Bitstream Fault Injections (BiFI) – Automated Fault Attacks against SRAM-based FPGAs

Pawel Swierczynski, Georg T. Becker, Amir Moradi, Christof Paar, *Fellow, IEEE*

Abstract—This contribution is concerned with the question whether an adversary can automatically manipulate an unknown FPGA bitstream realizing a cryptographic primitive such that the underlying secret key is revealed. In general, if an attacker has full knowledge about the bitstream structure and can make changes to the target FPGA design, she can alter the bitstream leading to key recovery. However, this requires challenging reverse-engineering steps in practice. We argue that this is a major reason why bitstream fault injection attacks have been largely neglected in the past. In this paper, we show that malicious bitstream modifications are *i*) much easier to conduct than commonly assumed and *ii*) surprisingly powerful. We introduce a novel class of bitstream fault injection (BiFI) attacks which does *not* require any reverse-engineering. Our attacks can be automatically mounted without any detailed knowledge about either the bitstream format of the design or the crypto primitive which is being attacked. Bitstream encryption features do not necessarily prevent our attack if the integrity of the encrypted bitstream is not carefully checked. We have successfully verified the feasibility of our attacks in practice by considering several publicly available AES designs. As target platforms, we have conducted our experiments on Spartan-6 and Virtex-5 Xilinx FPGAs.

Index Terms—FPGA security, bitstream fault injection, automated key recovery, AES, bitstream encryption vulnerability.

1 Introduction

FPGAs are used in a wide range of applications and systems, e.g., in automotive, communications, computer networks, avionics, defense and many other industries. Since many of these applications require security, they need cryptographic primitives as building blocks for services such as integrity, authenticity, and confidentiality. Methods of how to attack cryptographic implementations and how to secure them have been studied for a long time in the scientific literature. As one of the earlier references, Boneh et al. [6] demonstrated in 1997 that the RSA public-key scheme as well as authentication protocols are vulnerable to fault injections. The idea is to exploit transient hardware faults that occur during the computations of the cryptographic algorithm. Due to the injected faults, faulty intermediate values may propagate sensitive information to the output revealing the private key. This concept was extended by Biham and Shamir [5] – known as Differential Fault Analysis (DFA) – to recover the secret key from symmetric block ciphers such as DES. In 2003, Piret and Quisquater [24] introduced a sophisticated fault model for AES which enables an attacker to recover the secret key with only two faulty ciphertexts.

In the last two decades, numerous other implementation attacks on hardware have been proposed, including power and EM side-channel attacks [22], glitch-based fault attacks [15], [8], laser fault attacks [27] and photonic emission attacks [26], each of which require different expertise and equipment. For a classification of fault injection attacks, we refer to contribution [35]. Notably, all proposed methods have in common that they cannot be executed automatically for different designs. They always require an experienced engineer to adjust the attack to each new target that may become a time-consuming task in a black-box scenario. Moreover, the majority of these attacks are generic hardware attacks, i.e., they do not exploit the specific nature of field programmable gate arrays.

An alternative attack strategy is to attempt to directly read-out the key from a non-volatile memory or the bitstream. However, this is often a very difficult task. The keys may be stored externally (e.g., in a hardware security module), hard-coded in an obfuscated manner in the bitstream or can be internally derived using a Physical Unclonable Function (PUF). Due to the proprietary bitstream file formats that dominate the industry and hardware architecture which are often complex, any direct read-out attempt of the key from the bitstream seems rather difficult. It should be noted that recovering the entire netlist from a Xilinx FPGA design is neither trivial nor completely possible, cf. [9]. For more information about the FPGA security, see [10] and [34].

In this paper, we introduce a new strategy to efficiently and automatically extract secrets from FPGA designs which we coin bitstream fault injection (BiFI) attack. The goal is to reduce the required expertise as well as the engineering hours. Instead of reverse-engineering an FPGA design, we *manipulate an unknown bitstream without any knowledge of the design* resulting in faulty ciphertexts. These faulty ciphertexts can then be used to recover the secret key. The general idea that one might recover secret keys by manipulating bitstreams without reverse-engineering was first mentioned in [33], but no concrete attack was proposed and it remained unclear if such an attack is feasible in practice. In this paper we not only show that such attacks are indeed feasible, but also that they are much more powerful than assumed. A

Pawel Swierczynski, Amir Moradi and Christof Paar are with the Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany. E-mail: {firstname.lastname}@rub.de. Georg T. Becker is with the Digital Society Institute (DSI) at the ESMT Berlin. The work was conducted while he was with the Ruhr-Universität Bochum. E-mail: {firstname.lastname}@esmt.org.
The work described in this paper has been partially supported by the German Federal Ministry of Education and Research BMBF (grant 16KIS0015, project PhotonFX^2) and under NSF grant CNS-1318497. It has also been supported in part by the Bosch Research Foundation.

surprising large number of bitstream manipulations result in exploitable faulty ciphertexts. A key finding of our analysis is that it is not necessary to make targeted manipulations based on knowledge of the target design. Instead, a set of design-independent manipulation rules can be applied automatically to different regions of the target bitstream until the attack succeeds. Thus, one only needs to develop an attack tool once and can apply it to any design that implements the same cryptographic algorithm. Crucially, no FPGA reverse-engineering expertise is needed to perform the BiFI attack on different targets. We verified the feasibility of the attack with 16 different AES implementations on a Spartan-6 FPGA. Out of those, 15 designs could be successfully attacked with BiFI in an automated fashion.

While it might be tempting to think that bitstream encryption can prevent BiFI attacks, this is not necessarily true. Already in [33] it was noted that bitstream manipulations might be possible in theory on encrypted bitstreams if no integrity checks is used. However, it was also noted that the CRC feature as it is implemented in Virtex-2 through Virtex-5 FPGAs should prevent bitstream manipulation attacks such as BiFI. In this paper, we show for the first time that it is possible to apply BiFI to encrypted bitstreams. We demonstrate this by successfully attacking 12 out of 13 AES cores on a Virtex-5 FPGA *with* enabled bitstream encryption. Hence, bitstream encryption in combination with a CRC feature is not necessarily enough to stop BiFI attacks.

1.1 Related Work

Reverse-engineering the proprietary bitstream structure of FPGAs has been the focus of several works [4], [9], [25], [30], [38]. It was shown by Ziener *et al.* that reverse-engineering LUT contents in an FPGA bitstream is relatively easy for Virtex-II FPGAs [38]. It has also been demonstrated how to recover LUT contents from a bitstream file [30]. However, reverse-engineering all elements in an FPGA including routing information is considerably harder as stated by Benz *et al.* [4]. The best results in the open literature were achieved in 2013 by Ding *et al.*, where between 87% and close to 100% of the routing of target designs were reverse-engineered [9]. Hence, we conclude that while reverse-engineering LUTs is rather simple, reverse-engineering the entire bitstream is considerably more challenging.

There are very few works that discuss the manipulation of non-encrypted bitstreams. It has been recently shown how to weaken a cryptographic primitive to leak a key by altering a bitstream [30]. The targets were AES designs that use a table-lookup implementation of the S-box. The authors showed how to automatically detect corresponding LUTs or BRAMs which realize the AES S-boxes, and fix the truth table outputs to zero. This enables a key recovery requesting one response. Similarly, Aldaya *et al.* proposed to modify the BRAM content of AES implementations based on T-tables, cf. [2]. Both of these attacks have in common that they target a specific design-dependent part of the employed cryptographic algorithm, which might be missing in other implementations. It was later demonstrated how a Trojan can be realized in practice by manipulating the bitstream of the FPGA used in a commercial USB flash drive offering user data encryption [29]. Even though this shows that bitstream manipulations are feasible in practice, it is design-dependent and requires major efforts. In contrast to these works, the BiFI attack is much more general and targets a wide range of possible designs with many different implementation styles. Furthermore, in contrast to all previous works, BiFI (up to some extent) can still deal with encrypted bitstreams.

We should stress that while most research in the areas of fault attacks focus on transient faults in the processed data, our attack targets permanent faults in the control logic. In our experiments, most designs could be attacked with *only one* faulty ciphertext. In many cases, neither the plaintext nor the fault-free ciphertext are required.

2 Background and Adversary Model

This section gives a brief overview of the different building blocks of an FPGA and also provides the assumed attack and system model.

2.1 FPGA Building Blocks

One of the most important building blocks in FPGAs are look-up tables (LUTs). Small truth tables (commonly either 16 or 64 bits wide) implement together with multiplexers the combinatorial logic of a design. They can realize simple gates such as an AND or XOR, but can also implement more general Boolean expressions. LUTs are truth tables with one output bit, cf. Figure 1. Besides LUTs and multiplexers,

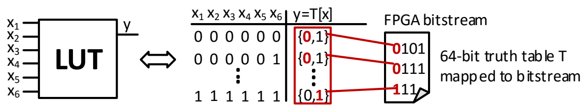


Fig. 1: On the left a 6-to-1 LUT with 6 input bits and 1 output bit is depicted. The LUT is a truth table T with 64 entries that is stored in the bitstream (right side).

FPGAs consist of D-flip-flops, BRAM blocks, DSPs and other resources that define the wiring between these building blocks. They can be connected with millions of wires to form a complex circuitry. The entire configuration of these elements is described in the binary bitstream file. LUTs, being the main logic elements in most modern FPGAs, are used for a multitude of tasks like storing constants, processing input signals, saving/loading data from flip-flops, copying signals from one location to another, controlling data buses or writing/loading BRAM contents. Given that LUTs play such a key role in FPGA designs, they are a particular interesting target for bitstream manipulation attacks.

