FPGA Trojans through Detecting and Weakening of Cryptographic Primitives

Pawel Swierczynski*, Marc Fyrbiak*, Philipp Koppe*, and Christof Paar*†, *Fellow, IEEE*
*Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
†University of Massachusetts Amherst, USA
{*pawel.swierczynski,marc.fyrbiak,philipp.koppe,christof.paar*}@rub.de

***Abstract*—This paper investigates a novel attack vector against cryptography realized on FPGAs, which poses a serious threat to real-world applications. We demonstrate how a targeted bitstream modification can seriously weaken cryptographic algorithms, which we show with the examples of AES and 3DES. The attack is performed by modifying the FPGA bitstream that configures the hardware elements during initialization. Recently, it has been shown that cloning of FPGA designs is feasible, even if the bitstream is encrypted. However, due to its proprietary file format, a meaningful modification is very challenging. While some previous work addressed bitstream reverse-engineering, so far it has not been evaluated how difficult it is to detect and modify cryptographic elements. We outline two possible practical attacks that have serious security implications. We target the S-boxes of block ciphers that can be implemented in look-up tables or stored as precomputed set of values in the memory of the FPGA. We demonstrate that it is possible to detect and apply meaningful changes to cryptographic elements inside an unknown, proprietary and undocumented bitstream. Our proposed attack does not require any knowledge of the internal routing. Furthermore, we show how an AES key can be revealed within seconds. Finally, we discuss countermeasures that can raise the bar for an adversary to successfully perform this kind of attack.**

***Keywords*—Hardware security, FPGAs, Trojans, bitstream manipulation, reverse-engineering, DES, AES.**

I. Introduction

FIELD-Programmable Gate Arrays (FPGAs) play an important role in the field of embedded systems. They are used in a wide spectrum of applications, e.g., computer networks, data centers, automation, signal processing and the automotive industry. Many of these applications are security-sensitive and use FPGAs for cryptographic operations such as random number generation, key establishment, digital signatures as well as encryption. Despite a large body of research addressing various aspects of FPGA security [1], the issue of maliciously manipulating the configuration data of FPGAs has not been addressed to our knowledge. During initialization, the so-called bitstream is loaded into the FPGA, which configures the internal hardware elements. The majority of FPGAs used in practice employ bitstreams that are stored externally, e.g., on dedicated flash chips. This setup provides an unfortunate attack surface which allows to learn about the security mechanisms implemented, and more damaging, to introduce Trojan-like manipulations of the hardware. Even though the two market leaders, Altera and Xilinx, offer bitstream encryption as a security measure, the schemes of both have been broken [2], [3], [4]. These attacks leak the symmetric encryption keys stored inside the FPGA utilizing side-channel analysis. After key extraction, the encrypted bitstream stored in the external flash can be read and decrypted. It is also possible to re-encrypt a modified bitstream and load it into the FPGA to ultimately change its hardware configuration.

Even though it can be assumed that the bitstream is known to an adversary, there are still two major problems: the attacker has to overcome a considerably obfuscation hurdle (proprietary bitstream) and has to find the cryptographic components in a large FPGA design. The bitstreams of all commercial FPGAs make use of proprietary file formats. It is neither documented which parts of the file belong to which hardware components within the FPGA nor how different bits of the file influence the specific configuration. There has been research on bitstream reverse-engineering to uncover some of the bitstream features. Nonetheless, it is not publicly documented what the file format details of popular commercial FPGA vendors are. Even with a full understanding of the bitstream, it poses a great challenge for an attacker to detect and identify cryptographic components within an unknown design. However, this is a prerequisite for "meaningful" manipulations. To the best of our knowledge, the only previous work in this direction is by Chakraborty et al. [5]. They proposed a technique which allows to merge new logic into an existing bitstream. The inserted logic is restricted to unused logic blocks, meaning the inserted logic has to be completely distinct from the existing one. The fundamental limitation of the approach is the inability to interact with or modify the existing design.

In this paper, we introduce methods to detect and manipulate crucial cryptographic components like S-boxes in the bitstream. These can either be implemented as lookup-table or they can be stored in the embedded memory. The applied modifications weaken the cryptographic algorithm or leak (parts of) the key, while we require no knowledge of the internal routing information. We demonstrate our approach with AES, DES, and Triple-DES, which are most widely used block ciphers in current and legacy applications. The weakened algorithms are incompatible with their genuine counterparts.

Part of the research was conducted at the University of Massachusetts Amherst. This material is based upon work partially supported by the National Science Foundation under Grant No. CNS-1318497. This work has been also partially supported by the Hans L. Merkle Foundation.

Thus, the attack is limited to scenarios in which encryption and decryption are computed by the same device, e.g., USB flash drives, solid-state disks or in encrypted cloud storage. The manipulations can also be used in systems in which all involved devices can be altered. We practically verified our techniques on a commercially available FPGA. Finally, we briefly discuss countermeasures to raise the bar for an attacker.

II. First Point of Attack: Look-up Tables in Field Programmable Gate Arrays

Lookup-tables (LUTs) are one of several element types embedded in an Field Programmable Gate Array (FPGA). They are responsible for implementing the logic of a design. When combining LUTs with multiplexers, an FPGA can implement combinatorial and more complex logic functions. FPGAs use thousands of LUTs that can implement either logic functions or serve as distributed Random Access Memory (RAM). Usually two or four LUTs are embedded in a "logic block". As depicted in Figure 1, a group of logic blocks is connected to a switch-box. The switch-box is used for managing all wire connections.

The outputs of the switch-boxes are connected with the input pins of the LUTs or with the embedded multiplexers. Thus, the output of the switch-box provides the permutation of the input bits of any LUT.

Since LUTs represent the primary logic in an FPGA, they are promising targets for an attacker that wants to maliciously change the functionality of a third-party FPGA design. This is especially critical in cryptographic applications. Thus, it is also quite important to analyze the LUTs contents in terms of security. In the real world an attacker usually only possesses the bitstream of an FPGA design, but not the corresponding netlist. We have practically verified that an attacker is able to detect and modify the appropriate bits in order to change a genuine bitstream to a malicious version.
For this purpose, an adversary needs to know details of the (proprietary) bitstream mapping that is responsible for configuring the LUT contents. To be more precise, the bitstream file format has to be partially reverse-engineered.

Section II-A provides detailed information on how to fully reverse-engineer all LUT bit positions. Note that the LUT bits are distributed over the bitstream following specific and unknown patterns. We successfully obtained the bit mapping of two devices that are based on a 4-bit-to-1 bit and a 6-bit-to-1 bit LUT architecture. Note that the following approach can be applied for most of those FPGAs belonging to the same vendor.

A. Extracting the LUT Mapping From a Bitstream

As an example, we consider an FPGA that uses a 6-bit-to-1 bit architecture. We assume the following properties:

- Four 6-bit-to-1 bit LUTs are embedded in one logic block with the ability to store 4×64 bits.
- Three multiplexers that can combine LUT outputs.

To extract the bitstream mapping of all LUTs, an attacker simply conducts a profiling phase. The approach of learning the LUT contents from a bitstream relies on generating appropriate netlists that specify the rules of reconfiguring the hardware for the given FPGA target. The netlist can be used to manually configure any LUT with any arbitrary 6-input Boolean function. To give an example, Listing 1 shows the configuration of four LUTs. Note that the presented netlist uses a fictional syntax.

Listing 1: Netlist example for setting LUT contents

```
FPGA design "minimal_lut_implementation" ...,
  other configuration "...";
instance "logic block X Y",
config {
 LUT1 = {0x0000000000000000} //64 inputs output 0
 LUT2 = {0xFFFFFFFFFFFFFFFF} //64 inputs output 1
 LUT3 = {0x0000000000000001} //Last input outputs 1
 LUT4 = {0x8000000000000000} //First input outputs 1
}
```

As further illustrated by Listing 1, each LUT of one logic block can be configured by specifying a 64-bit LUT content representing a Boolean function.

In the following, we describe a generic reverse-engineering strategy that reveals all LUT bit positions from a bitstream. An attacker configures two different values for exactly one LUT, thus, she has to create two different netlists (c.f., Listing 1). The first netlist configures a LUT, whose output is always a logical zero for all 64 input values (6-bit-to-1 bit architecture). It should be noted that for each input value one output bit has to be stored. All outputs bits together (64-bit) form a LUT content. In this case, 64 "0"-bits, which is the resulting LUT content of the currently discussed Boolean function, are stored in the bitstream. Analogously, in a 4-bit-to-1 bit architecture (16 input values), only sixteen "0"-bits are stored in the bitstream due to less input value entries.

