Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs

PAWEL SWIERCZYNSKI, Ruhr University Bochum
AMIR MORADI, Ruhr University Bochum
DAVID OSWALD, Ruhr University Bochum
CHRISTOF PAAR, Ruhr University Bochum

In order to protect FPGA designs against IP theft and related issues such as product cloning, all major FPGA manufacturers offer a mechanism to encrypt the bitstream that is used to configure the FPGA. From a mathematical point of view, the employed encryption algorithms, e.g., AES or 3DES, are highly secure. However, it has been shown that the bitstream encryption feature of several FPGA families is susceptible to side-channel attacks based on measuring the power consumption of the cryptographic module. In this paper, we present the first successful attack on the bitstream encryption of the Altera Stratix II and Stratix III FPGA families. To this end, we analyzed the Quartus II software and reverse-engineered the details of the proprietary and unpublished schemes used for bitstream encryption on Stratix II and Stratix III. Using this knowledge, we demonstrate that the full 128-bit AES key of a Stratix II as well as the full 256-bit AES key of a Stratix III can be recovered by means of side-channel attacks. In both cases, the attack can be conducted in a few hours. The complete bitstream of these FPGAs that are (seemingly) protected by the bitstream encryption feature can hence fall into the hands of a competitor or criminal — possibly implying system-wide damage if confidential information such as proprietary encryption schemes or secret keys programmed into the FPGA are extracted. In addition to lost IP, reprogramming the attacked FPGA with modified code, for instance, to secretly plant a hardware Trojan, is a particularly dangerous scenario for many security-critical applications.

Categories and Subject Descriptors: K.6.5 [**Security and Protection**]: Physical security; D.4.6 [**Security and Protection**]: Cryptographic controls

General Terms: Security, Experimentation

Additional Key Words and Phrases: Side-channel attack, bitstream encryption, AES, Altera, Stratix II, Stratix III, hardware security, reverse-engineering

ACM Reference Format:
Pawel Swierczynski, Amir Moradi, David Oswald, Christof Paar, 2013. Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs. *ACM Trans. Reconfig. Technol. Syst.*

This is an extended version of an already published conference paper. The brief list of differences includes Section 2.5 (Potential IV security problem), Section 3 (Reverse-engineering of Stratix III), and section 6 (Side-channel attack on Stratix III).

1. INTRODUCTION

Ubiquitous computing has become reality and has began to shape almost all aspects of our life, ranging from social interaction to the way we do business. Virtually all ubiquitous devices are based on embedded digital technology. As part of this development, the security of embedded systems has become an increasingly important issue. For instance, digital systems can often be cloned relatively easily or Intellectual Property (IP) can be extracted. Also, ill-intended malfunctions of the device or the circumvention of business models based on the electronic content — which is regularly happening in the pay-TV sector — are also possible. Another flavor of malicious manipulation of digital systems was described in a 2005 report by the US Defense Science Board, where the clandestine introduction of hardware Trojans was underlined as a serious threat [DSB 2011]. In order to prevent these and other forms of abuse, it is often highly desirable to introduce security mechanisms into embedded systems which prevent reverse-engineering and manipulation of designs.

In the field of digital design, Field Programmable Gate Arrays (FPGAs) close the gap between powerful but inflexible Application Specific Integrated Circuits (ASICs) and highly flexible but performance-limited microcontroller (μC) solutions. FPGAs combine the advantages of software (fast development, low

Author's addresses: Horst Görtz Institute for IT-Security, Ruhr University Bochum, Germany.

non-recurring engineering costs) with those of hardware (performance, relative power efficiency). These advantages have made FPGAs an important component in embedded system design, especially for applications that require heavy processing, e.g., for routing, signal processing, or encryption.

Most of today's FPGAs are (re-)configured with bitstreams, which is the equivalent of software program code for FPGAs. The bitstream determines the complete functionality of the device. In most cases, FPGAs produced by the dominant vendors use volatile memory, e.g., SRAM to store the bitstream. This implies that the FPGA must be reconfigured after each power-up. The bitstream is stored in an external Non-Volatile Memory (NVM), e.g., EEPROM or Flash, and is transferred to the FPGA on each power-up.

One of the disadvantages of FPGAs, especially with respect to custom hardware such as ASICs, is that an attacker who has access to the external NVM can easily read out the bitstream and clone the system, or extract the IP of the design. The solution that industry has given for this issue is a security feature called *bitstream encryption*. This scheme is based on symmetric cryptography in order to provide confidentiality of the bitstream data. After generating the bitstream, the designer encrypts it with a secure symmetric cipher such as the AES, using a secret key k_{design}. The encrypted bitstream can now be safely stored in the external NVM. The FPGA possesses an internal decryption engine and uses the previously stored secret key k_{FPGA} to decrypt the bitstream before configuring the internal circuitry. The configuration is successful if and only if the secret keys used for the encryption and decryption of the bitstream are identical, i.e., $k_{design} = k_{FPGA}$. Now, wire-tapping the data bus or dumping the content of the external NVM containing the encrypted bitstream does not yield useful information for cloning or reverse-engineering the device, given the adversary does not know the secret key.

The cryptographic scheme used by Xilinx FPGAs starting from the old and discontinued Virtex-II family to the recent 7 series is 3DES or AES in Cipher Block Chaining (CBC) mode [Krueger 2004; Tseng 2005]. Recent findings reported in [Moradi et al. 2011] and [Moradi et al. 2012] show the vulnerability of these schemes to state-of-the-art Side-Channel Analysis (SCA). Indeed, it has been shown that a side-channel adversary can recover the secret key stored in the target FPGA and use it for decrypting the bitstream. More recently, similar findings have been reported for bitstream security feature of a family of flash-based Actel FPGAs of Microsemi [Skorobogatov and Woods 2012].

Side-channel attacks exploit physical information leakage of an implementation in order to extract the cryptographic key. In the particular case of power analysis, the current consumption of the cryptographic device is used as a side channel for key extraction. The underlying principle is a divide-and-conquer approach, i.e., small parts of the key, e.g., 8 bits, are guessed, and the according hypotheses are verified. This process is repeated until the whole key has been revealed [Eisenbarth et al. ; Kocher et al. 1999].

In this paper, we analyze the bitstream protection mechanism (called *design security*) of Altera's Stratix II and Stratix III FPGA families. We give a detailed description of this real-world attack illustrating the steps required to perform a black-box analysis of a mostly undocumented target, i.e., the design security feature of the targeted FPGA families. Similar to the attacks on the bitstream encryption of Xilinx and Actel FPGAs, our attack on the targeted Altera FPGA makes use of the physical leakage of the embedded decryption module. However, a detailed specification of the design security scheme is not publicly available. By reverse-engineering the Quartus II software application, we recovered all details and proprietary algorithms used for the design security scheme. Our results show the vulnerability of the bitstream encryption feature of both Altera's Stratix II and Stratix III FPGAs to side-channel attacks, leading to a complete break of the security feature and the anti-counterfeiting mechanism.

The remainder of this paper is organized as follows. In Section 2, we describe the steps needed to reverse-engineer the Quartus II application in order to reveal the details of the design security scheme of Stratix II and Stratix III. Also, basic security problems of the according scheme are illustrated. The details of our side-channel attacks on Stratix II are presented in Section 4 and Section 5. Section 6 deals with our side-channel attack on the security feature of Stratix III, and finally in Section 7 we conclude and sum up our research results.

2. REVERSE-ENGINEERING — DESIGN SECURITY SCHEME OF STRATIX II

For an SCA, all details of the bitstream encryption scheme are required. However, this information cannot be found in the public documents published by Altera. In this section, we thus illustrate the method we followed to reveal the essential information, including the proprietary algorithms used for the key derivation and the encryption scheme.

