Journal of Hardware and Systems Security (2023) 7:11–24
https://doi.org/10.1007/s41635-022-00130-y

FPGA Design Deobfuscation by Iterative LUT Modification at Bitstream Level

Michail Moraitis[1] · Elena Dubrova[1]

Received: 18 May 2022 / Accepted: 14 November 2022 / Published online: 16 February 2023

Abstract
Hardware obfuscation is a well-known countermeasure against reverse engineering. For FPGA designs, obfuscation can be implemented with a small overhead by using underutilised logic cells; however, its effectiveness depends on the stealthiness of the added redundancy. In this paper, we show that it is possible to deobfuscate an SRAM FPGA design by ensuring the full controllability of each instantiated look-up table input via iterative bitstream modification. The presented algorithm works directly on bitstream and does not require the possession of a flattened netlist. The feasibility of our approach is verified on the example of an obfuscated SNOW 3G design implemented on a Xilinx 7-series FPGA.

Keywords Obfuscation · Hardware opaque predicate · SRAM FPGA · Bitstream modification · Reverse engineering

1 Introduction

Our world is being transformed by the fourth industrial revolution which is marked by the rapid development and integration of life-changing technologies such as cloud computing, artificial intelligence (AI), and the internet of things (IoT). These technologies have an increasing demand for more powerful, low-power, agile, and low-cost devices. This can be offered through hardware acceleration. Two popular candidates for this role are application-specific integrated circuits (ASICs) and static random access memory field-programmable gate arrays (SRAM FPGAs). ASICs have an excellent performance and power consumption profile which can offer a very efficient acceleration. However, they are severely lacking in agility, having a constant configuration and a very slow time to market. Furthermore, they require high engineering effort to design and their cost per chip becomes viable only for large chip orders making them an expensive solution for small companies and startups. On the other hand, SRAM FPGAs offer lower performance and consume more power compared to ASICs but they require lower engineering effort and most importantly hold the advantage of reconfigurability making them a very agile device with low time to market. Therefore, SRAM FPGAs are a very attractive choice for many computationally heavy applications such as cryptographic algorithm implementation and AI acceleration. This growth in popularity, however, gives rise to SRAM FPGA-specific security challenges.

The programming of SRAM FPGAs is performed through a file called bitstream that contains the configuration information describing a given design in a hidden and proprietary format. The bitstream has to be loaded to the device at every power-on due to the volatile nature of SRAM. This fact renders bitstreams particularly vulnerable to threats such as reverse engineering and modification. Reverse engineering can lead to intellectual property theft and facilitate bitstream modification attacks. It has been demonstrated that, with bitstream modification, it is possible to recover the secret key from FPGA implementations of cryptographic algorithms [1–9].

These attacks assume an adversary that has access to the bitstream of a design under attack. According to the design flow stages presented in [10], the adversary can acquire a bitstream during the *bitstream-at-rest* and *bitstream-loading*

This work was supported in part by the research grant 2021-02426 from Vinnova and by the Vinnova Competence Center for Trustworthy Edge Computing Systems and Applications at KTH Royal Institute of Technology.
We would also like to thank the anonymous reviewers whose constructive comments helped us improve the quality of the present manuscript.

✉ Michail Moraitis
micmor@kth.se

Elena Dubrova
dubrova@kth.se

[1] Department of Electrical Engineering, Royal Institute of Technology (KTH), Electrum 229, Stockholm 196 40, Sweden

stages. The adversary is typically assumed to be able to reverse engineer the bitstream to a certain degree and the goal of the attack is to recover and/or manipulate the logic of a given design to meet various ends, e.g. trojan injection, secret key recovery, and intellectual property theft. In the future, when AI algorithms become a natural part of many systems, the extraction of neural network models from FPGA bitstreams through reverse engineering, or the tampering of the neural networks through bitstream modification, can pose a serious threat.

A popular method of defense against bitstream reverse engineering and modification is to conceal the design's functionality using obfuscation techniques. Typically this is accomplished by redundancy addition (e.g. injection of redundant combinational logic). In this paper, we focus on FPGA obfuscation techniques that make use of constant values to change the function implemented in underutilised LUTs without changing their behavior during execution.

Our Contributions:

- We demonstrate that, by assuring the full controllability of each input of each instantiated LUT in a design via iterative LUT modification, we can defeat obfuscation based on constant values and potentially unlock bitstreams locked using combinational logic locking [11].
- Our approach is not impacted by the level of stealthiness of the constant values or the circuit that generates them. Therefore, it can be used to remove obfuscation that uses constants created by hardware opaque predicates *regardless of how stealthy they are*. This is achieved by searching for the LUT inputs that behave as undetectable stuck-at faults during the execution of the algorithm under attack rather than the hardware opaque predicate itself.
- Our method uses bitstream reverse engineering to determine the logic functions implemented by LUTs and the wires connected to the LUT inputs[1]. This is an advantage over methodologies that require a netlist.
- We demonstrate the feasibility of our approach on the example of an obfuscated SNOW 3G design implemented in a Xilinx 7-series FPGA.

Paper Organization: The rest of the paper is organized as follows. Section 2 presents background information on FPGA technology. Section 3 presents an overview of the related literature pertaining to bitstream encryption, design obfuscation, and fault identification. Section 4 gives a high-level overview of the proposed deobfuscation method. Section 5 presents the adversary model along with three attack scenarios. Section 6 presents the formulation of our method into an algorithm. In Section 7, the method is applied to an obfuscated SNOW 3G design to display its feasibility in practice. Section 8 discusses issues related to the presented approach. Finally, Section 9 concludes the paper.

[1] A short, 2-page, version of the paper was presented at [75].

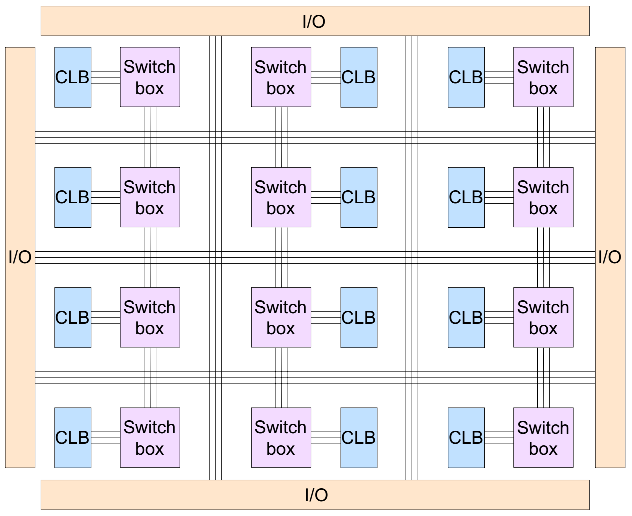


Fig. 1 A typical SRAM FPGA architecture

2 Background on FPGA Technology

This section covers the basics of FPGA technology, with a focus on Xilinx 7 series FPGAs, a popular line of FPGA devices.

2.1 Bitstream Format of FPGA Basic Building Blocks

An FPGA fabric is a mesh of *configurable logic blocks* (CLBs) implementing user-defined logic that is connected through routing channels that pass through programmable switch boxes. By defining both, the functionality of the logic elements and their interconnections, a physical circuit is created on this mesh. In Fig. 1, an abstract view of a typical SRAM FPGA architecture is presented. In this subsection, we describe the basics of logic and routing in Xilinx 7 series FPGAs and their representation in the bitstream.

2.1.1 Look-Up Tables

In SRAM FPGAs, CLBs typically consist of k-input LUTs. In Xilinx 7 series FPGAs, $k = 6$; thus, a LUT can implement a Boolean function of up to 6 variables. Regardless of the actual number of inputs the LUT's function depends on, its truth table appears in the bitstream as a 64-bit vector, called *initialization vector*, partitioned into four 16-bit words which are placed at a fixed distance from each other. Each bit of the initialization vector represents the output value of the LUT for a specific input assignment. If some of the LUT inputs are not used, they are treated as *don't-cares* in the truth table and their value is by default fixed to constant-1.

2.1.2 Programmable Interconnect Points

The routing in FPGAs is performed through *programmable interconnect points* (PIPs). A PIP is a connection between two points called *source* and *destination PIP junction*. Therefore, activating or deactivating a PIP in the bitstream results in creating or removing the connection between the corresponding source and destination PIP junctions. Each FPGA device has a predefined set of PIPs which dictates the possible ways that PIP junctions can be connected to each other. In other words, two PIP junctions cannot be connected unless there is a PIP that allows this connection.