Since we will later introduce various LUT modification rules in the bitstream, we first provide some low-level background information on how LUTs are used in most Xilinx FPGAs. An FPGA area is virtually divided into several Configurable Logic Blocks (CLB), each of which consists of – usually four – so-called slices. One slice includes various flip-flops, multiplexers, and input/output pins which are interconnected to a switch-box. It also offers four native 64-bit LUTs. Fig. 2 illustrates a simplification of the most common slice configurations. Note that we ignored some hardware elements such as flip-flops and switch-boxes for the sake of simplicity.

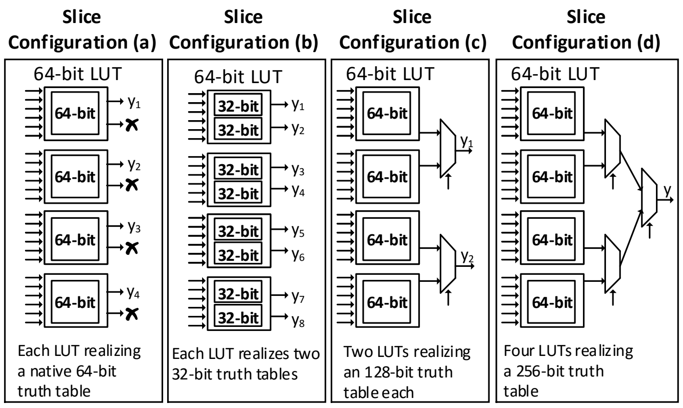

Slice Configuration (a)
Slice Configuration (b)
Slice Configuration (c)
Slice Configuration (d)
64-bit LUT
64-bit
32-bit
Each LUT realizing a native 64-bit truth table
Each LUT realizes two 32-bit truth tables
Two LUTs realizing an 128-bit truth table each
Four LUTs realizing a 256-bit truth table

Fig. 2: Subset of the most commonly used possible slice configurations with focus on look-up tables.

Each 6-bit LUT can implement one out of 2^{64} possible $6 \mapsto 1$ Boolean functions, cf. configuration (a) in Fig. 2. Alternatively, each 64-bit LUT can be split into two 32-bit subtables in order to realize two different $5 \mapsto 1$ Boolean functions with shared inputs, cf. configuration (b) in Fig. 2. Two (resp. four) LUTs within one slice can also be combined to use larger truth tables with 128 bits (resp. 256 bits) to realize $7 \mapsto 1$ (resp. $8 \mapsto 1$) Boolean functions, cf. configuration (c) and (d) in Fig. 2.

2.2 System Assumptions and Adversary Model

In the following, we describe a generic system model for SRAM-based FPGA systems which can be often found in various practical applications. Additionally, we describe our adversary model. SRAM-based FPGAs employ volatile memory, thus they require an external (usually untrusted) storage, e.g., a flash or EEPROM chip that stores the bitstream file. It needs to be loaded upon each power-up of the FPGA, cf. Fig. 3. As described above, many modern

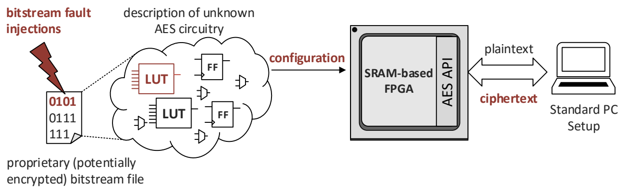

bitstream fault injections
description of unknown AES circuitry
configuration
LUT
FF
0101
0111
111
proprietary (potentially encrypted) bitstream file
SRAM-based FPGA
AES API
plaintext
ciphertext
Standard PC Setup

Fig. 3: Overview of system model. A proprietary bitstream file implements an unknown AES circuit, which configures an SRAM-based FPGA once it is powered-up. It then provides an AES interface to any possible real-world application.

applications call for cryptographic primitives to be realized on FPGAs. In this system model, we target particular designs where encryption/decryption algorithms (e.g., AES) are implemented. It is assumed that the key is not directly accessible to the adversary. For example, it is encoded or obfuscated in external memory, stored in a secure on-chip memory, hard-coded in the design/bitstream, or generated internally by a PUF. We furthermore assume that the attacker can query the AES implementation (with either known or chosen plaintexts) and observe the ciphertexts. However, in many cases we need to observe only one faulty ciphertext.

Like most implementation attacks (either side-channel or fault-injection), our attack requires physical access to the target FPGA, an assumption which is often given in practice. We suppose that the adversary has read- and write-access to the external memory which stores the bitstream of the target device. Consequently, he can arbitrarily modify the bitstream, and observe the behavior of the FPGA accordingly. More precisely, we suppose that the attacker is able to extract and modify the content of LUTs in the target bitstream. To achieve such an ability we refer to [30], where the corresponding techniques are described.

3 Bitstream Manipulation Attacks

The first step of the attack is to read out the bitstream of the device under attack from the non-volatile memory or by wiretapping the configuration data bus. The attack tool then repeatedly i) manipulates the bitstream by changing some LUT contents, ii) configures the target device, and iii) queries the manipulated design to collect faulty ciphertexts.

The faulty ciphertexts that are collected in this way are then used to recover the key by testing a set of hypotheses, e.g. the hypothesis that the collected faulty ciphertext is simply the plaintext XORed with the key. However, there are several LUTs in even small FPGAs, and testing all possible modifications on all LUT bits is not practically feasible. Therefore, we try to reduce the space for manipulations by defining particular *rules* in such a way that the faulty ciphertexts can be still used for key recovery.

3.1 Manipulations Rules

All of the conducted manipulations target the LUTs of the FPGA, i.e., only the combinatorial logic of the design is changed. Let LUT_i be the i^{th} occupied LUT of n available LUTs on the target FPGA. A LUT is a 64-bit truth table T that implements a Boolean function $y = f(x)$ with $x \in \{0,1\}^6$ and $y \in \{0,1\}$. Let us denote the j^{th} binary element in this truth table as $T[j]$ with $0 \leq j \leq 63$. As stated before, we suppose that the location of the truth table in the bitstream for each LUT is known to the attacker, and hence he can directly modify any single bit of this truth table.

With T as the original truth table and its corresponding manipulated truth table $\tilde{T}$, we define three basic operations:

1) Clear: $\tilde{T}[j] = 0, \quad \forall j \in \{0, ..., 63\}$
2) Set: $\tilde{T}[j] = 1, \quad \forall j \in \{0, ..., 63\}$
3) Invert: $\tilde{T}[j] = T[j] \oplus 1, \quad \forall j \in \{0, ..., 63\}$,

and accordingly we define three manipulation rules as

- $R_1[i]/R_2[i]/R_3[i]$: Clear/Set/Invert the i^{th} 64-bit LUT,

which cover the cases where the entire LUT forms a $6 \mapsto 1$ function, cf. configuration (a) in Fig. 2.

Besides modifying the entire LUT, we also consider the cases where only the *upper* or *lower* half of the LUT is manipulated. As an example, we can form $\tilde{T}$ by

- $\tilde{T}[j] = 1, \quad \forall j \in \{0, ..., 31\}$
- $\tilde{T}[j] = T[j], \quad \forall j \in \{32, ..., 63\}$.

In other words, we only modify the upper half (first 32 bits) of the truth table. The motivation of considering these operations is the cases where a LUT realized a $5 \mapsto 2$ function, cf. configuration (b) in Fig. 2. Hence we define three other rules as

- $R_4[i,h]/R_5[i,h]/R_6[i,h]$: Clear/Set/Invert the h^{th} half of the i^{th} LUT.

To cover other two configurations – case (c) and (d) in Fig. 2 – where two or four LUTs are grouped to form larger truth tables, we define the next four rules as

- $R_7[i]/R_8[i]$: Clear/Set all 4 LUTs within the i^{th} slice.
- $R_9[i,h]/R_{10}[i,h]$: Set/Clear ($h = 1$) upper or ($h = 2$) lower 2 LUTs within the i^{th} slice.

Let us define the Hamming weight of the content of all 4 LUTs within a slice by HW. Accordingly we define two rules as

- $R_{11}[n], R_{12}[n]$: Clear/Set all 4 LUTs within slices with $HW = n$,

with $n \in \{1, ..., 256\}$. In other words, by these rules we clear/set all slices that have a specific Hamming weight. The motivation for these rules is to potentially alter multiple instances of the same Boolean function simultaneously. This may result in manipulating all instances of the same S-Box in the design at once.

Based on our observations, the LUTs of the control logic that examine whether a counter (e.g., AES round counter) reaches a certain value (e.g., to issue a *done* signal) have a considerably low or high HW. In other words, the content of such LUTs have a high imbalance between the number of '1's and '0's. As an example, a LUT with 4 inputs $c_3c_2c_1c_0$ which checks whether $c_3c_2c_1c_0$=`1010` (10_{dec} as the number of AES-128 rounds) has HW=1. Therefore, we consider the following rules accordingly

- $R_{13}[i,j]$: Invert bit $T[j]$ of the i^{th} LUT, if $1 \leq HW \leq 15$,
- $R_{14}[i,j]$: Invert bit $T[j]$ of the i^{th} LUT, if $49 \leq HW \leq 64$.

Finally, we cover the case where a LUT is replaced by a random Boolean function (in our experiments we applied this rule 10 times to each LUT):

- $R_{15}[i]$: Set the i^{th} LUT to a random 64-bit value.

3.2 Key Recovery

By applying any of the above-explained manipulation rules ($R_1 - R_{15}$), we have hit control logic and/or data processing part if a faulty ciphertext is observed. All collected faulty ciphertexts potentially exhibit a sensitive intermediate value allowing for key recovery. Hence, all faulty ciphertexts are further processed depending on which intermediate value hypothesis ($H_1 - H_{11}$) is tested to derive a set of AES key candidates. All those derived AES key candidates are then automatically tested by a C++ tool with the help of one valid plaintext-ciphertext pair (p, c).

