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

***Abstract*—This paper investigates a novel attack vector against cryptography realized on FPGAs, which can pose a serious threat to real-world implementations. We demonstrate how a simple bitstream modification can seriously weaken crypto algorithms, which we show by example of the AES and 3DES. The attack is performed by modifying the FPGA bitstream that configures the hardware elements during initialization. It has been known for a long time that cloning of FPGA designs, even if the bitstream is encrypted, is a relatively easy task. However, due to the proprietary format of the bitstream, a meaningful modification of an unknown FPGA bitstream is very challenging. While some previous work had 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 can lead to serious security implications. We target the non-linear S-boxes of crypto algorithms of a synthesized FPGA design that can be either implemented as Boolean equations in look-up tables, or as precomputed set of values that are stored in the memory of the FPGA. We demonstrate that it is possible to detect and apply meaningful changes to cryptographic elements inside an unknown propriety and undocumented bitstream. Furthermore, we also show how an AES key can be revealed within seconds by modifying the bitstream. Finally, we propose countermeasures that can raise the bar for an adversary to successfully perform an 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, signal processing, automation and the automotive industry. Many of these application 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 FPGAs and 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 set-up 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]. The attacks leak the symmetric encryption keys stored inside the FPGA utilizing side-channel analysis. After key extracting, 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, she still faces two major problems: She has to overcome a considerably obfuscation hurdle and she 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 publically documented what the bitstream details of popular commercial FPGAs 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 our best 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 logic. 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 serve the purpose to 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. The weakened algorithms are incompatible with their genuine counterparts. Thus, the attack is limited to certain scenarios, in which encryption and decryption are computed by the same device, e.g., USB sticks, solid-state disks or in encrypted cloud storage. Also, the manipulations can be used in systems in which all involved devices can be altered. We practically verified our techniques for a well known FPGA vendor.

Part of the research was conducted at the University of Massachusetts Amherst.

Finally we propose countermeasures to raise the bar for an attacker.

DISCLAIMER - We cannot publish detailed results regarding bitstream reverse-engineering due to potential legal issues. Hence we describe the technique in a generic manner.

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 realizing the main 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 medium in an FPGA, they are promising targets for an attacker that wants to maliciously change the functionality of an FPGA design. This is especially critical in cryptographic applications. Thus, it is also quite important to analyze the LUT 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 of 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 patterns of two Device Under Tests (DUTs) 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

First, we provide information about our DUT that uses a 6-bit-to-1 bit architecture. It has 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 can use the vendor's tools. To our best knowledge, this can be done for any FPGA that belongs to the same vendor. The approach of reverse-engineering the LUT contents from a bitstream relies on generating appropriate netlists that specify the rules of reconfiguring the hardware. The netlist can be used to manually configure any LUT with any arbitrary 6-input Boolean function. It can be converted to the FPGA's bitstream using the vendor's tool. To give an example, Listing 1 shows the configuration of four LUTs. LUT$_1$ implements a 6-input AND gate. The inputs are denoted by $i_6, ..., i_1$. The &-character represents a logical AND, while a ∼-character represents a logical NOT. Note that the presented netlist uses a fictional syntax, but it is very similar to the syntax of our targeted vendor.

Listing 1: Netlist example for setting LUT contents

```
FPGA design "minimal_lut_implementation" ...,
  other configuration "...";
instance "logic block X Y",
config {
        LUT1 = {"i1 & i2 & i3 & i4 & i5 & i6"}
        LUT2 = {"i1 & i5"}
        LUT3 = {"i2 + i4 + i6"}
        LUT4 = {"∼i3 + (i5 & i6")}
}
```

As further illustrated by Listing 1, each LUT of one logic block can be configured by specifying a Boolean equation.

In the following, we describe a reverse-engineering strategy that reveals all LUT bit positions from a bitstream. Then, all LUT contents can be easily dumped from a bitstream, i.e., the corresponding dump reveals the FPGA's synthesized Boolean functions. This way, an attacker is able to search for specific logic like cryptographic S-boxes. The idea is as follows: An attacker configures two Boolean functions for exactly one LUT, thus, he has to create two different netlists (c.f., Listing 1). The first netlist configures a Boolean function, 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. The only difference is the specified Boolean function. 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, 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 Boolean function (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. Note that this can be realized by configuring a 6-input AND as it is shown in the table below. 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 bitstream per LUT, while for a 4-bit-to-1 bit architecture only 17 bitstream generations are sufficient. This approach has to repeated for all given LUT of the underlying FPGA in order to be able to dump all LUT contents from the bitstream.

Equations for Boolean functions	**LUT content**
$\sim i_6$ & $\sim i_5$ & $\sim i_4$ & $\sim i_3$ & $\sim i_2$ & $\sim i_1$	0x00000000000000001
$\sim i_6$ & $\sim i_5$ & $\sim i_4$ & $\sim i_3$ & $\sim i_2$ & i_1	0x00000000000000002
$\sim i_6$ & $\sim i_5$ & $\sim i_4$ & $\sim i_3$ & i_2 & $\sim i_1$	0x00000000000000004
... ...	...
i_6 & i_5 & i_4 & i_3 & i_2 & $\sim i_1$	0x40000000000000000
i_6 & i_5 & i_4 & i_3 & i_2 & i_1	0x80000000000000000
0	0x00000000000000000

TABLE I: Setting boolean equations for LUT

Note that the bits of one LUT, as indicated by Table I, are not stored next to each other in the bitstream. Rather, they are 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 equations) and by creating the corresponding bitstream. Then, the LUT contents can be dumped from the bitstream and compared to the Boolean functions of the previously generated netlist.

Algorithm 1 illustrates this straightforward and time-consuming reverse-engineering approach in more detail. A more sophisticated (and much 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 reverse-engineering process, then, approximately takes 1-2 days, while the straight-forward approach takes much longer. Note that several offset patterns can be found, which depend on the LUT coordinates in the FPGA's grid. The paper's intention is not to provide very deep details of the bitstream file format. Rather, it illustrates how basic the approach can be. This is supposed to raise awareness of the moderate reverse-engineering effort of an attacker. The next sections deal with the detection and impacts of an adversary's modification.