In Xilinx 7 series FPGAs, there are two types of PIPs: PIPs that appear in the bitstream and PIPs that do not. Following the terminology of project Xray [12], we call PIPs that do not appear in the bitstream *fake* and PIPs that do appear *regular*. Our deobfuscation method makes use of three PIP junction types, $PJ1$, $PJ2$, and $PJ3$, which correspond to the immediate connections of LUT inputs[2].

2.2 Architecture of Xilinx 7 Series FPGA

The fabric of Xilinx 7 series FPGAs is a grid of *tiles* uniquely identified by their X and Y coordinates. There are several different types of tiles but in this paper, we are concerned with the two most basic ones, the *interconnect tiles* (INT tiles) and the *configurable logic block tiles* (CLB tiles).

The INT tiles are responsible for the majority of routing. An INT tile is a large switchbox consisting of a set of PIP junctions. A CLB tile has a small switchbox connected horizontally to an INT tile on one side and to two blocks called *slices* (which constitute the main body of the CLB) on the other. If a CLB tile is on the right side of its corresponding INT tile, then they are both labeled as *right*; otherwise, they are labeled as *left*.

Each slice contains four LUTs, eight flip flops (FFs), a fast carry logic unit, and multiplexers (MUXes) to control the internal routing. The slices are positioned vertically inside a CLB; thus, they are usually referred to as *top* and *bottom* slices. Slices are also categorized as *SliceM* or *SliceL* depending on whether they contain ordinary LUTs (SliceL) or special LUTs that can be also configured into a 32-bit shift register or a distributed LUT-based RAM (SliceM).

[2] The names of PIP junctions are given by us and are not related to the terminology used in [12].

3 Background on Bitstream Encryption, Design Obfuscation, and Fault Identification

This section reviews previous work on bitstream encryption, design obfuscation and fault identification. It starts by presenting attacks against bitstream encryption schemes of several popular FPGA models and proceeds to present obfuscation techniques used to enhance the security of the designs. Finally it presents an overview of fault identification techniques.

3.1 Bitstream Encryption

Acknowledging the importance of securing the bitstream file, commercial FPGA vendors offer the option of secure configuration through proprietary bitstream encryption mechanisms. On an abstract level, these mechanisms work as follows. First, the user has to enable the bitstream encryption feature in the FPGA vendor's design tool and define the encryption key. Typically, the encryption algorithm used is the advanced encryption standard (AES) with a 256-bit key. With this feature enabled, the tool generates an encrypted bitstream. On the FPGA side, there is a dedicated decryption core that uses a key commonly stored in either e-fuses or battery-backed RAMs (BBRAMs) that are embedded in the FPGA device. If the key stored in the FPGA matches the one used to encrypt the bitstream, then the bitstream gets decrypted correctly and configures the FPGA.

Unfortunately, in many cases, such protection schemes have been shown vulnerable to physical attacks. In [13–15], the bitstream encryption key is recovered through side-channel analysis for several different commercial FPGAs. State-of-the-art FPGAs like the Xilinx Ultrascale+ have implemented a key-rolling mechanism to thwart side-channel attacks by limiting the number of blocks that are encrypted/decrypted by the same key [16]. However, the use of key-rolling comes with a performance-security trade-off. In [17], after thorough experimentation, it was found that to be protected against current side-channel attacks, the key-rolling factor has to be set between 20 and 30. This imposes a considerable performance overhead. In [18], contactless optical probing is used to read the decrypted bitstream from the output bus of the dedicated decryption core of a Xilinx 7-series FPGA. In [19], the decrypted bitstream is obtained using the FPGA itself as a decryption oracle by exploiting a design flaw of Xilinx 7-series FPGAs. In principle, this vulnerability does not affect the Xilinx Ultrascale+ FPGAs given that the vendor-recommended settings are used. However, in [20], it is shown that when settings outside of the recommendations are used, these devices can display security weaknesses that can be exploited to re-enable the attack vector presented in [19]. Finally, in [21], thermal laser stimulation is used to recover the key stored in a BBRAM of a Xilinx Kintex Ultrascale FPGA while it is powered off.

To summarise, the security of bitstream encryption is an open topic as FPGA vendors constantly strive to make their implementations more secure. Currently, no attack has been shown to be entirely successful against the bitstream encryption mechanism of Xilinx Ultrascale+ FPGAs. However, this series of FPGAs is very new (making its debut a bit more than half a decade ago). One of the main reasons to use an FPGA is its reconfigurability, which gives them a very long life cycle; thus, replacing FPGAs with the latest available models is typically not practiced. This offers a large attack surface of older FPGAs in which the bitstream encryption has been shown to be vulnerable.

3.2 Obfuscation

Since for many FPGA models the current implementations of bitstream encryption cannot effectively protect a design, additional protection mechanisms have to be applied. A popular countermeasure against SRAM FPGA bitstream reverse engineering and modification is design obfuscation.

3.2.1 ASIC

Obfuscation attempts to transform a design into a functionally equivalent, but structurally different, representation which is more difficult to understand. For ASICs, there are well-studied obfuscation techniques such as gate camouflaging (low-level obfuscation) [22–24], combinational logic locking [25–28], and sequential logic locking [29–33]. *Gate camouflaging* makes it hard to recover the functionality of the logic blocks in a circuit while *logic locking* makes it hard to understand the functionality of the whole netlist. Logic locking is one of the most popular approaches for protecting intellectual property and is based on embedding a secret key that needs to be supplied for the design to function correctly. In combinational logic locking, this is achieved by injecting redundant logic controlled by key bits in the design which introduces faults in the case that the key bits are not set correctly. In sequential logic locking, the finite state machines (FSMs) in a design are given extra states from which the FSM cannot escape and move to the original states unless a correct key is supplied. A comprehensive overview of logic locking techniques can be found in [34].

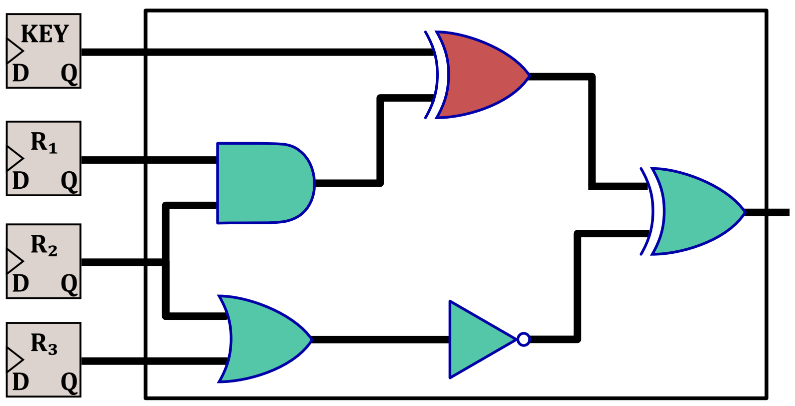
3.2.2 FPGA

Transferring ASIC obfuscation methods to FPGAs requires adaptation to the unique characteristics of the FPGA technology. In FPGAs, logic is implemented by look-up tables (LUTs) with a predefined number of inputs (typically between four and six) and outputs (typically one or two). When a gate-level netlist is translated into LUTs, many of them use fewer inputs than there are available. This affects the way gate camouflaging and combinational logic locking is implemented.

In [11, 35–37], combinational logic locking schemes dedicated to FPGAs are presented. The basic idea is to insert key bits to the unused inputs of already instantiated LUTs and define the locked logic that corresponds to the wrong values of the key bits by modifying the LUT's truth table. Following the terminology introduced in [11], we refer to this unused portion of instantiated LUTs as *FPGA dark silicon*. In the same paper, the term *occupancy* is defined as the percentage of the LUT inputs that are actually used in the instantiated LUTs of a design. The authors reported an average of 30% occupancy while studying nine benchmark designs which indicates that finding such LUTs is very common. Even in the case where the occupancy is high, by splitting large LUTs into smaller ones, we can create FPGA dark silicon. Therefore, finding unused LUT inputs to embed the key for logic locking is typically not a problem.

Since LUTs are the basic logic elements (gates) in FPGAs, the aforementioned combinational logic locking methods, given a correct key, also function as gate camouflaging. That is because the truth table of the LUT is changed, but its actual functionality remains the same. This aspect of FPGA dark silicon modification is highlighted in [5] where the truth table entries corresponding to unused LUT inputs are modified. Such type of camouflaging is effective against an adversary capable of reverse engineering the bitstream format of LUT truth tables, but not the routing that would reveal which inputs of the LUT are used.