In practice, an adversary can check whether the computation $AES^{-1}_{\text{key candidate}}(c)$ outputs the known plaintext p. Given this is true, the tested AES key candidate is equal to the correct AES key k. A known plaintext-ciphertext pair (p, c) can for example be obtained from our encryption oracle itself or the attacker might know part(s) of the plaintext, i.e., due to constant header information. In cases where no plaintext-ciphertext pair is available (p unknown, encryption can be invoked, ciphertext c observable), an adversary may also analyze the entropy of decrypted ciphertexts, e.g., if the decryption yields text which typically has a low entropy.

To understand our key recovery approach, we use the following notations for the AES computation:

- $p, k, c, \tilde{c}$: 16-byte plaintext, key, ciphertext, and faulty ciphertext with $c = AES128_k(p)$.
- 0^{128}: 128-bit value consisting of all 0's.
- rk_j : 16-byte j^{th} round key with $j \in \{0, ..., 10\}$ with $rk_0 = k$.
- $SB(st), SR(st), MC(st)$: SubBytes, ShiftRows, and MixColumns operations on the current 16-byte state st. Analogously, $SB^{-1}(st), SR^{-1}(st)$, and $MC^{-1}(st)$ represent the respective inverse functions.
- sb_j, sr_j, mc_j, ka_j: 16-byte state after SubBytes, ShiftRows, MixColumns, and KeyAdd operations respectively at round j.

Based on our observations, in several cases the key can be extracted without knowing the corresponding ciphertext. We hence define the following hypotheses:

- $H_1[j]$: $\tilde{c} = rk_j$
- $H_2[j]$: $\tilde{c} = SB(0^{128}) \oplus rk_j$,

for $j \in \{0, \ldots, 10\}$. The hypothesis H_1 mainly deals with those cases where the state st becomes 0^{128}. Further, H_2 targets those faults which hit the control logic in such a way that the S-box input register becomes always inactive. We give a more detailed information about this concept in Section 5.1.3. If only one round key rk_j is extracted, the main key can be easily recovered (e.g., see [14]). As a side note, these hypotheses should be also examined on the invert of the faulty ciphertext $\tilde{c}$.

Further, we consider the following hypotheses:

- $H_3[j]$: $\tilde{c} = c \oplus rk_j$
- H_4: $\tilde{c} = ka_9$
- H_5: $\tilde{c} = sb_{10}$

To recover the key using H_3 is straightforward as $rk_j = \tilde{c} \oplus c$. Hypotheses H_4 and H_5 check the dependency between the faulty ciphertext and the state in the last AES round. With hypothesis H_4 the last roundkey rk_{10} can be recovered: $rk_{10} = SR\left(SB\left(\tilde{c}\right)\right) \oplus c$. A similar approach can be followed for hypothesis H_5. The next set of hypotheses are defined as:

- $H_6[j]$: $\tilde{c} = p \oplus rk_j$
- H_7: $\tilde{c} = sr_1$
- H_8: $\tilde{c} = sb_1$
- H_9: $\tilde{c} = mc_1$
- H_{10}: $\tilde{c} = AES_{k'}(p)$
- H_{11}: $\tilde{c} = SR(SB(p)) \oplus rk_{10}$,

where k' is defined as $rk'_0 = rk_0$, and $rk'_{j \in \{1,\ldots,10\}} = 0^{128}$. Using H_6 is straightforward. H_7, H_8, and H_9 can also be checked by applying the corresponding inverse functions, e.g., $rk_0 = SB^{-1}\left(SR^{-1}\left(MC^{-1}\left(\tilde{c}\right)\right)\right) \oplus p$ for H_9.

To examine H_{10}, we need to apply a specific form of the decryption function as $rk_0 = AES^{-1}_{k''}(\tilde{c}) \oplus p$ with $\forall j \in \{0,\dots,10\}$, $rk''_j = 0^{128}$. Hypothesis H_{11} can be seen as an AES encryption where only the last round is executed. In this case, the last round key can be trivially computed with $rk_{10} = \tilde{c} \oplus SR(SB(p))$.

In each of the above hypotheses only one faulty ciphertext is used. One can also define hypotheses that used two faulty ciphertexts generated by two different bitstream manipulations for a certain plaintext. As an example, $\tilde{c}_1 = ka_j$ and $\tilde{c}_2 = ka_{j+1}$, which can lead to a full key recovery. In this scenario, the adversary needs to try all possible combinations between different faulty ciphertexts, and the computation complexity of the attack increases quadratically. Since it is – to some extent – in contradiction with our goal (i.e., limiting the number of rules as well as the hypotheses for key recover), we omit the corresponding results although we have observed such successful cases in 6 designs in our experiments.

4 Experimental Setup and Results

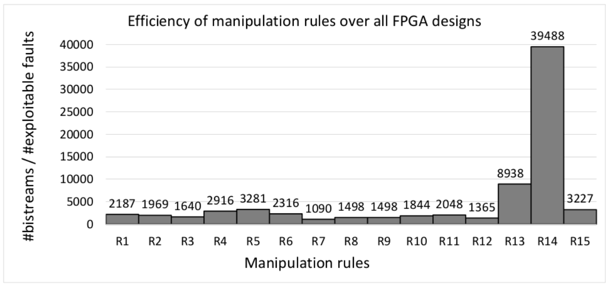
For practical evaluations, we collected 16 AES encryption designs $D_0, ..., D_{15}$, four of which were developed by our group. The other 12 cores have been taken from publicly-available websites, e.g., NSA homepage, OpenCores, GitHub, SASEBO project[1].

Most of the designs (D_0, $D_2 - D_4$, D_6, $D_9 - D_{15}$) operate on a round-based architecture. D_1 is based on a word-based architecture, where 32-bit blocks are processed at each clock cycle. D_7 and D_8 follow a serialized architecture, i.e., with only one instance of the S-Box, where at each clock cycle one byte is processed. Finally, D_5 is an unrolled pipeline design with 10 separately instantiated round functions, which can give one ciphertext per clock cycle if the pipeline is full.

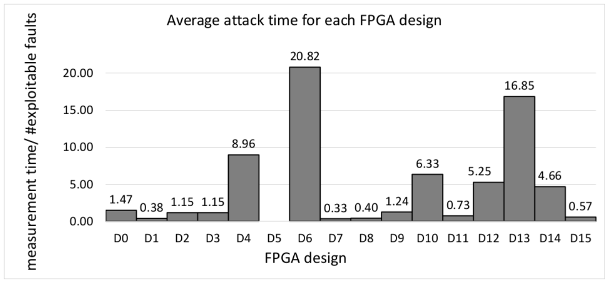
Each core is provided by an interface to set the key k and the plaintext p, and to fetch the ciphertext c. We developed an FPGA design template to play the role of an RS-232 interface between a PC and the target AES core, where the key is kept constant by the template. During the integration of any of the target AES cores, we did not modify the AES circuitry, but adopted the template to fit to the sequences required by the underlying AES interface. As an example, some of the AES cores require first to perform a key schedule, while the others process the actual encryption and key schedule in parallel.

In order to perform the attacks, we developed a program to automatically apply all the rules R_1-R_{15} given in Section 3.1 one after each other. To this end, our tool first queries the original design with a particular plaintext and captures the fault-free ciphertext (p, c). Afterwards, for each rule our tool manipulates the bitstream accordingly, configures the FPGA, queries the design with the same particular plaintext p and collects the corresponding faulty ciphertext $\tilde{c}$. Another program (which we have also developed) examines all the hypotheses H_1-H_{11} described in Section 3.2 by analyzing each faulty ciphertext.

1. D_0 [16], D_1 [13], D_2 [11], D_4 [12], D_5 [32], D_6 [21], D_{10} [23], D_{11}-D_{15} [1]



(a)



(b)

Fig. 4: a) The ratio the number of performed bitstream manipulations over the number of exploitable faults. b) The average attack time (in hours) until an exploitable faulty ciphertext is obtained for each of the targeted design (using modification rules R_1-R_{12}).

4.1 Results without Bitstream Encryption

Our setup is based on a Spartan-6 (XC6SLX16) FPGA, where the JTAG port is used for configuration. The entire bitstream manipulation, configuration, query and collection of the faulty ciphertext takes around 3.3 seconds. We should emphasize that by manipulating the bitstream, the Cyclic-Redundancy-Check (CRC) checksum should be correct [30]. Alternatively, the bitstream can be modified in such a way that the CRC check is simply disabled.

As stated before, we conducted our attack on 16 different designs. The corresponding results are depicted in Table 1 indicating which manipulation rule R_1-R_{15} on which AES design D_0-D_{15} led to an exploitable faulty ciphtertext. Similarly, Table 2 shows for each AES design which hypotheses H_1-H_{11} led to successful key recovery.

For all designs, except the unrolled pipeline one (D_5), at least one hypothesis could make use of the faulty ciphertexts generated by the manipulation rules to recover the key. In Section 5.1 we give a detailed analysis on the exploitable faults. In short, many exploitable faults hit the control logic (i.e., the AES state machine is modified). We predict this to be the reason why the design D_5 cannot be successfully attacked, since the unrolled pipeline design makes use of the simplest state machine.

