1 stream cipher. Our main contributions are:

- **IP Infringement.** In our first case study (Sect. III), we carefully analyze the security of constraint-based watermarking tailored to FPGAs and show how to automatically identify and tamper watermarks. Additionally, we present novel improvements to mitigate reverse engineering by use of opaque predicates. Moreover, we demonstrate flaws in proposed hardware opaque predicate implementations.
- **Hardware Trojan Injection.** In our second case study (Sect. IV and Sect. V), we demonstrate how benign designs of Present and A5/1 can be surreptitiously weakened. In particular, we provide novel aspects on how these cryptographic algorithms can be automatically reverse engineered and custom-tailored key leakage mechanism that allow a man-in-the-middle eavesdropper to decrypt any communication.

II. Technical Background and Related Work

Our work builds on previous research in IP protection, reverse engineering, and hardware Trojans. Below, we present a concise technical background and related work on each area.

A. IP Protection

The design of digital systems involves valuable effort in terms of manpower and money. To reduce both production time and costs, the reuse of IP design components is a common practice. Unfortunately, the protection of the IP owner's rights becomes a major problem [11]. In recent years, various solutions have been proposed to protect valuable IP [12]. Generally, there exists numerous defenses (e.g., watermarking or fingerprinting) targeting different attacks (e.g., overbuilding or cloning).

B. Hardware Reverse Engineering

Hardware reverse engineering can be generally divided in FPGA and ASIC-oriented techniques. In the reverse engineering context of FPGAs existing works can be mainly divided into bitstream reverse engineering to recover the netlist, and netlist reverse engineering to recover high-level Register Transfer Level (RTL) information. Obtaining of the bitstream from a deployed SRAM-based FPGA requires to extracting it from the storing non-volatile memory. While several works developed automated file format reverse engineering techniques to recover the (partial) bitstream, c.f. [13], there also exist works on directly selecting the bitstream as an target for manipulations, c.f. [14], [15]. Although prominent FPGAs provide bitstream encryption to achieve confidentiality, it could be shown that in many cases this encryption can be circumvented [16], [17]. In the case of ASIC netlist reverse engineering several different areas of research can be divided. Where one area focuses on recovering high-level extraction [18], [19], [20], [21], other works elaborate on best practices for a human analyst [22].

C. Hardware Trojans

After the publication of the initial Department of Defense (DoD) report [23] the research community started to investigate both offensive and defensive aspects of malicious hardware manipulations [7], [8].

Defense-driven research focuses on the detection of hardware Trojans using several characteristic properties [24], [25]. Albeit, Zhang *et al.* [26] were able to present a solution to evade detection algorithms targeting the netlist. In contrast to the defense-driven side much less work has been performed on the attack-driven side. The majority of proposed Trojan designs are inserted during the design phase while accessing high-level information [10]. Furthermore there are several works target novel Trojan design methodologies such as dopant-level Trojans [27], analog malicious hardware [28], or parametric Trojans [29].

D. Attacker Model

We suppose that the adversary has access to a flattened gate-level netlist without any high-level information such as names, module hierarchies, and synthesis options. The goal of the adversary is to perform an illegitimate application such as hardware Trojan injection or IP infringement. In consequence the attacker has to at least partially reverse-engineer the design.

Note that our adversary model is realistic in multiple scenarios, i.e. SRAM-based FPGAs (since the netlist can be recovered from a bitstream [30]) and malicious foundries.

III. Case Study: Watermarking

In this case study, we present how IP infringement can be performed for watermark protected designs. To demonstrate the efficacy of gate-level netlist reverse engineering, we analyze a scheme which aims to protect valuable FPGA IP cores at netlist level.

A. LUT-based Watermarking

As described in Sect. II, various works have addressed IP protection by use of watermarks. In particular, constraint-based watermarking [31] is suited for FPGAs since the additional satisfiability constraints are suited for the Look-up table (LUT)-based FPGA structures [32].

Watermarking Scheme [32]. The high-level idea of the scheme by Schmid *et al.* is to exploit not addressable LUT memory space, see Fig. 1. Since input pin I1 is connected to GND, there are 48 Bits that can be arbitrarily changed without altering any functionality of the design. These LUT memory bits are used to embed the watermark.