In Fig. 2, an example of how the FPGA dark silicon can be leveraged to obfuscate the logic functions of LUTs is presented. In the example, a LUT implements a function with three inputs, R_1, R_2, and R_3, the truth table of which is shown in blue background. Assuming that the LUT has four inputs, the initialization vector of this LUT would be 16 bits long with the output values in the blue background appearing twice. To obfuscate the logic of the LUT (or lock with logic locking), the fourth unused input of the LUT is connected to a key value that is constantly zero. The new input is used to define redundant logic (the red XOR gate) that changes the overall function described in the LUT initialization vector by defining the values on the red background. The red output values cannot appear but this is not known to the adversary. Alternatively, instead of activating the fourth LUT input,



KEY	R_1	R_2	R_3	O
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Fig. 2 Example of FPGA Dark Silicon-Based LUT Obfuscation

assuming that unconnected inputs are constant-1, the bits of the initialization vector corresponding to combinations where the value of this input is zero can be modified as proposed in [5]. However, as explained earlier, detecting such constants is easy with adequate bitstream format knowledge.

From the testing perspective, the utilization of unused LUT inputs is equivalent to the injection of *undetectable stuck-at faults*, which do not cause incorrect output values for any input assignment during the execution of the protected algorithm. The stuck-at faults can be in the form of the correct key values in the case of logic locking, or predefined constants, e.g. the default value on an unused input pin, or the output of a combinational logic circuit with redundancy like $x + \bar{x}$ for constant-1 or $x \cdot \bar{x}$ for constant-0 in the case of logic obfuscation. However, these combinational methods of stuck-at fault injections are vulnerable to static analysis (given an adversary with adequate reverse engineering capabilities). To make the identification of these faults harder, *hardware opaque predicates* can be used for constant value generation.

An opaque predicate is a concept widely used in software obfuscation and in principle is a function that provides a constant Boolean output regardless of its inputs. The output is known to the designer but not to the user/adversary. The first implementation of a hardware opaque predicate, proposed in [38], is an n-stage linear feedback shift register (LFSR) with all state registers connected to an n-input OR gate. Given that an LFSR state always has a hamming weight (HW) greater than zero[3] the output of the OR gate is constant-1. The weakness of this design is that an LFSR has a distinct structure and if identified (e.g. by reverse engineering as in [8]), the constant output of the OR gate can be deduced.

In [39], hardware opaque predicates based on FSMs and counters are presented. Since FSMs and counters are common structures, distinguishing hardware opaque predicates from functional elements is a difficult task. Furthermore, the authors of [39] have demonstrated that, in some cases, even the existing FSMs in a design can be used to implement opaque predicates, making them (and their produced stuck-at outputs) even harder to detect. To the best of our knowledge, no methods for identifying such constructions are known at present, especially if a netlist is not available.

Finally, another type of redundancy that can be used for obfuscation is *functional duplication* which occurs when different sub-circuits implement the same function.

3.3 Fault Identification

In combinational circuits, undetectable stuck-at faults can be identified using automatic test pattern generation (ATPG), Boolean satisfiability problem (SAT) solvers, and fault-independent methods [40–42]. The ATPG and SAT algorithms [43] can guarantee the detection of all undetectable stuck-at faults, but their worst-case time complexity is exponential. Fault-independent methods cannot always find all undetectable faults, but they have the advantage of polynomial worst-case time complexity.

SAT-based attacks against logic locking in particular have drawn a lot of attention, with many methodologies proposed to counter them and equally as many to enhance them [44].

[3] An LFSR cannot have the all-zero state since that would make it unable to transition to another state.

Regarding functional duplication in combinational circuits, it can be identified using SAT [45], BDD sweeping [46], and structural hashing [47]. Both SAT and BDD sweeping guarantee the detection of all functional duplicates, but they have an exponential worst-case time complexity. Structural hashing can identify structurally isomorphic equivalent sub-circuits in linear time. For this reason, obfuscation methods using functional duplication typically implement duplicated blocks in a diverse manner [35].

4 Proposed Method

In this section, we give an overview of the proposed deobfuscation method.

In the presented deobfuscation scheme, the goal is to find which LUT inputs are connected to a net that is or behaves as a constant during the execution of the implemented algorithm. To do that we iteratively set the inputs of LUTs to constant values, upload the modified design instances to an FPGA, and observe the output. Our methodology is based on the observation that the constant values used in logic camouflaging and the key bits of logic locking essentially behave as stuck-at-faults. These faults are undetectable since they do not influence the output of the circuit for any input assignment[4]. This occurs because the intended functionality is the one enabled by the fault since the logic that depends on stuck inputs is injected for the purpose of obfuscation and is not part of the original circuit. Therefore, when the correct stuck-at fault is applied to a redundant LUT input, there is no deviation from the expected output. Having identified the stuck LUT inputs and their values removes the obfuscation since it allows the reconstruction of the original LUT truth table. As a result, an adversary can find the LUT implementation of targeted functions (as in the case of attacks on cryptographic algorithms mentioned in Section 1) or remove logic locking by either updating the LUT truth table to express the original function or manipulating the PIPs to connect the key inputs to equivalent constant values.

5 Attacking Obfuscated Designs

In this section, we present the adversary model and three scenarios of attacks against obfuscated and combinational logic locked designs that can be enabled with the application of the proposed method.

[4] In the case of logic locking we assume that the correct key is provided.

5.1 Adversary Model

The assumed adversary model has the following requirements.

FPGA Access The adversary has access to an FPGA device compatible with the bitstream under attack.

The proposed technique involves loading multiple bitstreams and observing the output of a design; thus, a compatible FPGA has to be available. Depending on the attack scenario, the FPGA can be the property of the adversary or the property of a victim.

Bitstream Access The adversary has access to a non-encrypted bitstream of the implementation under attack.

In SRAM FPGAs, the configuration bitstream has to be loaded at every device power-on due to the volatility of SRAM. For that reason, the bitstream is typically stored in an external, non-volatile memory. This puts the bitstream in a vulnerable position since, given physical access to the target FPGA, the contents of the external memory can be read, or the bus that connects the FPGA to the external memory can be wiretapped to retrieve the bitstream while it is loaded to the FPGA.

Another popular way of loading the bitstream is through a microcontroller. Again, having access to the microcontroller can also give access to the bitstream that is stored in its firmware. Furthermore, if the microcontroller is connected to a network, it becomes possible to extract the bitstream remotely [48].

If the bitstream is encrypted, one of the methods mentioned in Section 3.1 can be used to decrypt it. The selection of the method depends on the model of the FPGA under attack and the equipment of the adversary. Each method has different requirements and not all FPGA models are vulnerable to attacks on bitstream encryption as explained in Section 3.1.

Access Level According to the above, physical access to the device under attack is often required as means of acquiring a bitstream, breaking the encryption of an encrypted bitstream, or loading modified versions of the original bitstream.

The assumption of physical access can be realistic in several cases, especially considering that FPGAs are used more and more in unsupervised environments.

Examples of this are FPGAs used as IoT edge devices [49] and the FPGA-as-a-service (FaaS) (e.g. Amazon Web Services (AWS)[50]) setting where cloud-based access to FPGAs is provided. In FaaS, the FPGA provider has unobstructed physical access to FPGAs programmed with designs owned by different clients. As a result, an insider can potentially access the bitstreams of the client designs. Furthermore in such environments, space and resource sharing between devices is a common practice. This allows clients to attack implementations of other clients through side-channel and covert-channel attacks. Several works have presented attacks in multi-tenant settings (multiple users per FPGA) [51, 52] but also single-tenant[5] settings (one user per FPGA) [53–55]. Note that for applying exclusively the method proposed in this paper, there are attack scenarios where physical access to a device under attack is not required (e.g. scenarios 1 and 2 in Section 5.2).

Bitstream Reverse Engineering Capability The adversary can reverse engineer the bitstream format of LUT initialization vectors and the PIPs associated with LUT inputs. This is necessary for making the LUT inputs controllable and recovering the relation between the LUT's physical inputs and the LUT's truth table.