It can be seen from Table 1 that many different manipulations rules lead to exploitable faulty ciphtertexts. It is also worth mentioning that each manipulation rule was successful for at least one design. To compare the efficiency of the manipulation rules, we computed a ratio between the number of performed bitstream manipulations and the number of exploitable faults, cf. Fig 4a. Note that a

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	d.att.
Changing all 64 bits of a LUT in the bitstream																	
$R_1[i]$: Clear LUT	1	3	.	6	.	.	.	1	1	3	1	.	.	.	.	2	8
$R_2[i]$: Set LUT	2	4	3	1	.	.	.	2	2	2	.	3	.	.	.	1	9
$R_3[i]$: Invert LUT	1	9	2	2	1	.	.	.	1	2	2	2	.	.	.	2	10
Changing only 32 bits of a LUT in the bitstream																	
$R_4[i,h]$: Clear half LUT	.	4	.	7	.	.	1	4	3	3	.	.	1	1	1	2	10
$R_5[i,h]$: Set half LUT	1	3	2	3	.	.	.	3	3	2	.	3	1	1	1	1	12
$R_6[i,h]$: Invert half LUT	.	7	2	5	1	.	.	1	1	3	1	2	2	2	3	4	13
Changing two or four 64-bit LUTs in the bitstream																	
$R_7[i]$: Clear slice	.	1	.	4	.	.	.	1	.	3	1	.	.	.	.	1	5
$R_8[i]$: Set slice	1	1	1	1	.	.	.	1	.	1	.	1	.	.	.	1	3
$R_9[i,h]$: Set 2 LUTs	1	2	.	5	.	.	.	1	2	3	1	.	.	.	.	1	8
$R_{10}[i,h]$: Clear 2 LUTs	1	2	1	1	.	.	.	2	2	1	.	2	.	.	.	1	9
Clearing only LUTs with a specific slices' HW in the bitstream																	
$R_{11}[n]$: Clear slice if HW=n	.	.	.	1	.	.	.	1	.	.	.	.	.	.	.	.	2
$R_{12}[n]$: Set slice if HW=n	.	1	.	1	.	.	.	1	.	.	.	.	.	.	.	.	2
Inverting single LUT bits (64 times) for a specific HW in the bitstream																	
$R_{13}[i,j]$: Invert bits if HW ≤ 15	.	2	2	3	.	.	2	5	5	4	.	.	1	1	1	.	7
$R_{14}[i,j]$: Invert bits if HW ≥ 49	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
Configuring random Boolean functions (10 times) in the bitstream																	
$R_{15}[i]$: Set LUT randomly	7	31	16	16	1	.	.	3	7	13	1	12	5	2	5	3	14
Statistics																	
$\sum$ exploitable faulty ciphertexts	15	71	29	56	3	0	3	26	27	40	7	25	10	7	11	19	
Number of vulnerable LUTs (R_1-R_{14})	5	20	4	18	1	0	2	9	7	13	3	5	3	3	4	6	
Measurement time R_1-R_{12} (hours)	20	26	18	74	22	64	26	9	9	42	62	12	26	246	28	12	
Measurement time R_{13}-R_{14} (hours)	110	12	11	38	16	53	18	5	5	26	34	8	18	61	21	7	

TABLE 1: Overview of the experiments with regard to the different modification rules. Each entry in the table represents the number of times for which applying the manipulation rule R_i lead to an exploitable fault for design D_j. The last column "d.att." (designs attacked) shows the number of different designs D_j that could be attacked with the corresponding rule. In the experiment, several different modification rules resulted in an exploitable faulty ciphertext when applied to the same LUT. The number of LUTs that lead to at least one exploitable faulty ciphertext for at least one of the manipulation rules $R_1 - R_{14}$ is shown in row "Number of vulnerable LUTs" as a reference.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	d.att.
$H_1 : rk_0$	.	.	.	.	.	.	.	.	.	.	.	12	.	.	.	.	1
$H_1 : rk_{10}$	.	.	.	33	.	.	.	2	4	15	.	3	.	.	.	5	6
$H_2 : S(0^{128}) \oplus rk_0$	.	.	.	.	.	.	.	.	.	.	.	.	4	2	4	.	3
$^{\dagger}H_2 : S(0^{128}) \oplus rk_1$	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	1
$^{\dagger}H_2 : S(0^{128}) \oplus rk_2$	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	1
$H_2 : S(0^{128}) \oplus rk_{10}$	.	6	.	.	.	.	.	2	2	.	.	.	.	.	.	.	3
$H_3 : c \oplus rk_{10}$	.	.	.	.	.	.	.	2	2	.	.	.	.	.	.	.	2
$H_4 : ka_{10}$	.	.	.	.	3	.	2	.	.	.	.	.	.	.	1	1	4
$H_6 : p \oplus rk_0$	4	.	19	10	.	.	.	.	.	14	2	10	1	3	4	7	10
$^{\dagger}H_6 : p \oplus rk_2$	1	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	2
$H_6 : p \oplus rk_4$	6	.	.	1	.	.	.	.	.	1	.	.	.	.	.	.	3
$^{\dagger}H_6 : p \oplus rk_5$	.	.	.	3	.	.	.	.	.	1	.	.	.	.	.	.	2
$^{\ddagger}H_6 : p \oplus rk_6$	.	.	.	1	.	.	.	.	.	2	.	.	.	.	.	.	2
$H_6 : p \oplus rk_{10}$	4	.	.	8	.	.	.	.	.	5	.	.	.	.	.	.	3
$^{\ddagger}H_7 : sr_1$	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
$^{\dagger}H_9 : mc_1$	.	.	.	.	.	.	.	.	.	.	.	.	3	.	.	.	1
$H_{10} : AES_k(p), rk_j = 0$	.	63	10	.	.	.	1	17	17	.	4	.	2	2	2	4	10
$H_{11} : SR(S(p)) \oplus rk_{10}$	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	1
Statistics																	
Collected responses	20333	26455	18044	75296	22442	65051	26522	8914	9080	42620	62991	12261	26226	249813	28188	12077	
Unique faulty responses	6411	7412	6022	28512	8955	3772	15887	2587	2675	15760	10137	5171	15484	45971	17246	5317	

TABLE 2: Overview of the experiments with regard to the different hypotheses. Each entry in the table represents the number of times a hypotheses H_i for each design D_j could be used to recover the key from faulty ciphertexts being the result of applying the modification rules R_1-R_{15}. Some of the hypotheses (marked by †) succeed only for R_{15} while some other hypotheses with ‡ could make use of only R_1-R_{14}. The last column "d.att." shows the number of different designs that could be successfully attacked by the corresponding hypothesis. The last two rows summarize the number of collected responses (which are equivalent to the number of times a bitstream manipulation was conducted) and the number of observed unique faulty ciphertexts.

lower ratio means that the underlying manipulation rule is more efficient, since the average number of manipulations required for an attack becomes smaller. As stated before, each manipulation rule led to at least one exploitable faulty ciphertext. However, some of them are more efficient than the others. The most efficient one is R_7 (i.e., clear an entire slice), and R_{13} and R_{14} are among the worst manipulation rules. On the other hand, we should emphasize that in R_{13} and R_{14} each bit of the target LUT is independently manipulated. Hence, the number of manipulated bitstreams in these two rules is considerably higher compared to the other rules.

We would like to stress that in average every 3227 random manipulations (R_{15}) led to an exploitable faulty ciphertext (cf. Fig 4a) indicating that it is also a solid strategy. Nevertheless, manipulations rules R_1-R_{12} are a bit more efficient than random manipulations with an average 1971 manipulations required to observe an exploitable faulty ciphertext.

As stated before, the entire manipulation, configuration, and query takes around 3.3 seconds. Hence, in average $1971 \times 3.3 = 1.8$ hours of bitstream manipulations are needed per exploitable faulty ciphertext for rules R_1-R_{12}. However, this time varies significantly depending on the targeted design. Figure 4b shows the average time of bitstream manipulations (over manipulation rules R_1-R_{12}) needed for an exploitable fault for each of the 16 AES designs.

4.2 Experimental Setup with Bitstream Encryption

To prevent reverse-engineering and IP-theft some FPGAs are equipped with so-called bitstream encryption. We also investigated to what extent the above-presented attack can be efficient when the underlying bitstream is encrypted. To this end, we take a closer look at this feature integrated in several Xilinx FPGAs.

When this protection mechanism is enabled in the vendor's software, the user can chose a 256-bit AES key k as well as a 128-bit initial vector IV. Excluding the header, the main body of the bitstream is encrypted using $AES256_k(\cdot)$ in Cipher Block Chaining (CBC) mode. The corresponding bitstream data of size m is divided into n 16-byte plaintext blocks with $P_{i\in\{1,\dots,\frac{m}{16}\}}$, and sequentially encrypted as

$$C_i = AES256_k(P_i \oplus C_{i-1}), \text{ for } i > 0 \text{ and } C_0 = IV. \quad (1)$$

Analogously, the decryption is performed by a dedicated hardware module on the FPGA as

$$P_i = AES256_k^{-1}(C_i) \oplus C_{i-1}, \text{ for } i > 0 \text{ and } C_0 = IV. \quad (2)$$

The key needs to be programmed once into the target FPGA either in its volatile (BBRAM) or non-volatile memory (eFUSE). At every power-up, if the FPGA receives an encrypted bitstream, it runs the corresponding CBC decryption and configures its internal elements accordingly. In relatively old Xilinx FPGAs, i.e., Virtex-4, Virtex-5, and Spartan-6 families, the integrity of the encrypted bitstream is examined by a 22-bit CRC. In Virtex-4 and Virtex-5 FPGAs, the CRC checksum is not included in the encrypted part, and the corresponding command to enable the CRC is involved in the (*unencrypted*) header of the bitstream. Hence, the attacker can easily disable such an integrity check by patching the encrypted bitstream. However, in case of Spartan-6 the encrypted part of the bitstream contains the CRC as well. Therefore, any bitstream manipulation most likely leads to CRC failure (see Appendix 8.1 for more information). Further, in more recent Xilinx products, e.g., Virtex-6 and the entire 7-series, the integrity (as well as authenticity) is additionally examined by an HMAC, which also disables any bitstream manipulation.