In the next step, a second netlist is created that only differs in the specified LUT content. Instead of outputting zeros only, the function is chosen in such a way that it always outputs a one regardless of the input value. Again, the corresponding bitstream is generated. This leads to the storage of 64 "1"-bits in the bitstream.

When comparing both bitstreams, one can observe that exactly 64 bits toggle from "0" to "1", while all other bits remain unchanged. Therefore, one can easily determine and store the mapping rules of all 64 bits that are related to one LUT, but obviously the correct order of these 64 bits stays unclear. It is important to know the correct order to be able to reconstruct the correct Boolean function. Thus, an attacker has to extend the previous approach: Now, the idea is to additionally create 64 bitstreams from 64 different netlists.

Each netlist configures an appropriate value (c.f. Table I) for the same LUT such that only one bit of the LUT content is set, while all other 63 bits are cleared. All 64 generated bitstreams can be compared with the bitstream, whose LUT content bits are all cleared, because then only one bit toggles.

To be more precise, each LUT content bit is recovered separately by observing the toggling positions, and thus, the correct order can be revealed. In a 6-bit-to-1 bit architecture, one needs to generate 65 bitstreams for each LUT, while for a 4-bit-to-1 bit architecture only 17 bitstream generations are

sufficient. This approach has to be repeated for all given LUTs of the underlying FPGA in order to be able to extract all LUT contents from a third-party bitstream.

Generation of	Content of exactly one LUT
Bitstream 1	0x0000000000000001 //Only input 0 outputs a 1
Bitstream 2	0x0000000000000002 //Only input 1 outputs a 1
Bitstream 3	0x0000000000000004 //Only input 2 outputs a 1
...	...
Bitstream 63	0x4000000000000000 //Only input 62 outputs a 1
Bitstream 64	0x8000000000000000 //Only input 63 outputs a 1
Bitstream 65	0x0000000000000000 //Each input outputs a 0

TABLE I: Generating 65 bitstreams for one LUT

Note that the bits of one LUT, as indicated by Table I, are not necessarily stored next to each other in the bitstream. Rather, they could be distributed in the bitstream file by following specific offsets rules. To give an example, the first bit of one LUT content can be stored in the bitstream at position (Byte Y, Bit 0), while the second bit may be located at position (Byte $Y - 8$, Bit 5). We were able to practically verify the correctness of our recovered bitstream mapping for any single LUT. This can be done by setting a random configuration for any LUT (in a netlist describing all LUTs) and by creating the corresponding bitstream. Then, the LUT contents can be parsed from the bitstream and compared to the LUT contents of the previously generated netlist.

Algorithm 1 illustrates that straightforward but time-consuming approach in more detail. A more sophisticated and considerably faster method is to learn the offset patterns of one or several LUTs that can be applied to all other LUTs. For a mid-sized FPGA the computation time, then is approximately 1-2 days, whereas the straightforward approach needs much longer. The profiling phase has to be performed only once per device. The paper's intention is not to provide detailed insights into the bitstream file format. Rather, it illustrates the feasibility of the approach. In summary, this approach is generic and is always applicable using the standard FPGA design flow. The next sections deal with the detection and impacts of a potential adversary's modification to weaken third-party cores in bitstreams.

Algorithm 1 LUT content extraction for a 6-bit-to-1 bit FPGA architecture

Input: FPGA device file describing LUTs
Output: Bitstream position table of LUT content

set_lut_content(·) sets the LUT content in netlist file
bs_ref: Reference file with zeroized LUT content
bs_mod: File with specific LUT content

```
1:  for lut_index = 0 to num_of_luts - 1 do
2:      Create bitstream bs_ref with zeroised LUT content
        for the LUT lut_index
3:      for bit = 0 to 2^6 − 1 do
4:          lut_content = 2^bit
5:          set_lut_content(bs_mod, lut_index, lut_content)
6:          Generate bitstream bs_mod
7:          Compare bs_ref and bs_mod
8:          Store difference in position_table[lut_index][bit]
9:      end for
10: end for
11: return position_table
```

III. Detecting S-boxes in FPGA Bitstreams

In this section, several S-box detection approaches are discussed for the Data Encryption Standard (DES) and Advanced Encryption Standard (AES). Besides a search pattern strategy for specific S-box implementations, general strategies to identify non-linear elements in FPGA bitstreams are described. We note that AES is the most widely used symmetric cipher and 3DES, which is based on DES, is popular in both legacy applications, e.g., in banking, and in recent systems such as the e-Passport.

A. Detection of DES S-boxes

This section covers the detection of DES S-boxes from a bitstream. The corresponding FPGA design is based on a 6-bit-to-1 bit architecture. The DES algorithm is described in Section V in more detail. DES uses eight different predefined 6-bit-to-4 bit S-boxes. Since our target device provides 6-bit-to-1 bit LUTs, one DES S-box column[1] fits into one LUT. Therefore, one complete S-box (4 columns) can be realized by four LUTs. Hence, a round-based DES implementation requires 32 LUTs for all eight S-boxes. A general 6-bit-to-4 bit S-box is illustrated in Table II. Note that each column LUT_1, LUT_2, LUT_3, and LUT_4 stores a unique 64-bit sequence describing a Boolean equation. These might be the fixed bit-sequences of a DES S-box. Each value a_i, b_i, c_i, and d_i with $i \in \{1, ..., 64\}$ stores exactly one bit.

Input values						Output columns			
i_6	i_5	i_4	i_3	i_2	i_1	LUT_1	LUT_2	LUT_3	LUT_4
0	0	0	0	0	0	a_1	b_1	c_1	d_1
0	0	0	0	0	1	a_2	b_2	c_2	d_2
...	...	...	...	...	...	...	...	...	...
1	1	1	1	1	1	a_{64}	b_{64}	c_{64}	d_{64}

TABLE II: General shape of a 6-bit-to-4 bit S-box

To give an example, the four patterns of the first DES S-box are as follows.

- $LUT_1 = (a_{64}, ..., a_1) =$ 0x869D497A86E67619
- $LUT_2 = (b_{64}, ..., b_1) =$ 0xB0C7871B497826BD
- $LUT_3 = (c_{64}, ..., c_1) =$ 0x27E9D492609F1F29
- $LUT_4 = (d_{64}, ..., d_1) =$ 0x917BE9066F81B478

Note that each 64-bit pattern is unique for each DES S-box. An adversary can learn the bitstream mapping for all LUTs, thus she can now analyze the extracted contents. As stated above, an attacker may search for the presented patterns. Specifically, all

[1] It is equal to the LUT content describing one output bit of the S-box. A DES S-box column can be understood as $LUT_1 = (a_{64}, a_{63}, \ldots a_1)$, c.f., Table II.

720 of its permutations have to be examined, due to possible input transposition.

The synthesis tools determine an optimal routing path. For this purpose, the internal algorithms permute the input bits of a LUT. This also leads to permuted output bits in the FPGA design and the corresponding bitstream, respectively. Due to this fact, the LUT content, e.g., has to be viewed as $\mathrm{perm}(LUT_1)$ instead of LUT_1. One may think that an attacker needs further knowledge of the FPGA's routing to obtain the given unknown input permutation (denoted by $perm(\cdot)$), but this is not necessary due to the uniqueness of DES S-box patterns: the basic idea is to precompute the possible input permutations for all given DES patterns and to compare them with all extracted LUTs. If there is one match, the permutation is successfully recovered. The corresponding DES pattern search algorithm is depicted in Algorithm 2.

Algorithm 2 DES S-box detection for a 6-bit-to-1 bit architecture

Inputs: Bitstream *bs*, Bitstream position table of LUT content
Output: File with localized DES LUTs

$S_1(x), S_2(x), ..., S_8(x)$ represent the DES S-boxes
$S_p^j(x)$ denotes to the j'th output bit of $S_p(x)$
$\mathrm{perm}_i(\cdot)$ denotes the i'th permutation out of 720
get_lut_content(·) extracts the LUT content of the bitstream
mark_lut(·) writes the parameter to an output file

```
//Generate DES search patterns
for sbox = 0 to 7 do
    for output_bit = 0 to 3 do
        des_pattern[sbox][output_bit] =
            S_sbox^output_bit(63)|S_sbox^output_bit(62)|...|S_sbox^output_bit(0)
    end for
end for
//Get LUT content of bitstream by using Algorithm 1
LUT[num_of_luts] ← get_lut_content(bs)
//Search for DES pattern
for lut_index = 0 to num_of_luts - 1 do
    for perm_index = 0 to 719 do
        for sbox = 0 to 7 do
            for output_bit = 0 to 3 do
                if (perm_perm_index(LUT[lut_index])
                  == des_pattern[sbox][output_bit]) then
                    mark_lut(lut_index, sbox, output_bit)
                end if
            end for
        end for
    end for
end for
```

