Configuration Tampering of BRAM-based AES Implementations on FPGAs

Daniel Ziener
Computer Architecture for Embedded Systems
University of Twente
Email: d.m.ziener@utwente.nl

Jutta Pirkl and Jürgen Teich
Hardware/Software Co-Design
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Email: {jutta.pirkl,juergen.teich}@fau.de

***Abstract*—Fault injection attacks constitute a major attack vector on cryptographic implementations, such as the Advanced Encryption Standard (AES). On Field Programmable Gate Arrays (FPGAs), the circuit can be altered by tampering the configuration data and thereby causing a desired faulty execution that leaks information about the secret key. Often it is not even necessary to conduct extensive reverse engineering of the propriety bitstream file format. In this paper, we present a novel strategy to recover the secret AES key by exploiting the properties of the FPGA's memory elements called Block RAM (BRAM) that are often used to store the Rijndael S-boxes. The attack can be performed by a single reconfiguration with a faulty bitstream without any knowledge of either design properties or plaintext input. The advantage of our approach is that this attack works also with encrypted bitstreams. However, our experiments show that the number of reconfigurations might increase in this case.**

***Index Terms*—FPGA, Advanced Encryption Standard (AES), bitstream manipulation, hardware security**

I. Introduction

As embedded systems are getting more and more networked, security has become an important issue. In order to ensure that confidential information is only accessible to those authorized to read it, cryptographic algorithms, such as the well-known *Advanced Encryption Standard* (AES), are deployed. For implementing such cryptographic algorithms, FPGAs represent an efficient platform. However, in order to be hardened against any physical attacks, each security-critical implementation must be protected. One major class of physical attacks on cryptographic hardware implementations is *fault injection* which aims at manipulating the execution and afterwards deducing the key from one or more faulty chiphertext outputs. Whereas circuit manipulation via focused laser beams requires much effort and expertise, Field Programmable Gate Array (FPGA) implementations can be altered easily by tampering the bitstream and thereby changing the configuration memory to create the desired faulty behavior. Nevertheless, it can be extremely difficult if a specific fault has to be injected that requires to reverse-engineer parts or the whole bitstream. If the bitstream is additionally encrypted, the method seems to be infeasible anyhow. However, knowledge of the whole or part of the design is not always necessary in order to conduct a successful recovery of the AES key.

In [1], an attack is proposed that automatically manipulates Look-Up Table (LUT) content in the bitstream with several strategies, such as toggling an entire LUT configuration. The resulting faulty ciphertexts and corresponding plaintext inputs are afterwards tested against fault hypotheses which leads to a recovery of the secret key. The authors were even able to show that bitstream encryption delivers no secure protection against their attack.

Often, the Rijndael S-box are implemented with Block RAMs (BRAMs) in order to optimize the logic resource utilization or performance. In this paper, we show that this strategy opens vulnerabilities against fault injection attacks which require extremely low effort to extract the key. We recover the secret key by manipulating the configuration bits for controlling the BRAM elements in order to disable or reset the S-box instances. Such BRAM-based implementations are vulnerable to the attack proposed in [1], too. However, with the attack proposed in this paper, the key can be extracted in seconds to minutes. For example, if the FPGA configuration is not encrypted, only *one faulty ciphertext* is required without any knowledge of either design properties or the corresponding plaintext input. For encrypted configurations, more tries might be necessary. However, the extraction time is lowered to less than one minute compared to hours by applying [1].

The remaining part of the paper proceeds as follows: The related work on similar attack are discussed in Section II. Section III outlines the background for our proposed attack on the AES algorithm. In Section IV, we provide an overview of the practical implementation of the attack for both unencrypted and encrypted bitstreams including bitstream reverse-engineering. Subsequently, the impact of this attack and possible and already existing countermeasures are discussed in Section V. Finally, Section VI concludes the paper.

II. Related Work

To date, several publications have investigated the security of BRAM primitives in FPGAs. Besides side-channel attacks [2], [3], there also exists prior work on successful fault injection by bitstream manipulation targeting the embedded memory units.

By analyzing the AES algorithm, it is clear that by setting the output of the S-box in the last round to zero, the (round) key can be obtained at the encryption or decryption output [4]. For FPGA-based implementations, the S-box can be either implemented with lookup tables (LUTs) or block memories (BRAMs). By setting the whole content of S-boxes to zero, the key can be immediately recovered.

For example, Swierczynski et al. [5] propose an attack that exploits BRAM-based S-box implementations by reverse

978-1-7281-1968-7/18/$31.00

engineering the corresponding bits of the BRAM configuration content, searching for the S-box function within these locations, and subsequently replacing the bitstream data with either zeros or ones. An extension of this concept to T-boxes, which store a merged *SubBytes* and *MixColumns* operation, is shown in [6]. Similar to [5], the T-box content is identified in the bitstream and afterwards modified in order to create an information leak of the AES circuit which in turn enables the adversary to retrieve the secret encryption key. An advantage of their approach is that a known plaintext query is not necessary, the observation of a single faulty ciphertext is sufficient to successfully obtain the key. Yet, the S- or T-box identification can be easily prevented by either obfuscation techniques and more effectively, with bitstream encryption.