Therefore, in order to investigate the efficiency of our BiFI attack when the bitstream is encrypted, we conducted our experiments on a Virtex-5 FPGA. Obviously it is desirable to control the effect of manipulation of the bitstream, e.g., to avoid the propagation of the changes. If l bits of ciphertext block C_{i+1} – in CBC mode – are toggled, its effect on the plaintext block P_{i+1} is not predictable. However, it also directly changes the corresponding l bits of the next plaintext block P_{i+2}, and interestingly such a manipulation does not propagate through the entire bitstream. This concept is illustrated in Figure 5.

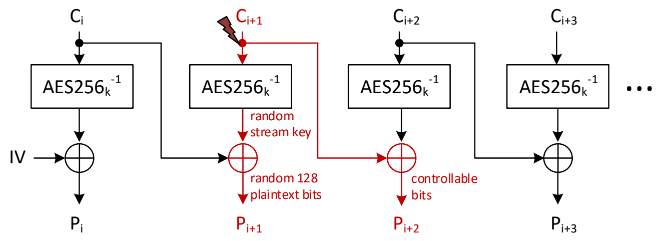


Fig. 5: The impact of faulting one ciphertext block in case of CBC decryption.

4.3 Results with Enabled Bitstream Encryption

On our Virtex-5 (XC5VLX50) setup – with bitstream encryption enabled – we examined 13 AES designs (D_0, D_2-D_6, D_9-D_{15}) out of the previously expressed 16 designs[2]. If we ignore the unpredictable changes on plaintext P_{i+1}, toggles on the bits of ciphertext C_{i+1} lead to the same toggles on plaintext P_{i+2} (see Fig. 5). Therefore, we can only apply the rules R_3 and R_6 which toggle the entire or a half of a LUT. Further, the unpredictable manipulation of plaintext P_{i+1} may also hit a utilized LUT. In short, manipulation of ciphertext C_{i+1} based on R_3 and R_6 indirectly applies the rule R_{15} to other LUTs as well. More importantly, the unpredictable changes on plaintext P_{i+1} can lead to misconfiguration of switch-boxes, and hence short circuits[3]. In such scenarios, the FPGA cannot be configured, and needs to be restarted.

We should emphasize that we assume that the attacker has a deep knowledge about the underlying bitstream structure. As an example, he knows which parts of the bitstream (either unencrypted or encrypted) belong to LUTs' configuration. However, in case of the encrypted bitstream he cannot realize which LUTs are utilized. Therefore, the rules R_3 and R_6 need to be applied to all available LUTs (28,800 in our Virtex-5 (XC5VLX50) FPGA). Hence, the attack takes longer compared to targeting an unencrypted bitstream. Further, since the Virtex-5 FPGA equipped in our setup is larger (hence, larger bitstream) than the Spartan-6 one, each configuration takes around 6.6 seconds, i.e., two times slower than the previously-shown experiments, which in sum turns into 6.8 days to apply both R_3 and R_6 rules on all available LUTs. Table 3 shows the corresponding result of the attacks on the targeted 13 AES designs.

Similar to the unencrypted case, only the unrolled pipeline design D_5 cannot be successfully attacked. Notably, an average of 11.5 hours is needed for a successful attack over all designs. For further details, we refer to Table 4 which shows all successful hypotheses leading to the exposure of the key.

We also conducted another attack in which we considered the encrypted part of the bitstream as a complete

2. Due to their e.g., hard-coded macros not compatible with Virtex-5, the designs D_1, D_7, and D_8 could not be synthesized on this FPGA.

3. Based on our observations, the currently-available FPGAs in the market are protected against such short circuits, preventing them being destroyed.

	D_0	D_2	D_3	D_4	D_5	D_6	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	d.att.
$R_3[i]$	12	7	11	1	.	2	6	1	8	6	4	5	6	12
$R_6[i,h]$	25	11	14	2	.	2	8	2	11	8	7	13	13	12
$\sum$ exploitable faulty ciphertexts	**37**	**18**	**25**	**3**	**.**	**4**	**14**	**3**	**19**	**14**	**11**	**18**	**19**	**12**

TABLE 3: Overview of the BiFI attack on encrypted bitstreams. Two modification rules R_3 and R_6 were tested and each table entry represents the number of exploitable faulty ciphertexts.

	D_0	D_2	D_3	D_4	D_5	D_6	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	d.att.
$H_1 : rk_0$	.	.	.	.	.	.	.	.	7	.	.	.	5	2
$H_1 : rk_{10}$	.	.	16	.	.	.	7	.	.	.	.	.	.	2
$H_2 : S(0^{128}) \oplus rk_0$	.	.	.	.	.	.	.	.	.	3	4	13	.	3
$H_2 : S(0^{128}) \oplus rk_1$	2	.	.	.	.	.	.	.	.	.	.	.	.	1
$H_3 : c \oplus rk_{10}$	.	.	.	.	.	.	.	1	.	.	.	.	.	1
$H_4 : ka_{10}$	.	.	.	3	.	.	.	.	1	.	.	.	.	2
$H_6 : p \oplus rk_0$	17	12	.	.	.	.	.	.	11	7	6	5	13	7
$H_6 : p \oplus rk_2$	9	.	1	.	.	.	1	.	.	.	.	.	.	3
$H_6 : p \oplus rk_3$	.	.	2	.	.	.	.	.	.	.	.	.	.	1
$H_6 : p \oplus rk_5$	4	.	5	.	.	.	.	.	.	.	.	.	.	2
$H_6 : p \oplus rk_6$	.	.	1	.	.	.	.	.	.	.	.	.	.	1
$H_6 : p \oplus rk_8$	2	.	.	.	.	.	.	.	.	.	.	.	.	1
$H_6 : p \oplus rk_{10}$	3	.	.	.	.	.	6	.	.	.	.	.	.	2
$H_{10} : AES_k(p), rk_j = 0^{128}$	.	6	.	.	.	4	.	1	.	2	1	.	1	6
$H_{11} : SR(S(p)) \oplus rk1$	.	.	.	.	.	.	.	.	.	2	.	.	.	1
Statistics														
Collected responses	86400	86400	86400	86400	86400	86400	86400	86400	86400	86400	86400	86400	86400	
Unique faulty responses	4655	7659	16959	12118	9432	12124	8089	19638	4677	11706	12469	21945	5269	

TABLE 4: Overview of the exploitable faulty ciphertexts of the different hypotheses for 13 different designs with enabled bitstream encryption.

black-box, i.e., without directly targeting the LUTs. In order to minimize the effect on plaintext block P_{i+2}, we only toggled the most significant bit of one ciphertext block. In other words, we tried to apply only R_{15} on plaintext block P_{i+1}.We conducted this bitstream manipulation to each encrypted block once and could successfully attack 11 out 13 designs. Attacking one AES core takes approximately 8 days, and led again to various exploitable faulty ciphertexts[4]. To conclude, knowing the exact locations of the LUT contents in the (encrypted) bitstream is not necessarily essential.

4.4 Discussion on Altera Bitstream Encryption Scheme

The underlying mode of encryption and the employed integrity check determine whether a BiFI attack can be mounted on an encrypted bitstream. While we used Xilinx FPGAs in our practical experiments, below we discuss about the feasibility of our attack on Altera's bitstreams with enabled encryption. In the recent families of Altera FPGAs (similar to that of Xilinx FPGAs), an HMAC authentication process is integrated. Hence, such devices are not susceptible to our attacks (unless the bitstream encryption and authentication is circumvented e.g., by a side-channel analysis attack [19], [31]). However, the older Stratix-II and Stratix-III families use AES in counter mode and a simple CRC for integrity check. The underlying scheme in Stratix-II and Stratix-III are similar except i) AES-128 replaced by AES-256 in the later one, and ii) arithmetic counter (of the counter mode) replaced by a sophisticated pseudo-random-number generator (for more information see [19], [31]).

4. It is noteworthy that in this experiment several times the FPGA "crashed" i.e. could not be programmed until it was restarted manually. This never happened for the manipulation rules that only targeted LUTs.

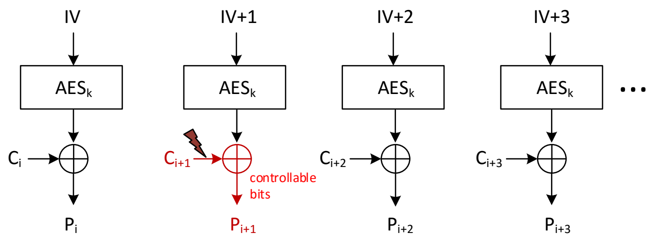


Fig. 6: The decryption in Counter Mode as it is used for bitstream encryption in Stratix-II and III FPGAs. Toggling a single ciphertext bit results in a predictable toggle of a plaintext bit.

Both devices generate a stream key which is XORed with the plaintext blocks to form the encrypted bitstream. The decryption process (performed on the FPGA) follows the same concept as depicted in Figure 6. In this case, if an adversary manipulates the bitstream by toggling l bits of the ciphertext block C_{i+1}, the corresponding l bits of the same plaintext block P_{i+1} toggle, and the changes propagate neither to other bits of the same block nor subsequent blocks. Therefore, compared to the encryption feature of Xilinx FPGAs, the attacker has more control over the manipulations, hence higher efficiency of BiFI attacks. More importantly, since the CRC is linear, it can be trivially predicted how the CRC checksum should change by any toggle made on a ciphertext block (similarly on a plaintext block). More precisely, the attacker can toggle any arbitrary bit(s) of the encrypted bitstream and correspondingly modify the CRC checksum. Therefore, the counter mode makes BiFI attacks considerably easier if a CRC integrity check is employed. Although we have not yet practically examined it, we are confident that our attack can be easily and successfully applied on Altera Stratix-II and Stratix-III FPGAs.

5 Analysis

So far we have only expressed the manipulation rules as well as the hypotheses which we used to conduct successful attacks. Below, we give more details by focusing on a few cases, where observed faulty ciphertexts led to key recovery.