In practice, we were able to detect all S-box instances. Besides to the exact location of the LUTs on the FPGA's grid, we obtained the exact permutation order of the corresponding input pins (without any knowledge of the routing) for every single S-box column. The obtained knowledge is extremely useful for an attacker, e.g., if side-channel attacks based on electromagnetic emanation are used. Knowing the exact location an attacker can try to locate the best probe position for the measurement while a target device performs its cryptographic operations. The bitstream can also expose information about the utilized architecture of the design. Knowing the architecture can indicate, whether an implementation is round-based, unrolled, or whether other cryptographic instances run in parallel. Table III illustrates that we were able to locate all DES S-boxes from three tested FPGA implementations. Note that one can also easily identify the S-boxes of a Triple-DES (3DES) architecture. The Algorithm 2 can also be applied

Impl.	Architecture	Found LUTs	Detection rate
#1	Round-based	32	100 %
#2	Round-based	32	100 %
#3	Unrolled (16 rounds)	$16 \cdot 32$	100 %

TABLE III: Overview of evaluated DES implementations

to a 4-bit-to-1 bit LUT FPGA architecture. In this case, we evaluated whether one can also detect the corresponding 4-bit-to-4 bit S-boxes of the lightweight cipher PRESENT [6]. We could again identify all S-box instances from the bitstream. As long as a Boolean function candidate is known to an attacker, she is able to search for it in the bitstream. Since the S-boxes are usually the only non-linear function of a block cipher, they represent a potential security risk from a cryptographic perspective, if they can be altered by an attacker. Under certain conditions, the identification of S-box columns can be more challenging as discussed in the following.

B. Generalization of Arbitrary y-bit-to-1 bit LUTs

If the FPGA's architecture uses y-bit-to-1 bit LUTs (2^y bits of memory) and x-bit-to-1 bit Boolean functions (2^x bits of memory) need to be synthesized, two cases may occur:

a) Case 1: $x \leq y$: If x is less than or equal to y, then the whole S-box column is placed in exactly one LUT. The LUT contents can be matched with the reference patterns as described in Algorithm 2. It is thus straightforward to detect single x-bit-to-1 bit S-box columns. This case holds, e.g., for the S-boxes of the DES algorithm in a 6-bit-to-1 bit architecture ($x = y = 6$).

b) Case 2: $x > y$: If x is larger than y, then it is a more challenging task to find x-bit-to-1 bit S-box columns within an FPGA design. Due to the dimensions, one S-box column must be split into $\frac{2^x}{2^y}$ LUTs that have to be combined by $\frac{2^x}{2^y} - 1$ multiplexers. We have developed a search strategy for S-box columns that exceed the common 16-bit (4-bit-to-1 bit) and 64-bit (6-bit-to-1 bit) memory limitations of one LUT. This technique is described for the AES in Section III-C. AES uses 8-bit-to-8 bit (eight 8-bit-to-1 bit functions) S-boxes and thus $\frac{2^8}{2^6} = 4$ LUTs implement one S-box column in a 6-bit-to-1 bit architecture.

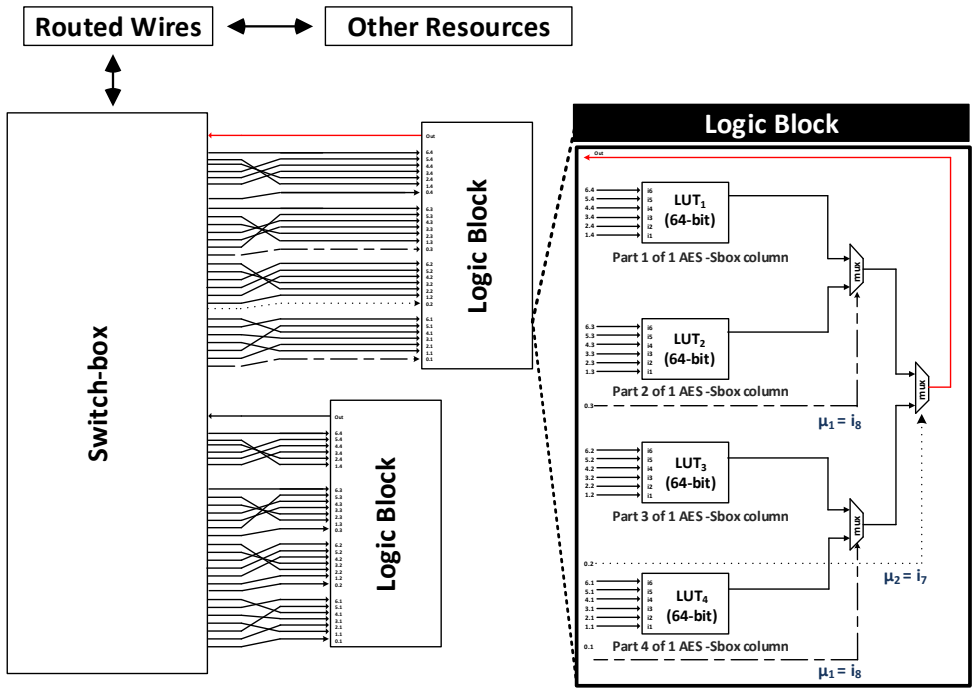

Fig. 1: Simplified overview of a logic block realizing an 8-bit-to-1 bit Boolean function (256 bits of memory) with four 6-bit-to-1 bit LUTs (64 bits of memory each)

C. Detection of AES S-boxes

In the following, the detection of decomposed AES S-boxes, which are realized in a 6-bit-to-1 bit architecture, is considered. At first the attacker computes the 8-bit-to-8 bit AES S-box. The decomposition is necessary, because the degree x of the Boolean function is higher than the number of LUT inputs y. Thus case 2 of the previous section holds here. Each out of the 8 AES S-box columns represents an 8-bit-to-1 bit Boolean function and is stored as a 256-bit value denoted by $(a_{256}, a_{255}, ..., a_1)$.

Such an AES S-box column is similar to the bit-string of Table II but is larger (256 bits instead of 64 bits). The AES S-box columns have to be split-up into 4 LUTs each, because one LUT of the FPGA can only store 64 bits. All LUTs are denoted by LUT_i with $i \in \{1, 2, 3, 4\}$ that need to be multiplexed with 2 input bits. They are denoted by μ_1 and μ_2. The purpose of these two input bits is to select one of the 4 lookup-tables LUT_i with the help of 3 multiplexers, c.f., Fig. 1. The synthesizer has to chose 64 bits from $(a_{256}, a_{255}, ..., a_1)$ and assigns them to one LUT. An AES S-box column has eight input bits that we denote by (i_8, i_7, ..., i_1). The straightforward example of how to distribute the 256 bit values $(a_{256}, a_{254}, ..., a_1)$ to 4 LUTs is illustrated in the following.

- $LUT_1 = (a_{256}, ..., a_{193})$ sel. by $(\mu_1, \mu_2) = (0, 0)$
- $LUT_2 = (a_{192}, ..., a_{129})$ sel. by $(\mu_1, \mu_2) = (0, 1)$
- $LUT_3 = (a_{128}, ..., a_{65})$ sel. by $(\mu_1, \mu_2) = (1, 0)$
- $LUT_4 = (a_{64}, ..., a_1)$ sel. by $(\mu_1, \mu_2) = (1, 1)$
- The multiplexer configuration is $\mu_1 = i_8$ and $\mu_2 = i_7$
- The remaining input bits (i_6, i_5, ..., i_1) are not permuted

If the multiplexer configuration is different, then the assignment of $(a_{256}, a_{255}, ..., a_1)$ to the lookup-tables LUT_1, LUT_2, LUT_3, and LUT_4 has to be re-organized by the synthesizer. An attacker would observe that the tools proceed as follows: for each AES S-box input value $i \in \{0, ..., 255\}$, for which $(\mu_1, \mu_2) = (0, 0)$ holds, add the corresponding bit (one bit of $(a_{256}, ..., a_1)$) to the same LUT group. This is repeated for $(\mu_1, \mu_2) \in \{(0, 1), (1, 0), (1, 1)\}$. The contents of LUT_i can vary due to one out of $6! = 720$ possible permutations. Also, there are $\binom{8}{2}$ possibilities to pick two multiplexer bits (μ_1, μ_2) from (i_8, i_7, ..., i_1). So there are $\binom{8}{2} \cdot 720 \cdot 4 = 80640$ patterns for one AES S-box column. To be able to search for all 8 AES S-box output columns, one needs to generate $8 \cdot 80640 = 645120$ patterns in total. Algorithm 3 provides the necessary steps for detecting all AES S-boxes from the bitstream.