Several works have presented methods for reverse engineering FPGA bitstream formats [56–59]. For the Xilinx 7 series FPGAs (on which our experiments are based), project Xray [12] maintains a database that documents the format of almost every FPGA element. It should be noted that even in the absence of such a database, the bitstream format knowledge required for the application of our method is minimal. This makes it a more viable approach than one that requires a netlist (reverse engineering of every activated PIP and flattened netlist reconstruction).

However, depending on the attack scenario, further bitstream reverse engineering might be required as we explain in the next subsection.

5.2 Attack Scenarios

Summarizing the adversary model, to apply the method proposed in this paper, an adversary needs to have access to an unencrypted bitstream of the design under attack, an FPGA compatible with the bitstream and bitstream reverse engineering skills. The proposed method is typically used to enable other attacks the requirements of which can add to the overall adversary model. What follows is a description of three attack scenarios and their requirements.

• Scenario 1: Bitstream Modification Attack on a Design with Obfuscated Logic

In this scenario, the attacker has general knowledge of the functionality of an implementation (e.g. that runs a specific encryption algorithm) and aims to find and modify some critical functions. The goal of such an attack can be to inject a Trojan, degrade the performance of the design, recover the secret key of a cryptographic implementation (as in the attack in Section 7), etc. Since the design is protected by logic obfuscation, the deobfuscation method presented in this paper can be used as a pre-processing step. In this scenario, the FPGA used by the adversary can be any FPGA of the same model as the device under attack.

• Scenario 2: Unlocking of a Legally Owned Design Locked with Combinational Logic Locking

In this scenario, a design locked to a device owned by the adversary is legally acquired. The goal here is to unlock the design and redistribute it to unlicensed devices. In the assumed combinational logic locking method, a structure (e.g. a physical unclonable function (PUF)[6] or a nonlinear-feedback shift register (NLFSR)) is used to supply the logic locking key to the locked LUTs [11]. The application of the proposed method here will reveal the value of the key bits in the locked LUTs. However, in the case that there are false-positive detections (discussed in Section 8.2), further reverse engineering is required. This process will involve recovering the nets of the detected stuck-at inputs and evaluating if their source is the key-providing circuit.

• Scenario 3: Unlocking a Design Under Attack Locked with Combinational Logic Locking

This scenario is similar to scenario 2 with the additional requirement of having prolonged physical access to the device under attack (that is not owned by the adversary). That is because the locked bitstream can only work on this specific device; therefore, the proposed method has to also be applied on it.

Table 1 summarises the requirements for applying the proposed method for the three scenarios.

Table 1 Summary of the requirements of the presented attack scenarios

Scenario	Reverse engineering	Physical access to the device under attack
1	LUT initialization vector and inputs	Not required
2	Potentially netlist	Not required
3	Potentially netlist	Required

[5] Single-tenant settings are considered more secure and are adopted by providers such as AWS.

[6] A PUF is a circuit whose output depends on the physical properties of the silicon it is implemented on. Thus, the same PUF design gives different responses when it is implemented on different physical components/devices.

6 Deobfuscation Algorithm

In this section, we present our deobfuscation algorithm FindObfuscated(). Its pseudo-code is shown as Algorithm 1.

Algorithm 1 An algorithm for finding obfuscated LUTs.

```
Name: FindObfuscated($\mathcal{B}$)
Input: Bitstream $\mathcal{B}$
Output: Set of deobfuscated LUT candidates
 1: $\mathcal{S} = \emptyset; \mathcal{R} = \emptyset; \mathcal{M} = \emptyset;$
 2: $\mathcal{P}$ = PIP_Extract($\mathcal{B}$);
    /*$\mathcal{P} := ((p_{1,1}, \dots, p_{1,k_1}), \dots, (p_{n,1}, \dots, p_{n,k_n}))$, where $p_{i,j}$ is the PIP associated with the $j$th input of LUT $l_i$, $i \in \{1, \dots, n\}$, $j \in \{1, \dots, k\}$*/
 3: $\mathcal{L}$ = LUT_Extract($\mathcal{B}$);
    /*$\mathcal{L} := ((l_1, c_1), \dots, (l_n, c_n))$, where $c_i$ is the coordinate of LUT $l_i$ in $\mathcal{B}$, $i \in \{1, \dots, n\}$*/
 4: Clean($\mathcal{L}, \mathcal{P}$)
 5: for each $i \in \{1, \dots, n\}$ do
 6:   for each $j \in \{1, \dots, k_i\}$ do
 7:     for each $\alpha \in \{0, 1\}$ do
 8:       $\mathcal{B}^*$ = Replace ($\mathcal{B}, l_i, c_i, j, \alpha$); /* Replaces LUT $l_i$ at coordinate $c_i$ by a LUT $l_i^\alpha$ in which the $j$th input is stuck-at-$\alpha$ */
 9:       Upload $\mathcal{B}^*$ to the FPGA
10:       if $\mathcal{B}^*$ generates the same output as $\mathcal{B}$ then
11:         if $(l_i^{\overline{\alpha}}, c_i, j, \overline{\alpha}) \notin \mathcal{S}$ then
12:           $\mathcal{S} = \mathcal{S} \cup (l_i^\alpha, c_i, j, \alpha)$;
13:         else
14:           $\mathcal{S} = \mathcal{S} - (l_i^{\overline{\alpha}}, c_i, j, \overline{\alpha})$;
15:           $\mathcal{R} = \mathcal{R} \cup (l_i^\alpha, c_i, j)$; /* Reserve list */
16:         end if
17:       end if
18:     end for
19:   end for
20: end for
21: Count the number of occurences of each LUT $l$ in $\mathcal{S}$, $N(l)$
22: for each $l$ in $\mathcal{S}$ such that $N(l) > 1$ do
23:   for each subset $J \subseteq \{j_1, \dots, j_{N(l)}\}$ of size $|J| > 1$ do
24:     $\mathcal{B}^*$ = Replace($\mathcal{B}, l, c, J, \mathrm{A}$); /* A is a set of multiple stuck-at fault values */
25:     Upload $\mathcal{B}^*$ to the FPGA
26:     if $\mathcal{B}^*$ generates the same output as $\mathcal{B}$ then
27:       $\mathcal{M} = \mathcal{M} \cup (l^{\mathrm{A}}, c, J, \mathrm{A})$;
28:     end if
29:   end for
30: end for
31: return $\mathcal{S} \cup \mathcal{M}$
32: return $\mathcal{R}$
```

FindObfuscated() takes as input a bitstream, $\mathcal{B}$, and returns a list of potential deobfuscated LUT candidates (false-positive detections are possible).

First, a list of all active PIPs connected to utilised LUT inputs is extracted from the bitstream. This list is represented by a vector $\mathcal{P} = ((p_{1,1}, \dots, p_{1,k_1}), \dots, (p_{n,1}, \dots, p_{n,k_n}))$, where $p_{i,j}$ is the PIP associated with the jth input of the LUT l_i, for $i \in \{1, \dots, n\}$, $j \in \{1, \dots, k\}$.

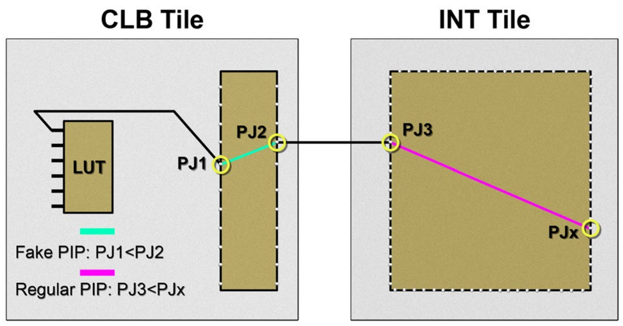


Fig. 3 Visualisation of PIPs connected to LUT inputs

Each input of a LUT is connected to a PIP junction of type *PJ*1 in the CLB's switchbox (see Fig. 3). This PIP junction forms a fake PIP with a PIP junction of type *PJ*2, which in turn is connected to a PIP junction of type *PJ*3 located in the corresponding INT tile switchbox. If the input is not used, then *PJ*3 forms a fake PIP with PIP junction *VCC_WIRE* (constant-1). If the input is used, then *PJ*3 forms a regular PIP with one out of 25 possible PIP junctions in the INT tile switchbox (denoted with *PJx* in Fig. 3). Therefore, if the bitstream contains an activated PIP with destination *PJ*3, it means that the corresponding LUT input is connected somewhere in the design.