In [1], an attack is proposed that automatically manipulates LUT content in the bitstream with several strategies, such as toggling an entire LUT configuration. The resulting faulty ciphertexts and corresponding plaintext inputs are afterwards tested against fault hypotheses which leads to a recovery of the secret key.

Xilinx introduced configuration bitstream encryption for Virtex-II devices in the year 2000. During the loading, the bitstream is decrypted with a Triple-DES hardcore inside the configuration controller of the FPGA. Later on, Xilinx choose an AES-256 algorithm for encryption. Trimberger et al. discovers in [7] weaknesses in the used CBC mode which might be exploited by fault injection attacks. Single targeted bits can be toggled in the decrypted bitstream configuration data, as will be explained in Fig. 5 in Section IV-C.

In [1], Swierczynski et al. transfer the above mentioned attack also to encrypted bitstreams. Despite limitations to bit toggles and involuntary destruction of the previous 128-bit block in the raw bitstream data, the authors were able to launch successful fault injection attacks. In this way, it was possible to change the LUTs' content, e.g., toggling an entire LUT, and collect faulty ciphertexts. From these erroneous outputs in addition to the corresponding plaintext inputs, the key can be recovered by testing against some hypotheses.

Taken together, these publications show that bitstream manipulation attacks are feasible and thus pose a serious threat to FPGA-based cryptographic implementations, such as the AES. In comparison to our work, their attack requires significantly more effort, e.g., they have to generate 86400 faulty bitfiles [1].

III. Background and Adversary Model

Before our attack is presented, the background of AES encryption as well as fault injection attacks are presented, followed by the assumptions and adversary model.

A. Advanced Encryption Standard (AES)

The AES was established by the US National Institute of Standards and Technology (NIST) in 2001 [8]. It is a symmetric block cipher, i.e., the same key is used for en- and decryption. There are three versions that differ in the bit length of the key and consequently the number of rounds required to calculate the ciphertext: AES-128, AES-192, and AES-256 with 10, 12, and 14 rounds, respectively. The plaintext is processed in separate 128-bit blocks which are encrypted in a 128-bit ciphertext output. Except for the last round, each round includes four operations called *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. They operate on a 4×4 column-major order matrix of the 16-byte (128-bit) block. In *SubBytes*, each byte is replaced by another byte in a non-linear fashion as defined in the 8-bit Rijndael S-box. In *ShiftRows*, the bytes in each row are cyclically shifted to the left by an offset varying with the row number. In *MixColumns*, the four bytes of each column are scrambled by a matrix multiplication, such that each input byte influences all four output bytes. In *AddRoundKey*, the state matrix is combined with a round key by bit-wise XOR-operation. The i-th round key K^i can be calculated with the master key K using Rijndael's key schedule. Thus, *AddRoundKey* is the only operation influenced by the key.

B. Fault Injection to Attack the Final Round

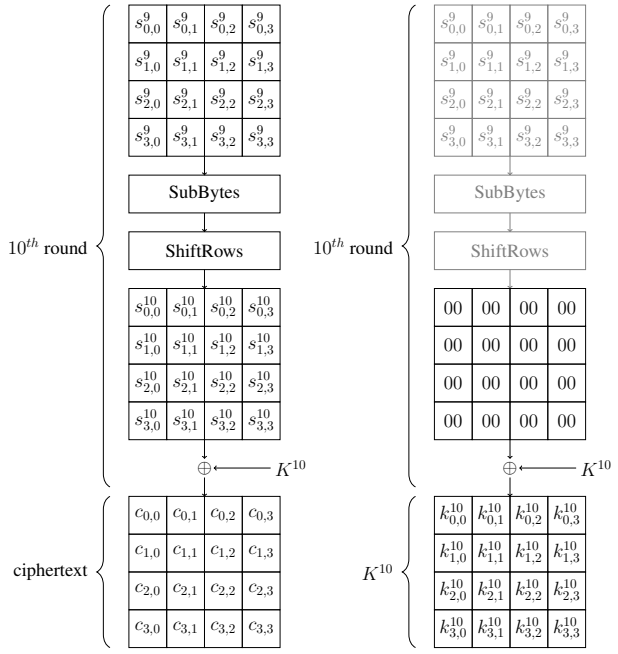
As shown in Fig. 1a, the last calculation is a bit-wise XOR-operation of the state matrix and the 10-th round key. This step constitutes a known vulnerability [4]–[6]. If the state can be fixed to a constant value known by the attacker, e.g., all-zero, the ciphertext output equals the last round key K^{10} independent of the plaintext input. In turn, the last round key can be used to recover the key from the key schedule algorithm. The difference in the execution between the undisturbed and faulty implementation can be compared in Fig. 1b. In order to realize this scenario, an attacker can manipulate the S-box within the *SubBytes* operation to always output the zero-byte.