From an attacker's point of view, it is an advantage that all four LUTs are placed within one logic block. This property simplifies the detection of one single AES S-box column.
These results are a proof-of-concept that in many cases an attacker only has to reverse-engineer the LUT content part of the bitstream, and does not need any further knowledge about the routing to be able to detect and modify S-boxes. Thus, the reverse-engineering effort is minimal. Nevertheless, there are various ways to implement the S-boxes in an FPGA design and thus the presented detection method does not always neccessarily lead to a successful finding. Note that in some cases, an adversary needs to figure out the exact input permutation and multiplexer configuration of a logic block,

Algorithm 3 AES S-box detection for a 6-bit-to-1 bit FPGA architecture

Input: Bitstream *bs*, Bitstream position table of LUT content
Output: File with localized AES LUTs

$S^j(x)$ denotes the j'th output bit of $S(x)$
$\mathrm{perm}_i(\cdot)$ denotes the i'th permutation of all 6!
get_lut_content(·) extract the LUT content of the bitstream
mark_lut(·) writes the parameter to output file

```
//Generate AES search patterns
for sbox_bit = 0 to 7 do
    for mux_cfg = 0 to (8 2) − 1 do
        Pick multiplexer configuration (μ1, μ2)
        Set cnt_i to 0, i = 1,2,3,4
        for i = 0 to 255 do
            switch(get_mux_value(i))
            case(0,0):
        LUT1[mux_cfg][sbox_bit][cnt1++] = S^sbox_bit(i)
            case(0,1):
        LUT2[mux_cfg][sbox_bit][cnt2++] = S^sbox_bit(i)
            case(1,0):
        LUT3[mux_cfg][sbox_bit][cnt3++] = S^sbox_bit(i)
            case(1,1):
        LUT4[mux_cfg][sbox_bit][cnt4++] = S^sbox_bit(i)
            end switch
        end for
    end for
end for
//Get LUT content of bitstream by using Algorithm 1
LUT[num_of_luts] ← get_lut_content(bs)
//Search for AES pattern
for lut_index = 0 to num_of_luts do
    for sbox_bit = 0 to 7 do
        for mux_cfg = 0 to (8 2) − 1 do
            for perm_index = 0 to 719 do
                for i = 0 to 255 do
                    if perm_perm_index(LUT[lut_index])
        == LUT_{i/64}[mux_cfg][sbox_bit][i mod 64] then
                        mark_lut(lut_index, sbox_bit,
                                 mux_cfg, perm_index)
                    end if
                end for
            end for
        end for
    end for
end for
```

while for some other potential detection strategies this is not necessary at all. These cases are discussed in Section VI and V in more detail.

To verify our results practically, we evaluated 8 publicly available AES cores on a 6-bit-to-1 bit FPGA in order to study the corresponding bitstreams. These cores are offered by OpenCores[2]. With the help of Algorithm 3, we were able to detect all S-boxes being implemented by LUTs. The corresponding results are given in Table IV.

Impl.	Architecture	AES	LUTs with S-box logic	S-boxes in memory	Detection
#1	Round-based	128	**(16+4)**$\cdot 32 = 640$	no	100 %
#2	$\frac{1}{4}$ Round	128	0	yes	100 %
#3	$\frac{1}{4}$ Round	192	0	yes	100 %
#4	$\frac{1}{4}$ Round	256	0	yes	100 %
#5	Round-based	128	**(0+4)** $\cdot$ 32 $= 128$	yes	100 %
#6	Round-based	128	0	yes	100 %
#7	Round-based	128	0	yes	100 %
#8	Round-based	128	**(16+4)**$\cdot 32 = 640$	no	100 %

TABLE IV: Overview of evaluated AES implementations

Implementation #1 and #8 used 640 LUTs for realizing AES S-boxes, while the S-boxes of implementation #2, #3, #4, #6, and #7 were placed in the embedded memory of the FPGA. It should be noted that in Sect. IV we explain how to extract the S-boxes from the embedded memory.

During our evaluations, we observed that only implementation #5 used both parts of the hardware, namely the embedded memory and the offered 6-input LUTs. In this implementation, 4 S-boxes were used for the key schedule step of AES, while all other S-boxes were placed in the embedded memory.

As already mentioned before, implementation #1 and #8 use exactly 640 LUTs belonging to AES S-boxes. Having extracted such information from the bitstream, it can be inferred that a round-based AES implementation is used since there are $20 = \frac{640}{32}$ synthesized S-box instances of the AES algorithm. By obtaining such a result, an attacker can guess that sixteen S-boxes belong to the processing of the AES SubBytes step, while the other four S-boxes are synthesized for the key schedule step of AES, which we afterwards confirmed with the help of the netlist. It should be noted that we were also able to identify all sixteen inverse S-boxes belonging to the AES decryption.

D. Exploiting the Non-linearity of LUT Contents

In this section, another potential approach of detecting decomposed S-boxes is introduced. In particular their non-linear nature is exploited. One advantage is that the permutation configuration of the input pins do not have to be considered anymore. We show that measuring the degree of linearity of all LUT contents may also be helpful for obtaining information from third-party FPGA designs.

As mentioned in the previous sections, the analysis of LUT content of a bitstream can reveal valuable information for an attacker. A block cipher can be implemented using several strategies. Each implementation strategy has an inherent characteristic that may be revealed through measuring the linearity of certain components of the design. For example,

[2]http://opencores.org/

in an unrolled design, it is common that many more S-box instances are needed, compared to the number of S-boxes in an iterated design. A central requirement for S-boxes is that they possess a high degree of non-linearity. This is a necessary characteristic in order to prevent cryptanalysis. Thus, the more S-box instances are used, the more LUTs that realize non-linear Boolean functions can be expected, if the S-box is implemented in LUTs. To measure the linearity of a LUT content, we use the *Walsh Coefficient*. It is a well-established measure, and we use in the following the notation from [7].
For two vectors $a, b \in \mathbb{F}_2^n$, we denote the inner product of a and b by

$$\langle a, b \rangle = \sum_{i=0}^{n-1} a_i b_i$$

We call a LUT a *non-linear LUT* if its corresponding content is non-linear according to the *Walsh Coefficient*. A large number of non-linear LUTs may indicate the usage of S-boxes. For a Boolean function in n variables $f\colon \mathbb{F}_2^n \to \mathbb{F}_2$ the *Walsh Coefficient* is defined by

$$\mathrm{wal}_f(a) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x)+\langle a,x \rangle}$$

Note that the function f is the LUT content representation. For an FPGA with 6-bit-to-1 bit LUTs, the function f is a Boolean function $f\colon \mathbb{F}_2^6 \to \mathbb{F}_2$. The linearity of the Boolean function f is denoted by

$$\mathrm{Lin}(f) = \max_{x \in \mathbb{F}_2^n} |\mathrm{wal}_f(x)|$$

If $\mathrm{Lin}(f)$ is large, this means that there exists an affine or linear function that is a good approximation to the function f. Having introduced the *Walsh Coefficient*, this measure is used to evaluate the AES design of Section III-C.

1) Evaluation of an AES implementation on a 6-bit-to-1 bit Architecture: In order to evaluate the suitability of the *Walsh Coefficient*, we provide the corresponding results for an AES design (implementation #8, c.f., Table IV) that uses 20 S-box instances, implemented in LUTs. The results are depicted in Fig. 2. The x-axis represents the *Walsh Coefficient* that ranges from 16 to 64. Note that a *Walsh Coefficient* of 16 represents a low degree of linearity, while a value of 64 indicates a very high degree. The y-axis provides the number of occurrences regarding the LUTs of the underlying FPGA design.