2.1. Preliminaries

The main design software for Altera FPGAs is called "Quartus II". To generate a bitstream for an FPGA, the Hardware Description Language (HDL) sources are first translated into a so called .SOF file. In turn, this file can then be converted into several file types that are used to actually configure the FPGA, cf. Table I.

For the purposes of reverse-engineering the bitstream format, we selected the .RBF type, i.e., a raw binary output file. This format has the advantage that it can be used with our custom programmer, cf. Section 4.1.

For transferring the bitstream to the FPGA, Altera provides several different configuration schemes [Str 2007, p.131-132]. Table II gives an overview on the different available schemes. For our purposes, we used

Table I. Bitstream file formats generated by Quartus II

File extension	**Type**
.HexOut	Hexdecimal Output
.POF	Programmer Object File
.RBF	**Raw Binary File**
.TTF	Tabular Text File
.RPD	Raw Programming Data
.JIC	JTAG Indirect Configuration

the Passive Serial (PS) configuration scheme, because it supports bitstream encryption and moreover, because the configuration clock signal is controlled by the configuration device.

Table II. Configuration modes for the Stratix II

Mode	**Bitstream Enc.**
Fast Passive Parallel (FPP)	Yes
Active Serial (AS)	Yes
Passive Serial (PS)	**Yes**
Passive Parallel Asynch. (PPA)	No
JTAG	No

Regarding the actual realization of the bitstream encryption, relatively little information is known. In the public documents [AN3 2009] it is stated that Stratix II uses the AES with 128-bit key. Furthermore, a key derivation scheme is outlined that generates the actual encryption key given two user-supplied 128-bit keys. Apart from that, no information on the file format, mode of operation used for the encryption, etc. was initially available to us. Thus, in the following, we analyze the functional blocks of Quartus II and completely describe the mechanisms used for bitstream encryption on the Stratix II.

2.2. RBF File Format (Stratix II)

In order to understand the file structure of an .RBF file, we generated both the encrypted and the unencrypted .RBF files for an example design and compared the results. We found that the file can be divided into a header and a body section. Comparing the encrypted and the unencrypted .RBF files, we figured out that only a few bytes vary in the header. In contrast, the bodies containing the — possibly encrypted — actual bitstream are completely different. The unencrypted file's body contains mainly zero, while the encrypted file consists of seemingly random bytes.

We encrypted the same input (.SOF file) twice, using the same key both times. It turned out that the resulting encrypted bitstreams are completely different, with differences in some header bytes and the complete body. Thus, the encryption process appears to be randomized in some way. Experimentally, we found that this randomization is based on the current PC clock only. Using a small batch script, we fixed the PC clock to a particular value and again generated two encrypted .RBF files. The resulting files were completely identical, confirming the conjecture that the PC clock is used as an Initialization Vector (IV) for the bitstream encryption.

To gain further insight into the internals of the file format, we used the reverse-engineering tool Hex-Rays IDA Pro [IDA 2012]. This program allows to analyze the assembly code of an executable program (i.e., in our case the Quartus II bitstream tool) and run a debugger (i.e., display register values etc.) while the target program is running. Using IDA Pro, we obtained the file structure depicted in Figure 1 (for the specific FPGA fabric EP2S15F484C5N).

Both the unencrypted and encrypted .RBF files start with a fixed 33-byte "pre-header". The following 40 bytes include the IV used for the encryption. For the unencrypted file, the IV is always set to `0xFF...FF`, while for the encrypted file the first (left) 32-bit half is randomized (using the PC clock). The right 32-bit half is set to a fixed value. However, the IV is not directly stored in plain; rather, the single bits of the IV are distributed over several bytes of the header. Using IDA Pro, we determined the byte (and bit) positions in the header at which a particular IV bit is stored.

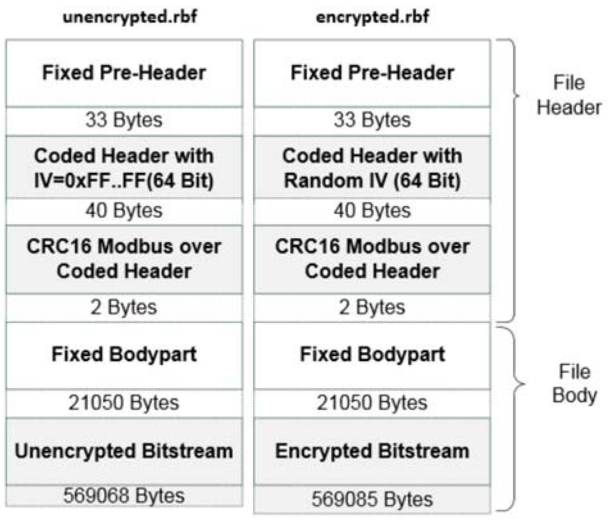


Fig. 1. Structure of an unencrypted and an encrypted .RBF file

Table III shows the resulting IV bit positions. The notation Y_{bitX} refers to bit X (big endian, $X \in [0, 7]$) of the byte at position Y in the .RBF file. Note that the byte positions are counted starting from the beginning of the .RBF file, i.e., including the fixed 33-byte pre-header.

Table III. Mapping between the IV bits and the header bytes

IV bit	63	62	61	60	59	58	57	56
Position	49_{bit3}	48_{bit3}	47_{bit3}	46_{bit3}	45_{bit3}	44_{bit3}	43_{bit3}	42_{bit3}
IV bit	55	54	53	52	51	50	49	48
Position	57_{bit3}	56_{bit3}	55_{bit3}	54_{bit3}	53_{bit3}	52_{bit3}	51_{bit3}	50_{bit3}
IV bit	47	46	45	44	43	42	41	40
Position	65_{bit3}	64_{bit3}	63_{bit3}	62_{bit3}	61_{bit3}	60_{bit3}	59_{bit3}	58_{bit3}
IV bit	39	38	37	36	35	34	33	32
Position	33_{bit4}	72_{bit3}	71_{bit3}	70_{bit3}	69_{bit3}	68_{bit3}	67_{bit3}	66_{bit3}
IV bit	31	30	29	28	27	26	25	24
Position	41_{bit4}	40_{bit4}	39_{bit4}	38_{bit4}	37_{bit4}	36_{bit4}	35_{bit4}	34_{bit4}
IV bit	23	22	21	20	19	18	17	16
Position	49_{bit4}	48_{bit4}	47_{bit4}	46_{bit4}	45_{bit4}	44_{bit4}	43_{bit4}	42_{bit4}
IV bit	15	14	13	12	11	10	9	8
Position	57_{bit4}	56_{bit4}	55_{bit4}	54_{bit4}	53_{bit4}	52_{bit4}	51_{bit4}	50_{bit4}
IV bit	7	6	5	4	3	2	1	0
Position	65_{bit4}	64_{bit4}	63_{bit4}	62_{bit4}	61_{bit4}	60_{bit4}	59_{bit4}	58_{bit4}

Only the third and fourth bit of a byte is used to store the IV bits. The other bits of the header are constant and independent of the IV. We assume that these bits store configuration options, e.g., whether the bitstream is encrypted. The header is followed by a two-byte Modbus CRC-16 [CRC 2012] computed over the preceding 40 header bytes for integrity check purposes.

The body starts with a 21050-byte block identical for both encrypted and unencrypted files. This block is followed by the actual (encrypted or unencrypted) bitstream. The unencrypted bitstream has a length of 569068 bytes. For the encrypted bitstream, 17 additional bytes are added. This is due to the fact that for the encrypted format several padding bytes are added. For the purposes of our work, the details of this padding are irrelevant, as the additional block does not carry data belonging to the actual bitstream.