III. Detecting DES S-boxes

This section explains how to detect DES S-boxes from an FPGA's bitstream. The corresponding FPGA design is based on a 6-bit-to-1 bit architecture. Note that the Data Encryption

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

```
1:  **Input**: FPGA device file describing LUTs
2:  **Output**: Bitstream position table of LUT content
-----
3:  bool_eq(·) generates the boolean equation (see Table I)
4:  set_lut_content(·) sets the LUT content in netlist file
5:  Bitstream *bs_ref*: Bitstream file with zeroised LUT content
6:  Bitstream *bs_mod*: Modified LUT content
-----
7:  for *lut_index* = 0 to num_of_luts - 1 do
8:      Create bitstream *bs_ref* with zeroised LUT content
        for LUT *lut_index*
9:      for *bit* = 0 to $2^6 - 1$ do
10:         *lut_content* = bool_eq(*bit*)
11:         set_lut_content(*bs_mod, lut_index, lut_content*)
12:         Synthesize bitstream *bs_mod*
13:         Compare *bs_ref* and *bs_mod*
14:         Store difference in *position_table*[*lut_index*][*bit*]
15:     end for
16: end for
17: return *position_table*
```

Standard (DES) algorithm is described in Section V in more detail. DES uses eight different predefined 6-bit-to-4 bit S-boxes. Since our DUT 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. Because of that, a round-based DES implementation, using 8 S-boxes, requires an instantiation of 32 LUTs. A general 6-bit-to-4 bit LUT is illustrated in Table II. Note that each column f_1, f_2, f_3, and f_4 stores a unique 64-bit sequence describing a Boolean equation. These might be the fixed bit-sequences of a DES S-box.

Input values						**Output columns**			
$\mathbf{i_6}$	$\mathbf{i_5}$	$\mathbf{i_4}$	$\mathbf{i_3}$	$\mathbf{i_2}$	$\mathbf{i_1}$	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{f_4}$
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:

- $f_1(i_6, ..., i_1)$ = a$_{64}$...a$_1$ = 0x869D497A86E67619
- $f_2(i_6, ..., i_1)$ = b$_{64}$...b$_1$ = 0xB0C7871B497826BD
- $f_3(i_6, ..., i_1)$ = c$_{64}$...c$_1$ = 0x27E9D492609F1F29
- $f_4(i_6, ..., i_1)$ = d$_{64}$...d$_1$ = 0x917BE9066F81B478

Note that each 64-bit pattern is unique for each DES S-box. Because we were able to reverse-engineer the bitstream mapping for each LUT, we can now analyze the corresponding (dumped) LUT contents. As stated above, an attacker may

[1]It is equal to the LUT content describing the Boolean function of one output bit of the S-box. A (S-box) column might be $f_1(\cdot) = a_1 ... a_{64}$, c.f., Table II

search for the presented patterns. Additionally, all 6! permutations of a pattern have to be examined, because of the possible input permutations.

The vendor's tools determine an optimal routing path. For this purpose, the tools permute the input bits of a LUT. Thus, this also leads to permuted output bits. Due to this fact, the LUT content has to be viewed as $f(\mathrm{perm}(i_6,\ldots,i_1))$ instead of $f(i_6,\ldots,i_1)$. One may think that an attacker needs further knowledge of the FPGA's routing, but this is not necessary due to the uniqueness of DES S-box patterns: The basic idea is to compute all possible input permutations for all given DES patterns and to compare them with all dumped LUTs. 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

```
1: Inputs: Bitstream bs, Bitstream position table of LUT content
2: Output: File with localized DES LUTs
3: $S_1(x), S_2(x), ..., S_8(x)$ represent DES S-Boxes
4: $S_i^j(x)$ denotes to the $j$'th output bit of $S_i(x)$
5: $\mathrm{perm}_i(\cdot)$ denotes the $i$'th permutation out of all 6!
6: mark_lut(·) writes the parameter to an output file
   //Generate DES search patterns
7: for sbox = 1 to 8 do
8:     for output_bit = 1 to 4 do
9:         des_pattern[sbox][output_bit] =
10:                $S_{\text{sbox}}^{\text{output\_bit}}(63)|\ldots|S_{\text{sbox}}^{\text{output\_bit}}(0)$
11:    end for
12: end for
13: LUT[num_of_luts] ← DumpLUT(bs)
   //Search for DES pattern
14: for lut_index = 0 to num_of_luts - 1 do
15:    for perm_index = 1 to 6! do
16:        for sbox = 1 to 8 do
17:            for output_bit = 1 to 4 do
18:                if ($\mathrm{perm}_{perm\_index}$(LUT[lut_index])
                       == des_pattern[sbox][output_bit]) then
19:                    mark_lut(lut_index, sbox, output_bit)
20:                end if
21:            end for
22:        end for
23:    end for
24: end for
```

In practice, we were able to detect all S-box instances of our synthesized bitstream. Next to the exact location of LUTs on the FPGA's grid (belonging to DES S-boxes), we obtained the exact permutation order of the corresponding input pins. (without any knowledge of the routing) for every single S-box column.

Note that the provided knowledge might be extremely useful for an attacker, e.g., if side-channel attacks based on Electromagnetic Emanation (EMA) are used. Knowing the exact location an attacker can try to locate the best probe position for the measurement while a DUT 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 (known) cryptographic instances run in parallel. Table III illustrates that we were able to locate all DES S-boxes from two of our FPGA implementations. Note that one can also easily identify the S-boxes of a Triple-DES (3DES) architecture.

Implementation	Architecture	Found LUTs	Detection rate
#1	fully unrolled	512	100 %
#2	round-based	32	100 %

TABLE III: Overview of evaluated DES implementations

The Algorithm 2 can also be applied to a 4-bit-to-1 bit LUT FPGA architecture. In this similar scenario, 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, he is able for 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, if they can be altered by an attacker, but under certain conditions, the identification of S-box columns can be more challenging. We discuss this in Section III-A.