As one can see in Fig. 2, there are 728 LUTs[3] possessing a *Walsh Coefficient* smaller or equal to 28 (low degree of linearity). With the help of the detection approach of Algorithm 3, we observed only two S-box LUTs having a higher linearity (*Walsh Coefficient* of 26) than expected. Nevertheless, they basically fit into the set of non-linear LUTs. An attacker can

[3]It is the sum of the first 7 counted occurrences: $162 + 65 + 239 + 171 + 86 + 2 + 3 = 728$

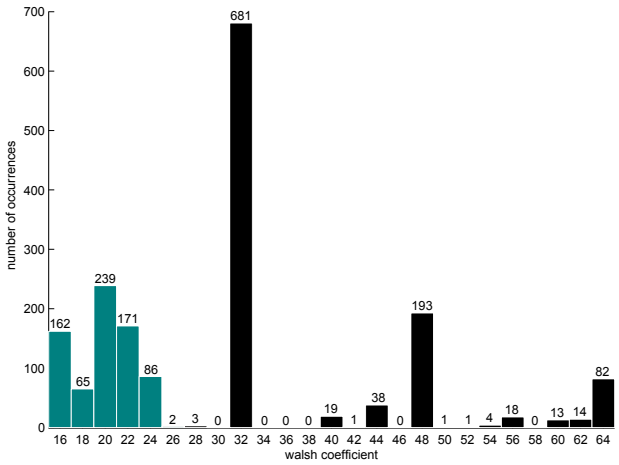

Fig. 2: Histogram of Walsh Coefficients evaluating the LUT contents of a LUT-based AES implementation

assume that a LUT-based AES implementation uses 32 LUTs per S-box instance (6-bit-to-1 bit architecture). In this case, she would estimate $\lfloor \frac{728}{32} \rfloor = 22$ utilized S-boxes for the discussed implementation. When using the Walsh Coefficient, an adversary also can try to guess the architecture of the design. Remember that the previous approach, for identifying S-boxes using Algorithm 3, found all 640 LUTs (100% detection rate) belonging to AES S-boxes. Therefore, it is the best solution, but the main weakness is the inability of identifying proprietary encryption algorithms and especially its proprietary S-boxes. In those cases, the *Walsh Coefficient* may indicate, which LUTs potentially implement proprietary S-boxes. This can be valuable information, e.g., if the FPGA design only uses a limited amount of logic resources. Consider for example that only a proprietary block cipher and a communication interface are implemented: the results should be then quite accurate when predicting the architecture as this is the case for the analyzed AES implementation. The non-linear LUTs may be further studied by an attacker, who is interested in the modification of the proprietary algorithm. During our evaluations, we also found LUTs that simply forward, e.g., a plaintext or ciphertext. The corresponding LUTs possess a very high linearity. Such information may also reveal further implementation details.

In summary, the presented approach may help in the following cases:

- Identification of known non-linear S-boxes
- Identification of proprietary non-linear S-boxes
- Prediction of the implementation architecture

Thus, the *Walsh Coefficient can* indeed be a helpful tool for an attacker. The adversary is able to identify, whether a bitstream contains non-linear parts such as S-box instances and where they are located on the FPGA's grid.

IV. Another Point of Attack: Embedded Memory in FPGAs

Another common implementation strategy for realizing cryptographic S-boxes is to store them in the embedded memory of the FPGA. We briefly describe how the corresponding bitstream mapping of the embedded memory can be obtained. Knowing this mapping, critical data like cryptographic symmetric/asymmetric keys or S-boxes may be extracted from the bitstream since one obtains the plain representation of the embedded memory content.

Suppose that a fixed AES-{128,192,256} key with its corresponding subkeys has been placed in the embedded memory. An attacker then may easily find the corresponding main key by searching XOR-dependencies. This can be done, e.g., with a tool called *aesfindkey* written by Haldermann et al. [8]. For the reverse-engineering process, we need to create a Very High Speed Integrated Circuit Hardware Description Language (VHDL) file in order to derive the appropriate netlist that again serves for learning the bitstream mapping.

A. S-box Instances in Embedded Memory

A simplified VHDL code example, realizing an AES S-box, is depicted in Code Listing 2.

Listing 2: AES S-box instantiation in the embedded memory

```
architecture rtl of sbox_embd_mem is
...
type rom_array is array (0 to 255) of
      std_logic_vector(7 downto 0);
      signal ROM : rom_array := (
            X"63", X"7C", X"77", X"7B",
            ...
            X"B0", X"54", X"BB", X"16"
      );
...
process(clk)
...
if(rising_edge(clk)) then
      data <= ROM(conv_integer(addr));
end if;
end process;
```

When using Code Listing 2, the embedded memory of the FPGA is filled with the specified bytes of the given signal *rom_array*. In this case, it contains the S-box values of AES. This kind of embedded memory requires to use a clock. The S-box input is evaluated on the rising edge of the clock. The corresponding netlist of this design can be generated from the above VHDL file.

B. Extraction of Embedded Memory Content from FPGA Bitstreams

The idea of obtaining the bitstream mapping of the embedded memory is similar to the approach of extracting the mapping of the LUT contents. Again, an attacker can create certain netlists, for which, she changes all memory values bitwise. For each change, the bitstream is synthesized and the corresponding toggling bits are observed. Algorithm 4 shows a generic approach. Having obtained the mapping, we verified the correctness for several FPGA families. Note that there are certain setups for the memory layout that can be chosen by the user. We could verify that the contents of the embedded memory can be extracted – regardless of the chosen memory layout. This can be done with moderate programming efforts.

Algorithm 4 Extracting bitstream mapping of embedded memory content

Input: FPGA device file describing embedded memory
Output: Bitstream position table of embedded memory

set_bit_in_block(·) sets the memory content in netlist file
bs_ref: Reference file with zeroised memory content
bs_mod: Modfied embedded memory content
.net represents the netlist file

```
1:  for block = 0 to num_of_memory_blocks - 1 do
2:      Create bitstream bs_ref with zeroised memory content
        for memory block block
3:      for bit = 0 to num_of_bits_per_memory_block - 1 do
4:          set_bit_in_block(bs_mod.net, block, bit)
5:          Generate bitstream bs_mod
6:          Compare bs_mod and bs_ref
7:          Store difference bit in position_table[block][bit]
8:          clear_bit_in_block(bs_mod.net, block, bit)
9:      end for
10: end for
11: return position_table
```

Note that Algorithm 4 has to be executed only once per device. With the help of the recovered bitstream mapping describing the contents of the embedded memory, an attacker is able to extract and modify the contents using the bitstream file.

C. Practical Evaluation

We were able to extract all S-box bytes residing in embedded memory from the corresponding bitstreams, c.f., Table 3. After having presented several detection approaches, we describe the potential security issues, in the next sections, for the case that an attacker is able to detect and modify S-boxes in a bitstream that corresponds to a DES or AES implementation.

V. Analysis of DES

The DES and especially the 3DES algorithms are still widely used, e.g., in financial systems and SSL/TLS applications. Therefore, both algorithms represent an attractive target to be weakened in FPGA bitstreams. This can be conducted, e.g., by directly modifying the bits of the bitstream related to the DES S-boxes. As demonstrated in the previous sections, we can clearly locate these bits. Figure 3 shows the general Feistel structure of the DES algorithm. The DES algorithm processes a 64-bit plaintext using a 56-bit main key [9]. Sixteen subkeys are derived from the main key by following a fixed scheduling plan. Before demonstrating how an adversary can modify the

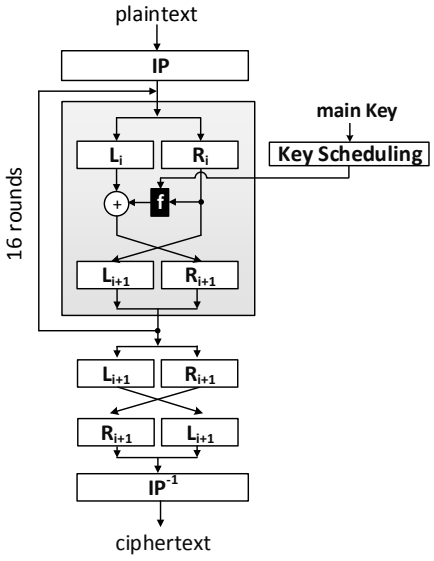

Fig. 3: Overview of the DES encryption algorithm

FPGA bitstream to weaken the DES, the round function f is described. The basic properties of diffusion and confusion are realized by the f-function. Each of the 16 round keys is processed by this function. Figure 4 shows the internal structure of the DES f-function. As can be seen, all eight S-boxes process an intermediate value that has been previously XOR-ed with a subkey. The goal is to directly modify these S-boxes in a way that a ciphertext (computed by this modified DES) can be easily decrypted by an adversary. This is supposed to hold for all plaintext blocks being encrypted by the modified algorithm.