2.3. AES Key Derivation (Stratix II)

In the publicly available documents, it is stated that the 128-bit AES key used for the bitstream encryption is not directly programmed into the Stratix II. Rather, two 128-bit keys denoted as KEY1 and KEY2 are sent to the FPGA during the key programming. These keys are then passed through a key derivation function that generates the actual "real key" used to decrypt the bitstream. The idea behind this approach is that if an adversary obtains the real key (e.g., by means of a side-channel attack), he should still be unable to use the same (encrypted) bitstream to program another Stratix II (e.g., to create a perfect clone of a product). Since the real key (of the second Stratix II) can only be set given KEY1 and KEY2, an adversary would have to invert the key derivation function, which is supposed to be hard. We further comment on the security of this approach in the case of the Stratix II in Section 2.3.2.

Initially, the details of the key derivation were hidden in the Quartus II software, i.e., the software appears as a complete black-box. As depicted in Figure 2, Quartus II produces a key file (in our case Keyfile.ekp) that stores the specified KEY1 and KEY2. This key file is later passed to the FPGA, e.g., via the Joint Test Action Group (JTAG) port using a suitable programmer.

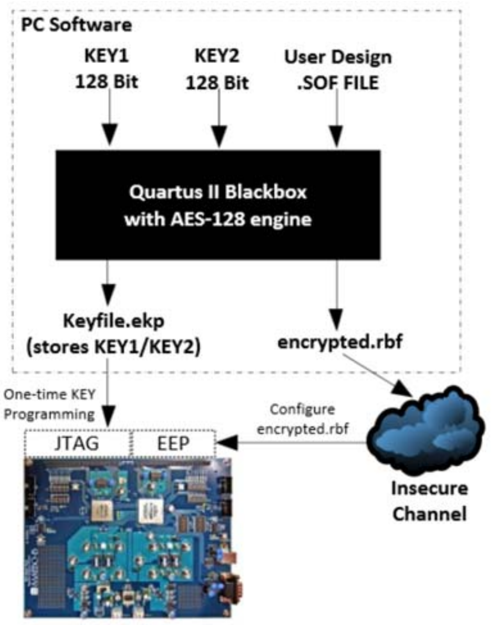


Fig. 2. Quartus II black-box generating encrypted Stratix II bitstreams

However, the key derivation function obviously has to also be implemented in Quartus II because the real key is needed to finally encrypt the bitstream. Hence, we again reverse-engineered the corresponding scheme from the executable program. Most of the cryptographic functions are implemented in the DLL file `pgm_pgmio_nv_aes.dll`. Apparently, the developers of Quartus II did not remove the debugging information from the binary executable; hence the original function names are still present in the DLL.

Figure 3 shows the corresponding function calls for the key derivation and the bitstream encryption. First, we focus on the key derivation, i.e., the upper part of Figure 3. Note that due to the available debugging information, all function names are exactly those chosen by the Altera developers.

First, the `do_something()` function checks the used key length. Then, the `make_key()` function copies the bytes of KEY1 to a particular memory location. The `key_init()` function then implements the key schedule

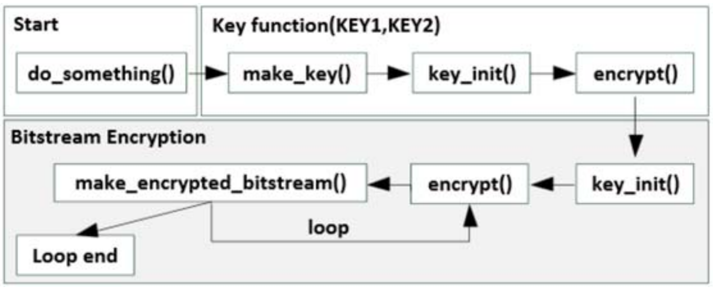


Fig. 3. Quartus II call sequence during the bitstream encryption (Stratix II)

algorithm of the AES, generating 160 bytes of round keys in total. `encrypt()` then encrypts KEY2 with KEY1. Hence, the — previously unknown — key derivation function is given as

$$\textbf{Real Key} := \textbf{AES128}_{\textbf{KEY1}}(\textbf{KEY2}),$$

where KEY1 and KEY2 are those specified in the Quartus II application.

2.3.1. Example for the Key Derivation. In order to further illustrate the details of the key derivation function, in the following we give the inputs and outputs for the chosen KEY1 and KEY2 we used for our analysis.

KEY1 (Quartus II input, little endian)
```
0x0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00
```

KEY2 (Quartus II input, little endian)
```
0x32 00 31 C9 FD 4F 69 8C 51 9D 68 C6 86 A2 43 7C
```

Real Key = $\textbf{AES128}_{\textbf{KEY1}}$(KEY2) (big endian)
```
0x2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
```

2.3.2. Security of the Key Derivation Function. At first glance, the approach of deriving the real key within the device appears to be a reasonable countermeasure to prevent cloning of products even if the real key has been discovered. Yet, it should be taken into account that an adversary knowing the real key is still able to decrypt the bitstream and re-encrypt it with a different key for which he has chosen KEY1 and KEY2. Nevertheless, a product cloned in such a way could be still identified, because the re-encrypted bitstream will differ from the original one.

However, the way the AES is used for the key derivation in the case of the Stratix II does not add to the protection against product cloning in any way: a secure key derivation scheme requires the utilized function to be one-way, i.e., very hard to invert. For the Stratix II scheme, this is not the case. An adversary can pick *any* KEY1 and then decrypt the — previously recovered — real key using this KEY1. The resulting KEY2 together with KEY1 then forms one of 2^{128} pairs that lead to the same (desired) real key when programmed into a blank Stratix II. The device will thus still accept the original (encrypted) bitstream, and the clone cannot be identified as such because KEY1 and KEY2 are never stored in the FPGA by design.

2.4. AES Encryption Mode (Stratix II)

Having revealed the key derivation scheme, we focus on the details of the actual AES encryption, i.e., analyze the lower part of Figure 3. First, the `key_init` function is executed in order to generate the round keys for the (previously derived) real key. Then, `encrypt()` is invoked repeatedly in a loop. Using the debugger functionality of IDA Pro, we exemplarily observed the following sequence of inputs to `encrypt()`:

```
0xB4 52 19 50 76 08 93 F1 B4 52 19 50 76 08 93 F1
0xB5 52 19 50 76 08 93 F1 B5 52 19 50 76 08 93 F1
0xB6 52 19 50 76 08 93 F1 B6 52 19 50 76 08 93 F1
...
```

Note that the first and the second eight bytes of each AES input are equal. Moreover, this 64-bit value is incremented for each encryption, yielding (in this case) the sequence `B4`, `B5`, `B6` for the first byte. Apparently, the AES is not used to directly encrypt the bitstream. Rather, it seems that the so-called Counter (CTR) mode [NIST 2001] is applied. Figure 4 shows the corresponding block diagram.