A significant advantage of this approach is that the adversary only needs to observe one faulty ciphertext output, i.e., the last round key. Yet, the corresponding plaintext input does not need to be known. In this work, the S-box tampering will be performed by injecting corresponding faults in the bitstream of an FPGA.

C. Assumptions and Adversary Model

The following is a brief description of the applied invasive physical attack scenario for injecting malicious bitfiles into Static Random-Access Memory (SRAM)-based FPGAs.

These devices require an external storage of the bitstream in, e.g., a flash memory or Electrically Erasable Programmable Read-Only Memory (EEPROM), which has to be transferred into the internal configuration memory of the FPGA upon each power-up. We assume that the attacker has access to the bitstream data. This might be achieved, for example, by either possessing read and write permission on the external memory or electronic eavesdropping on the configuration data bus. Furthermore, access to a configuration interface must be provided in order to configure the FPGA with the manipulated bitstream. The goal of the adversary is to extract the secret key of an AES encryption implementation. It is evident that no one has direct access to this confidential data. In order to retrieve the encryption key with the proposed fault injection attack, the adversary must be able to observe the ciphertext output of the AES implementation. However, the initiation of a query



(a) Untampered execution. (b) Faulty execution.

Fig. 1. Overview of the final round of the AES-128 encryption algorithm. Fig. 1a presents the fault-free execution of the 10^{th} round and its resulting ciphertext output. In comparison, Fig. 1b shows the faulty execution leading to a leak of the last round key K^{10} which explains the underlying idea the presented attack is based on.

with a known or specifically selected plaintext input is not necessary.

IV. BRAM-based Bitstream Tampering Attack

BRAMs are one of the major site types on the FPGA fabric in addition to LUTs and Digital Signal Processing (DSP) blocks. On current Xilinx FPGAs, each BRAM site can be configured as either one 36 Kbit memory (RAMB36) or two standalone 18 Kbit RAMB18 with in turn two access ports each. Thus, an S-box can be implemented using one port of an 18 Kbit block with an address and data width of 8 bit. In order to provide several S-box lookups per clock cycle, as required in parallel AES implementations, one BRAM primitive can host up to four parallel S-boxes. Consequently, all four access points of the two independent 18 Kbit blocks are occupied.

If an attacker wants to blank the S-box output to achieve the scenario described in Section III-B, we propose she can utilize the enable (EN[A|B]) or synchronous set/reset ports (SSR[A|B]), as shown in Fig. 2. These ports can be configured in the bitstream with an inverted polarity, i.e., high-active reset ports can be converted to low-active ones. When the input wiring remains unchanged this leads to a permanent reset or disabled state, respectively, forcing the BRAM output to a constant predefined value which is in the most case zero. The predefined initial and reset values can be configured and are part of the bitstream. However, on all evaluated AES implementations, it was always set to zero.

In order to launch this attack, the relevant configuration bits of the BRAMs must be reverse engineered first. This can be achieved by changing the above mentioned attributes for each individual RAMB36 site in a netlist occupying all available

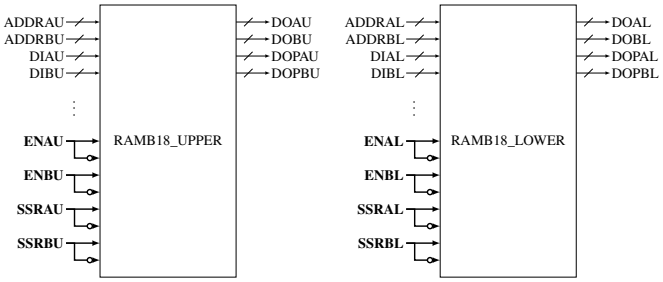


Fig. 2. Interface of a RAMB36 Site configured as two independent 18 Kbit RAMB18. The enable (EN[A|B]) and synchronous set/reset ports (SSR[A|B]) can be configured as high- or low-active.

BRAMs, generating a bitstream and subsequently comparing the result with the original bitstream. Note that there exist 16 possible combinations per site for each EN and SSR port due to the two ports per upper and lower part. The whole process can be automated with the commands available in Xilinx ISE's FPGA Editor.

The results for a Virtex-5 device (xc5vfx130t) revealed a division of the 298 BRAMs in ten segments. A segment corresponds to a vertical clock region which includes the left and the right clock region. As can be seen from Fig. 3, each segment usually contains four BRAM lines, i.e., 32 RAMB36 primitive instances[1]. At first, the BRAMs in the upper half of the FPGA are configured starting from $Y = 20$. Afterwards, the BRAMs in the lower half are configured, also starting from $Y = 19$.