I5	0	0	0	0		0	0	0			1
I4	0	0	0	0		0	1	1			1
I3	0	0	0	0		1	0	0			1
I2	0	0	0	0	...	1	0	0	...		1
I1	0	0	1	1		1	0	0			1
I0	0	1	0	1		1	0	1			1
O	0	1	1	0		1	X	X			X

Figure 1: Example of a LUT-6. I0 to I5 are input pins, and O is the output pin. I4 and I5 are connected to GND. Bits marked with X can be used for watermarks.

Reverse Engineering. In order to identify the LUT memory bits which implement the watermark, we analyze the Boolean function of each LUT. More precisely, we inspect each clause of a LUT's Disjunctive Normal Form (DNF). If there is any clause where an GND or VCC input signal is required to be logical 1 or 0, respectively, we successfully identified a watermark bit. We practically verified that we are able to automatically disclose the watermark of our target design (an Advanced Encryption Standard (AES) IP core synthesized for an xc6slx16 with Xilinx ISE 14.7). Furthermore, we are able to automatically remove and alter the watermark in the gate-level netlist.

Note that Schmid *et al.* [32] proposed to utilize LUTs configured as shift register LUTs to prevent optimization, but since the *shift-enable* input pin is assigned to GND (to ensure the designs functionality), we treat them as general LUTs in our reverse engineering algorithm.

B. Opaque Predicates

Even though Schmid *et al.* [32] noted that the security can be increased by use of bogus constant generating signals (instead of the plain connection to GND), it appears challenging how such signals can be implemented with consideration of a reasonable reverse engineer.

To this end, we propose to leverage *opaque predicates* which have been mainly used in the context of software watermarking and obfuscation [33]. An opaque predicate is an expression that either evaluates to *true* or *false* irregardless of the input and thus this function implements a constant generating signal. Hence, instead of GND the *unused* LUT's input pins (e.g., I5 and I6 in Fig. 1) are connected to the output of an opaque predicate which mitigates our reverse engineering approach presented in the previous section. Despite numerous works that address opaque predicates for software, there is to the best of our knowledge only one work by Sergeichik *et al.* [34] applying them to the hardware reverse engineering context.

Implementation [34]. To implement opaque predicates, Sergeichik *et al.* suggest to exploit Linear Feedback Shift Registers (LFSRs) as constant signal generators. Their general idea is to connect all registers of the LFSR to an OR or NOR gate to generate a constant high or low signal, respectively. Note that the standard LFSR with XOR feedback never enter the zero state (where all registers hold a logic 0) thus there is at least one register which holds a logic 1.

Reverse Engineering. Even though the reverse engineering strategy from the previous section is generally prevented by opaque predicates, the proposed opaque predicate instantiation by LFSRs is not sufficient to mitigate reverse engineering and thus IP infringement is again possible. The high-level idea of our automated reverse engineering strategy is to identify LFSRs and subsequently whether they are used to implement constant generators. For the detection of LFSRs, we exploit their typical characteristic: a chain of Flip Flops (FFs) elements to store and shift the current state. Note that we skip any pass-through LUTs or buffers in each step. To find the FF chains, we search for the *initial* FF (which stores the next state bit) by checking whether its preceding gate is a FF. For any candidate, we execute a modified Depth-First Search (DFS), and we search for circles considering taps defined by the underlying feedback polynomial of the LFSR. Since we can identify the position of the initial FF and the taps, we are able to algorithmically identify the feedback polynomial as well.

After the LFSRs are automatically reverse engineered, we topologically analyze each LFSR for the constant generator part, i.e. an OR or NOR gate that is connected to all registers of the LFSR. Note that the OR and NOR gates might be implemented across multiple LUTs depending on the type and size of the LFSR. To identify the final constant generation gate, we manually inspected our target design (an AES IP core synthesized for an xc6slx16 with Xilinx ISE 14.7) which typically requires only several minutes but can be easily automated. In case other types of LFSRs such as XNOR-based LFSRs [34], the search for the final constant generation gate has to be adapted.

In summary, LUT-based watermarking is a promising approach to protect valuable FPGA IP and particularly in combination with advanced hardware obfuscation techniques it might provide an adequate security level to hamper reverse engineering.