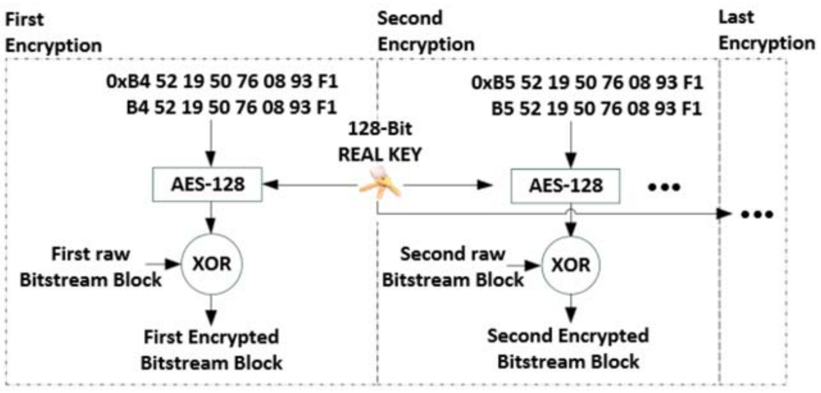

First Encryption
Second Encryption
Last Encryption
0xB4 52 19 50 76 08 93 F1
B4 52 19 50 76 08 93 F1
0xB5 52 19 50 76 08 93 F1
B5 52 19 50 76 08 93 F1
128-Bit REAL KEY
AES-128
AES-128
XOR
XOR
First raw Bitstream Block
Second raw Bitstream Block
First Encrypted Bitstream Block
Second Encrypted Bitstream Block

Fig. 4. AES in CTR mode (Stratix II)

In CTR mode, an IV is encrypted using the specified key (in our case the real key). The output (i.e., ciphertext) of the AES is then XORed with the 16-byte data block to perform the encryption (of the bitstream blocks for the case of Stratix II). For each block, the IV is incremented to generate a new ciphertext to be XORed with the corresponding data block. The XOR operation is implemented in the function `make_encrypted_bitstream()`.

As mentioned in Section 2.2, the IV is generated based on the PC clock. Indeed, we found that the first four bytes of the IV correspond to the number of seconds elapsed since January 1, 1970. More concretely, the (little endian) value `0xB4 52 19 50` represents the date `2012.08.01 18:00:52`. The remaining four bytes are constant. The overall structure of the IV is thus:

$$0\text{x}\underbrace{\mathbf{B4}\ 52\ 19\ 50}_{\text{Timestamp}}\underbrace{76\ 08\ 93\ \text{F1}}_{\text{Fixed bytes}}\underbrace{\mathbf{B4}\ 52\ 19\ 50}_{\text{Timestamp}}\underbrace{76\ 08\ 93\ \text{F1}}_{\text{Fixed bytes}}.$$

Having figured out the details of the AES key derivation and encryption, we implemented the aforementioned functions to decrypt a given encrypted bitstream. Given the correct real key and IV, we successfully decrypted the bitstream of an encrypted .RBF file. Figure 5 summarizes the details of the bitstream encryption process of Stratix II.

2.5. Security of the IV Computation (Stratix II)

A different (potential) security problem that we noticed is the utilized IV computation scheme. Suppose a design rbf is encrypted using KEY1 and KEY2. This leads to a specific real key rk. Hence, the i'th encrypted block of this design is given as:

$$E_{IV_i,rk} = AES128_{rk}(IV + i) \oplus rbf_i$$

For the case that a second configuration design is encrypted using the same KEY1 and KEY2, and hence, the same real key rk, we obtain:

$$\widehat{E}_{\widehat{IV}_i,rk} = AES128_{rk}(\widehat{IV} + i) \oplus \widehat{rbf}_i$$

Note that the first IV depends on the system clock for both configuration designs. Hence, if the second configuration design is being encrypted with a delay of x seconds, $\widehat{IV} = IV + x$ holds. Thus, we get:

$$\widehat{E}_{\widehat{IV}_i,rk} = AES128_{rk}(\widehat{IV} + i) \oplus \widehat{rbf}_i = AES128_{rk}(IV + i + x) \oplus \widehat{rbf}_i$$

Incrementing the IV x times — as it is performed due to the full encryption of the first configuration design — leads to the IV $IV + i + x$. Thus, the x'th IV of the first encryption matches the first IV ($\widehat{IV} + i = IV + i + x$) of the second encryption. This also implies that the x'th XORed value (i.e., $AES128_{rk}(IV + x)$) of the first configuration design encryption is identical to that of the first corresponding value of the second configuration design encryption. Using the same key for two different encryptions can lead to a serious problem because an attacker is able to cancel the key by XORing both ciphertext blocks. Table IV gives the first five encrypted blocks, whereas the second encryption was invoked with a delay of three seconds, i.e., $x = 3$.

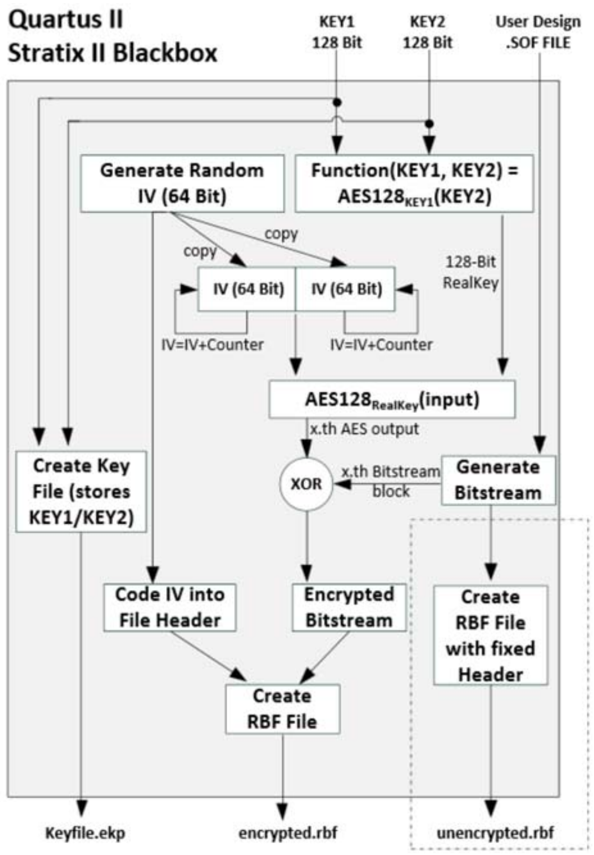


Fig. 5. Overview of the bitstream encryption process for the Stratix II FPGAs

Table IV. Resulting IVs for two encryptions invoked at different points in time (Stratix II)

1st **encryption**	2nd **encryption (delay of $x = 3\,s$)**
$E_{IV_0,rk}$	–
$E_{IV_1,rk}$	–
$E_{IV_2,rk}$	–
$E_{IV_3,rk}$	$\widehat{E}_{\widehat{IV}_0,rk} = \widehat{E}_{IV_3,rk}$
$E_{IV_4,rk}$	$\widehat{E}_{\widehat{IV}_1,rk} = \widehat{E}_{IV_4,rk}$

Computing the XOR between the values of the fourth row leads to:

$$E_{IV_3,rk} \oplus \widehat{E}_{\widehat{IV}_0,rk} = (AES128_{rk}(IV+3) \oplus rbf_3) \oplus (AES128_{rk}(\widehat{IV}+0) \oplus \widehat{rbf}_0)$$
$$= (AES128_{rk}(IV+3) \oplus AES128_{rk}(IV+3)) \oplus (rbf_3 \oplus \widehat{rbf}_0) = rbf_3 \oplus \widehat{rbf}_0$$

An attacker is able to retrieve the XOR sum $rbf_3 \oplus \widehat{rbf}_0$. Analogously, the computation can be repeated for the fifth, sixth, ... row to obtain $rbf_4 \oplus \widehat{rbf}_1$, $rbf_5 \oplus \widehat{rbf}_2$, ..., i.e., in general for $i \geq x$ (and a delay of

x seconds) $diff_i := rbf_i \oplus \widehat{rbf}_{i-x}$. This XOR sum can only be computed if the second configuration design is encrypted with a maximum delay of `#bitstream blocks` seconds, because then, no overlap of the utilized IVs occurs between the encrypted configuration designs.