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

If the FPGA's architecture uses y-bit-to-1 bit LUTs and x-bit-to-1 bit Boolean functions 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.

b) Case 2: $x > y$: If $x > y$ holds, then it is a more challenging task to find x-bit-to-1 bit S-box columns. Due to the dimensions, one S-box column must be split into at least $\lceil \frac{x}{y} \rceil$ LUTs that have to be multiplexed. 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 Advanced Encryption Standard (AES) in Section III-B.

B. Detection of AES

To detect decomposed AES[2] S-boxes (for a 6-bit-to-1 bit architecture) within a bitstream, we first synthesized the corresponding 8-bit-to-8 bit AES S-boxes. Thus, roughly speaking, we have to handle the detection for Case 2 of the previous

[2]The AES is described in Section VI.

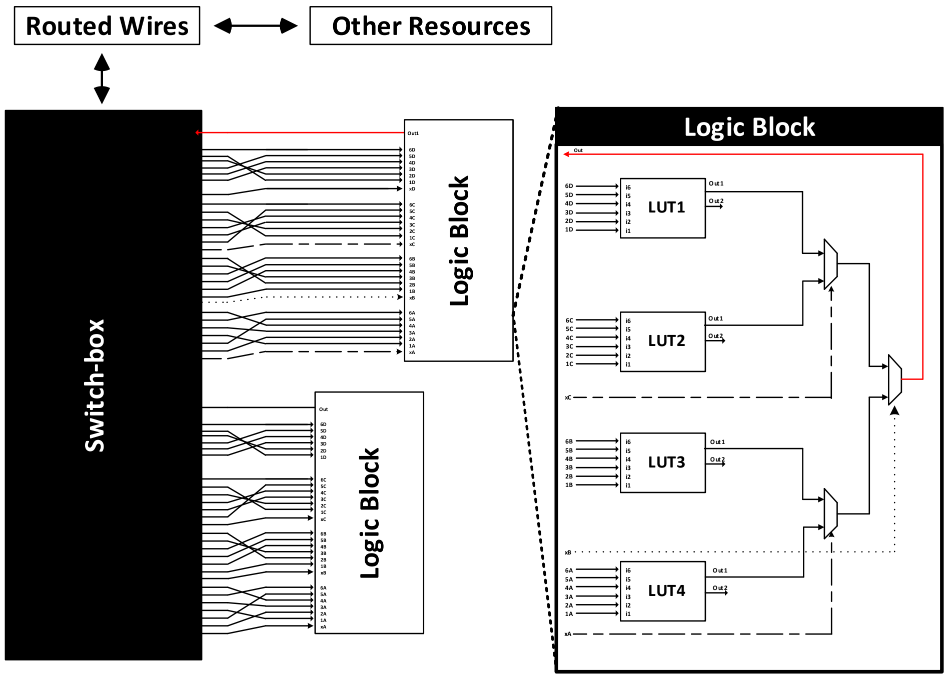

Fig. 1: Simplified overview of a logic block realizing an 8-bit-to-1 bit Boolean function with four 6-bit-to-1 bit LUTs

section. In the next step, we analyzed the place and route behavior of the underlying FPGA tool chain.

During this step, we observed that any 8-bit-to-1 bit function is usually placed and routed as depicted in Fig. 1. Thus, for the realization of one single AES S-box, eight of the illustrated logic blocks need to be synthesized by the FPGA tool chain. It seems that this is the most efficient way of realizing an 8-bit-to-1 bit Boolean function as otherwise the logic has to be distributed over several logic blocks. To be more precise, the FPGA's architecture makes use of $8 \cdot 4$ LUTs and $8 \cdot 3$ multiplexers for implementing one AES S-box.
In the following we briefly explain what we could observe from the FPGA's tools. To be more precise, we show how an 8-bit-to-1 bit function is separated into four 6-bit-to-1 bit LUTs and how they are multiplexed. One AES S-box output column can be written as $f(i_8, i_7, i_6, i_5, i_4, i_3, i_2, i_1) = a_{256}...a_1$. First, two multiplexer inputs, denoted by mu_1 and mu_2, are selected that control which LUT is selected as output, c.f. Fig. 1. Without loss of generality, let $mu_1 = i_8$ and $mu_2 = i_7$. These two multiplexer inputs are routed to the three multiplex units inside the logic block. Note that there are $\binom{8}{2} = 28$ possibilities to chose two input bits of the AES as multiplexer bits.
The 8-bit-to-1 bit function is divided into four $\lceil \frac{256}{64} \rceil = 4$ LUTs that are denoted by LUT$_i$ (or: subfunctions f_i) with $i = 1, 2, 3, 4$. Thus, the synthesizer has to pick 64 outputs bits from $a_{256}...a_1$ that have to be stored together in one LUT. Once the two multiplexer inputs have been chosen, it is automatically arranged which 64 bits have to grouped together. To give an example, with the previously defined multiplexer configuration, the first 64 output bits $a_{256}...a_{193}$ have to stored in one LUT, while $a_{192}...a_{129}$ belong to a second LUT, etc. For any multiplexer configuration, this is the way how the output bits $a_{256}...a_1$ have to be divided into four LUTs:

For any AES S-box input value $x \in \{0, ..., 255\}$, for which $(mu_1, mu_2) = (0, 0)$ holds[3], add the corresponding AES S-box output bit to the same LUT group. This is also repeated for $(mu_1, mu_2) \in \{(0, 1), (1, 0), (1, 1)\}$. Note that all LUT$_i$ contents can again be permuted by one out of 6! possible permutations. Knowing this, one can see that for an 8-bit-to-1 bit AES S-box column, there are $\binom{8}{2} \cdot 6! \cdot 4 = 80640$ patterns that have to be generated and searched for. To be able to search for all AES S-box output columns, one needs to generate $8 \cdot 80640 = 645120$ patterns in total. Algorithm 3 (no further explanation) provides the necessary steps for detecting all AES S-boxes from a 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.
This algorithm is 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.
Note that in some attack scenarios an adversary needs to figure out the exact input permutation and multiplexer configuration of a logic block, while for some other attack strategies this is not necessary at all. These scenarios are