IV. Case Study: Block Ciphers

This case study presents an approach to detect and weaken block ciphers using Substitution-Boxes (S-Boxes). Here, we extend the technique described in Swierczynski *et al.* [14] by weakening block ciphers normally found in space and power constraint Internet of Things (IoT) devices on the netlist level. By an example we describe the reverse engineering and Trojan injection process for an FPGA based design for an Xilinx Spartan-6 device. Using PRESENT [35] as the targeted design we demonstrate an fully algorithmic solution to detect (Sect. IV-A) and weaken (Sect. IV-B) the S-Boxes of the implementation, even if the S-Box logic is merged with the control logic of a design.

Block Ciphers. In modern systems block ciphers like AES are the standard for symmetrically encrypted communication. In contrast to asymmetric ciphers, many prominent symmetric schemes use a Substitution Permutation Network (SPN) structure which allows a higher data throughput. In combination with asymmetric schemes, which enable the on-demand key generation over an untrusted channel, they allow the encryption of plaintext in blocks of several bytes. In contrast to stream ciphers, where each bit is encrypted independently, block ciphers use a predefined block size to specify the amount of data that is encrypted at once. The SPN structure consists here of alternative substitutions and permutations. The characteristic non-linear nature of each S-Box leads to a specific representation in the synthesized FPGA netlist.

PRESENT cipher. Although AES is the de-facto standard for block-ciphers, due to it size it has some drawbacks in constrained scenarios, as smart-cards, medical, or battery powered IoT devices. One prominent cipher to use in this context is the PRESENT cipher. In contrast to AES, which uses a 8-bit to 8-bit S-Box, it uses a 4-bit to 4-bit S-Box while increasing the number of rounds. For LUT-6 architectures, LUTs with six inputs and (usually) one output, like Spartan-6 the S-Box logic can be potentially merged by the synthesizer. Here two input pins are not used by the S-Box logic and can therefore be freely used by the synthesizer to reduce the space consumption of a design.

In the following we discuss our approach based on Swierczynski *et al.* [14] of detecting S-Boxes in FPGA bitstreams, and extend it to detect merged S-Boxes on the netlist level. Therefore we facilitate an existing PRESENT implementation [36] and utilize Xilinx ISE 14.7 to produce a gate-level, Hardware Description Language (HDL)-based netlist of the implementation. Such a modification can be of interest to extract the secret key from a Device Under Test (DUT), like an Universal Serial Bus (USB) flash drive, discussed in [37].

A. S-Box detection

In contrast to AES, where one LUT-6 is not able to represent the calculation of one S-Box output bit, PRESENT using a 4-to-4 S-Box allows the implementation in one LUT and also allows two bits to be freely used by the synthesizer. Alg. 1 provides an approach to generate the LUT-4 based output pattern, LUTs with four inputs and one output, for the PRESENT cipher. In order to search for these pattern in the LUT-6 based netlist we iterate over all LUT elements and need to extract the sub-configurations by sequentially removing one input bit from the LUT equation. Here we iterate over all combinations of two input pin tuples and generate the sub-pattern, c.f. Alg. 2. In our case we were able to detect 68 S-Box implementation, 64 for each bit of a state and four used by the key-schedule.

Algorithm 1 PRESENT S-Box pattern generation for a LUT-6 FPGA architecture

Input: Present S-Box *sbox*
Input: Number of input bits used for the S-Box l
Output: Vector of all configuration for PRESENT LUTs v

Map of all permutation to the generated configuration vector m
S-Box value at position n *sbox[n]*

```
for i := 0; i < 2^l; ++i do
    for p in Bitpermutations of i do
        m[p].push_back(sbox[p])
return m.values()
```

Algorithm 2 LUT Sub-Configuration Generator for FPGA based designs

Input: Configuration to decompose *orig_conf*
Input: Vector of input pins to sequentially remove from configuration: p
Output: Vector of all generated Sub-Configurations s
Output: Vector of pins in order of removal v