One might argue that this scenario is not a real threat because an attacker only possesses the XOR difference between two configuration designs blocks, and hence is not able to reconstruct one of both bitstreams. However, assume that a company encrypts two configuration designs rbf and $\widehat{rbf}$ (both within a short time range) using the same key rk. Later, one of the bitstreams, e.g., rbf, becomes public (e.g., is leaked or extracted by means of a side-channel attack). The attacker is then able to decrypt most of the second configuration $\widehat{rbf}$ with the help of the known bitstream rbf. To do so, he computes the XOR sum $diff_i \oplus rbf_i = \widehat{rbf}_{i-x}$. The point of this section is to emphasize that — in certain situations — problems may appear due to the bad design practice of encrypting two different configuration designs with the same key. In general, a designer of cryptographic functions should try to avoid the usage of a timestamp as an IV. At least, the IV should not be simply incremented (when using a timestamp) to avoid the described problem of overlaps. This scheme has indeed changed in the newer FPGA families of Altera. As explained in Section 3.3, the IVs for Stratix III FPGAs are not generated using a counter but a more complex update function.

3. REVERSE-ENGINEERING — DESIGN SECURITY SCHEME OF STRATIX III

Similar to Section 2, we describe the reverse-engineering of the bitstream encryption scheme for Stratix III FPGAs. The Stratix III series is the third generation of Altera FPGAs. According to [Corporation 2012b], Stratix III is manufactured using a 65 nm process (while Stratix II FPGAs are based on a 90 nm process) and has a lower static and dynamic power consumption than Stratix II. According to [Corporation 2012a], Stratix III FPGAs additionally features a volatile key for the bitstream encryption. Moreover, the AES engine uses 256-bit instead of 128-bit keys. Again, we were facing a black-box scenario, i.e., had no further knowledge on the inner workings of the key derivation scheme, the utilized encryption mode, or the IV update function. Thus, we continued reverse-engineering the Quartus II application and recovered the design security scheme for Stratix III (FPGA fabric EP3SC150).

In the following, we first compare the structure of RBF files for the unencrypted and encrypted file. Next, we reveal the changed key derivation scheme and provide the rules for computing the 256-bit real key given 256-bit keys KEY1 and KEY2. Furthermore, we describe the utilized AES encryption mode, which is different compared to Stratix II devices. The simple IV increment used by Stratix II devices (presented in Section 2.4) has been replaced by a more complex update function f. The details of this function are given in Section 3.4.

3.1. RBF File Format (Stratix III)

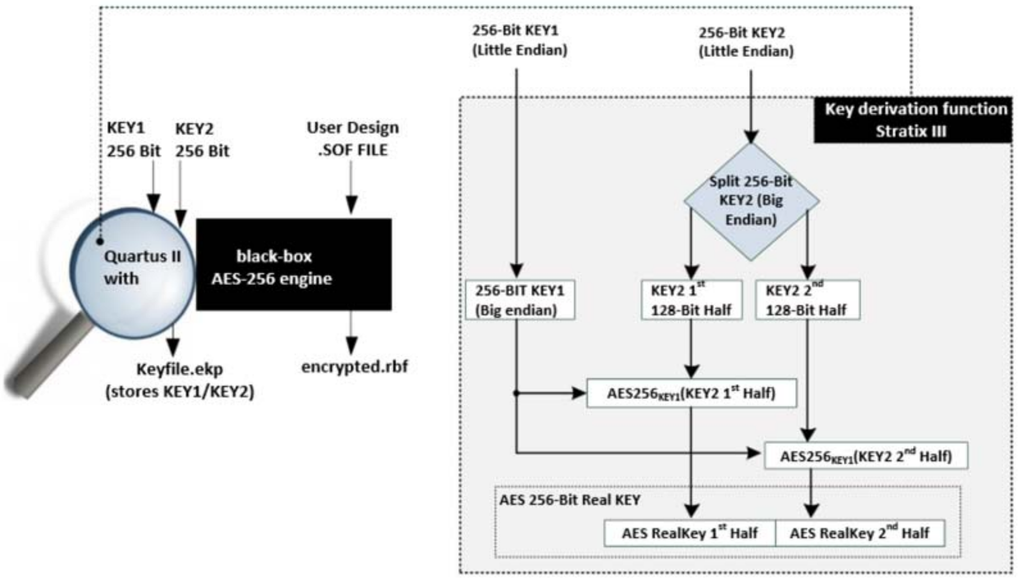
Comparing the unencrypted and encrypted RBF file (as done for an Stratix II in Section 2.2), we found that the file format is basically the same as for Stratix II: The file header of an Stratix III RBF file is identical to that of Stratix II. Again, the bits of an IV are distributed (and stored) in the header, applying the same mapping rules as for Stratix II, cf. Table III. The CRC checksum is computed in an identical manner. The file body of the Stratix III RBF file is similar to that of an Stratix II RBF file (cf. Figure 1). Only the size of the fixed body and the bitstream differ. The fixed body of the targeted Stratix III consists of 78,636 bytes, while for the Stratix II 21,050 bytes are used. The bitstream of Stratix III has a size of 5,847,954 bytes, compared to 569,068 bytes for Stratix II.

3.2. AES Key Derivation (Stratix III)

The key derivation scheme of Stratix III FPGAs is similar to that of Stratix II FPGAs. Given two 256-bit inputs KEY1 and KEY2, Quartus II first performs the AES key schedule for KEY1 (i.e., fourteen round keys are derived). This implies that KEY1 is the AES key, while KEY2 is the AES plaintext. Since the AES engine operates on 128-bit blocks, Quartus II splits KEY2 into two 128-bit halves and then encrypts each half separately, using both times KEY1 as the key. Thus, the previously unknown black-box key derivation function of Stratix III FPGAs is:

$$\textbf{Real Key} := \textbf{AES256}_{\textbf{KEY1}}(1^{\text{st}}\ \textbf{128-bit KEY2})\,||\,\textbf{AES256}_{\textbf{KEY1}}(2^{\text{nd}}\ \textbf{128-bit KEY2})$$

where the || symbol denotes concatenation. As described above, the output block length is 128-bit, and thus, the real key has (due to the concatenation) a length of 256 bit. The real key serves as the cryptographic key for the actual encryption of all bitstream blocks. Figure 6 depicts the implementation of the secret key derivation scheme in Quartus II. Note that the same security issue mentioned for the Stratix II key derivation scheme (Section 2.3.2) holds for that of Stratix III. Having recovered the real key, one can select an arbitrary KEY1 and obtain a valid value for KEY2 by decrypting both halves of the real key separately.



256-Bit KEY1 (Little Endian)
256-Bit KEY2 (Little Endian)
Key derivation function Stratix III
KEY1 256 Bit
KEY2 256 Bit
User Design .SOF FILE
Quartus II with
black-box AES-256 engine
Keyfile.ekp (stores KEY1/KEY2)
encrypted.rbf
Split 256-Bit KEY2 (Big Endian)
256-BIT KEY1 (Big endian)
KEY2 1st 128-Bit Half
KEY2 2nd 128-Bit Half
AES256KEY1(KEY2 1st Half)
AES256KEY1(KEY2 2nd Half)
AES 256-Bit Real KEY
AES RealKey 1st Half
AES RealKey 2nd Half

Fig. 6. Quartus II black-box generating the Stratix III 256-bit real key

3.3. AES Encryption Mode (Stratix III)

Analyzing the respective program code of Quartus II, we noticed that the first AES input is equal to the first IV, which is stored in the RBF file. Observing that the first AES output is being XORed with the first unencrypted 16-byte bitstream block, we first assumed that the AES block mode is identical to that of Stratix II FPGAs. Against our expectations, we figured out that the second AES input was not the incremented IV. Instead, the second AES input appeared as — initially — random bytes.