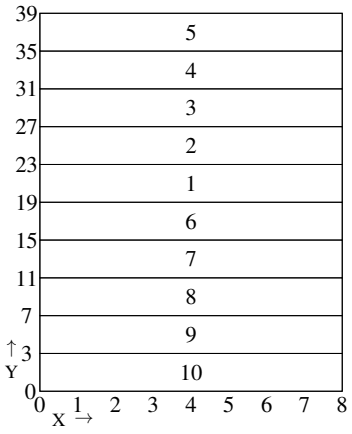


Fig. 3. Order of BRAM configuration bytes in Virtex-5 bitstreams (xc5vfx130t). Within each of the depicted segments, the configuration is arranged in vertical frames, i.e., the control bits of four vertical BRAMs are configured in one frame. Each segment consists of 2598 frames and each frame is 164 Bytes long. Please note that for $X = \{1, 2, 8\}$, some BRAMs do not exist for the used device.

The control bits for the enable and reset polarity resides in one frame for all four vertical BRAMs in one segment. For each polarity change, only one bit has to be toggled. The polarity control bits for the ENA, ENB, SSRA, and SSRB

[1]Apart from the exceptions due to the lack of BRAMs with X-coordinates = $\{1, 2\}$ and Y-coordinates = $\{8, \ldots, 15, 24, \ldots, 31\}$ due to the embedded processors and with X-coordinates = 8 and Y-coordinates = $\{4, \ldots, 21, 24, 25, 28, \ldots, 37\}$ due to the PCIE and TEMAC primitives.

are directly together in one byte. For each RAMB38 instance, two bytes are affected, one for the upper and on for the lower RAMB18 instance. Fig. 7 shows the affected bytes of the whole FPGA.

A. Experimental Setup

As indicated above, we conducted our experiments with a Virtex-5 device (xc5vfx130t). It was selected because it uses configuration bitstream encryption without using secure integrity check methods used by newer devices. The used Cyclic Redundancy Check (CRC) integrity check is directly embedded in the bitstream footer [1]. Since this part of the bitstream is always unencrypted, the corresponding command can be adjusted even if bitstream encryption is enabled. This means, that the CRC check can either be disabled, or the checksum can be recalculated. Consequently, the manipulated bitstreams can be loaded into the FPGA without causing a most likely CRC failure. In newer Xilinx FPGA families, such as the 7-series architecture, the integrity is evaluated by an Hash Message Authentication Code (HMAC). The latest UltraScale$^+$ devices even enhance this security feature by offering bitstream authentication with either AES-Galois/Counter Mode (GCM) or Rivest–Shamir–Adleman (RSA) authentication. Thus, an attack that requires to modify the bitstream is infeasible on the more recent Xilinx FPGAs unless the integrity check can be circumvented. However, the underlying idea of our presented attack can generally be transferred to these devices. Especially, the scenario without bitstream encryption can be applied directly.

Table I shows the evaluated AES implementations and their resource utilization. Note that the AES implementations which are used for evaluating the attack in [1] are included in this list. The table shows a representative selection of open-source AES implementations. The half of these implementations using BRAMs for the S-Boxes. Furthermore, the BRAM-based implementations are usually utilizing fewer lookup tables than the implementations without using BRAMs. The number of BRAM-based AES implementations could be increased, if the implementations will be optimized for FPGAs. Of course, only the BRAM-based implementations are investigated to attack. To communicate with a standard PC, a UART-interface was used.

TABLE I
DIFFERENT OPEN-SOURCE AES DESIGNS, IMPLEMENTED ON A XILINX VIRTEX-5 DEVICE.

Name	Ref.	FFs	LUTs	RAMB18s
AES_128_ENC	[9]	922	562	9
AES_128_fast	[10]	789	11,213	0
AES	[11]	668	1,074	12
aesenc	[12]	329	1,098	4
AES_ENC	[13]	404	802	9
AES_Comp_ENC	[13]	533	2,361	0
AES_PPRM1_ENC	[13]	591	2,439	0
AES_PPRM3_ENC	[13]	536	2,590	0
AES_TBL_ENC	[13]	407	893	8
AES_top_example	[14]	1,533	3,629	0
aes_cipher_top	[15]	1,423	8,872	8
RIJNDEL_ITER	[16]	668	4,445	0
RIJNDEL_PIPE	[16]	3,214	25,090	0
aes_pipe	[17]	6,273	5,196	100

B. Single-shot Attack without Bitstream Encryption

When an adversary has access to the unencrypted bitstream data, the key used by the AES implementation under attack can be recovered in three steps:

- Firstly, the corresponding bits in the configuration data must be reset to disable all BRAMs on the chip.
- Secondly, the FPGA must be configured with the manipulated bitstream and the output of an arbitrary AES query must be observed.
- Thirdly, the key must be calculated with an inverse key schedule.