Next, a list containing the truth tables of all instantiated LUTs, along with their coordinates in the bitstream, is extracted. This list is represented by a vector $\mathcal{L} = ((l_1, c_1), \dots, (l_n, c_n))$, where c_i is the coordinate of LUT l_i in $\mathcal{B}$, $i \in \{1, \dots, n\}$.

In step 4, the procedure Clean() is called with $\mathcal{P}$ and $\mathcal{L}$ as arguments to remove possible don't-cares in the LUT's function truth table. Obfuscation techniques such as [5] use these don't-cares to camouflage a LUT's truth table without adding any new input to the LUT and so does the watermarking scheme presented in [60]. Since there is a one-to-one mapping between LUT inputs and PIPs involving *PJ*3, the sub-vectors $(p_{i,1}, \dots, p_{i,k_i})$ of $\mathcal{P}$ provide information about k_i input variables on which the function of the LUT l_i actually depends. Leveraging that, Clean updates the truth table of every LUT in $\mathcal{L}$ accordingly. Note that, in a non-obfuscated bitstream, this step would be unnecessary since this is how the vendor tools format LUT truth tables by default.

In steps 5–20, for each LUT $l_i \in \mathcal{L}$ and each of its instantiated inputs $j \in \{1, \dots, k_i\}$, the truth table of l_i in $\mathcal{B}$ is modified to a truth table in which the jth variable is stuck-at-α. This is done by replacing $f|_{x_j=\overline{\alpha}} = f|_{x_j=\alpha}$ where $f|_{x_j=\alpha}$ denotes a subfunction of the function $f(x_1, \dots, x_k)$ of the LUT l_i in which $x_j = \alpha$ and $\overline{\alpha}$ is the Boolean complement (NOT) of α. The modifications are done directly in the bitstream.

The resulting modified bitstream $\mathcal{B}^*$ is uploaded to the FPGA to compare its output sequence to the one of the original bitstream $\mathcal{B}$. If the sequences are the same and l_i with the jth input fixed to $\overline{\alpha}$ is not yet in the list of candidates, $\mathcal{S}$, then l_i is added to $\mathcal{S}$ along with its coordinate c_i, input j, and stuck-at fault value α. If l_i with the jth input fixed to $\overline{\alpha}$ is already in $\mathcal{S}$, it is removed from $\mathcal{S}$ and added to a reserve list $\mathcal{R}$. In this way, the full controllability of each single instantiated LUT input is assured.

Since $k_i \leq 6$ for any $i \in \{1, \dots, n\}$, the computational complexity of steps 5–20 is $O(12n(t_1 + t_2))$, where t_1 is the time to upload $\mathcal{B}^*$ into the FPGA (step 9) and t_2 is the time required to observe the output of $\mathcal{B}^*$ in order to check its equivalence with $\mathcal{B}$ (step 10). Although the worse case complexity of equivalence checking is exponential in the number of primary inputs of the design implemented by $\mathcal{B}$, we found that cryptographic algorithms are quite sensitive to changes. In our SNOW 3G case study, observing 20 output words (640 keystream bits) was enough to get a list that contained all obfuscated LUTs in the design.

In steps 21–30, we repeat the process for multiple stuck-at faults at the instantiated inputs of each LUT in $\mathcal{S}$. First, the number of occurrences of each LUT l in $\mathcal{S}$, $N(l)$, is counted. Since 4-tuples representing the same LUT with different instantiated inputs appear in $\mathcal{S}$ in order, the counting can be performed in $O(|\mathcal{S}|)$ time by recording the number of LUTs with the same coordinate c in $\mathcal{S}$.

Let $\{\alpha_1, \dots, \alpha_{N(l)}\}$ be a set of constants assigned to the inputs $\{j_1, \dots, j_{N(l)}\}$ of a LUT l in $\mathcal{S}$. In steps 22–30, for each l in $\mathcal{S}$ and each subset J of the set $\{j_1, \dots, j_{N(l)}\}$ of size greater than 1, the truth table of l in $\mathcal{B}$ is modified to a truth table in which all inputs in the subset J are stuck-at the corresponding constants in the subset A of the set $\{\alpha_1, \dots, \alpha_{N(l)}\}$. As in the single stuck-at fault case, the modifications are done directly in the bitstream.

The resulting modified bitstream $\mathcal{B}^*$ is uploaded to the FPGA and emulated to compare its output sequence to the one of $\mathcal{B}$. If the sequences are the same, then l is added to the set $\mathcal{M}$ along with its coordinate c, inputs J, and multiple stuck-at fault values A.

Since $N(l) \leq 6$ for any l, we would need to consider $\sum_{i=2}^{6}({}_6C_i) = 57$ faults. Thus, the computational complexity of steps 22–30 is $O(57n(t_1 + t_2))$. It should be noted that when $N(l) = 6$ the output of the LUT is also a constant since all the inputs are constant values. However, this would not apply if one or more stuck input detections were false-positives. Therefore, testing multiple faults for these cases is relevant.

The algorithm terminates by returning the union of $\mathcal{S} \cup \mathcal{M}$.

The reason for creating the reserve list $\mathcal{R}$ is to include cases where logic obfuscation in a LUT is performed through multiple key values that mask single stuck-at faults (e.g. $(key_1 + key_2) \cdot z$, where $(key_1, key_2) = (1, 1)$ are the key values, and z is the obfuscated signal). Note that such a function is unlikely to be used for logic locking since three out of four possible key combinations can be used to unlock the design. If the execution of the presented algorithm does not provide sufficient deobfuscation, list $\mathcal{R}$ can be merged with list $\mathcal{S}$ and analysed as in steps 21–30. Alternatively, steps 11-16 can be replaced by $\mathcal{S} = \mathcal{S} \cup (l_i^{\alpha}, c_i, j, \alpha)$ to include the elements of $\mathcal{R}$ in $\mathcal{S}$ from the beginning[7]. The latter approach can introduce unnecessary overhead since any logic that does not contribute to the output is also included in the elements of $\mathcal{R}$.

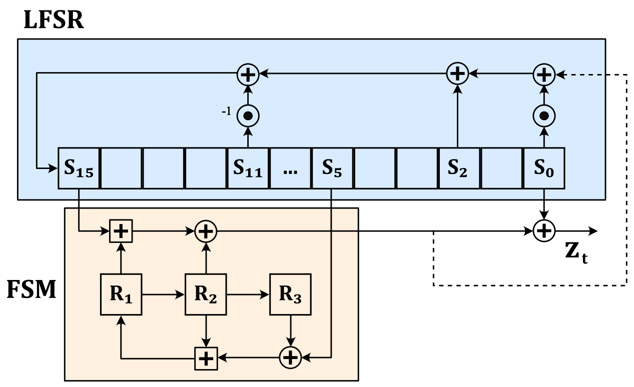


Fig. 4 SNOW 3G Block Diagram

7 Case Study: SNOW 3G Stream Cipher

We demonstrate the feasibility of FindObfuscated() algorithm on the example of SNOW 3G stream cipher obfuscated with constants given by a simple hardware opaque predicate. The design is implemented on a Xilinx 7-series FPGA (XC7A35T-2CPG236) using a VHDL description of SNOW 3G kindly provided by the authors of the stream cipher.

7.1 SNOW 3G Design Description

SNOW 3G is the backbone of the 3GPP confidentiality and integrity algorithms UEA2 and UIA2 [61] in UMTS, 128-EEA1 and 128-EIA1 in LTE [62], GEA5 and GIA5 in Extended Coverage GSM for IoT (EC-GSM-IoT) [63], and 128-NEA1 and 128-NIA1 in 5G New Radio (NR) [64].

SNOW 3G is a word-oriented binary additive stream cipher [65] which takes as input a 128-bit *Initialization Vector* (IV) and a 128-bit secret key, and produces a pseudorandom sequence called *keystream*. Each keystream element is

[7] Step 31 will also be omitted since there will be no list $\mathcal{R}$.