[3]This is always the case for exactly 64 bits

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

```
1: **Input**: Bitstream $bs$, Bitstream position table of LUT content
2: **Output**: File with localized AES LUTs
3: $S^j(x)$ denotes the $j$'th output bit of $S(x)$
4: $\mathrm{perm}_i(\cdot)$ denotes the $i$'th permutation of all 6!
5: mark_lut($\cdot$) writes the parameter to output file
6: //Generate AES search patterns
7: for sbox_bit = 1 to 8 do
8:     for mux_cfg = 1 to $\binom{8}{2}$ do
9:         Pick muxer configuration $(mu_1, mu_2)$
10:        Set cnt$_i$ to 0, $i$ = 1,2,3,4
11:        for $i$ = 0 to 255 do
12:            switch(get_mux_value(i))
13:            case(0,0):
               LUT$_1$[mux_cfg][sbox_bit][cnt$_1$++] = $S^{sbox\_bit}(i)$
14:            case(0,1):
               LUT$_2$[mux_cfg][sbox_bit][cnt$_2$++] = $S^{sbox\_bit}(i)$
15:            case(1,0):
               LUT$_3$[mux_cfg][sbox_bit][cnt$_3$++] = $S^{sbox\_bit}(i)$
16:            case(1,1):
               LUT$_4$[mux_cfg][sbox_bit][cnt$_4$++] = $S^{sbox\_bit}(i)$
17:            end switch
18:        end for
19:    end for
20: end for
21: //Dump LUT content with Algorithm 2
22: LUT[num_of_luts] $\leftarrow$ DumpLUT(bs)
23: //Search for AES pattern
24: for lut_index = 0 to num_of_luts do
25:     for sbox_bit = 1 to 8 do
26:         for mux_cfg = 1 to $\binom{8}{2}$ do
27:             for perm_index = 1 to 6! do
28:                 for $i$ = 0 to 255 do
29:                     if $perm_{perm\_index}$(LUT[lut_index])
                        == LUT$_{i/64}$[mux_cfg][sbox_bit][$i \mod 64$] then
30:                         mark_lut(lut_index, sbox_bit,
                                     mux_cfg, perm_index)
31:                     end if
32:                 end for
33:             end for
34:         end for
35:     end for
36: end for
```

discussed in Section VI and V in more detail.

To verify our results practically, we synthesized a publicly available AES core on a 6-bit-to-1 bit FPGA in order to analyze the corresponding bitstream. With the help of Algorithm 3, we could identify all S-box instances. We synthesized also one our own implementations and could identify exactly 640 LUTs that belong to AES S-boxes.

From this information, it can be inferred that a round-based AES implementation is used since there are $20 = \frac{640}{32}$ S-box instances of an AES S-box. By obtaining such a result, it is very likely 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. It should be noted that we were also able to identify all sixteen inverse S-boxes belonging to the AES decryption.

C. Measuring the Non-linearity of LUT Contents

In this section, we introduce another potential approach of detecting decomposed S-boxes, by again targeting the LUT contents. One advantage is that we do not have to consider the permutation configuration of the input pins anymore. We show that measuring the degree of linearity of all LUT contents may also be helpful for obtaining information from unknown FPGA designs.

As mentioned in the previous sections, the analysis of LUT content of a bitstream can reveal valuable information for an attacker. An algorithm can be implemented using several strategies. Each implementation strategy has an inherent characteristic that may be revealed through measuring the linearity. For example, in an unrolled design, it is very common that more S-box instances are utilized compared to the amount of S-boxes used by an iterated design.

In the case of cryptographic applications the underlying S-boxes have a very high degree of non-linearity. This is a necessary characteristic in order to defeat crypt-analytical attacks. Thus, the more S-box instances are used, the more (decomposed) non-linear LUTs can be expected. We call a LUT a "non-linear LUT", if its corresponding content is non-linear according to the *Walsh Coefficient*. A large amount of non-linear LUTs can already indicate the usage of S-boxes.

To measure the linearity of a LUT content, we use the *Walsh Coefficient*. It is a well-established measure, thus, we introduce the corresponding notation based on Leander et al. [7].

For two vectors $a, b \in \mathbb{F}_2^n$, we denote the inner product of a and b by

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

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*, we now use this measure to evaluate the AES design of Section III-B.

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 our AES design that uses 20 S-box instances. 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 in the opposite, a value of 64 indicates a very high degree. Due to $n = 6$, only 24 possible *Walsh Coefficients* can occur. The y-axis provides the number of occurrences regarding the LUTs of the underlying FPGA design.

As one can see, there are given 728 LUTs possessing a

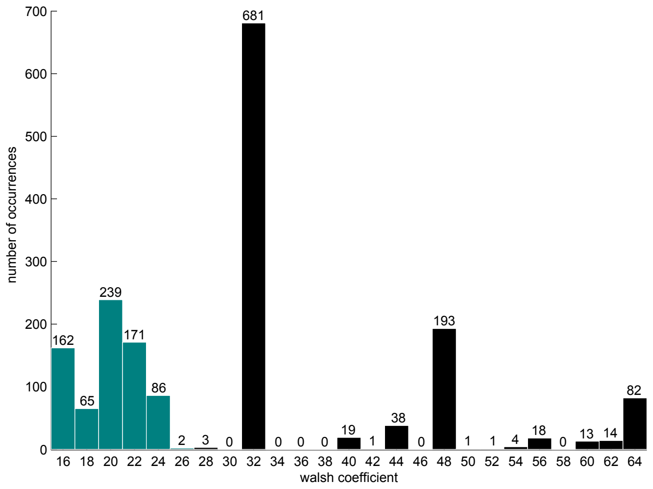

Fig. 2: Histogram of Walsh Coefficients evaluating the LUT contents of a LUT-based AES implementation

mid-high non-linearity, because for these LUTs, the *Walsh Coefficient* is smaller or equal to 28. Remember, that the previous approach of Algorithm 3, for identifying S-boxes, yielded 640 LUT belonging to AES S-boxes. With the help of the detection approach of Alg. 3, we could only observe 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.