As mentioned in Section III-B, the output of the encryption directly leaks the last round key K^{10} when all S-boxes of the last round are blanked. Yet in this described attack, every BRAM on the entire chip is disabled and consequently also the S-boxes used in the key schedule for calculating the round keys output zero-bytes. This leads to a simplification of the key expansion process as shown in Fig. 4. The calculation of the i-th word in the expanded key of length 44 words is defined as follows:

$$w_i = w_{i-4} \oplus F_i(w_{i-1}), \tag{1}$$

where w_0, w_1, w_2, and w_3 contain the key. If i is a multiple of four, $F_i(w_{i-1})$ corresponds to

$$F_i(w_{i-1}) = SubWord(RotWord(w_{i-1})) \oplus RC_i, \tag{2}$$

i.e., the input word is first rotated byte-wise and afterwards its output of the S-box is XOR-ed with the i-th round constant. Otherwise, $F_i(w_{i-1})$ resembles the identity function. With blanked S-boxes the expression simplifies to

$$F_i(w_{i-1}) = RC_i, \tag{3}$$

where the word RC_i contains the one byte round constant padded with three zero-bytes.

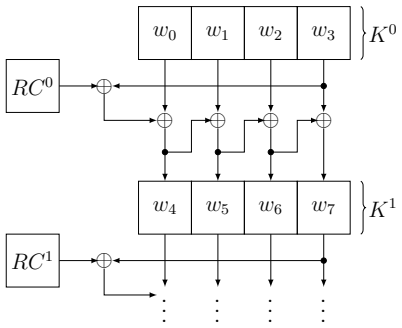


Fig. 4. Simplified AES-128 key schedule due to tampered S-boxes.

For the inverse key schedule, Eq. (1) can be rewritten with $j = i - 4$:

$$w_j = w_{j+4} \oplus F_i(w_{j+3}). \tag{4}$$

Thus, the expanded key can be computed backwards starting with w_{40} to w_{43} containing the faulty ciphertext output. In the end, w_0 to w_3 result in the secret key.

C. Attack with Bitstream Encryption

In case of bitstream encryption on the target device, the relevant configuration bits cannot be set to a specific value. Since a modification of one bit in the ciphertext, i.e., encrypted bitstream, affects all bits in the same 128-bit block in an arbitrary manner during the decryption process on the FPGA, the attack described above must be adjusted. Yet, the bitstream encryption for Xilinx devices uses an AES-256 with Cipher Block Chaining (CBC) mode and thus opens a security loophole, as described in [7].

The decryption in CBC mode is defined as follows:

$$P_i = \text{AES-256}_K^{-1}(C_i) \oplus C_{i-1} \qquad \forall i > 0 \tag{5}$$

and

$$P_0 = \text{AES-256}_K^{-1}(C_0) \oplus IV, \tag{6}$$

where P_i and C_i denote the i-th 128-bit block of the decrypted plaintext and ciphertext, respectively. The initialization vector (IV) required in CBC mode is used to determine the first 128 bits of the plaintext and can be found in the bitstream header. It is not necessary to keep the initialization vector secret in order to maintain the security aspects of the AES encryption. Fig. 5 provides an overview of this decryption mode.

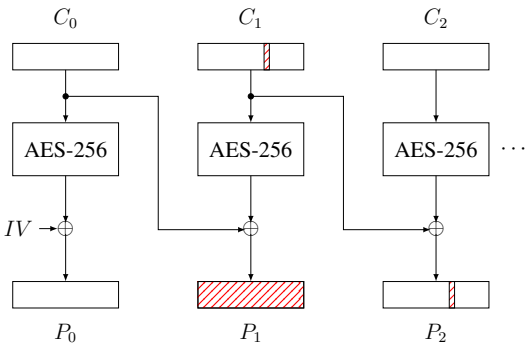


Fig. 5. Decryption of the AES-256 in CBC mode, as used for bitstream decryption in Xilinx devices. A bit flip in one ciphertext block leads to the destruction of the decrypted plaintext block, but also a bit flip at the same position in the following block.

Furthermore, it can be seen from the figure, that it is possible to toggle a specific bit in the decoded plaintext block P_i when toggling the same bit position in the previous ciphertext block C_{i-1} due to the XOR-operation. However, this also leads to an unknown scrambling in the plaintext block P_{i-1} and thus the adversary might destroy relevant bits of the circuit configuration. Additionally, it is not possible to toggle a specific bit in neighboring blocks P_i and P_{i+1}, because the fault in C_i will destroy P_i.

To investigate this behavior, an FPGA design was created which utilize all 596 RAMB18 of the used xc5vfx130t device. The BRAMs are configured as ROMs with preinitialized values and dual port access. Over an UART-interface, all 1192 ports can be accessed and the content of all BRAMs can be read out. The resulting bitstream was encrypted with the Xilinx tools. The BRAM control bits are toggled according to the above described attack. We toggle only SSR, only EN, and EN & SSR bits, as well as two different additional bits. From the read out results, we determine the number of successfully switched off BRAM ports with constant zero outputs as well as the failures. Different encryption keys for the bitstream encryption are used to generate different corrupted blocks even if the toggled bit positions are the same.

TABLE II
RESULTS FROM THE BRAM CONTROL BIT TAMPERING WITH ENCRYPTED BITSTREAMS.