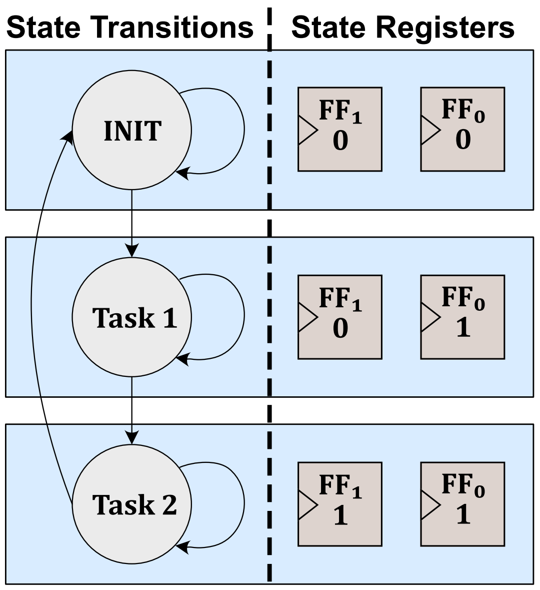


Fig. 5 FSM implementing a simple opaque predicate

a 32-bit word. The encryption/decryption is performed by combining the keystream with plaintext/ciphertext.

Figure 4 shows a block diagram of SNOW 3G. The cipher consists of a 16-stage LFSR and a non-linear FSM. Like most stream ciphers, SNOW 3G has two modes of operation—initialization and keystream generation. In the initialization mode, marked by a dashed line in Fig. 4, the LFSR is loaded with a combination of the key and IV, the FSM is loaded with an all-0 state, and the cipher is clocked for 32 cycles without producing any output. After that, the cipher enters the keystream generation mode, marked by a solid line in Fig. 4, in which one keystream word is generated per clock cycle.

SNOW 3G is resistant to classical cryptanalysis [66–70]; however, physical attacks on its implementations through cache timing side-channels [71], electromagnetic inference analysis [72], transient fault injection [73], and bitstream modification [3] have been reported.

7.2 Obfuscated SNOW 3G Implementation

We implemented a protected version of SNOW 3G in which the part sensitive to fault injections, the FSM output, is obfuscated using constants created by a simple FSM-based hardware opaque predicate shown in Figs. 5 and 6.

The FSM opaque predicate illustrated in Fig. 5 has three states: one initialization state, *INIT* , and two states corresponding to two different tasks, *Task 1* and *Task 2*. The state

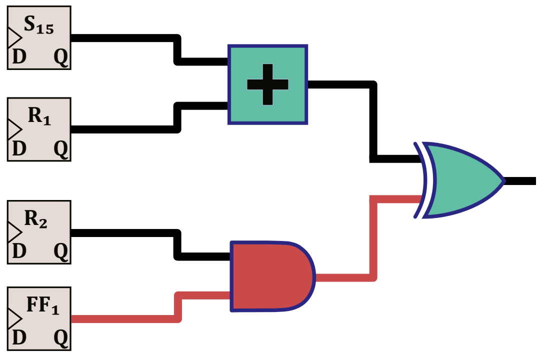


Fig. 6 Obfuscated SNOW 3G FSM output logic

machine stays in the initialization state until the execution of SNOW 3G is initiated. After that, it stays in Task 1 for a period smaller than the computation of the first SNOW 3G FSM output, and then traverses to Task 2, where it stays until the execution of the algorithm is completed. To represent the different states of such an FSM, two registers are used, FF_0 and FF_1. From the values they take in each state (shown in Fig. 5), it is evident that both FF_0 and FF_1 can be used to supply a constant-1 for obfuscation purposes during the evaluation of the SNOW 3G FSM's output.

As shown in Fig. 4, the output function of the SNOW 3G FSM is $(S_{15} \boxplus R_1) \oplus R_2$. To obfuscate this function, we add an AND operation between the SNOW 3G FSM register R_2 and the state register FF_1 of the hardware opaque predicate as shown in Fig. 6. Since FF_1 is constant-1, the injection of the AND causes no deviation from the original functionality.

In [3], it is demonstrated that the injection of a stuck-at-0 fault at the FSM output during the initialization can be exploited to extract the secret key of SNOW 3G. This is because, in this case, the LFSR state after the initialization depends entirely on the characteristic polynomial of the LFSR. Thus, by analysing the keystream, it is possible to reverse the LFSR to its initial state and recover the key-IV combination which is loaded in it. To perform the attack, the LUTs implementing the SNOW 3G FSM output have to be identified and modified. However, the logic of the function is now changed in a way unknown to the potential adversary; thus, the attack fails since locating the relevant LUTs in the bitstream is not possible.

7.3 Deobfuscating SNOW 3G

We developed a software package implementing the FINDOBFUSCATED() algorithm. The package uses the project Xray [12] to reverse engineer the bitstream format, *python* scripts to automate the processing of the PIP and LUT lists

Table 2 Number of LUTs with multiple candidate stuck-at inputs

# of stuck inputs	# LUTs
1	354
2	464
3	178
4	38
5	10
6	0

extracted from the bitstream, and *tcl* scripts to automate the uploading of the bitstreams into the FPGA. We used the package to deobfuscate the protected implementation of SNOW 3G described in the previous subsection. The size of the LUT list $\mathcal{L}$ recovered by reverse engineering in step 3 of FindObfuscated() was $n = 3107$ (the number of LUTs reported by Vivado is $n = 3053$). The size of the PIP list $\mathcal{P}$ recovered by reverse engineering in step 2 was $\sum_{i=1}^{n}(n \cdot k_i) = 12,533$ (compare to $6n = 18,642$). It takes 24 seconds to compute both lists.

The modified bitstreams $\mathcal{B}^*$ created in steps 5–8 were uploaded to the FPGA one by one in step 9. To upload one bitstream into the FPGA, generate 20 keystream words (640 bits) of $\mathcal{B}^*$ and verify the equivalence of the keystreams of $\mathcal{B}^*$ and $\mathcal{B}$ requires $t_1 + t_2 = 6.3$ secs on average.

The number of LUTs that contain candidate stuck at faults is 1044. In Table 2, the distribution of the candidate stuck-at faults in these LUTs is presented. To test multiple stuck-at faults in the algorithm steps 22–30, a total of $\sum_{i=2}^{2}({}_2C_i) \times 464 + \sum_{i=2}^{3}({}_3C_i) \times 178 + \sum_{i=2}^{4}({}_4C_i) \times 38 + \sum_{i=2}^{5}({}_5C_i) \times 10 = 1854$ multiple faults have to be evaluated with one bitstream for each.

The set of deobfuscated LUT candidates returned by FindObfuscated() contained all LUTs implementing SNOW 3G FSM output because redundant inputs of these LUTs behave as undetectable stuck-at faults during the execution of SNOW 3G. Since all points of interest for fault injection are discovered, after deobfuscation it is possible to extract the secret key of SNOW 3G through a bitstream modification attack as in [3].

The bitstreams analysed in our experiments are available at https://github.com/MichailM7/FPGA-Design-Deobfuscation.

8 Discussion

In this section, we discuss the critical factors that affect the runtime of the proposed algorithm, the problem of false-positive detections, and how fault masking can affect the algorithm's performance.

8.1 Runtime

In the experimental results of Section 7, the presented method is evaluated against a simple FSM-based opaque predicate where no considerations about its stealthiness are made. Replacing it with a more sophisticated and stealthy one will not affect the success rate of our algorithm. This is because our method does not search for the hardware opaque predicate itself, but for the LUT inputs that behave as undetectable stuck-at faults during the execution of the implementation under attack. Our approach evaluates exhaustively every used LUT input in a brute-force manner. This guarantees that every LUT input connected to a constant (or to a signal that behaves as constant during execution) will be identified as a candidate by FindObfuscated() regardless of the way the constant is generated.

The runtime of the proposed algorithm depends on the number of LUTs and the degree of LUT occupancy or, in other words, the total number of LUT inputs that are active in the design. Sophisticated opaque predicates have a minimal area overhead which contributes to their stealthiness, for example, the LUT overhead in [39] is reported to be 1–2.2%. As a result, such opaque predicates will cause an equally minimal increase in the runtime of FindObfuscated().

8.2 False-Positive Analysis

Apart from stuck-at LUT inputs, the candidate list returned by FindObfuscated() will include any unobservable single stuck-at fault in the design. In the context of our method, these unobservable stuck-at faults are considered false-positives. However, the identification of an unobservable stuck-at fault requires exhaustive simulation. In our experiments we do not exhaustively test every possible input assignment; instead, we run SNOW 3G with a constant key and observe a limited number of outputs (640 bits). As a result, many of the false-positive detections are not unobservable faults but faults that we either failed to propagate with an appropriate input assignment or faults for which we did not observe a sufficiently long output sequence to detect.