As it is quite likely that a LUT-based AES implementation uses 32 LUTs per S-box instance (6-bit-to-1 bit architecture), an attacker should be able to estimate the amount of utilized S-boxes as $\lfloor \frac{728}{32} \rfloor = 22$. Considering false positives and implementation strategies for an AES design, an attacker could gain information regarding the implementation.

During our experiments, we also found LUTs that simply pass, e.g., a plaintext or ciphertext. The corresponding LUTs possess a very high linearity. Such information may also reveal further implementation details.

Measuring the degree of linearity can also be helpful in other scenarios. Consider a proprietary encryption algorithm that also uses proprietary S-boxes. Then, the approach with search patterns cannot be applied, because the corresponding S-box is unknown. In this case, the *Walsh Coefficient* may indicate which LUTs potentially implement a proprietary S-box. The corresponding LUT may be an attractive target for an attacker, who is able to modify the bitstream. To sum it up, the presented approach may help in the following cases:

- Identification of known S-boxes
- Identification of proprietary S-boxes
- Identification of key-dependent S-boxes
- Make predictions regarding 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 like 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 reverse-engineering.

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_bram 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, we create certain netlists, for which, we change all memory values bitwise. For each change, the bitstream is synthesized and the corresponding toggling bits are observed. We implemented the steps that are given in Algorithm 4 (no further explanation). Having obtained the mapping, we verified the correctness for several FPGA families. Note that there are certain set-ups for the memory layout that can be chosen by the user. We could verify that the contents of the embedded memory can be reverse-engineered – regardless of the chosen memory layout. This can be done with moderate programming efforts.