Toggled bits Type	No.	Output Corr.	Const.	Zero	Oth.	Success
Bitstream encryption with key K_1						
only EN	1,192	20	62	1,102	8	92.4%
only SSR	1,192	21	143	1,018	10	85.4%
EN & SSR	2,384	16	82	1,086	8	91.1%
EN & SSR & a	3,576	23	71	1,093	5	91.7%
EN & SSR & b	3,576	18	51	1,115	8	93.5%
Bitstream encryption with key K_2						
only EN	1,192	18	73	1,090	11	91.4%
only SSR	1,192	19	165	990	18	83.1%
EN & SSR	2,384	25	64	1,094	9	91.7%
EN & SSR & a	3,576	13	76	1,098	5	92.1%
EN & SSR & b	3,576	20	73	1,093	6	91.7%
Bitstream encryption with key K_3						
only EN	1,192	22	70	1,084	16	90.9%
only SSR	1,192	19	184	983	6	82.5%
EN & SSR	2,384	17	82	1,090	3	91.4%
EN & SSR & a	3,576	12	78	1,091	11	91.5%
EN & SSR & b	3,576	14	78	1,083	17	90.9%
Bitstream encryption with key K_4						
only EN	1,192	28	58	1,101	5	92.4%
only SSR	1,192	16	143	1,027	6	86.2%
EN & SSR	2,384	17	59	1,103	13	92.5%
EN & SSR & a	3,576	16	87	1,077	12	90.4%
EN & SSR & b	3,576	32	69	1,078	13	90.4%

The results in Table II with bitstream encryption with four different keys (K_1 – K_4) show that in average 90.2% of all BRAMs can be set to zero with only one tampered bitstream. Around 1.7% of the BRAMs are still working and $\approx$ 7% producing a constant output which is not zero. Moreover, $\approx$ 0.7% of the BRAMs producing random or unstable outputs. Some of these other BRAMs further deliver different values if the memory is read out more often. I must be noted that the BRAMs whose output cannot be set to zero are not the same at each try. With only the five tries with different fault injections, the output of all BRAMs can be at least for one try set to zero. However, some BRAM locations are more susceptible to produce the correct or constant non-zero results. For the correct results, it can be explained by the alignment of the 128 bit encryption blocks. For example, the positions for the affected bytes at the BRAMs with the coordinates $X = 1$, $Y = \{0, \ldots, 3\}$ are shown in Fig. 6 with the 16 byte alignment of the 128 bit encryption blocks. It can be seen that the sixth modified byte destroys the fifth modified byte by generating random values for all 16 bits in the previous encryption block. Fig. 7 shows all modified bytes with the destroyed ones in red.

In Fig. 7, it can be seen that always one lower RAMB18 block is affected on the eight vertical RAMB18 instances in one segment. All BRAMs with correct output values belongs to this group of 149 affected BRAMs. However in Table II, much fewer BRAMs have correct output values. If the corresponding control byte is randomized, the chance that the

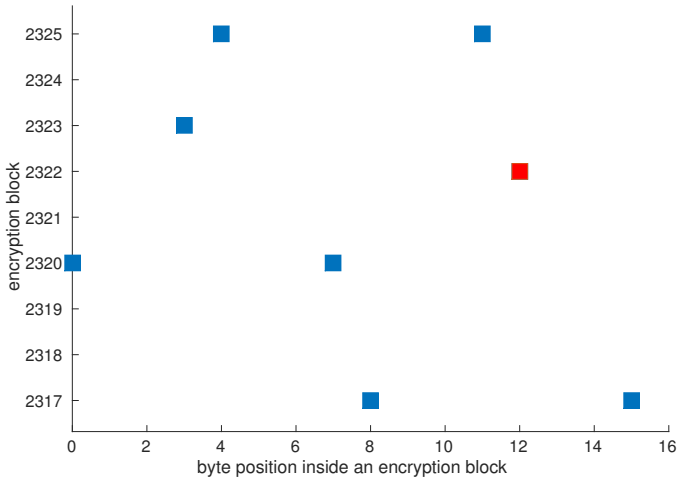


Fig. 6. The alignment into the 16 byte encryption blocks of the modified bytes of the BRAMs with the coordinates $X = 1$ and $Y = \{0, \ldots, 3\}$ are shown. The red modified byte is destroyed by the corrupted encryption block 2322 generated by the sixth modified byte inside the encryption block 2323.