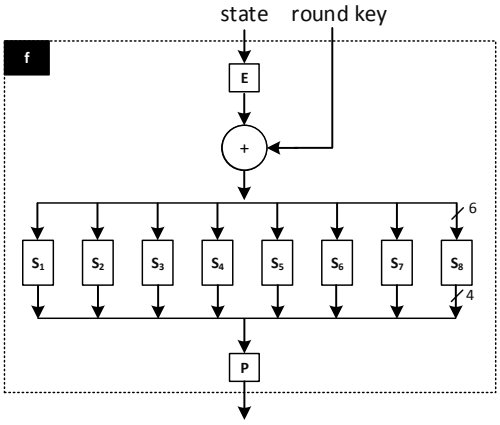

Fig. 4: DES round function f

A. Modification of DES S-boxes

The idea of DES S-box alteration is discussed by Kerins et al. [10], which we briefly present for completeness. If all S-boxes $(S_1, S_2, ..., S_8)$ can be modified in such a way that they always output a zero – regardless of all 64 possible input values – an attacker has successfully performed a malicious alteration of the DES algorithm. To be more precise, the following modification should be applied to the DES S-boxes implemented in the FPGA bitstream:

$$\text{S-box}^0_{\text{DES}}(i) = 0, \quad \forall\, i \in \{0, \ldots, 63\}$$

Due to the presented modification, the whole DES algorithm turns into a key-independent permutation. The modified DES is visible in Figure 5. In a normal operating f-function, the S-box outcomes (32 bits) are permuted according to the mapping rules of function P. The evaluated result of P is concurrently the output of the function f. Since in the modified version, all S-boxes outputs are zero, the output of the permutation P is also completely zero. Hence, the output of the function f is zero, independent of the input and subkey. Because a zero outcome of f is XOR-ed with the left state L_i, it remains unchanged as XOR-ing a value with zero is equal to the identity function. Thus, the state after $\text{IP}(\cdot)$ is not affected when having processed all 16 DES rounds. This is because the number of swaps is even. In the end, a final swap is performed, which is followed by a permutation denoted by $\text{IP}^{-1}(\cdot)$.

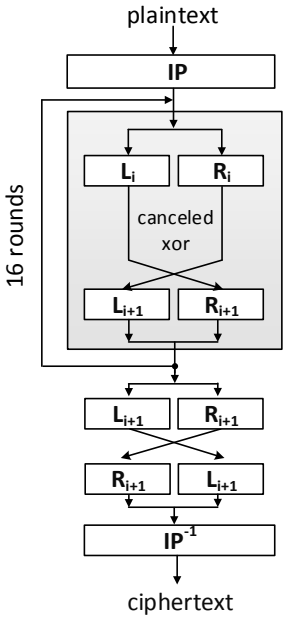

Fig. 5: Modified DES with canceled f-function

The following two equations compare the computation steps of a normal DES encryption with the one of a modified $\widetilde{\text{DES}}$-encryption using $\text{S-box}^0_{\text{DES}}$. The modified encryption only applies three permutations on the plaintext (denoted by p) that

can be easily inverted by an attacker.

$$\mathrm{DES}_k(p) = \mathrm{IP}^{-1}(\mathrm{Swap}(R_{16,k_{16}}(\ldots(R_{1,k_1}(\mathrm{IP}(p)\ldots)$$

$$\widetilde{\mathrm{DES}}_k(p) = \mathrm{IP}^{-1}(\mathrm{Swap}(\mathrm{IP}(p))) = \widetilde{c}$$

An attacker has to perform the following computation to obtain the plaintext from the modified ciphertext $\widetilde{c}$:

$$p = \mathrm{IP}^{-1}(\mathrm{Swap}(\mathrm{IP}(\widetilde{c})))$$

This attacks works likewise for the Triple-DES encryption that is computed as follows [9]:

$$\widetilde{c} = \widetilde{DES}_{k_3}(\widetilde{DES}^{-1}_{k_2}(\widetilde{DES}_{k_1}(p)))$$

A plaintext from the modified 3DES with $\text{S-box}^0_{\mathrm{DES}}$ can also be easily recovered:

$$\begin{aligned} p &= \widetilde{\mathrm{DES}}^{-1}_{k_1}(\widetilde{\mathrm{DES}}_{k_2}(\widetilde{\mathrm{DES}}^{-1}_{k_3}(\widetilde{c}))) \\ &= \mathrm{IP}^{-1}(\mathrm{Swap}(\mathrm{IP}(\mathrm{IP}(\mathrm{Swap}(\mathrm{IP}^{-1}(IP^{-1}(\mathrm{Swap}(\mathrm{IP}(\widetilde{c})\ldots)\end{aligned}$$

As one can see, in this case, an attacker only has to modify eight S-boxes (or: 32 decomposed LUTs in a 6-bit-to-1 bit architecture) within the bitstream to significantly weaken the DES algorithm[4]. The S-box changes were directly applied on the bitstreams and we verified that the alteration of the design was successful. The presented attack of the DES algorihm works, for both, a LUT-based implementation and for an implementation based on embedded memory. Due to the fact that the DES algorithm does not exhibit any inverse S-boxes, the decryption also functions correctly. This severe bitstream modification may remain undetected in applications such as data storage where encryption and decryption are performed by the same device, or if all ciphers in a given system are modified in this way. Possible countermeasures include self-tests or integrity checks.

VI. Analysis of AES

The Advanced Encryption Standard (AES) is the most commonly used symmetric cipher today. In this section, we present further results of our analysis regarding malicious AES modifications in FPGA bitstreams. Similar to DES, an attacker may be able to silently weaken the algorithm such that an encryption and decryption are still being performed correctly, but the corresponding ciphertexts are cryptographically weak. For this purpose, the S-box instances are again modified. Furthermore, a key leakage approach and a corresponding scenario are discussed. As described in the previous sections, we are able to detect S-box instances by analyzing the corresponding bitstream of an FPGA.

Before describing both S-box modifications on the FPGA bitstream, the AES algorithm is briefly introduced. Figure 6 shows an overview of the AES-{128,192,256} encryption scheme for the three different key sizes 128, 192, and 256 bit leading to the execution of $\mathrm{Nr} \in \{10, 12, 14\}$ rounds, respectively [11]. The AES operates on 128-bit blocks, independent of the key size. One AES round consists of the

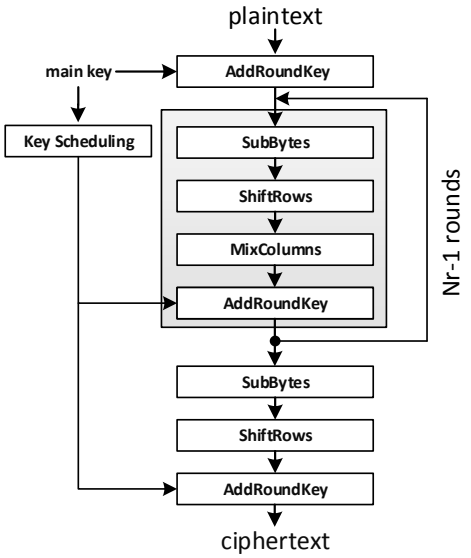

Fig. 6: Overview of the AES encryption algorithm

operations *SubBytes*, *ShiftRows* ($\mathrm{SR}(\cdot)$), *MixColumns* ($\mathrm{MC}(\cdot)$), and *AddRoundkey* that are executed consecutively. Thereby, the *SubBytes* step processes sixteen intermediate bytes by using a fixed S-box. For round-based implementations it is common to synthesize multiple S-boxes such that each input byte can be processed in parallel. In addition to that, the key schedule step also needs to process four S-box instances. Two of the three key schedule algorithms are depicted and described in Section VI-B. Section VI-A describes the impact of replacing all S-boxes by the identity function, while Section VI-B demonstrates the influence of setting all S-box outcomes to zero.

A. Replacing S-boxes to the Identity Function

a) Impact of S-box modification to the AES encryption: When setting all AES S-box instances to the identity mapping as given in the equation below, the encryption and decryption function turn into a linear bijection. The corresponding modified AES can correctly encrypt and decrypt, but is extremely vulnerable to cryptanalytical attacks.

$$\text{S-box}^{id}_{\mathrm{AES}}(i) = i, \quad \forall i \in \mathrm{GF}(2^8)$$

An attacker is able to decrypt all faulty ciphertext blocks, because the altered AES-128 can be described as:

$$\begin{aligned} \widetilde{c} = \widetilde{\mathrm{AES}}_k(p) &= \mathrm{SR}(\ldots\mathrm{MC}(\mathrm{SR}(p \oplus K_0) \oplus K_1)\ldots) \oplus K_{10} \\ &= \mathrm{SR}(\ldots\mathrm{MC}(\mathrm{SR}(p)\ldots) \\ &\quad \oplus \mathrm{SR}(\ldots\mathrm{MC}(\mathrm{SR}(K_0) \oplus K_1)\ldots) \oplus K_{10} \\ &= \mathrm{SR}(\ldots\mathrm{MC}(\mathrm{SR}(p)\ldots) \oplus \widetilde{K} \end{aligned}$$

[4]Potentially more S-box instances have to be modified in the bitstream, depending on the design architecture, e.g., in an unrolled implementation.