We performed further experiments to rule out certain modes of operation for the AES. For example, we manipulated the first unencrypted bitstream block (using IDA's debugger) before being XORed with the first AES output. We noticed that the second AES input does not change when using the same first IV. Hence, the Cipher Feedback Mode (CFB) mode can be excluded, since in this mode, the second AES input depends on the XOR result of the first block. In addition, we noticed that the second and all following AES inputs are not influenced by the choice of KEY1 or KEY2, and thus, are independent of the real key. Finally, we exclude the CBC mode, because the first input would then be IV $\oplus$ (1^{st} bitstream block), which is not the case.

Performing additional reverse-engineering and tracing the respective values in IDA's debuggger, we found that the AES is used to generate a stream of blocks that are XORed with the according bitstream blocks for encryption, similar to the method used for Stratix II, cf. Section 2.4. However, the IV is not incremented, but updated with a more complex function f. Figure 7 summarizes the bitstream encryption process for Stratix III FPGAs.

The rationale for the introduction of f for Stratix III may be to avoid the potential security issue outlined in Section 2.5. Moreover, the additional function f has to be reverse-engineered again to be able to decrypt a bitstream, raising the bar for an adversary. However, f must be a deterministic function implemented both on the FPGA and in the Quartus II application. Hence, it can be analyzed using IDA Pro. In the following Section 3.4, we describe the results of this reverse-engineering process and reveal the details of f.

3.4. IV Update Function (Stratix III)

In this section, we present the realization of the IV update function f. With this knowledge, all AES inputs required to decrypt a bitstream file can be reconstructed given the first IV and the real key. The first IV can be extracted from the RBF file header with the help of Table III (as explained in Section 3.1). For reverse-engineering f, we again observed the Quartus II application in the debugger. Figure 8 gives a simplified overview of the executed functions.

As depicted in Figure 8, an initially unknown function `sub_10007310(3)` (implementing f) is executed to generate the next input for the AES. The integer value "3" is passed as an argument to this function. When observing the memory locations being read by this function, it turned out that the bytes located at address

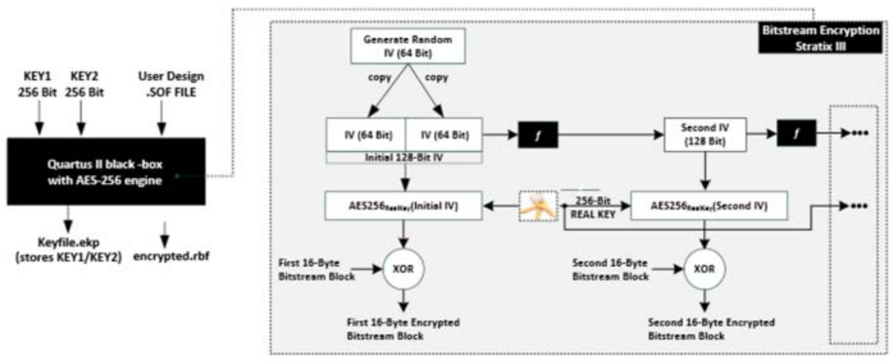

KEY1 256 Bit
KEY2 256 Bit
User Design .SOF FILE
Quartus II black -box with AES-256 engine
Keyfile.ekp (stores KEY1/KEY2)
encrypted.rbf
Bitstream Encryption Stratix III
Generate Random IV (64 Bit)
copy
copy
IV (64 Bit)
IV (64 Bit)
Initial 128-Bit IV
f
Second IV (128 Bit)
f
AES256(Initial IV)
256-Bit REAL KEY
AES256(Second IV)
First 16-Byte Bitstream Block
XOR
First 16-Byte Encrypted Bitstream Block
Second 16-Byte Bitstream Block
XOR
Second 16-Byte Encrypted Bitstream Block

Fig. 7. Bitstream encryption mechanism of Stratix III

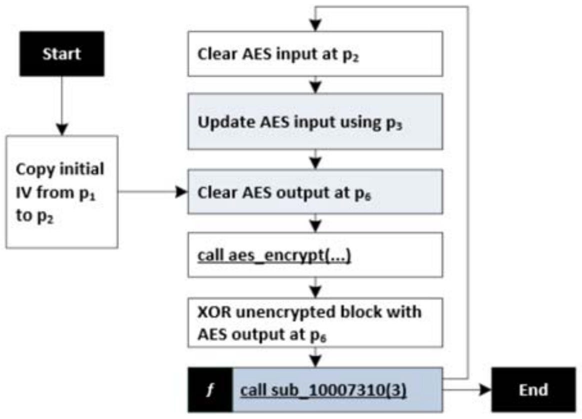

Start
Copy initial IV from p_1 to p_2
Clear AES input at p_2
Update AES input using p_3
Clear AES output at p_6
call aes_encrypt(...)
XOR unencrypted block with AES output at p_6
f
call sub_10007310(3)
End

Fig. 8. Execution order of the encryption on Stratix III

p_3 are repeatedly accessed. We found that the value at address p_3 is updated three times (as specified by the argument "3") in a loop and that the result is directly related to the next AES input.

Subsequently, we analyzed the assembly instructions of f and found that the update is performed as depicted in Figure 9. It turned out that f (`sub_10007310(3)`) implements a 64-bit Linear Feedback Shift Register (LFSR). If the Least Significant Bit (LSB) of the first byte IV_1 is set to "1", the value `0x20 24 00 10 10 00 00 01` is XORed to the current 64-bit state and the whole LFSR is rotated to the right by one bit. If the LSB is set to "0", the XOR operation is skipped. This process is repeated three times, yielding the next AES input.

Algorithm 1 gives a possible implementation of the function f. With a `C` implementation of f, it took less than one second to compute all IVs. As a side note, an adversary could alternatively also use a workaround for decrypting an encrypted RBF file, for which no knowledge on f is required at all. It is possible to use the Quartus II application itself as an oracle that provides all required AES inputs by setting the first IV in the debugger and tracing all AES inputs. We exemplarily implemented an IDC script that is able to dump all IVs directly using the IDA Pro debugger. All IVs can be obtained within approximately 3 hours of offline computation using a standard PC.

4. SIDE-CHANNEL PROFILING OF STRATIX II

With the knowledge of the bitstream encryption process presented in Section 2, we are able to analyze the Stratix II from a side-channel point of view. To this end, in this section we first describe the measurement setup and scenario. Then, as a prerequisite to the according key extraction attack (Section 5), we apply SCA to find out the point in time at which the AES operations are executed. In the following, we refer to the used

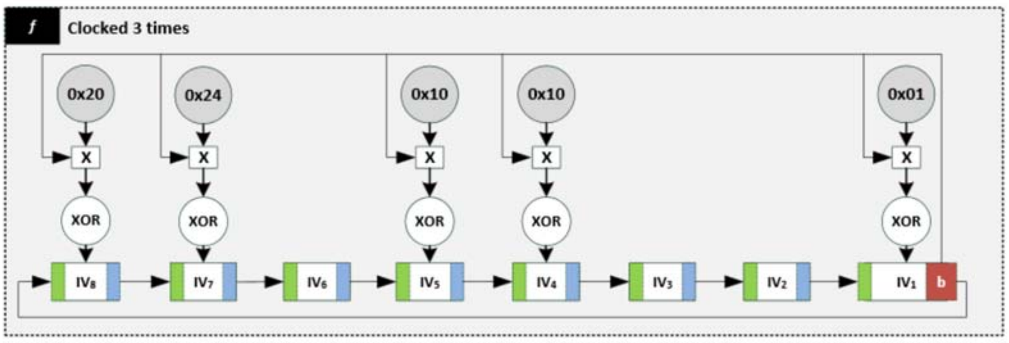


Fig. 9. Function f for IV update

ALGORITHM 1: Pseudo-code for IV update

```
**Input**: $IV = IV_8 \mid\mid IV_7 \mid\mid IV_6 \mid\mid IV_5 \mid\mid IV_4 \mid\mid IV_3 \mid\mid IV_2 \mid\mid IV_1$, $IV_i$ one byte
**Output**: Updated $IV$
**for** $i = 1 \ldots 3$ **do**
    **if** LSB($IV_1$) = 1 **then**
        $IV \leftarrow IV \oplus 0x2024001010000001$
    **end if**
    $IV \leftarrow \text{rotate\_right}_1(IV)$
**end for**
```

Stratix II FPGA as Device Under Test (DUT). Also, we call — following the conventions in the side-channel literature — the current consumption curves during the configuration process *(power) traces*.