Algorithm 4 Extracting bitstream mapping of embedded memory content

```
1: Input: FPGA device file describing embedded memory
2: Output: Bitstream position table of embedded memory
3: set_bit_in_block(·) sets the memory content in netlist file
4: bs_ref: Reference file with zeroised memory content
5: bs_mod: Modfied embedded memory content
6: .net represents the netlist file
7: for block = 0 to num_of_memory_blocks - 1 do
8:     Create bitstream bs_ref with zeroised memory content
       for memory block block
9:     for bit = 0 to num_of_bits_per_memory_block - 1 do
10:        set_bit_in_block(bs_mod.net, block, bit)
11:        Synthesize bitstream bs_mod
12:        Compare bs_mod and bs_ref
13:        Store difference bit in position_table[block][bit]
14:        clear_bit_in_block(bs_mod.net, block, bit)
15:    end for
16: end for
17: 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, we were practically able to extract and modify the contents using the bitstream file.

C. Practical Evaluation of Open Cores

We evaluated several FPGA designs of the Advanced Encryption Standard that are offered by OpenCores[4]. After synthesizing some of the designs, using the standard options, each AES S-box instance was placed in the embedded memory. We were able to extract all S-box bytes from the corresponding bitstreams.

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 Data Encryption Standard (DES) and especially the Triple-DES (3DES) algorithms are still used nowadays. Therefore, both algorithms represent an attractive target to be weakened. This can be done, e.g., by directly modifying the bits of the bitstream that are related to the DES S-boxes. As we have demonstrated in the sections before, 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 width main key. Sixteen subkeys are derived from the main key by following a fixed scheduling plan. We demonstrate how easy it may be for an attacker to cancel the influence of the key by modifying the bitstream of the FPGA. Before doing so, we have to take a look at the inner working of the f-function.

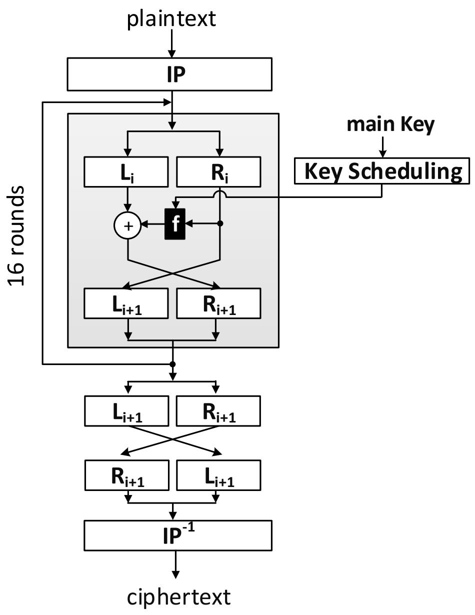

Fig. 3: Overview of the DES encryption algorithm

The basic properties of diffusion and confusion are realized by the f-function. More importantly, each subkey is usually 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 XORed with a subkey. Our goal is to directly modify the S-boxes in such a way that a modified ciphertext – carrying a scrambled plaintext

[4]http://opencores.org/

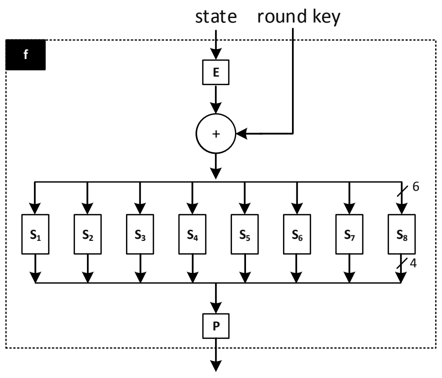

Fig. 4: DES round function f

– is computed. This automatically holds for all plaintext blocks being encrypted by the modified algorithm.

A. Modification

If all S-boxes ($S_1, ..., S_8$) 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 to the DES algorithm. To be more precise, the following equation has to be valid for all modified S-boxes:

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

Due to the presented modification, the whole DES algorithm turns into a (key-independent) permutation. The modified DES is visible in Figure 5. Usually, in a normal operating f-function, the S-box outcomes (32 bits) are permutated 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, consequently, the output of the permutation P is also completely zero. Hence, the output of the function f also becomes zero. Because the outcome (which is zero) of f is XORed with the left state L_i, L_i remains unchanged.

Thus, the state after $\text{IP}(\cdot)$ is not affected after 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 that we denote by $\text{IP}^{-1}(\cdot)$.

The following two equations compare the computations steps of a normal DES encryption with those of a modified $\widetilde{\text{DES}}$-encryption using $\text{S-box}^0_{\text{DES}}$. The modified encryption only applies three permutations on the plaintext that can be easily inverted by an attacker.

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

$$\widetilde{\text{DES}}_k(\cdot) = \text{IP}^{-1} \circ \text{Swap} \circ \text{IP}(p)$$

An attacker has to perform the following computation to obtain the plaintext from the ciphertext:

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

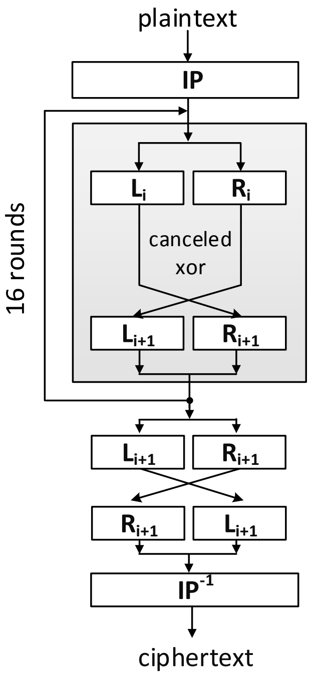

Fig. 5: Modified DES with canceled f-function

This attacks works likewise for Triple-DES. As described in [9], the 3DES encryption is computed as follows:

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

A plaintext from the modified 3DES with a $\text{S-box}^0_{\text{DES}}$ can be computed as follows:

$$\begin{aligned} p = &\text{IP}^{-1}(\text{Swap}(\text{IP}(&& \text{//invert } \text{DES}_{k_1}(\cdot) \\ &\text{IP}(\text{Swap}(\text{IP}^{-1}(&& \text{//invert } \text{DES}^{-1}_{k_2}(\cdot) \\ &\text{IP}^{-1}(\text{Swap}(\text{IP}(c)\dots) && \text{//invert } \text{DES}_{k_3}(\cdot) \end{aligned}$$

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. Moreover, the ciphertext appears to be a true one as it possesses a random looking shape and cannot be identified with visual inspection. We applied the S-box changes directly on two bitstreams and were able to successfully alter the design. The presented attack practically worked, either for a LUT-based and a RAM-based implementation.

Due to the fact that the DES algorithm does not exhibit any inverse S-boxes, the decryption also functions correctly. This severe modification may remain undetected, if, e.g., the bitstream encryption scheme is circumvented or if there are not any further selftests or integrity checks.

VI. Analysis of AES

The Advanced Encryption Standard is the most commonly used symmetric cipher today. In this section, we present further results of our analysis regarding malicious AES modifications. Similar to DES, an attacker may be able to silently weaken the algorithm such that the encryption and decryption process still works. For this purpose, we again alter all S-box instances. Furthermore, we discuss a key leakage approach, and in which scenarios it is feasible.

As described in the previous sections, we are able to detect any single S-box instance by reading in and analyzing the corresponding bitstream of an FPGA. We fully control all 20 AES S-boxes, i.e., can perform any desired manipulation. Because of that, we briefly introduce the AES algorithm [10]. Figure 6 shows an overview of the AES-{128,192,256} encryption scheme. The algorithm supports three different key sizes 128, 192, and 256 bit leading to the execution of 10, 12 and 14, respectively. One AES round consists of the operations

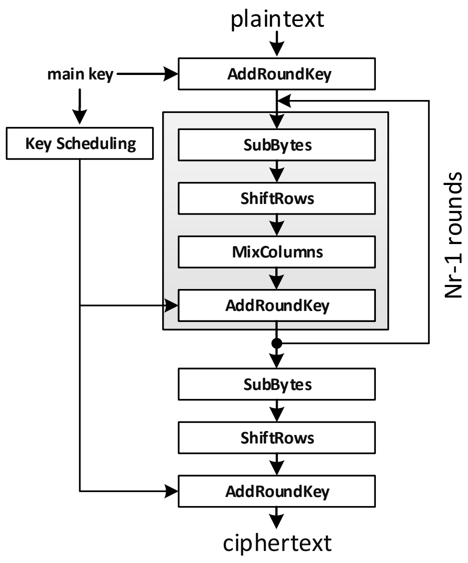

Fig. 6: Overview of the AES encryption algorithm

SubBytes, *ShiftRows*, *MixColumns*, and *AddRoundkey* that are executed consecutively. Thereby, the *SubBytes* step processes sixteen intermediate bytes by calling a fixed S-box. It is very common, for round-based implementations, to synthesize multiple S-boxes such that each input byte can be processed in parallel. In addition to that, the key scheduling also needs to process four S-box instances. Section VI-A describes the impact of replacing all S-boxes to an identity table, 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 like given in the equation below, the encryption and decryption function turns into a linear bijection. The property of non-linearity is then completely canceled. The corresponding modified AES can correctly encrypt and decrypt, but is extremely vulnerable to cryptanalytical attacks.

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

With the knowledge of at least one plaintext and ciphertext pair, an attacker can decrypt all other ciphertext blocks. In practice, this is an imaginable scenario, because there are a lot of applications that include, e.g., fixed metadata or file header.

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

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

Note that p denotes by a plaintext, $K_0, K_1, ..., K_{10}$ denote by faulty or normal subkeys, and $\tilde{c}$ denotes by a faulty ciphertext. The above equation holds, because the $\text{MC}(\cdot)$ and the $\text{SR}(\cdot)$ functions are linear as described below.

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

It is important to understand that $\tilde{K}$ can be expressed as XOR sum of all subkeys. Note that the number of $\text{MC}(\cdot)$ and $\text{SR}(\cdot)$ operations depend on the utilized AES mode. Knowing this, we further describe how $\tilde{K}$ can be recovered with the help of one $(p, \tilde{c})$ pair.

b) Recovering $\tilde{K}$: When an attacker can obtain one faulty plaintext and ciphertext pair $(p, \tilde{c})$, for a specific AES mode, he is then able to compute the secret $\tilde{K}$. For this purpose, he can simply reconstruct $\text{SR}(\ldots\text{MC}(\text{SR}(p)\ldots)$, and then compute the following:

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

With the knowledge of $\tilde{K}$, an attacker can recover any plaintext p from any faulty ciphertext $\tilde{c}$. To do so, the adversary has to XOR the value $\tilde{c}$ with the previously recovered secret $\tilde{K}$. Afterwards, the MC and SR have to be inverted. The inversion differs depending on the AES mode and key size. Algorithm 6 illustrates this concept in more detail. As indicated above, this

Algorithm 5 Decrypt Faulty Ciphertexts