5.1 Evaluation of Observed Faults

For a couple of exploitable faulty ciphertexts, we investigated at the netlist level, what exactly caused this faulty behavior. To do so, we used the FPGA Editor (provided by the Xilinx ISE toolchain) to analyze the LUTs whose modification in the bitstream led to a key exposure. Due to the large number of faults, we only cover a small subset of exploitable faults that are representative for a class of similar possible faults. Hence, the provided analysis is not a comprehensive study and only aims at providing the reader with an intuition of what can happen during the attack. It is noteworthy that the presented figures are a simplified high-level representation of the usually more complex hardware structure.

5.1.1 Hitting the Control Logic

A successful key recovery on the round-based design D_0 was due to manipulating a LUT whose two output bits are used to control a 3-to-1 multiplexer controlled by two bits (m_1, m_2). A part of this design performs the following operations

- CLK cycle 1: $state = p \oplus rk_0$,
- CLK cycles 2-10: $state = mc_j \oplus rk_j, j \in \{1, ..., 9\}$,
- CLK cycle 11: $c = sr_{10} \oplus rk_{10}$.

Depending on the clock cycle (i.e., the round counter) the targeted LUT (which controls the multiplexer) switches between the plaintext p $(0,0)$, the state after MixColumns mc_j $(0,1)$, and the final state after the final ShiftRows sr_{10} $(1,0)$, cf. the upper part of Figure 7. The 128-bit multiplexer output is XORed to the corresponding 128-bit round key.

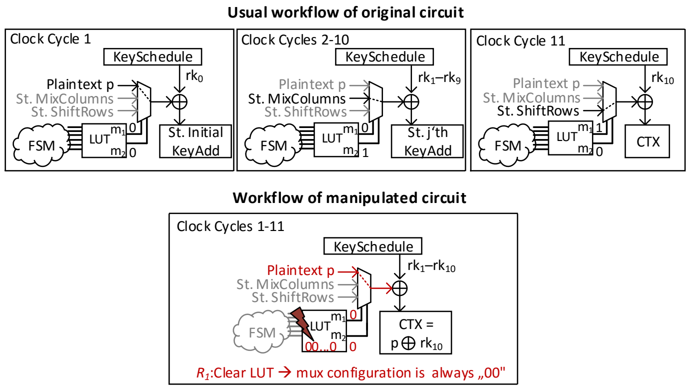


Fig. 7: Manipulation rule R_1 (*Clear LUT*), round-based design D_0, consequence: plaintext p (instead of sr_{10}) is XORed to the last AES round key rk_{10}.

By applying the rule R_1 (*Clear LUT*), the LUT outputs are permanently fixed to $(0,0)$, and hence, the multiplexer always outputs p regardless of the clock cycle, cf. the lower part of Figure 7. More precisely, by such a bitstream manipulation, the following operations are performed

1) CLK cycle 1: $state = p \oplus rk_0$,
2) CLK cycles 2-10: $state = p \oplus rk_j, j \in \{1, .., 9\}$,
3) CLK cycle 11: $\tilde{c} = p \oplus rk_{10}$.

The circuit outputs $\tilde{c} = p \oplus rk_{10}$ instead of $c = sr_{10} \oplus rk_{10}$, which is the motivation to include hypothesis H_6 for key recovery.

5.1.2 Update mechanism of flip-flops - Never Update 128-bit key register

We noticed a manipulated LUT whose output controls the update of a couple of flip-flops. As an example, a LUT might control the *CE* signal (*Clock Enable*) of a 128-bit state or key register. The flip-flops' content is updated on e.g., the rising edge of the clock, only if the CE signal is '1'. The manipulation rule R_1 (*Clear LUT*) turns such a LUT into constant '0', hence always disabling the update. We have observed many cases where the round key registers are never updated. Such a LUT modification can hence turn flip-flops into read-only elements, while they only output their initial value[5].

An example is depicted in Figure 8. It shows that the key schedule output is not stored by the flip-flops and they output always '0'. Therefore, such a manipulation rule can affect all round keys such that $rk_{j \in \{1,...,10\}} = 0^{128}$.

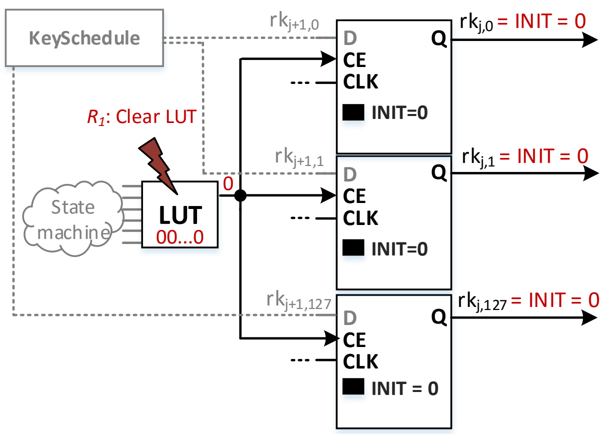


Fig. 8: Manipulation rule R_1 (*Clear LUT*), round-based design D_{15}, group of flip-flops forming a 128-bit round key register ($rk_{j,0}$ - $rk_{j,127}$) used for XOR with the current AES state. Due to the manipulation, none of the round key flip-flops are updated. Instead, they always remain '0'.

Based on our observations – depending on the design architecture – the initial KeyAdd operation is still conducted by the correct round key rk_0. More precisely, the manipulated AES core performs the following operations:

5. In Xilinx FPGAs the initial value of every flip-flop can be defined. Without any definition, the default value (usually '0') is taken by the synthesizer.

1) $state = p \oplus rk_0$,
2) $state = MC(SB(SR(state))) \oplus 0^{128}, j \in \{1, ..., 9\}$,
3) $\tilde{c} = SB(SR(state)) \oplus 0^{128}$.

Therefore, the hypothesis H_{10} (defined in Section 3.2) can examine whether a manipulation hit the corresponding LUT, and consequently recover the key.

5.1.3 Update mechanism of flip-flops - Never Update S-box Input Register

We have observed a similar scenario for the other registers. As an example, we focus on the AES design D_1 with 32-bit datapath, where updating the complete 128-bit AES states requires at least four clock cycles. Similar to the prior case, the *CE* pin of the registers are controlled by a LUT. If such a LUT is manipulated by R_1 (*Clear LUT*), the register will have always its initial value, cf. Figure 9.

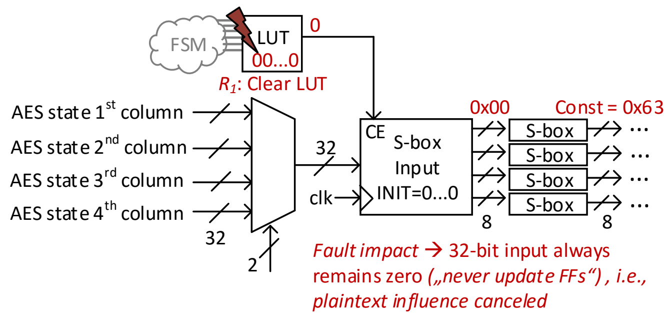


Fig. 9: Manipulation rule R_1 (*Clear LUT*), word-based design D_1, due to the bitstream manipulation the S-box inputs remain zero, this results into the leakage of the last round key rk_{10}.

It is noteworthy that in this design, the four aforementioned AES S-boxes are used only for the SubBytes operation, i.e., the key schedule circuitry employs separate S-box instances. Even though all the main AES operations (ShiftRows, AddRoundkey, MixColumns, etc.) operate correctly, the round output is not stored into the S-box input registers. Therefore, the manipulated design outputs $\tilde{c} = SB(SR(0^{128})) \oplus rk_{10} = SB(0^{128}) \oplus rk_{10}$, which trivially leads to recovering the last round key rk_{10}, i.e., hypothesis H_2.

These results indeed indicate that preventing registers from updating their state can lead to various exploitable ciphertexts. However, as expressed above, the feasibility of the key recovery depends on the initial value of the registers. Hence, if the registers (in HDL representation) are initialized by arbitrary values, the aforementioned attacks would need to guess the initial value instead of 0^{128}, which obviously complicates the attacks.

5.1.4 Update mechanism of flip-flops - Invert Updating

We also observed that the manipulation rule R_3 (*Invert LUT*) resulted in inverting a signal that controls when the output register (128-bit flip-flops so-called *out_reg*) should be updated by trivially being connected to its *CE* pin. In the underlying AES core, the *out_reg* is not updated during the encryption except when the encryption is terminated thereby storing the ciphertext. To this end, the corresponding LUT output (so-called *update_out_reg_enable*) is '1' at only one clock cycle, cf. upper part of Figure 10.

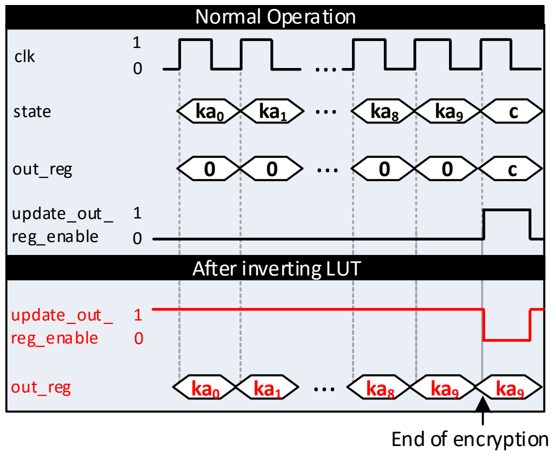


Fig. 10: Manipulation rule R_3 (*Invert LUT*), round-based design D_4, Due to the LUT inversion of the *update_out_reg_enable* control signal, the relevant output register *out_reg* is updated at the wrong clock cycles, i.e., the modified AES core fails to copy the correct ciphertext c and writes the leaking state ka_9.