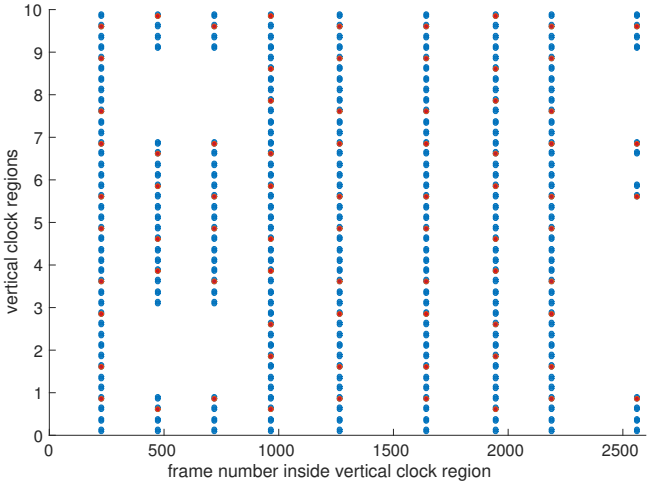


Fig. 7. In blue, all modified bytes to switch off the BRAMs of the whole FPGA are depicted. The space for the processors and the PCIe cores can be clearly seen. In red, the modified bytes which are overwritten by a corrupt encryption block, produced by the following modified byte are shown.

polarity is set correctly for a working BRAM is 25% (2 bits – one for EN and one for SSR). This would result in 37.25 correctly working BRAMs on average. The values in Table II are lower which indicate that an additional configuration bit could be involved.

The BRAMs with constant non-zero outputs could be generated by overwriting the initial or reset values for the internal registers in the BRAM primitive cell. However, the evaluation of the configuration location of the initial value (INITA, INITB) shows that these cannot be affected by the modified bytes.

In summary, we are able to set the output of on average 90.2% of all BRAMs to zero with only one falsified configuration even if the configuration is encrypted. By looking at the evaluated AES implementations, the chance to set all outputs of the used BRAMs in the last round and in the key schedule to constant zero with only one configuration is shown in Table III. This means that all used BRAMs in the last round and in the key schedule belongs to 90.2% of all BRAMs. If this is not the case, additional bits (e.g., a and b in Table II) could be toggled to generate new random values for the corrupted AES encryption blocks. However, these bits must be chosen carefully to avoid the generation of new corrupted blocks.

TABLE III
PROBABILITY FOR SUCCESSFUL ATTACKS WITH ENCRYPTED BITSTREAMS

Name	**Ref.**	**RAMB18s**		**Prob. with one try**	**No. tries to reach 50% chance**
		used	**to disable**		
AES_128_ENC	[9]	9	9	39.5%	2
AES	[11]	12	12	29.0%	3
aesenc	[12]	4	4	66.2%	1
AES_ENC	[13]	9	9	39.5%	2
AES_TBL_ENC	[13]	8	8	43.8%	2
aes_cipher_top	[15]	8	8	43.8%	2
aes_pipe	[17]	100	28	5.6%	13

V. DISCUSSION & COUNTERMEASURES

In contrast to the approach in [1], our approach focuses on disabling BRAMs with as few as possible configuration bits. The approach in [1] modifies the content of lookup tables which needs more faulty configurations with more configuration bits involved. On encrypted bitstreams, these generates also a lot of corrupted encryption blocks which might disturb the encryption process. For example, the key of the AES implementation of [15] could not be recovered. Note that we are able to extract the key with our approach also for this implementation. Furthermore, they need 86400 faulty bitstreams to extract the key which takes 11.5 hours. On larger FPGAs (e.g., our used FPGA is three times larger), the number of bitstreams as well as the time increases. Even on encrypted bitstreams, we need only few faulty configurations (see Table III) which leads to a key extraction time of less than one minute. This renders new possibilites for attacking hardware systems. On the other hand, the approach in [1] is more general and working also with implementations which do not utilize BRAMs.

In case that BRAMs are used also for other cores on the FPGA, our attack might not properly work, if the output of the AES core could not be transmitted to a location on which it can be recorded. This could be the case if after the encryption the data is send through other BRAMs. However, this is only true for BRAMs after the AES core since our approach is independent from the AES input, e.g., plain text. Nevertheless in this case, the BRAMs could be manipulated consecutively in order to deactivate only the BRAMs for the last AES round.

One of the most effective countermeasures against the proposed attack is to prevent the configuration of the FPGA with a modified bitstream by a bitstream authentication scheme. The FPGA manufacturers already offer solutions in their newer devices, such as HMAC authentication on Xilinx's 7-series architecture [18]. This feature is separated from the AES-256

bitstream encryption and can detect any bitstream tampering, e.g., single bit toggles. In contrast, the most recent UltraScale and UltraScale$^+$ devices offer the AES-GCM encryption standard for simultaneous encryption and authentication with the same key [19]. Additionally, RSA-2048 authentication is available. This asymmetric algorithm uses a public key for bitstream verification and therefore secure key storage is not necessary. Another advantage of this approach is the protection of the decryption process since no unauthorized data will be fed to the AES hardware.

However, systems using partial reconfiguration cannot use these protection schemes with the external configuration interfaces and are thus susceptible to bitstream manipulation attacks. Moreover, if the same key is used for authentication and encryption, the key might be extracted by using side-channel attacks as demonstrated in [2] and [3]. Furthermore, it can be assumed that a large amount of older FPGAs is still in use. Swierczynski *et al.* [1] cite annual reports of both Xilinx and Altera showing that about 50% of their revenue is made with older FPGAs which do not support bitstream authentication.