```
1: **Input**: Ciphertext $c'$ from a modified AES ($\text{S-box}^{id}_{\text{AES}}$)
2:         One previously obtained $(p, \tilde{c})$ pair
3: **Output**: Plaintext $p'$ corresponding to $c'$

   //Calculate $\tilde{K}$
4: $\tilde{K} = \tilde{c} \oplus \text{SR}(\text{MC}(\text{SR}(\ldots\text{SR}(p)\ldots)$
   //Cancel secret $\tilde{K}$
5: $c' \leftarrow c' \oplus \tilde{K}$
   //Calculate $p'$ depending on the number of rounds
6: $p' \leftarrow \text{SR}^{-1}(\text{MC}^{-1}(\text{SR}^{-1}(\ldots\text{MC}^{-1}(\text{SR}^{-1}(c')\ldots)$
```

attack works regardless of the key schedule, because the secret

$\tilde{K}$ can be canceled in any case. Thus, it does not matter how the S-boxes are altered that are related to the key schedule. Please note that this kind of attack works for all three AES modes.

B. Replacing S-boxes to the Zero Function

Analogous to the DES modification of Section V, all synthesized AES S-boxes can be reconfigured to always output a zero – regardless of the input value – as it is depicted in the following equation:

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

Obviously, after having altered all S-box instances in the presented manner (means by modifying the corresponding bitstream), the AES algorithm becomes unusable. That is because any information regarding the plaintext becomes lost, right after the first SubBytes step has been processed by the modified AES instance. Considering the last AES round, c.f. Fig. 6, the resulting ciphertext is equal to the last subkey. This is case, if one only modifies the S-boxes that are related to the SubBytes step.

Such kind of attack can be useful, if the underlying main key is, e.g., hardcoded in the FPGA design and if it is not stored in the embedded memory. Suppose, that a main key may be securely transfered after the power-up of the FPGA, e.g., by a Hardware Security Module (HSM), whose data bus cannot be eavesdropped. With the help of the presented alteration, an attacker though can obtain the key, if he is able to query the AES instance with any arbitrary plaintext.

Since the S-boxes of the key schedule are usually not distinguishable from the SubBytes S-boxes, an attacker probably has to modify all S-box instances to zero. Also, there are given little differences between the AES modes. Thus, in the following, we also deal with these cases. We further assume that all SubBytes S-boxes are altered.

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

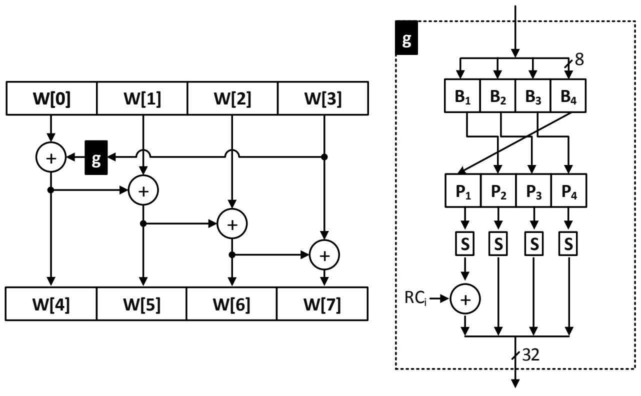

Fig. 7: Key schedule of AES-128

Algorithm 6 Reconstruction of the AES-128 main key

```
 1: Input: Ciphertext $c$ from modified AES (S-box$^0_{\text{AES}}$)
 2: Output: Fully recovered 128-bit AES main key.
 3: //Load modified ciphertext
 4: for $i$ = 0 to 3 do
 5:     $w[43 - i] = c[3 - i]$
 6: end for
 7: //Invert the 128-bit key schedule
 8: for $i$ = 39 to 0 do
 9:     if $i$ % 4 == 0 then
10:         $w[\text{i}] = w[i + 4] \oplus g(w[i + 3])$
11:     else
12:         $w[i] = w[i + 4] \oplus w[i + 3]$
13:     end if
14: end for
```

- The subkeys are determined using normal 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.
- The subkeys are determined using only S-box$^0_{\text{AES}}$ S-boxes. In this case, the g-function only returns the corresponding round constant $\text{RC}[i]$, also padded with three zeros. Code line 10 of Algorithm 6 should be then changed to

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

  in order to reveal the full key.

In the following, we also consider AES-192 and AES-256.

2) AES-192 and AES-256: Compared to AES-128, AES-192 and AES-256 only leak the key under special conditions. The graphical representations of both key schedule functions are shown in Figure 8 and 9. Similar to AES-128, two scenarios can occur if we have already modified all SubBytes AES S-boxes to zero:

a) The key schedule is calculated with normal AES S-boxes: The following explanation refers to AES-192, but also holds for AES-256.

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.

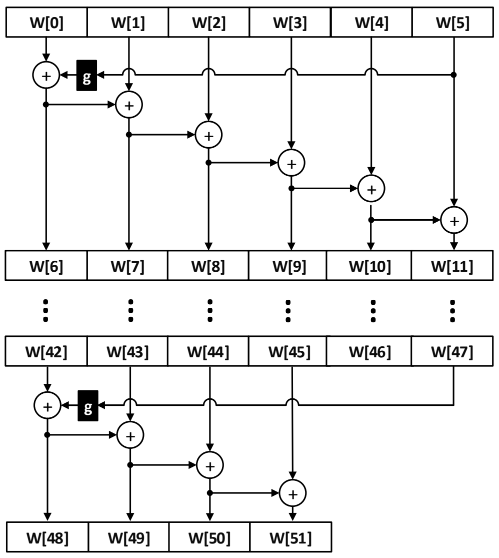

Fig. 8: Key schedule of AES-192

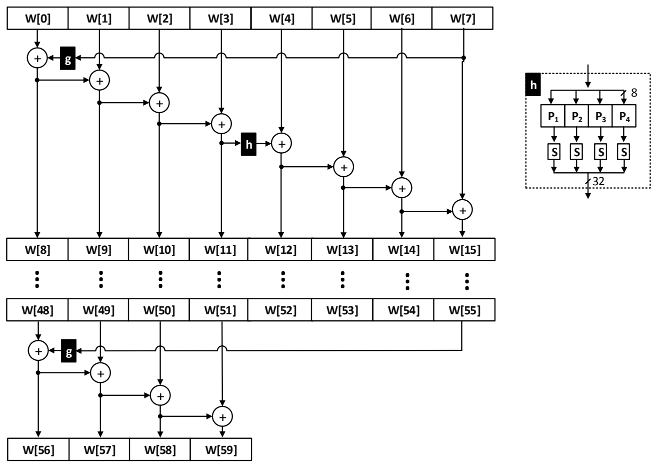

Fig. 9: Key schedule of AES-256

b) KeyExpansion with $\text{S-box}^0_{\text{AES}}$*:* In the case that the key schedule S-boxes are also set to zero, the first 128 bit of the main key can be derived. The explanation is related to AES-192, but 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 we know the *left* 4 words in the graphical representation of the key schedule. Even if the *right* part is not known, the first 4 words $w[0] - w[3]$ can be computed, c.f. Algorithm 7. The other bits cannot be computed. Having discussed the potential attack vectors, in the next section, we briefly describe some countermeasures.

Algorithm 7 Reconstruction of the AES-192 / AES-256 key