Ruling out the false-positives is a very hard task. A brute-force method leads to exponential complexity since it requires the evaluation of every possible combination of the candidate stuck-at faults. However, identifying false-positives is not always necessary to perform an attack. For bitstream modification attacks of scenario 1 (presented in subsection 5.2), the adversary needs to only identify a critical target function. Since the critical function will appear in the list returned by FindObfuscated(), the goal is completed and the remaining detections are ignored. On the other hand, to defeat logic locking (attack scenarios 2 and 3), all the LUT inputs connected to a key bit have to be modified; and therefore, all false-positives have to be detected. Since a brute-force strategy is infeasible, even for moderately sized designs, the adversary has to further reverse engineer the bitstream and recover a flattened netlist. By analysing the netlist, the candidate stuck-at inputs can be grouped into nets and the logic that generates their values can be traced. Then the adversary will evaluate which of these logic structures can be generating the logic keys and make a series of educated guesses to find the subset of the candidates that are the logic locking keys.

8.3 Fault Masking

The core idea of the presented method is to introduce stuck-at faults and observe their effect. If the effect of the faults cannot be observed in the output then our approach would not work. Therefore, applying fault masking in the obfuscated/locked logic can make the application of our approach significantly harder or even completely prevent it. In this subsection, we discuss how fault masking with redundancy addition can impact our approach and a possible way to work around it.

Fault masking is a fault-tolerant technique that is traditionally used to allow the correct functioning of a circuit in the presence of faults. The most popular fault masking scheme is the triple modular redundancy (TMR) [74]. In TMR, a critical module is triplicated and the outputs of the three modules are given to a majority voter. The voter gives the correct output as long as at least two of the modules are operating correctly. Therefore, TMR offers tolerance to any number of faults as long as they are concentrated on one module. However, this comes at the expense of a considerable hardware overhead. In [35], TMR is used as part of a logic locking scheme.

In our analysis, we assume a design that has critical logic functions obfuscated with sufficiently stealthy constant values (attack scenario 1) and is also protected with TMR. Each of the three modules is obfuscated in a different way to avoid detection (functional duplication). The majority voter unit is also obfuscated. The goal of the adversary is to remove the obfuscation from the critical function.

The presented approach relies on observing differences in the output after the injection of single stuck-at faults, something that TMR completely prevents. To be able to propagate a fault, the same fault needs to be injected in two modules at the same time. Without any knowledge about the location of the TMR modules, we would need to test ${}_{length(\mathcal{P})}C_2$ pairs of fault injections, where $\mathcal{P}$ is the list of all utilised inputs of all utilised LUTs in a design as defined in Section 6. Applying that on the SNOW3G implementation of Section 7 would require the testing of ${}_{12{,}533}C_2 \times 2^2 = 314{,}127{,}112$ faults which is infeasible.

To work around that, prior to applying our method, a bitstream modification attack targeting the TMR voter is required. Even though the voter unit is obfuscated, it is not protected with fault masking; thus, applying our method will deobfuscate it. After that, by considering possible voter implementations, mapping them into LUTs and searching for them in the deobfuscated LUT initialization vectors returned from our algorithm, the identification of the voter becomes possible. After the voter is identified, its logic can be easily modified to constantly output the response of one of the TMR modules. This modification removes the TMR since the remaining two modules get disconnected from the output and single stuck-at faults in the remaining module can propagate to the output. Therefore, with the proposed workaround, the adversary needs to execute our algorithm twice and perform a bitstream modification attack in between. Apart from doubling the runtime, this makes the application of our method much harder since the step of identifying the voter circuit requires further reverse engineering and a skilled attacker.

9 Conclusion

We proposed a new method for FPGA design deobfuscation based on ensuring the full controllability of each instantiated LUT input in a design via iterative LUT modification at bitstream level. We implemented the presented method in a software package and demonstrated its feasibility on the example of a SNOW 3G stream cipher FPGA implementation.

By providing a novel methodology for testing the resistance of obfuscation strategies, our findings are expected to contribute to the assurance of FPGA design security.

Funding Open access funding provided by Royal Institute of Technology.

Data Availability https://github.com/MichailM7/FPGA-Design-Deobfuscation.

References

1. Aldaya AC et al ([2015] 2016) AES T-Box tampering attack. J Cryptogr Eng 6(1):31–48
2. Ender M et al (2019) Insights into the mind of a trojan designer: the challenge to integrate a trojan into the bitstream. In: Proc of the 24th Asia and South Pacific Design Automation Conf pp. 112–119
3. Moraitis M, Dubrova E (2020) Bitstream modification attack on snow 3g. In: Design, Automation & Test in Europe conf. & Exhibition (DATE) pp. 1275–1278. IEEE
4. Moraitis M et al (2020) Breaking ACORN at Bitstream Level. In: IFIP/IEEE 28th Int Conf Very Large Scale Integration (VLSI-SOC) pp. 117–122. IEEE
5. Ngo K et al (2020) Attacking Trivium at the Bitstream Level. In: 38th Int Conf Comp Des (ICCD) pp. 640–647. IEEE
6. Swierczynski P et al ([2016] 2017) Interdiction in practice–Hardware Trojan against a high-Sec. USB flash drive. J Cryptogr Eng 7(3):199–211
7. Swierczynski P et al (2017) Bitstream fault injections (BiFI)–Automated fault attacks against SRAM-based FPGAs. IEEE Trans Comp 67(3):348–360
8. Wallat S et al (2017) A look at the dark side of hardware reverse engineering-a case study. In: 2nd Int Verification Sec Workshop (IVSW) pp. 95–100. IEEE
9. Ziener D et al (2018) Configuration tampering of BRAM-based AES implementations on FPGAs. In: Int Conf Reconfig Comput FPGAs pp. 1–7. IEEE
10. Duncan A et al (2019) FPGA bitstream Security: a day in the life. In: Int Test Conf (ITC) pp. 1–10. IEEE
11. Karam R etal (2016) Robust bitstream protection in FPGA-based systems through low-overhead obfuscation. In: 2016 Int Conf on ReConFigurable Computing and FPGAs (ReConFig) pp. 1–8. IEEE
12. SymbiFlow: Project X-Ray (2018) https://prjxray.readthedocs.io/en/latest/
13. Moradi A, Schneider T (2016) Improved side-channel analysis attacks on xilinx bitstream encryption of 5, 6, and 7 series. In: Int Workshop on Constructive Side-Channel Analysis and Secure Design pp. 71–87. Springer
14. Moradi A et al (2011) On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs. In: Proc 18th ACM Conf Comp Commun Sec pp. 111–124
15. Moradi A et al (2013) Side-channel attacks on the bitstream encryption mechanism of Altera Stratix II: facilitating black-box analysis using software reverse-Engineering. In: Proc ACM/SIGDA Int Symp Field Prog Gate Arrays pp. 91–100
16. Xilinx (2022) Using Encryption and Authentication to Secure an UltraScale/UltraScale+ FPGA Bitstream (XAPP1267)
17. Hettwer B et al (2021) Side-Channel Analysis of the Xilinx Zynq UltraScale+ Encryption Engine. IACR Trans Crypto Hardw Embed Syst 2021(1):279–304
18. Tajik S et al (2017) On the power of optical contactless probing: Attacking bitstream encryption of FPGAs. In: Proc 2017 ACM SIGSAC Conf Comp Commun Sec pp. 1661–1674
19. Ender M et al (2020) The unpatchable silicon: A full break of the bitstream encryption of xilinx 7-series fpgas. In: 29th USENIX Sec Symp pp. 1803–1819
20. Ender M et al (2022) A cautionary note on protecting xilinx' ultrascale(+) bitstream encryption and authentication engine. In: 30th Int Symp Field-Prog Custom Computing Machines (FCCM) pp. 1–9
21. Lohrke H et al (2018) Key extraction using thermal laser stimulation. IACR Trans Crypto Hardw Embed Syst pp. 573–595
22. Cocchi RP et al (2014) Circuit camouflage integration for hardware ip protection. In: 2014 51st ACM/EDAC/IEEE Design Automation Conf (DAC) pp. 1–5. IEEE
23. Erbagci B et al (2016) A secure camouflaged threshold voltage defined logic family. In: Int Symp Hardware Oriented Sec trust (HOST) pp. 229–235. IEEE
24. Rajendran J et al (2013) Security analysis of integrated circuit camouflaging. In: Proc ACM SIGSAC Conf Comp Commun Sec pp. 709–720
25. Dupuis S et al (2014) A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans. In: 20th Int On-Line Testing Symp (IOLTS) pp. 49–54. IEEE
26. Rajendran J et al (2012) Logic encryption: A fault analysis perspective. In: Design Automation & Test in Europe Conf & Exhibition (DATE) pp. 953–958. IEEE
27. Rajendran J et al (2013) Fault analysis-based logic encryption. IEEE Trans Comp 64(2):410–424
28. Roy JA et al (2008) EPIC: Ending Piracy of Integrated Circuits. In: Design Automation and Test in Europe pp. 1069–1074
29. Alkabani Y, Koushanfar F (2007) Active hardware metering for intellectual property protection and security. In: USENIX Sec Symp pp. 291–306
30. Chakraborty RS, Bhunia S (2009) Harpoon: An obfuscation-based soc design methodology for hardware protection. IEEE Trans CAD Int Circ Syst 28(10):1493–1502
31. Desai AR et al (2013) Interlocking obfuscation for anti-tamper hardware. In: Proc of the 8th cyber Sec and information intelligence research workshop pp. 1–4
32. Dofe J, Yu Q (2017) Novel dynamic state-deflection method for gate-level design obfuscation. IEEE Trans CAD Int Circ Syst 37(2):273–285
33. Meade T et al (2017) Revisit sequential logic obfuscation: Attacks and defenses. In: 2017 IEEE Int Symp Circ Syst (ISCAS) pp. 1–4. IEEE
34. Chakraborty A et al (2019) Keynote: A disquisition on logic locking. IEEE Trans CAD Int Circ Syst 39(10):1952–1972
35. Hoque T et al (2019) Hidden in plaintext: an obfuscation-based countermeasure against FPGA bitstream tampering attacks. ACM Trans Des Autom Electron Syst (TODAES) 25(1):1–32
36. Kamali HM et al (2018) Lut-lock: A novel LUT-based logic obfuscation for FPGA-bitstream and ASIC-hardware protection. In: Comp Soc Ann Symp VLSI (ISVLSI) pp. 405–410. IEEE
37. Olney B, Karam R (2020) Tunable FPGA bitstream obfuscation with boolean satisfiability attack countermeasure. ACM Trans Des Autom Electron Syst (TODAES) 25(2):1–22
38. Sergeichikand V, Ivaniuk A (2014) Implementation of opaque predicates for FPGA designs hardware obfuscation. J Info Control Manag Sys 12(2)
39. Hoffmann M, Paar C (2018) Stealthy opaque predicates in hardware-obfuscating constant expressions at negligible overhead. IACR Trans Crypto Hardw Embed Syst pp. 277–297
40. Harihara M, Menon P (1989) Identification of undetectable faults in combinational circuits. In: Proc 1989 IEEE Int Conf Comp Design: VLSI in Comp Processors pp. 290–291. IEEE Comp Society
41. Iyer MA, Abramovici M (1996) Fire: A fault-independent combinational redundancy identification algorithm. IEEE Trans VLSI Syst 4(2):295–301
42. Menon PR, Ahuja H (1992) Redundancy removal and simplification of combinational circuits. In: Digest Papers VLSI Test Symp pp. 268–273. IEEE
43. Biere A, Kunz W (2002) Sat and atpg: Boolean engines for formal hardware verification. In: Proceedings of the 2002 IEEE/ACM international conference on Computer-Aided Design pp. 782–785
44. Kamali HM, Azar KZ, Farahmandi F, Tehranipoor M (2022) Advances in logic locking: Past, present, and prospects. Cryptology ePrint Archive