Vector of Configuration: r
Number of Input Pins for Configuration c: $c.input\_pin\_size()$
Access *i'th* Bit of Number n (MSB at position: 0): $n[i]$

```
r.push_back(orig_conf)
while !p.empty() do
    pin := p.take_first()
    v.push_back(pin)
    Intermediate Vector of Configurations r_tmp := r
    for c in r_tmp do
        Configuration Vector μ1, μ2
        l := c.input_pin_size()
        for n = 0; n < 2^l; ++n do
            if n[p] == 0 then
                μ0.push_back(c[n])
            else
                μ1.push_back(c[n])
        r_tmp.push_back(μ0)
        r_tmp.push_back(μ1)
    r := r_tmp
return r, v
```

B. Trojan implementation

In order to weaken the provided design we altered the Boolean function for each pattern to return the identity function (input S-Box = output S-Box), which removes the non-linearity property of PRESENT. Here we again utilized the algorithm Alg. 1 by replacing the standard PRESENT S-Box by the identity S-Box. During the pattern search phase we now collect all sub-configurations for each LUT-6 element and replace the identified sub-pattern by the equivalent identity S-Box configuration based on the detected input pattern. Afterwards the partial equations of the LUT needs to be recombined, c.f. Alg. 3 Finally the resulting LUT-6 configuration is written back to the textual netlist representation.

Algorithm 3 LUT Sub-configuration Merger for FPGA based designs

Input: Vector of Sub-Configurations s
Input: Vector of pins in order of removal v
Output: Merged LUT configuration m

Number of input pins for configuration c: $c.input\_pin\_size()$
Access *i'th* bit of number n (MSB at position: 0): $n[i]$

```
while s.size() > 1 do
    pin := v.take_first()
    Configuration Vector μ1, μ2
    μ0 := s.take_first()
    μ1 := s.take_first()
    Merged configuration: merged
    μ0_counter := 0
    μ1_counter := 0
    l := c.input_pin_size() + 1
    for n = 0; n < 2^l; ++n do
        if n[p] == 0 then
            merged.push_back(μ0[μ0_counter++])
        else
            merged.push_back(μ1[μ1_counter++])
return m = s.take_first()
```

In summary, the presented detection and modification approach allows the modification of any provided S-Box implementation in FPGA architectures. Complementary to Swierczynski *et al.* we demonstrated ways to also handle S-Box logic merged with e.g. control logic. In consequence, this allows the detection and weakening of S-Box based block ciphers for FPGA based designs.

V. Case Study: Stream Ciphers

In this case study, we show how to algorithmically reverse engineer LFSR-based stream ciphers (Sect. V-A) and inject a Trojan into a third-party gate-level netlist based solely on the information inferred from reverse engineering (Sect. V-B).

A. LFSR-based Stream Ciphers

Stream ciphers are an important class of widely used encryption algorithms (e.g., in timing-critical voice transmission), since the encryption consists of a simple XOR-ing with a keystream. In practice, various stream ciphers are based on LFSRs such as E0 in the Bluetooth standard [38], A5/1 in GSM [39], or SNOW 3G in UMTS 3G [40]. LFSRs are advantageous since their implementation is lightweight and are mathematically well understood, see [41]. In addition, non-linear elements such as a non-linear combination of the output from multiple LFSRs are used to increase the security of a stream cipher.

A5/1 Stream Cipher. The A5/1 algorithm was defined for use in GSM networks and is supposed to guarantee data confidentiality for cellphone calls, however, severe flaws have been detected and various attacks proposed [42] [43] [44].

Besides cryptanalytic attacks, it has been shown that A5/1 can be broken in practice within seconds when using a pre-calculated table of ciphertext and plaintext pairs (rainbow tables) [45]. Note that our goal is not to demonstrate a cryptanalytic attack by exploiting inherent weaknesses of the A5/1 cipher, but to exemplify how the security of any LFSR-based stream cipher can be undermined by exploiting the underlying LFSR-based architecture.

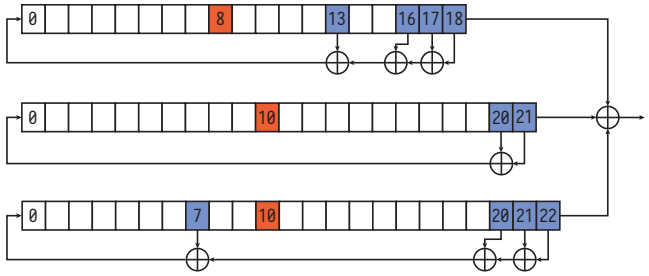

Figure 2: Block diagram of the A5/1 cipher.