```
 1: Input: Ciphertext $c$ from modfied AES with $\text{S-box}^0_{\text{AES}}$
 2: Output: First 128 bit of 192/256 main key
 3: $N_w \leftarrow 51$ for AES-192 ( $\leftarrow 59$ for AES-256)
 4: $N_k \leftarrow 6$   for AES-192 ( $\leftarrow 8$ for AES-256)
 5: //Load the ciphertext
 6: for $i$ = 0 to 3 do
 7:     $w[N_w - i] = c[3 - i]$
 8: end for
 9: //Invert the KeySchedule
10: for $i = N_w$ to 0 do
11:     if $i \mod N_k \geq 4$ then
12:         continue
13:     end if
14:     if $i \mod N_k == 0$ then
15:         $w[i] = w[i + N_k] \oplus \text{RC}[i]$
16:     else
17:         $w[i] = w[i + N_k] \oplus w[i + N_k - 1]$
18:     end if
19: end for
```

VII. Countermeasures

In this section, we discuss several countermeasures that may be deployed in order to raise the bar for an adversary, who is able to modify cryptographic S-boxes in FPGA designs.
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. Integrated Selftest

A simple integrated selftest 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 selftest has several drawbacks as listed below:

- The integrity value has to be stored somewhere. An adversary may be able to change it easily.
- The adversary could disable the selftest 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 easily detectable, because of their unique output patterns. Critical Boolean equations, generating the LUT contents, should be difficult to distinguish from other (linear) LUT patterns. This could be achieved with further decomposing the LUTs along its Disjunctive Normal Form (DNF). For example, in a 6-bit-to-1 bit architecture, a 64-bit LUT content may be splitted 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 $f(a, b, c) = ab + bc + abc$. 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 scheme is unknown to an attacker. The decomposition to multiple LUTs comes with a drawback, too:

- An adversary could look for these patterns or try to modify a chosen set of candidate LUTs in the bitstream and observe the FPGA's output. This can be repeated for several times until the S-box is detected.

Even, when the set of candidates is 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. For example, it might be more challenging, if the number of split up LUTs is chosen randomly for every S-box column.

C. Whitebox Cryptography

One could deploy whitebox cryptography as a countermeasure. The main idea is to hide the secret key inside the implementation [11]. Key-dependent LUTs together with random transformations generate the masking of a fixed key. Even for this kind of countermeasure there is one drawback: An adversary could create a copy of the bitstream and use it as an oracle to decrypt the ciphertext or encrypt the plaintext. To sum it up, there are possibilities to defeat the attacks proposed in this paper. Nevertheless, further research has to be investigated. One needs to evaluate how an FPGA design can be secured against bitstream modification attacks.

VIII. Conclusion

In this work, we have demonstrated how to detect and modify cryptographic primitives in FPGA bitstreams. The targeted cryptographic algorithms were DES and AES. We have successfully modified and weakened several FPGA designs (IP cores and own implementations) by altering the corresponding bits of a bitstream. To mount this kind of attack, we only need the FPGA's bitstream, which is a realistic scenario in the real world. We briefly described how the bitstream of an FPGA can be partially reverse-engineered with moderate efforts. We have shown that an attacker can obtain valuable information from a bitstream, which may help to improve side-channel attacks. This is due to the recoverable information of the design architecture and LUT localization inside the FPGA. We verified our attacks practically for DES and AES on an FPGA of a well-known vendor. For DES, we modified the bitstream in such a way that the encryption becomes key-independent and is altered to a revertible permutation.

The AES was first modified to leak (parts of) the AES main key, and in a second similar attack, we turned the AES into a simple linear function that produces weak ciphertexts, and thus, it is behaving like a hardware Trojan.

This work should arise awareness that an attacker can modify proprietary FPGA bitstreams by purpose. This must not necessarily be a cryptographic function. Thus, it is important to carefully check an Intellectual Property (IP) core, before using it in high-security applications. Our results also highlight the importance of integrity checks. Further, security mechanisms must be deployed around the FPGA. Because the bitstream file format cannot be completely changed anymore, future work demands for the development of new (low-level) countermeasures.

Acknowledgment

The authors would like to thank Christian Kison for providing help with the VHDL implementation. This work has been supported by the Hans L. Merkle Foundation.

References

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

[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 *CCS 2011*. 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 *CT-RSA 2012*, ser. LNCS, 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: ACM, 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, 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, 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," *CACM*, 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] NIST, "FIPS 197 Advanced Encryption Standard (AES)," 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

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