However, newer FPGA devices with proper bitstream authentication are vulnerable to laser attacks. A laser is able to locally randomize configuration bits. Our analysis shows that the probability to disable BRAMs by randomizing the BRAM control bits is relatively high (see Section IV-C). This could enable the proposed attack also on newer devices.

Another manufacturer-independent approach is a Built-In Self-Test (BIST) that can check the integrity of the FPGA configuration. In the presented attack scenario, an additional circuitry can, for example, be initiated which asserts the correct functionality of the S-boxes. If an inconsistency is detected, the faulty ciphertext output cannot leave the FPGA in order to keep the secret. Hardware redundancy schemes [20] are not suitable in case the copies also utilize BRAMs primitives since these will be disabled simultaneously.

Obviously, the simplest protection against the proposed attack is to avoid utilizing BRAMs for the S-box implementations. The intention of this paper is thus to raise awareness for the vulnerability of these primitives for cryptographic implementations due to the ability to be reset and/or disabled. However, it should be noted that also LUT-only based implementations have been successfully attacked [1].

VI. Conclusion

In this paper, we presented a novel attack vector against FPGA-based AES implementations which utilize BRAM blocks to store the *SubBytes* operation. Our analysis shows that this is true on many AES implementations. It can also be applied without modification for implementations merging both the *SubBytes* and *MixColumns* operation in memory elements called T-boxes. The attack is demonstrated on a Xilinx Virtex-5 device where the configuration of the BRAM elements can be easily tampered in the bitstream. In this way, all memory blocks can be simultaneously disabled and/or reset and the faulty ciphertext output will leak the secret information to recover the encryption key instantaneously. A major advantage of this approach is the independence of the plaintext input which must neither be known nor be manipulable. In addition, no knowledge of the particular design, such as routing information, etc., is required. Furthermore, our experiments have also shown that bitstream encryption alone is not a sufficient protection.

In conclusion, these findings recommend a careful use of BRAM primitives in cryptographic implementations, as such an attack can be conducted without special expertise. Further research might look into alternative protection schemes, besides bitstream authentication, in order to guarantee a more secure usage of BRAMs with marginal overhead.

Acknowledgment

This work has been supported by the German Federal Ministry for Eduction and Research (BMBF) within the collaborative research project "SecRec" (16KIS0609).

References

[1] P. Swierczynski, G. T. Becker *et al.*, "Bitstream Fault Injections (BiFI)–Automated Fault Attacks Against SRAM-Based FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 348–360, March 2018.

[2] S. Shah, R. Velegalati *et al.*, "Investigation of DPA Resistance of Block RAMs in Cryptographic Implementations on FPGAs," in *2010 International Conference on Reconfigurable Computing and FPGAs*, Dec. 2010, pp. 274–279.

[3] S. Bhasin, S. Guilley *et al.*, "From cryptography to hardware: analyzing and protecting embedded Xilinx BRAM for cryptographic applications," *Journal of Cryptographic Engineering*, vol. 3, no. 4, pp. 213–225, Nov. 2013.

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

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

[6] A. C. Aldaya, A. J. C. Sarmiento *et al.*, "AES T-Box tampering attack," *Journal of Cryptographic Engineering*, vol. 6, no. 1, pp. 31–48, Apr. 2016.

[7] S. Trimberger, J. Moore *et al.*, "Authenticated encryption for fpga bitstreams," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 83–86. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950432

[8] M. J. Dworkin, E. B. Barker *et al.*, "Advanced Encryption Standard (AES)," *Federal Inf. Process. Stds. (NIST FIPS) - 197*, Nov. 2001. [Online]. Available: https://www.nist.gov/publications/advanced-encryption-standard-aes

[9] M. C. McCoy, https://github.com/abhinav3008/inmcm-hdl/tree/master/AES/Basic_AES_128_Cipher, 2010.

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

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

[12] J. Gbur, http://opencores.org/project,aes_128_192_256, 2011.

[13] A. Satoh, http://www.aoki.ecei.tohoku.ac.jp/crypto/web/cores.html, 2007.

[14] M. Litochevski and L. Dongjun, http://opencores.org/project,aes_highthroughput_lowarea, 2013.

[15] T. Ahmad, http://opencores.org/project,aes-encryption, 2013.

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

[17] "Pipelined aes," https://github.com/freecores/aes_pipe, 2017.

[18] "Xilinx 7 series fpgas configuration user guide (ug470)," https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf, 2017.

[19] "Xilinx ultrascale architecture configuration user guide (ug570)," https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf, 2017.

[20] S. Ali, X. Guo *et al.*, "Fault Attacks on AES and Their Countermeasures," in *Secure System Design and Trustable Computing*. Springer, Cham, 2016, pp. 163–208, dOI: 10.1007/978-3-319-14971-4_5.