A5/1's architectural core consists of three LFSRs, see Fig. 2. The outputs of these LFSRs are XOR-ed and the output in turn is used as bit for the keystream. The LFSRs, L1, L2 and L3, have lengths of 19, 22 and 21 bits, respectively. To achieve an unpredictable output, they are clocked irregularly. To determine which LFSR exactly will be clocked, a so-called majority function is deployed. At positions 8 for L1, 10 for L2 and L3, the bits of the current state are tapped to determine whether 0 or 1 is in the majority. Each LFSR that contains the majority bit in aforementioned position will then be clocked. Before bits for the keystream are generated, a key must be loaded as initial state of the LFSRs. This ensures the output is unpredictable. Besides a 64-bit key, a public 22-bit frame number is loaded to the LFSRs by means of feeding the respective bit into all three LFSRs and subsequently clocking them. This creates an unpredictable initial configuration of the LFSRs. The majority function is then activated and the LFSRs are clocked 100 times without any keystream output being produced [46].

Reverse Engineering. To automatically reverse engineer A5/1 in a gate-level design, we use an extended approach of the LFSR detection presented in Sect. III. In particular, we search for the specified LFSR lengths and the search for the XOR gate that performs the combination of plaintext and keystream. We have practically verified the correctness of our algorithm on an open-source third-party IP core [47] synthesized with Xilinx ISE 14.7 for an xc6slx16. Note that we integrated the encryption/decryption functionality by addition of the XOR with the plaintext/ciphertext to possess a fully-fledged cryptographic core.

B. Hardware Trojan

In order to surreptitiously weaken the cipher, we show how to inject a hardware Trojan into the design's gate-level netlist. The goal of our Trojan is to leak the cryptographic key over the available communication channel.

Trigger. As a result of our previous analysis, we know the exact position of the LFSRs, thus we easily can attach them to our Trojan circuitry. Note that we additionally wire-tap a control signal which indicate whether the A5/1 core is ready to encrypt or decrypt user data. Once this signal is set to high it triggers our Trojan and we load a copy of the current 64-bit state of the LFSRs to our Trojan circuitry.

Payload. The high-level idea of our payload is that the first 64-bit of the ciphertext is the employed 64-bit state of the LFSR. To this end, we added a multiplexer after the final XOR gate, see Fig. 3 and a counter to our Trojan. For the first 64 ciphertext bits, we output the stored keystream and thus a man-in-the-middle can decrypt any user data.

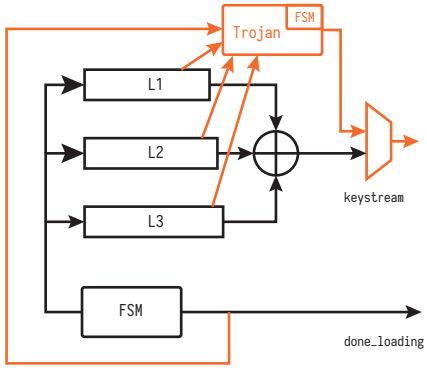

Figure 3: High-level design of the A5/1 Trojan.

One may argue that a user will notice that the encryption failed, as it cannot be decrypted properly. While this is true, we point to the fact that a transmission is organized in bursts of 114 bits. All 4.615 ms ($\sim$ 217 times a second) a burst is sent [39], so that the single unreadable burst will likely be unnoticed.

In summary, automated reverse engineering of LFSR-based stream ciphers and subsequent hardware Trojan injection is not as challenging as one might think. Due the general nature of our detection algorithm, our approach can be used on basically any LFSR-based stream cipher.

VI. Conclusion

Various works have highlighted the threats in modern Integrated Circuits production chain. Diverse threats such as IP infringement and injection of hardware Trojans require (partial) reverse engineering of the design.