4.1. Measurement Setup

Our DUT, a Stratix II FPGA (EP2S15F484C5N), is soldered onto a SASEBO-B board [AIST 2008] specifically designed for SCA purposes. The SASEBO-B board provides a JTAG port that allows one-time programing KEY1 and KEY2 into the DUT. For our experiments we set the real key to `0x2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C`, cf. Section 2.3.1.

We directly configure the DUT using the passive serial mode. For this purpose, we built an adapter that conforms to [Str 2007, p.599]. We developed a custom programmer based on an ATmega256 µC. Thus, we have precise control over the configuration process and are additionally able to set a trigger signal for starting the measurement process. This helps to record well-aligned power traces. Finally, our µC also provides the configuration clock signal to avoid (unwanted) internal clock effects that could, e.g., lead to clock jitter and therefore to misaligned traces.

According to [Str 2007, p.148], the DUT has three different supply voltage lines: V_{CCINT} (internal logic, 1.15V-1.255V), V_{CCIO} (input and output buffers, 3.00V-3.60V) and V_{CCPD} (pre-drivers, configuration, and JTAG buffers, 3.135V-3.465V).

For our analysis, we recorded the power consumption during the configuration of the DUT by inserting a small shunt resistor into the V_{CCINT} path and measuring the (amplified, AC-coupled) voltage drop using a LeCroy WavePro 715Zi Digital Storage Oscilloscope (DSO) as depicted in Figure 10. We acquired 840,000 traces with 225,000 data points each at a sampling rate of 500 MS/s. The respective (encrypted) bitstreams were generated on the PC built into the DSO and then sent to the DUT via the µC. The measurement process was triggered using a dedicated µC pin providing a rising edge shortly before the first bitstream block is sent.

During the decryption process of the encrypted bitstream, the AES is used in CTR mode. Hence, it might be possible that the DUT performs the first AES encryption when the header is being sent because from that time onwards, the DUT knows the IV (first AES input). Therefore, we decided to perform a new power-up of the FPGA for each power trace that we measured. The corresponding steps are described in more detail in Algorithm 2.

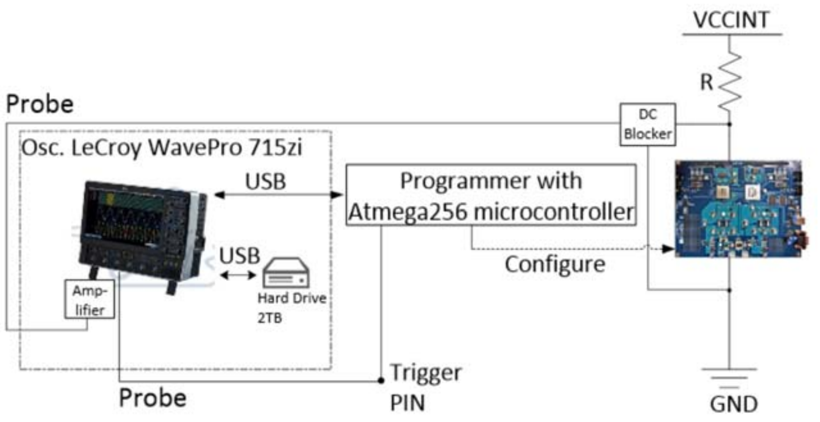


Fig. 10. Measurement setup for SCA

ALGORITHM 2: Measurement steps

```
for i=1 to numberOfTraces do
    [μC] Perform DUT reset
    [μC] Transfer fixed 33-byte pre-header to DUT
    [PC] myIV[0..7] ← rand
    [PC] myHeader[] ← Get header from .RBF file
    [PC] Code myIV[] into myHeader[] (Table III)
    [PC] Compute CRC-16 over coded header
    [PC] Send coded header with CRC-16 (42 bytes) to $\mu C$
    [μC] Transfer coded header (42 bytes) to DUT
    [μC] Transfer fixed body part (21050 bytes) to DUT
    [PC] Bitstream[0..47] ← rand
    [PC] Send Bitstream[] (48 bytes) to $\mu C$
    [μC] Set trigger. Transfer bitstream (48 bytes) to DUT
    [DSO] Record power trace of the DUT
    [PC] Store trace $i$
    [PC] Store myIV[]
end for
```

4.2. Difference between Unencrypted and Encrypted Bitstream

Using our measurement script, we recorded 10,000 power traces for the time range that includes the transmission of 48 fixed, encrypted bitstream bytes. The FPGA decryption engine hence has the same input each time. In addition to that, we performed the same measurements while sending 48 bytes of unencrypted bitstream. Finally, we computed the average power consumption over the set of our measured power traces, once for the unencrypted and once for the encrypted bitstream. Figure 11 illustrates the corresponding mean traces.

As it is clearly visible in Figure 11, there is a significant difference in the average power consumption between the processing of the unencrypted bitstream and the encrypted bitstream. While the FPGA processes an encrypted bitstream, it consumes more energy compared to the processing of an unencrypted bitstream. A difference is already visible at the point where the first bitstream block is being transferred to the DUT. Thus, we assume that the AES encryption engine processes the first AES input (IV) while the programmer transfers the first encrypted bitstream block to the DUT. We further conjecture that while the programmer sends the second encrypted bitstream block, the DUT computes the XOR of the first AES output with the encrypted bitstream and configures the corresponding FPGA blocks.

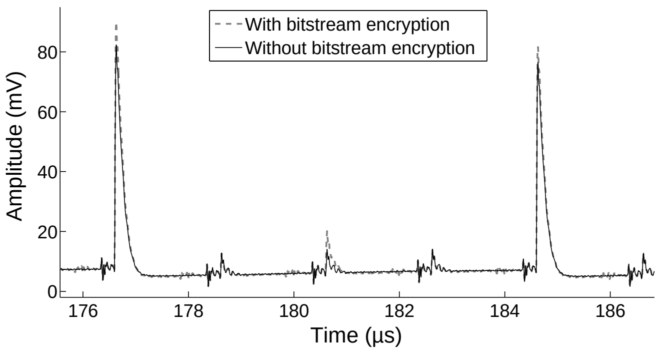


Fig. 11. Average power consumption (10k traces) while sending an unencrypted (solid) and an encrypted (dashed) bitstream (zoom on one byte)

4.3. Locating the AES Encryption

To verify our assumption on the correct time instance of the first AES encryption, we recorded another set of measurements and measured 840,000 power traces, this time exactly as described in Algorithm 2. Then, for our profiling, we used the known key to compute all intermediate AES values for each IV challenge/trace.

For a Correlation Power Analysis (CPA), [Brier et al. 2004], we used this set of power traces to compute the correlation curves of about 220 different prediction models, e.g., each S-box bit of the first AES round, several Hamming Distance (HD) models with different predicted register sizes, and several Hamming Weight (HW) models for the intermediate AES states. As a result, the majority of our power models revealed a data dependency between the predicted power models and the measured power traces. Hence, the FPGA evidently leaks sensitive information. Figure 12 shows nine of the correlation curves for the states after each AES round.