The plaintext is denoted by p, the subkeys by $K_0, K_1, ..., K_{10}$ and the faulty ciphertext by $\widetilde{c}$ (encrypted by the weak AES). The equation above holds, because the $\mathrm{MC}(\cdot)$ and the $\mathrm{SR}(\cdot)$ functions are linear as described below.

$$\forall a, b \;\; 4 \times 4 \text{ matrices with elements} \in \mathrm{GF}(2^8) :$$
$$\mathrm{MC}(a \oplus b) = \mathrm{MC}(a) \oplus \mathrm{MC}(b)$$
$$\mathrm{SR}(a \oplus b) = \mathrm{SR}(a) \oplus \mathrm{SR}(b)$$

It is important to understand that the XOR sum of all processed subkeys is constant and can be expressed by one variable $\widetilde{K}$. In addition, the number of $\mathrm{MC}(\cdot)$ and $\mathrm{SR}(\cdot)$ operations depends on the utilized AES key size, i.e., 128, 192, or 256 bits. In the following, we describe how $\widetilde{K}$ can be recovered with one plaintext/ciphertext pair $(p, \widetilde{c})$ encrypted by this modified AES.

b) Decryption of Ciphertexts: When an attacker can obtain one plaintext/ciphertext pair $(p, \widetilde{c})$ (encrypted by the modified AES), then she is able to compute the secret $\widetilde{K}$. For this purpose, she simply reconstructs $\mathrm{SR}(\ldots \mathrm{MC}(\mathrm{SR}(p) \ldots)$, and then computes the following:

$$\widetilde{K} = \widetilde{c} \oplus \mathrm{SR}(\ldots \mathrm{MC}(\mathrm{SR}(p) \ldots) \tag{1}$$

With the knowledge of $\widetilde{K}$, an attacker can recover any plaintext x from any faulty ciphertext $\widetilde{y}$. To do so, the adversary has to XOR the value $\widetilde{y}$ with the previously recovered secret $\widetilde{K}$. Afterwards, the $\mathrm{MC}(\cdot)$ and $\mathrm{SR}(\cdot)$ transformations have to be inverted. The number of inversions differs, depending on the AES key size. Algorithm 6 illustrates this concept in more detail. As indicated above, this attack works regardless of the key schedule, because the secret $\widetilde{K}$ can be canceled in any case. Thus, it does not matter whether any S-box of the key schedule is altered or not.

Algorithm 5 Decrypt Ciphertexts Encrypted with $\text{S-box}^{id}_{\mathrm{AES}}$

Input: Ciphertext $\widetilde{y}$ from a modified AES ($\text{S-box}^{id}_{\mathrm{AES}}$)
One previously obtained $(p, \widetilde{c})$ pair
Output: Plaintext x corresponding to $\widetilde{y}$

//Calculate $\widetilde{K}$
1: $\widetilde{K} \leftarrow \widetilde{c} \oplus \mathrm{SR}(\ldots \mathrm{MC}(\mathrm{SR}(p) \ldots)$
//Cancel secret $\widetilde{K}$
2: $\widetilde{y} \leftarrow \widetilde{y} \oplus \widetilde{K}$
//Calculate x depending on the number of rounds
3: $x \leftarrow \mathrm{SR}^{-1}(\mathrm{MC}^{-1}(\mathrm{SR}^{-1}(\ldots \mathrm{MC}^{-1}(\mathrm{SR}^{-1}(\widetilde{y}) \ldots)$

B. Replacing S-boxes to the Zero Function

Analogous to the DES modification of Section V, all AES S-boxes in the FPGA bitstream can be altered to always output a zero – regardless of the input value. This kind of modification is also presented by Kerins et al. [10], which we further extend with respect to an FPGA scenario. The modification is described in the following equation:

$$\text{S-box}^0_{\mathrm{AES}}(i) = 0, \quad \forall\, i \in \mathrm{GF}(2^8)$$

Obviously, after having altered all S-box instances in the this manner, the AES algorithm becomes unusable. That is because any information regarding the plaintext is lost, right after the first SubBytes step has been processed by the modified AES instance, hence the cipher is not bijective anymore. However, such an alteration could still be useful for an adversary since the output of the AES is now the last subkey.
Such kind of attack can be useful if the underlying main key is, e.g., hard-coded in the FPGA design and not stored in the embedded memory. Another scenario is given if a main key is securely transfered to the FPGA after power-up, e.g., by a Hardware Security Module (HSM) whose data bus cannot be eavesdropped. An adversary can obtain the key with this alteration, if she is able to query the AES instance with an arbitrary plaintext.

Since the S-boxes of the key schedule are usually not distinguishable from the SubBytes S-boxes, the attack will lead to the modification of all S-boxes, including those from the key schedule. In the following, we have a look at the two key schedule schemes of AES-128 and AES-192. We further assume that all SubBytes S-boxes are altered.

1) AES-128 Key Schedule: In the case of AES-128, the main key K_0 can always be fully recovered. The steps are given in Algorithm 6. In order to better understand Algorithm 6, the

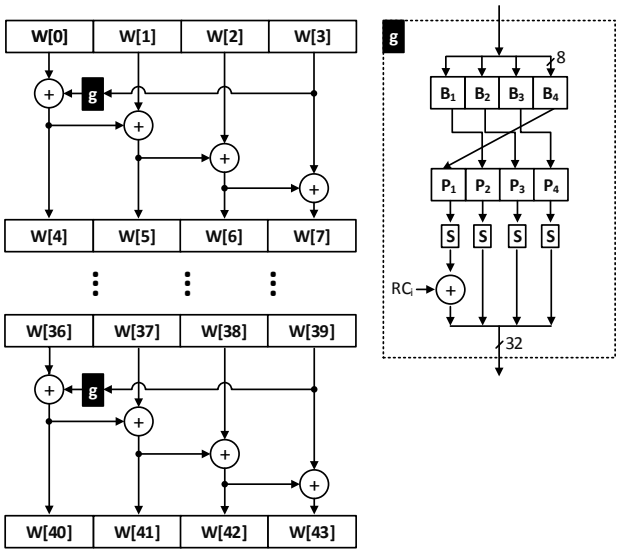

Fig. 7: Key schedule of AES-128

AES-128 key schedule is depicted in Figure 7. The following cases may occur:

- Function $g(\cdot)$ utilizes AES S-boxes: This can happen, if the round keys were computed before the modification. In this case, Algorithm 6 immediately reveals the full key.
- Function $g(\cdot)$ utilizes the modified S-boxes $\text{S-box}^0_{\mathrm{AES}}$: In this case, the g-function only returns the correspond-

Algorithm 6 Reconstruction of the full main key of AES-128

Input: Ciphertext $\widetilde{y}$ from modified AES ($\text{S-box}^0_{\text{AES}}$)
Output: Fully recovered 128-bit AES main key.

```
   //Load modified ciphertext (= last round key)
1: for i = 0 to 3 do
2:     w[43 − i] = ỹ[3 − i]
3: end for
   //Invert the 128-bit key schedule
4: for i = 39 to 0 do
5:     if i % 4 == 0 then
6:         w[i] = w[i + 4] ⊕ g(w[i + 3])
7:     else
8:         w[i] = w[i + 4] ⊕ w[i + 3]
9:     end if
10: end for
```

ing round constant $\text{RC}[i]$, also padded with three zero bytes. Code line 6 of Algorithm 6 should be then changed to

$$w[i] = w[i+4] \oplus \text{RC}[i]$$

in order to reveal the main key.

2) AES-192 (and AES-256): Compared to AES-128, AES-192 and AES-256, only leak the main key under special conditions. The graphical representations of the AES-192 key schedule function is shown in Figure 8. Similar to AES-

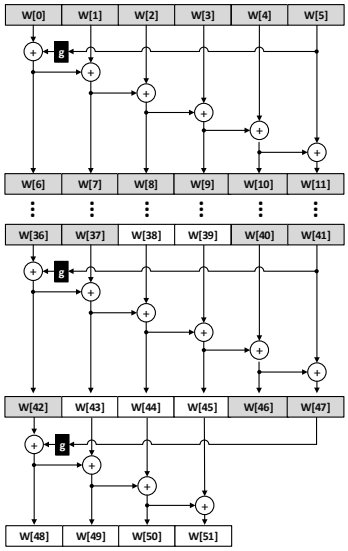

Fig. 8: Key schedule of AES-192

128, two cases can occur if all SubBytes S-boxes are already modified to a constant zero outcome function:

a) Key Schedule utilizes AES S-boxes: The following explanation refers to AES-192 that is illustrated in Figure 8, but also holds for AES-256.

Figure 8 shows the computable words (white background) and non-computable words (gray background). If the round keys are calculated utilizing normal AES S-boxes, then, $w[42]$ cannot be calculated from the modified ciphertext. This is because the output of the last g-processing is unknown to an attacker. Therefore, in the set of $w[36] - w[41]$ only the words $w[38]$ and $w[39]$ are computable. The other intermediate values belonging to the same set cannot be computed, because $w[42], w[46]$, and $w[47]$ are unknown. The last possible word that can be computed is $w[33]$. Hence, in this case, not any single byte of the main key can be recovered. This fact also holds for AES-256.

b) Key Schedule utilizes $\text{S-box}^0_{\text{AES}}$*:* In the case that the key schedule S-boxes are also set to zero, the first 128 bit of the AES-192 main key can be derived. This also holds for AES-256.