In this work, we presented how automated reverse engineering supports arbitrary destructive aspects. First, we provided a reviewed constrained IP watermarking scheme for FPGAs and improved the security by use of opaque predicates. Additionally, we revealed that proposed hardware opaque predicate implementations are not sufficiently secure against reverse engineering. Second, we provided automated reverse engineering strategies to detect cryptographic implementations and simultaneously inject malicious circuitry in third-party gate-level netlists. Our injected hardware Trojan exploits available communication channels, thus a man-in-the-middle is able to decrypt any communication.

Since our attacks are performed automatically, we believe that our work raises the awareness of the real-world attacker's capabilities targeting cryptographic designs and watermarking schemes.

ACKNOWLEDGMENT

This material is based upon work partially supported by NSF award CNS-1563829

REFERENCES

[1] NIST, "Suite B Cryptography," 2001. [Online]. Available: https://www.nsa.gov/ia/programs/suiteb_cryptography/

[2] S. Drimer, "Volatile FPGA design security – a survey (v0.96)," 2008. [Online]. Available: http://www.cl.cam.ac.uk/~sd410/papers/fpga_security.pdf

[3] A. Wild, A. Moradi, and T. Güneysu, "Evaluating the Duplication of Dual-Rail Precharge Logics on FPGAs," in *COSADE*, 2015, pp. 81–94.

[4] P. Sasdrich and T. Güneysu, "Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices," in *ARC*. Springer International Publishing, 2014, pp. 25–36.

[5] F. Schellenberg, M. Finkeldey, B. Richter, M. Schäpers, N. Gerhardt, M. Hofmann, and C. Paar, "On the Complexity Reduction of Laser Fault Injection Campaigns Using OBIC Measurements," in *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Sept 2015, pp. 14–27.

[6] G. T. Becker, M. Kasper, A. Moradi, and C. Paar, "Side-channel based watermarks for integrated circuits," in *HOST*, June 2010, pp. 30–35.

[7] R. Karri, J. Rajendran, and K. Rosenfeld, "Trojan Taxonomy," in *Introduction to Hardware Security and Trust*, M. Tehranipoor and C. Wang, Eds. Springer-Verlag, 2012.

[8] M. Tehranipoor and F. Koushanfar, "A Survey of Hardware Trojan Taxonomy and Detection," *Design Test of Computers, IEEE*, vol. 27, no. 1, pp. 10–25, 2010.

[9] ——, "A Survey of Hardware Trojan Taxonomy and Detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, Jan 2010.

[10] S. T. King et al., "Designing and Implementing Malicious Hardware," in *LEET*, ser. LEET'08. USENIX Association, 2008, pp. 5:1–5:8.

[11] U. Guin *et al.*, "Counterfeit Integrated Circuits: A Rising Threat in the Global Semiconductor Supply Chain," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1207–1228, 2014.

[12] J. Zhang and G. Qu, "A survey on security and trust of FPGA-based systems," in *2014 International Conference on Field-Programmable Technology (FPT)*, Dec 2014, pp. 147–152.

[13] E. Wanderley *et al.*, *Security FPGA Analysis*. Springer, 2011, pp. 7–46.

[14] P. Swierczynski, M. Fyrbiak, P. Koppe, and C. Paar, "FPGA Trojans Through Detecting and Weakening of Cryptographic Primitives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1236–1249, Aug 2015.

[15] P. Swierczynski, G. T. Becker, A. Moradi, and C. Paar, "Bitstream Fault Injections (BiFI) - Automated Fault Attacks against SRAM-based FPGAs," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2017.

[16] A. Moradi *et al.*, "On the Vulnerability of FPGA Bitstream Encryption against Power Analysis Attacks: Extracting Keys from Xilinx Virtex-II FPGAs," in *CCS*, 2011, pp. 111–124.

[17] ——, "Side-channel Attacks on the Bitstream Encryption Mechanism of Altera Stratix II: Facilitating Black-box Analysis Using Software Reverse-engineering," in *FPGA*, 2013, pp. 91–100.

[18] G. H. Chisholm *et al.*, "Understanding Integrated Circuits," *IEEE Design & Test of Computers*, vol. 16, no. 2, pp. 26–37, 1999.

[19] Y. Shi *et al.*, "A highly efficient method for extracting FSMs from flattened gate-level netlist," in *ISCAS*, 2010, pp. 2610–2613.