By the aforementioned manipulation, the LUT output *update_out_reg_enable* is inverted, and the output register *out_reg* stores the cipher state after KeyAdd operation at all cipher rounds except the last one. Consequently, the state after the AddRoundkey at round 9 (i.e., ka_9) is given as faulty output instead of the correct ciphertext c. By examining hypothesis H_4 it can be tested whether such a LUT is hit which directly leads to key recovery.

5.1.5 Hitting the State Machine or Round Counter

One of the most common observed exploitable faults was due to manipulation of a LUT that (partially) processes the state machine or counter signals. As a consequence, in many cases the manipulated AES core finished its operations earlier than the fault-free case. Such a manipulation leads to the exposure of various intermediate values, e.g., the state after the j^{th} AddKey (ka_j). A representative example is illustrated in Figure 11.

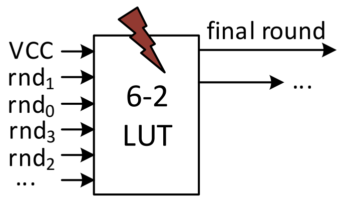


Fig. 11: Manipulation rule R_{13} (*Invert bits if HW* $\leq$ *15*), round-based design D_6, consequence: modification of AES round counter threshold.

We have observed that the manipulated LUT realizes the following function:

$$final\ round = \overline{rnd_0} \cdot rnd_3 \cdot rnd_1 \cdot \overline{rnd_2}. \quad (3)$$

Such a LUT controls the AES core to stop the operations when the round counter reaches $(rnd_3, rnd_2, rnd_1, rnd_0) = (1, 0, 1, 0) = 10_{dec}$, obviously corresponding to 10 cipher rounds of AES-128. Inverting certain bits of this LUT's content, e.g., by R_{13}, can lead to decrease or increase the number of rounds that the AES core operates.

Similarly, if manipulation rule R_2 (*Set LUT*) is applied, the targeted LUT (which e.g., makes the *DONE* signal) forces the AES core to terminate the operation right after the start, cf. Figure 12. The state machine control flow is therefore affected, and as a consequence an intermediate state (e.g., $p \oplus rk_0$) is given instead of the ciphertext.

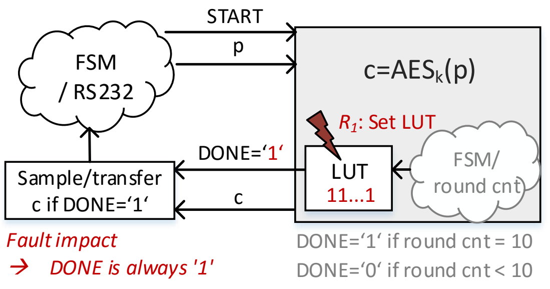

START
p
FSM / RS232
c=AES_k(p)
R1: Set LUT
DONE='1'
LUT 11...1
FSM/ round cnt
Sample/transfer c if DONE='1'
c
Fault impact → DONE is always '1'
DONE='1' if round cnt = 10
DONE='0' if round cnt < 10

Fig. 12: Manipulation rule R_2 (*Set LUT*), round-based design D_9, consequence: the AES core permanently observes *DONE*='1'.

6 Discussions and Countermeasures

One way to counter BiFI attacks might be to include built-in self tests (BIST) and conventional fault attack countermeasures such as redundancy and error-detection circuitry. However, not all countermeasures aimed at e.g. differential fault attacks are suitable for BiFI attacks and attacks especially aimed at overcoming a particular countermeasure might be able to defeat these. For example, let us assume a BIST as a countermeasure against BiFI. A BIST might prevent any manipulation attempt as conducted in this paper since it will detect that the encryption core is not functioning correctly. However, in such a case an attacker could try to perform a two-step BiFI attack: In the first step, the attacker tries to trigger a BIST failure by identifying/manipulating a LUT belonging to the AES circuitry.

The attacker then tries to disable the BIST by continuously modifying the bitstream until a faulty ciphertext is returned instead of the BIST failure. Although we did not verify this practically, the occurrence of a decision-making LUT, indicating a failure or success of the BIST, is likely. Manipulating such a LUT to always yield a "success" is then trivial and can be automatized by means of bruteforce.

Having figured out how to disable the BIST, in a second step one can proceed to apply the BiFI attack. Similar attack strategies might be applicable for other countermeasures. Which fault attack countermeasures are the most promising defenses against BiFI is still an open and interesting research question.

6.1 State of Bitstream Authentication

The most promising countermeasure against BiFI is to use a solid bitstream authentication scheme. The most recent Xilinx and Altera FPGAs have solid bitstream authentication and can therefore prevent such attacks. However, it is worth noting that actually most of the currently-deployed FPGAs do not support bitstream authentication. According to Altera's annual business report from 2014 [3] the peak production of an FPGA is roughly 6 years after introduction and the FPGAs are sold for more than 15 years. According to the annual reports of both Xilinx and Altera, around 50% of the revenue actually comes from older FPGA families which do not have bitstream authentication [3], [37]. Hence, our attack is applicable to a large share of FPGAs currently in use. It seems likely that it will take some time until FPGAs with bitstream authentication are widely used. It should also be noted that the bitstream encryption schemes of all Xilinx FPGAs except for the latest and rather expensive high-security devices (Kintex and Virtex Ultrascale), have shown to be vulnerable to side-channel attacks [17], [18], [20]. Similar to Xilinx, it was shown that Altera FPGAs are susceptible to side-channel attacks as well [19], [31]. For these seemingly protected schemes, overcoming the bitstream encryption is equivalent to bypassing the bitstream authentication. Therefore, the bitstream authentication of these devices could be defeated using a side-channel attack and subsequently a BiFI attack is possible again. In summary, the majority of currently-deployed Xilinx FPGAs appear to be vulnerable to the BiFI attack. In this case the aforementioned countermeasures such as BIST, redundancy and other fault attack countermeasures need to be employed. However, which countermeasures are the most effective and how well they might be defeated by advanced BiFI attack is yet unknown. Hence, moving to FPGAs with solid bitstream authentication therefore seems to be the most advisable countermeasure for security critical applications.

6.2 Impact on Other Fault Attack Types

This paper focused on generating exploitable permanent faults using bitstream manipulation. However, we would like to highlight that the presented approach to recover an AES key from permanent faults is not restricted to these attacks. Basically, several other fault techniques can be used to create similar exploitable faulty ciphertexts. For example, it was shown that laser fault attacks can also be used to change the configuration of the FPGA, e.g., in [7]. Typically, the goal of most laser fault attacks on FPGA designs is to cause transient faults in one round of the AES encryption to recover the key using a differential fault analysis. Permanent faults are usually not intended and are seen (e.g., in [7]) mainly as an obstacle from an attacker's perspective. However, the results in this paper show that permanent faults can actually be a very powerful attack vector. The key insight is that even random configuration errors (rule R_{15}) have a high chance to result in exploitable faulty ciphertexts. Hence, the same attack idea can also be performed with random (or targeted) laser fault injection. Clock glitch or power glitch during the configuration of the bitstream might also be used to cause such configuration faults. Therefore, investigating how the BiFI attack can be extended to other fault techniques or cryptographic algorithms is an interesting future research direction.

7 Conclusion

This paper introduces a new attack vector against cryptographic implementations on SRAM-based FPGAs. In this attack – so-called bitstream fault injection (BiFI) – the faults are injected repeatedly by configuring the target FPGA with malicious bitstreams. As a key insight of the BiFI attack, it can be automated so that no reverse-engineering of the target design is needed. Our attack, which is based on injecting permanent faults, is feasible in many practical realistic scenarios, where the attacker can manipulate the FPGA configuration and observe the faulty outputs of the target design. The adversary indeed manipulates the bitstream to alter and exploit the configuration maliciously.

Our experimental results with 16 AES implementations on a Spartan-6 FPGA showed that 15 out of these 16 designs could be successfully attacked in average in a few hours. The larger the design is (i.e., the more LUTs are utilized), the longer the attack takes. We furthermore showed that such a key recovery is even possible for *some* FPGAs when the bitstream encryption is enabled. The time required for the attack (in case of the encrypted bitstream) depends on the size of the underlying FPGA (more precisely on the number of available LUTs). It can range from hours (for low and mid-range FPGAs) up to weeks (for high-range FPGAs, e.g., with 1 million LUTs).

In short, the BiFI attack is non-invasive and requires neither a sophisticated setup nor a reverse-engineering tool. Indeed, it can be conducted by engineers without particular expertise. Hence, an interesting research question is how to develop the designs – rather than a sound integrity check – to defeat such attacks. For example, whether the concepts in [28], [36], developed mainly for software platforms, can be integrated into FPGA designs.

References

[1] Akashi Satoh. 2007. http://www.aoki.ecei.tohoku.ac.jp/crypto/web/cores.html.

[2] A. C. Aldaya, A. C. Sarmiento, and S. Sánchez-Solano. AES T-Box tampering attack. J. Cryptographic Engineering, 6(1):31–48, Springer, 2016.

[3] Altera. Altera annual report for form 10-k for 2014. https://www.sec.gov/Archives/edgar/data/768251/000076825115000008/altera10k12312014.htm, 2015.

[4] F. Benz, A. Seffrin, and S. Huss. BIL: A tool-chain for bitstream reverse-engineering. In Field Programmable Logic and Applications (FPL), pages 735–738. IEEE, Aug 2012.

[5] E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In Advances in Cryptology - CRYPTO '97, volume 1294 of Lecture Notes in Computer Science, pages 513–525. Springer, 1997.

[6] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In Advances in Cryptology - EUROCRYPT '97, volume 1233 of Lecture Notes in Computer Science, pages 37–51. Springer, 1997.

[7] G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, and M. Renaudin. Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA. J. Cryptology, 24(2):247–268, 2011.