45. Kim J et al (1997) RID-GRASP: Redundancy identification and removal using GRASP. In: Int. Workshop on Logic Synthesis. Citeseer
46. Kuehlmann A et al (2002) Robust boolean reasoning for equivalence checking and functional property verification. IEEE Trans CAD Int Circ Syst 21(12):1377–1394
47. Kuehlmann A, Krohm F (1997) Equivalence checking using cuts and heaps. In: Proc of the 34th annual Design Automation Conf pp. 263–268
48. Kataria J et al (2019) Defeating cisco trust anchor: A {Case-Study} of recent advancements in direct {FPGA} bitstream manipulation. In: 13th USENIX Workshop on Offensive Technologies (WOOT 19)
49. Biookaghazadeh S et al (2018) Are FPGAs suitable for edge computing? In: USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)
50. AWS: Amazon EC2 F1 instances (2017) https://aws.amazon.com/ec2/instance-types/f1/
51. Giechaskiel I et al (2019) Reading between the dies: Cross-SLR covert channels on multi-tenant cloud FPGAs. In: 37th Int Conf Comp Design (ICCD) pp. 1–10. IEEE
52. Glamočanin O et al (2020) Are cloud fpgas really vulnerable to power analysis attacks? In: 2020 Design, Automation & Test in Europe Conf & Exhibition (DATE) pp. 1007–1010. IEEE
53. Giechaskiel I et al (2020) C^{3}APSULe: Cross-FPGA Covert-Channel Attacks through Power Supply Unit Leakage. In: Symp Sec Privacy (SP) pp. 1728–1741. IEEE
54. Giechaskiel I et al (2022) Cross-VM Covert- and Side-Channel Attacks in Cloud FPGAs. ACM Trans Reconfig Technol Syst
55. Tian S, Szefer J (2019) Temporal thermal covert channels in cloud FPGAs. In: Proc ACM/SIGDA Int Symp. Field-Prog Gate Arrays pp. 298–303
56. Benz F et al (2012) Bil: A tool-chain for bitstream reverse-eng. In: 22nd Int Conf Field Prog Logic App pp. 735–738
57. Ding Z et al (2013) Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation. Microprocess Microsyst 37(3):299–312
58. Note JB, Rannaud É (2008) From the bitstream to the netlist. In: FPGA 8:264–264
59. Ziener D et al (2006) Identifying FPGA IP-cores based on lookup table content analysis. In: Int Conf Field Prog Logic App pp. 1–6. IEEE
60. Schmid M et al (2008) Netlist-level IP protection by watermarking for LUT-based FPGAs. In: Int Conf Field-Prog Tech pp. 209–216
61. 3GPP (2009) Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2. https://www.gsma.com/aboutus/wp-content/uploads/2014/12/uea2designevaluation.pdf
62. 3GPP (2018) 3GPP TS 33.401 version 14.5.0 Release 14. https://www.etsi.org/deliver/etsi_ts/133400_133499/133401/14.05.00_60/ts_133401v140500p.pdf
63. 3GPP (2017) 3GPP TS 43.020 version 14.3.0 Release 14. https://www.etsi.org/deliver/etsi_ts/143000_143099/143020/14.03.00_60/ts_143020v140300p.pdf
64. 3GPP (2022) Sec. architecture and procedures for 5G System. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3169
65. Robshaw M (1994) Stream ciphers. Tech Rep TR - 701. citeseer.ist.psu.edu/robshaw95stream.html
66. Biryukov A et al (2010) Analysis of SNOW 3G resynchronization mechanism. In: SECRYPT, pp. 327–333
67. Biryukov A et al (2010) Multiset collision attacks on reduced-round SNOW 3G and SNOW 3G$^{\oplus}$. In: Int Conf Appl Crypt Netw Sec pp. 139–153. Springer
68. Guan J et al (2013) Guess and Determine Attack on SNOW3G and ZUC. J Softw 6:1324–1333
69. Kircanski A, Youssef AM (2011) On the sliding property of SNOW 3G and SNOW 2.0. IET Info Sec 5(4):199
70. Nia MSN etal (2014) Improved Heuristic guess and determine attack on SNOW 3G stream cipher. In: 7'th Int Symp Telecom (IST'2014) pp. 972–976. IEEE
71. Brumley BB et al (2010) Consecutive S-box lookups: A Timing Attack on SNOW 3G. In: Int Conf Inf Commun Sec pp. 171–185. Springer
72. Takahashi J et al (2012) Feasibility of fault analysis based on intentional electromagnetic interference. In: Int Symp Electromag Compatibility pp. 782–787. IEEE
73. Debraize B et al (2009) Fault analysis of the stream cipher SNOW 3G. In: 2009 Workshop on Fault Diagnosis and Tolerance in Crypto (FDTC) pp. 103–110. IEEE
74. Dubrova E (2013) Fault-tolerant design. Springer
75. Moraitis M, Dubrova E (2022) FPGA Design Deobfuscation by Iterative LUT Modifications at Bitstream Level. In: 27th IEEE European Test Symp (ETS)