[20] W. Li *et al.*, "WordRev: Finding word-level structures in a sea of bit-level gates," in *HOST*, 2013, pp. 67–74.

[21] A. Gascón et al., "Template-based circuit understanding," in *FMCAD*, 2014, pp. 83–90.

[22] M. C. Hansen *et al.*, "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[23] "Report of the Defense Science Board Task Force on High Performance Microchip Supply," 2005.

[24] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: identification of stealthy malicious logic using boolean functional analysis," in *ACM CCS*, 2013, pp. 697–708.

[25] J. Zhang et al., "VeriTrust: Verification for Hardware Trust," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 34, no. 7, pp. 1148–1161, 2015.

[26] J. Zhang, F. Yuan, and Q. Xu, "DeTrust: Defeating Hardware Trust Verification with Stealthy Implicitly-Triggered Hardware Trojans," in *ACM CCS*, 2014, pp. 153–166.

[27] G. T. Becker et al., "Stealthy Dopant-Level Hardware Trojans," in *CHES 2013*, 2013, pp. 197–214.

[28] K. Yanet al., "A2: Analog Malicious Hardware," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 18–37.

[29] S Ghandali et al., "A Design Methodology for Stealthy Parametric Trojans and Its Application to Bug Attacks," in *CHES*, 2016, pp. 625–647.

[30] Z. Ding et al., "Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 37, no. 3, pp. 299–312, 2013.

[31] A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Constraint-based watermarking techniques for design IP protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 10, pp. 1236–1252, Oct 2001.

[32] M. Schmid, D. Ziener, and J. Teich, "Netlist-level IP protection by watermarking for LUT-based FPGAs," in *FTP*, 2008, pp. 209–216.

[33] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," *Electronic Commerce Research*, vol. 6, no. 2, pp. 155–171, 2006.

[34] V. Sergeichik and A. Ivaniuk, "Implementation of Opaque Predicates for FPGA Designs Hardware Obfuscation," *Journal of Information, Control and Management Systems*, vol. 12, no. 2, 01/2014.

[35] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," in *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 450–466.

[36] https://opencores.org/project,present.

[37] P. Swierczynski *et al.*, "Interdiction in Practice—Hardware Trojan Against a High-Security USB Flash Drive," *Journal of Cryptographic Engineering*, pp. 1–13, 2016.

[38] S. Bluetooth, "Bluetooth specification version 5," *Available HTTP: http://www. bluetooth. com*, 2016.

[39] J. Quirke, "Security in the GSM system," *AusMobile, May*, pp. 1–26, 2004.

[40] G. Orhanou, S. El Hajji, and Y. Bentaleb, "SNOW 3G stream cipher operation and complexity study," *Contemporary Engineering Sciences-Hikari Ltd*, vol. 3, no. 3, pp. 97–111, 2010.

[41] A. Menezes, P. van Oorschot, and S. Vanstone, "Handbook of Applied Cryptography. CRC, 1996," ISBN 0-8493-8523-7, Tech. Rep.

[42] E. Biham and O. Dunkelman, "Cryptanalysis of the A5/1 GSM stream cipher," in *International Conference on Cryptology in India*. Springer, 2000, pp. 43–51.

[43] A. Biryukov, A. Shamir, and D. Wagner, "Real Time Cryptanalysis of A5/1 on a PC," in *International Workshop on Fast Software Encryption*. Springer, 2000, pp. 1–18.

[44] P. Ekdahl and T. Johansson, "Another attack on A5/1," *IEEE transactions on information theory*, vol. 49, no. 1, pp. 284–289, 2003.

[45] J. Lu, Z. Li, and M. Henricksen, "Time–memory trade-off attack on the GSM A5/1 stream cipher using commodity GPGPU," in *International Conference on Applied Cryptography and Network Security*. Springer, 2015, pp. 350–369.

[46] M. Briceno, I. Goldberg, and D. Wagner, "A pedagogical implementation of a5/1 (1999)," *Dostupné na: http://www. scard. org/gsm/a51. html*, 2011.

[47] https://github.com/chemeris/airprobe-gprs/blob/master/A5.1/Verilog/a5.v.