The g-function returns the round constant value $\text{RC}[i]$, if all S-box outputs yield a zero (for every input), c.f., function g of Figure 7. Hence, $w[42]$ is derivable and all the first *left* 4 words of each subkey of the key schedule step are computable. Even if the *right* part is not known, the first 4 words $w[0] - w[3]$ can be computed, c.f., Algorithm 7. To

Algorithm 7 Partial key reconstruction of AES-192/256

Input: Ciphertext $\widetilde{y}$ from modified AES with $\text{S-box}^0_{\text{AES}}$
Output: First 128 bit of 192/256 main key

$N_w \leftarrow 51$ for AES-192 ($\leftarrow 59$ for AES-256)
$N_k \leftarrow 6$ for AES-192 ($\leftarrow 8$ for AES-256)

```
   //Load the ciphertext (= last round key)
1: for i = 0 to 3 do
2:     w[N_w − i] = ỹ[3 − i]
3: end for
   //Invert the KeySchedule
4: for i = N_w to 0 do
5:     if i mod N_k ≥ 4 then
6:         continue
7:     end if
8:     if i mod N_k == 0 then
9:         w[i] = w[i + N_k] ⊕ RC[i]
10:    else
11:        w[i] = w[i + N_k] ⊕ w[i + N_k − 1]
12:    end if
13: end for
```

be more precise, the first 128 bits of each subkey can be recovered. Therefore, in this case, an attacker can obtain the first 128 bits of the main key of AES-192 and AES-256. The other words (64 bits) of AES-192 and AES-256 (128 bits) cannot be computed. Having discussed the potential attack vectors, in the next section several countermeasures are briefly described.

VII. Countermeasures

In this section, we briefly discuss some countermeasures that may be deployed in order to raise the bar for an adversary. In our attack model the adversary can modify the LUT content and the embedded memory content of an FPGA bitstream, i.e., the parts of the hardware in which cryptographic S-boxes are implemented.

The countermeasures are based on obfuscation. In general, every obfuscation strategy helps to defeat such kind of modification attacks, but if a strategy is known to an attacker, it may be circumvented easily. In the following, several ideas and their drawbacks are listed.

A. Built-In Self-Test

Built-In Self-Tests are a well known concept to test different kinds of faults and circuits. A simple integrated self-test can be used to defeat the attacks presented in this work. For example, one can check if the algorithm outputs the correct ciphertext for a fixed key and plaintext. Such a self-test can be circumvented, however, by a more powerful adversary with the following approaches:

- The integrity value has to be stored somewhere in the FPGA bitstream. Thus, an adversary may be able to identify and modify this value.
- The adversary could disable the self-test or modify it in such a way that the test routine marks the test as "passed".

B. Forced Decomposition

Another approach targets the decomposed LUTs. They are detectable because of their characteristically non-linear patterns. Security critical Boolean equations, generating the LUT contents for the S-boxes, should be difficult to distinguish from other linear LUT content patterns to defeat detection and consequently modification of these parts. One possible way to achieve this is to further decompose the LUTs along the Disjunctive Normal Form. For example, in a 6-bit-to-1 bit architecture, a 64-bit LUT content may be split-up into 8 LUTs. The output of each LUT can be OR-ed together to compute the original LUT content.
To give an example, assume a Boolean function $f(a,b,c) = ab \vee bc \vee abc$. Suppose that this Boolean function is realized in one LUT. Following the idea described above, this LUT is separated into three LUTs:

$$f_1(a,b,c) = ab$$
$$f_2(a,b,c) = bc$$
$$f_3(a,b,c) = abc$$

The result of every function f_i is then OR-ed. Thus, it should be more difficult to identify f_1, f_2, and f_3, if this function splitting scheme is unknown to an attacker, but the decomposition to multiple LUTs has also a drawback:

- An adversary can search for the hardware part where the OR-ing of all f_i functions is processed. In a test implementation, we observed that one LUT is used to implement the corresponding OR function, hence an adversary could modify this LUT. For the alteration to an identity mapping, e.g., in the case of AES, the adversary would need to trace the path back to the f_i functions with the help of routing information.

Even when the set of candidates is too large for an adversary, it is possible to obtain the correct set of LUTs belonging to the S-box. The attacker's effort depends on the decomposition method and the corresponding parameters. It might be more challenging for an attacker, if the decomposition of the LUT content is chosen randomly for each S-box column.

C. White-box Cryptography

One could deploy white-box cryptography as a countermeasure. The main idea is to hide the secret key inside the implementation [12]. Key-dependent LUTs together with random transformations generate the masking of a fixed key. Even for this kind of countermeasure there is one drawback:

- The tables of a fixed-key implementation can be copied and used to decrypt the ciphertexts. In a non-fixed-key implementation, the table values have to be computed, thus the S-box is present on the FPGA. Again, these S-box can be modified to weaken the implementation.

In summary, there are methods available to defeat the attacks proposed in this paper. We recommend that further research should be conducted to evaluate how an FPGA design can be secured against bitstream modification attacks.

VIII. Conclusion

In this work, we demonstrated how to detect and maliciously modify an FPGA bitstream that implements the widely used cryptographic algorithms DES, 3DES, and AES in order to weaken their strong cryptographic properties. In this scenario, only an unknown third-party bitstream is obtained by the adversary. This work shows that she does not need to possess any high-level design information such as routing details to be able to significantly weaken cryptographic primitives. The presented attacks are practically feasible and pose a serious threat in several applications in the real world.
An attacker can easily decrypt all ciphertext blocks that were weakened due to the modified FPGA designs. The DES becomes a key-independent permutation that can be inverted by an adversary without any further information. Compared to that, the AES was modified in two ways: the first bitstream alteration turns the AES into a linear function and thus all further ciphertext blocks can be decrypted with only one known plaintext/ciphertext pair. The second modification leads to a (partial) key leakage of AES-$\{$128,192,256$\}$.
Furthermore, several scenarios were discussed where such a modification may remain undetected making a target device behave like an FPGA Trojan. The presented results highlight the importance of integrity checks and that further security mechanisms must be deployed around FPGAs.

This work should raise awareness that an attacker can manipulate proprietary bitstreams by purpose with moderate efforts. The modification must not necessarily be a cryptographic function. It is important to carefully check an intellectual property core before using it in security applications.

ACKNOWLEDGMENT

The authors would like to thank Christian Kison for providing help with the VHDL implementation and Georg Becker for his fruitful comments and discussions.

REFERENCES

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

[2] 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 Proceedings of the 18th ACM Conference on Computer and Communications Security, ser. CCS '11. ACM, 2011, pp. 111–124.

[3] 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 Topics in Cryptology - CT-RSA 2012, ser. Lecture Notes in Computer Science, vol. 7178. Springer, 2012, pp. 1–18.

[4] 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 Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ser. FPGA '13, New York, NY, USA, 2013, pp. 91–100.

[5] R. Chakraborty, I. Saha, A. Palchaudhuri, and G. Naik, "Hardware Trojan Insertion by Direct Modification of FPGA Configuration Bitstream," Design Test, IEEE, vol. 30, no. 2, pp. 45–54, April 2013.

[6] A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," in Cryptographic Hardware and Embedded Systems - CHES 2007, ser. Lecture Notes in Computer Science, P. Paillier and I. Verbauwhede, Eds. Springer-Verlag, 2007, vol. 4727, pp. 450–466.

[7] Leander, G. and Poschmann, A., "On the Classification of 4 Bit S-Boxes," in Arithmetic of Finite Fields, ser. Lecture Notes in Computer Science, C. Carlet and B. Sunar, Eds. Springer-Verlag, 2007, vol. 4547, pp. 159–176.

[8] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold-boot Attacks on Encryption Keys," Communications of the ACM, vol. 52, no. 5, pp. 91–98, May 2009.

[9] NIST, FIPS-46-3: Data Encryption Standard (DES), National Institute of Standards and Technology (NIST) Std., 1999, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[10] T. Kerins and K. Kursawe, "A cautionary note on weak implementations of block ciphers," in In 1st Benelux Workshop on Information and System Security (WISSec 2006, 2006, p. 12.

[11] NIST, "FIPS 197 Advanced Encryption Standard (AES)," 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[12] H. J. Stanley Chow, Philip A. Eisen and P. C. van Oorshot, "White-Box Cryptography and an AES Implementation," Selected Areas in Cryptography, vol. 2595, pp. 250–270, 2002.