[8] H. Choukri and M. Tunstall. Round Reduction Using Faults. In Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pages 13–14, 2005.

[9] Z. Ding, Q. Wu, Y. Zhang, and L. Zhu. Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation. Microprocessors and Microsystems - Embedded Hardware Design, 37(3):299–312, 2013.

[10] S. Drimer. Volatile FPGA design security – a survey (v0.96), April 2008.

[11] Fekete Balazs. 2014. http://opencores.org/project,aes_all_keylength.

[12] Hemanth. 2014. http://opencores.org/project,aes_crypto_core.

[13] Jerzy Gbur. 2011. http://opencores.org/project,aes_128_192_256.

[14] T. Kerins and K. Kursawe. A Cautionary Note on Weak Implementations of Block Ciphers. In Workshop on Information and System Security, page 12, Antwerp, BE, 2006.

[15] Y. Li, K. Sakiyama, S. Gomisawa, T. Fukunaga, J. Takahashi, and K. Ohta. Fault Sensitivity Analysis. In Cryptographic Hardware and Embedded Systems – CHES 2010, volume 6225 of Lecture Notes in Computer Science, pages 320–334. Springer, 2010.

[16] Michael Calvin McCoy. 2010. https://github.com/abhinav3008/inmcm-hdl/tree/master/AES/Basic_AES_128_Cipher.

[17] A. Moradi, A. Barenghi, T. Kasper, and C. Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from Xilinx Virtex-II FPGAs. In ACM Conference on Computer and Communications Security, pages 111–124. ACM, 2011.

[18] A. Moradi, M. Kasper, and C. Paar. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures - An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism. In The Cryptographers' Track at the RSA Conference, volume 7178, pages 1–18, Feb. 2012.

[19] A. Moradi, D. Oswald, C. Paar, and P. Swierczynski. Side-channel Attacks on the Bitstream Encryption Mechanism of Altera Stratix II: Facilitating Black-box Analysis Using Software Reverse-engineering. In ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, pages 91–100. ACM, 2013.

[20] A. Moradi and T. Schneider. Improved Side-Channel Analysis Attacks on Xilinx Bitstream Encryption of 5, 6, and 7 Series. In Workshop on Constructive Side-Channel Analysis and Secure Design. Springer, 2016.

[21] Moti Litochevski and Luo Dongjun. 2013. http://opencores.org/project,aes_highthroughput_lowarea.

[22] E. D. Mulder, P. Buysschaert, S. B. Ors, P. Delmotte, B. Preneel, G. Vandenbosch, and I. Verbauwhede. Electromagnetic Analysis Attack on an FPGA Implementation of an Elliptic Curve Cryptosystem. In EUROCON 2005 - The International Conference on "Computer as a Tool", volume 2, pages 1879–1882, Nov 2005.

[23] NSA. 1999. http://csrc.nist.gov/archive/aes/round2/r2anlsys.htm#NSA.

[24] G. Piret and J. Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In Cryptographic Hardware and Embedded Systems – CHES 2003, volume 2779 of Lecture Notes in Computer Scienes, pages 77–88. Springer, 2003.

[25] É. Rannaud. From the bitstream to the netlist. In Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays, pages 264–264, 2008.

[26] A. Schlösser, D. Nedospasov, J. Krämer, S. Orlic, and J. Seifert. Simple Photonic Emission Analysis of AES. J. Cryptographic Engineering, 3(1):3–15, 2013.

[27] S. P. Skorobogatov and R. J. Anderson. Optical Fault Induction Attacks. In Cryptographic Hardware and Embedded Systems – CHES 2002, volume 2523 of Lecture Notes in Computer Sciene, pages 2–12. Springer, 2002.

[28] B. Sunar, G. Gaubatz, and E. Savas. Sequential Circuit Design for Embedded Cryptographic Applications Resilient to Adversarial Faults. IEEE Transactions on Computers, 57(1):126–138, Jan 2008.

[29] P. Swierczynski, M. Fyrbiak, P. Koppe, A. Moradi, and C. Paar. Interdiction in practice—Hardware Trojan against a high-security USB flash drive. Journal of Cryptographic Engineering, pages 1–13, 2016.

[30] 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, 34(8):1236–1249, Aug 2015.

[31] P. Swierczynski, A. Moradi, D. Oswald, and C. Paar. Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs. ACM Trans. Reconfigurable Technol. Syst., 7(4):34:1–34:23, 2014.

[32] Tariq Ahmad. 2013. http://opencores.org/project,aes-encryption.

[33] S. Trimberger, J. Moore, and W. Lu. Authenticated encryption for FPGA bitstreams. In Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA '11, pages 83–86. ACM, 2011.

[34] S. M. Trimberger and J. J. Moore. FPGA Security: Motivations, Features, and Applications. Proceedings of the IEEE, 102(8):1248–1265, Aug 2014.
[35] I. Verbauwhede, D. Karaklajic, and J. M. Schmidt. The Fault Attack Jungle - A Classification Model to Guide You. In Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pages 3–8. IEEE, 2011.
[36] M. Werner, E. Wenger, and S. Mangard. Protecting the Control Flow of Embedded Processors against Fault Attacks. In Smart Card Research and Advanced Applications, pages 161–176. Springer, 2015.
[37] Xilinx. Xilinx annual report for form 10-k for 2014. https://www.sec.gov/Archives/edgar/data/743988/000074398815000022/xlnx0328201510k.htm, 2015.
[38] D. Ziener, S. Assmus, and J. Teich. Identifying FPGA IP-Cores Based on Lookup Table Content Analysis. In Field Programmable Logic and Applications, 2006. FPL '06. International Conference on, pages 1–6, Aug 2006.

8 Appendix

8.1 Testing the Bitstream Encryption Vulnerability of Xilinx FPGAs

In an initial step, we used the Xilinx tool to generate an encrypted bitstream for a Virtex-5 FPGA with enabled CRC-check (*bs_enc_crc_on*). For a quick test, we randomly modified one encrypted block in the bitstream and tried to configure the corresponding Virtex-5 FPGA. As expected, the FPGA refused to configure the manipulated bitstream. In the next step, we generated another encrypted bitstream for the same FPGA design and using the same key k and the same IV, but with disabled CRC-check (*bs_enc_crc_off*).

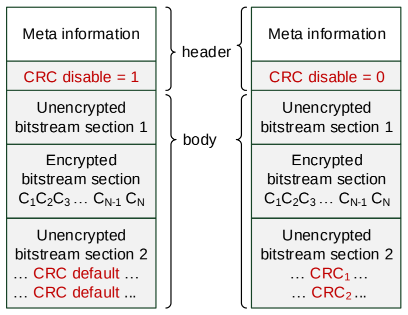

Meta information
CRC disable = 1
Unencrypted bitstream section 1
Encrypted bitstream section C1C2C3 … CN-1 CN
Unencrypted bitstream section 2
… CRC default …
… CRC default …
header
body
Meta information
CRC disable = 0
Unencrypted bitstream section 1
Encrypted bitstream section C1C2C3 … CN-1 CN
Unencrypted bitstream section 2
… CRC1 …
… CRC2 …

Fig. 13: Bitstreams Virtex-5 (VLX50), left: encryption enabled and CRC off (*bs_enc_crc_off*), right: encryption enabled and CRC on (*bs_enc_crc_on*).

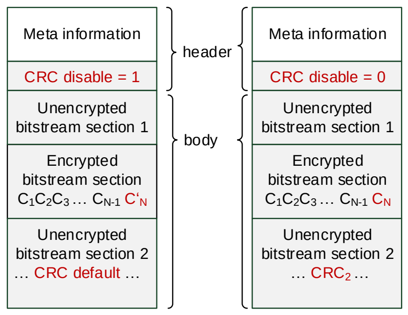

Meta information
CRC disable = 1
Unencrypted bitstream section 1
Encrypted bitstream section C1C2C3 … CN-1 C'N
Unencrypted bitstream section 2
… CRC default …
header
body
Meta information
CRC disable = 0
Unencrypted bitstream section 1
Encrypted bitstream section C1C2C3 … CN-1 CN
Unencrypted bitstream section 2
… CRC2 …

Fig. 14: Bitstreams Spartan-6 (SLX75), left: encryption enabled and CRC off (*bs_enc_crc_off*), right: encryption enabled and CRC on (*bs_enc_crc_on*.)

Disabling the CRC-check means that two 22-bit CRC values toggle to a fixed default sequence (defined by Xilinx) and that one bit toggles in the bitstream header which is responsible for disabling/enabling the CRC check.

The comparison of both encrypted bitstreams *bs_enc_crc_on* versus *bs_enc_crc_off* revealed that only the unencrypted parts of the file are different, i.e., the encrypted blocks are identical, cf. Fig. 13. Therefore, we concluded that the encrypted parts of the bitstream do not contain the checksum. Otherwise, due to the default CRC sequence at least one encrypted block would be different.

To evaluate our findings on Virtex-5, we i) first observed all bit toggles due to *bs_enc_crc_on* versus *bs_enc_crc_off*, then ii) applied the same bit toggles on the target encrypted bitstream (with enabled CRC-check), iii) applied various manipulation rules (Section 4.3), and finally iv) configured the manipulated bitstream into the FPGA device. It turned out that the Virtex-5 FPGA accepted the manipulated bitstream, and hence, there is no appropriate integrity check leading to feasible BiFI attacks.

We repeated the same experiment on a Spartan-6 (SLX75) FPGA and noticed that one (out of two) CRC sequences is part of the last encrypted block C_N, cf. Fig. 14. Therefore, for these kind of bitstreams the integrity is ensured providing protection against BiFI attacks unless side-channel attacks are used to recover the underlying bitstream encryption key.

One further observation we made is that the Spartan-6 FPGAs in general denies any encrypted bitstream with disabled CRC-check. This is not the case for Virtex-5.