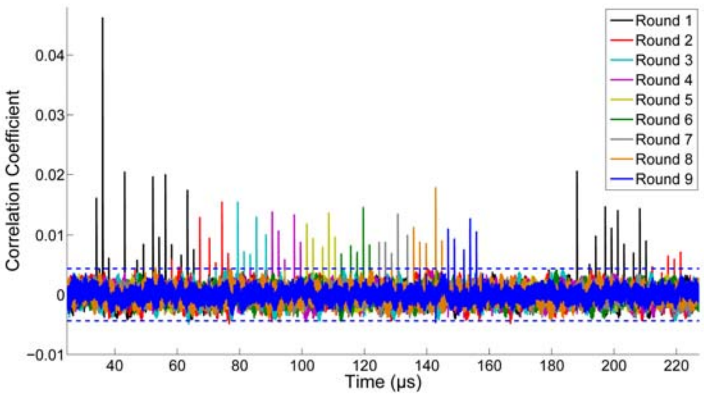


Fig. 12. Correlation coefficient for one full AddRoundKey 128-bit state (one curve for each round). Utilized models: 1st curve $\leftrightarrow$ HW of round 1, 2nd curve $\leftrightarrow$ HW of round 2, etc.

The first correlation curve (black) that exhibits a peak up to an approximate value of 0.05 between 30 and 65 microseconds is for the HW model of the 128-bit state after the first round of the first AES encryption. The second correlation curve (red) is almost the same prediction model as before, but this time for the second round, etc. . Each round of the first AES encryption leaks and therefore, the correct time instance of the first AES encryption is located between 30 and 160 μs.

In Figure 12, one can also spot the processing of the second AES encryption (starts at 180 μs). Due to the fact that only two bytes of the IV are incremented each time, for the second AES encryption, the first output state (128 bits) is similar to that of the first AES encryption. Therefore, the prediction of the first state of the first AES encryption automatically fits to the second encryption as well. Thus, the same leakage (black curve in Figure 12) appears for both the first and second AES encryption. Even the states after round 2 of

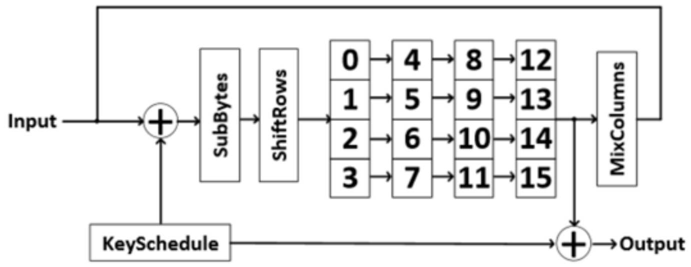


Fig. 13. Hypothetical architecture of the AES implementation

both encryptions are slightly similar, and the leakage peak (red curve) appears for both encryption runs. Since the states (starting from round 3) are completely different for both encryptions, the predicted state of round ≥ 3 does not leak for the second encryption anymore.

5. SIDE-CHANNEL KEY EXTRACTION OF STRATIX II

As shown in Section 4, the DUT exhibits a clear relationship between the power consumption and the internal states during the AES operation. In this section, we show how this side-channel leakage can be utilized to extract the full 128-bit AES key from a Stratix II with approximately three hours of measurements and a few hours of offline computation.

5.1. Digital Pre-Processing

As commonly encountered in SCA, the effect of the AES encryption on the overall power consumption is rather small, cf. Section 4. Hence, digital pre-processing of the traces to isolate the signal of interest (and thus reduce the Signal-to-Noise Ratio (SNR)) is often suggested in the literature in order to reduce the number of required measurements [Barenghi et al. 2010]. In the case of the Stratix II, we experimentally determined a set of pre-processing steps before performing the actual key extraction.

First, the trace is band-pass filtered with a passband from 500 kHz to 100 MHz. Then, the signal is subdivided into windows of 750 sample points (i.e., $1.5\,\mu$s at the sampling rate of 500 MHz), with an overlap of 50 percent between adjacent windows. Each window is zero-padded to a length of 7000 points. Then, the Discrete Fourier Transform (DFT) of each window is computed, and the absolute value of the resulting complex coefficients is used as the input to the CPA. Note that we found the frequency with the maximum leakage to be around 2 MHz, hence, we left out all frequencies above 8 MHz to reduce the number of data points as well as the computational complexity of the CPA. Hence, each window (0 . . . 8 MHz) has a length of 112 points.

This approach was first proposed in [Gebotys et al. 2005] under the name of Differential Frequency Analysis (DFA). Since then, several practical side-channel attacks successfully applied this method to improve the signal quality, cf. for instance [Oswald and Paar 2011; Plos et al. 2008].

5.2. Hypothetical Architecture

For a side-channel attack to succeed, an adequate model for the dependency between the internal architecture and the measured power consumption is needed. Common models include the HW, which states that the consumed power depends on the number of set bits in a register, and the HD, which predicts the power consumption to be proportional to the number of switching bits in a register.

In the case of the Stratix II, the internal realization of the AES was initially unknown. Hence, we experimentally tested many (common) different models, as mentioned in Section 4.3. As a result, it turned out that the leakage present in the traces is best modeled by the HD *within* the AES state after the `ShiftRows` step [NIST 2001]. More precisely, it appears that each column of the AES state is processed in one step, and that the result is shifted into a register, overwriting the previous column (that in turn is shifted one step to the right). The corresponding hypothetical architecture is depicted in Figure 13.

For the key extraction in Section 5.3, we thus use for instance the HD byte $0 \rightarrow 4$ (after `ShiftRows` in the first AES round) to recover the first key byte, byte $1 \rightarrow 5$ to recover the second key byte, and so on. As common in SCA, each key byte can be recovered separately from the remaining bytes, i.e., in principle 16×2^8 instead of 2^{128} key guesses for an exhaustive search have to be tested.

Note that, however, the initial state (i.e., the column overwritten with byte 0 . . . 3) is unknown. Hence, we consider each row of the first two columns together and recover the key bytes 0 and 4, 1 and 5, 2 and 6, and 3 and 7 together, corresponding to 2^{16} key candidates each. After that, the remaining eight key bytes 8 . . . 15 yield 8×2^8 candidates in total because the previous (overwritten) column values are known. The total

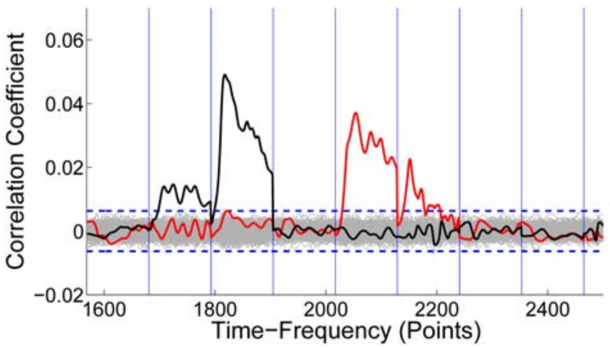


Fig. 14. Correlation coefficient for the first S-box after 400k traces using DFT pre-processing. Correct key candidate `0x2B`: black curve

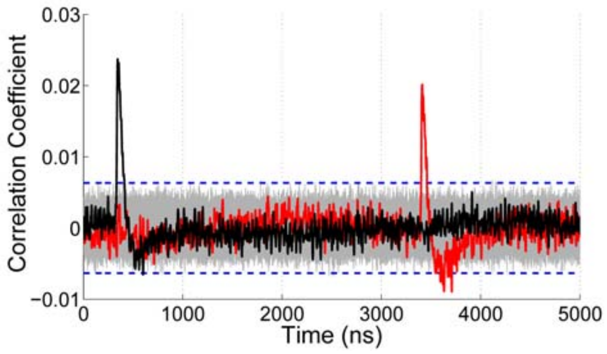


Fig. 15. Correlation coefficient for the first S-box after 400k traces without DFT pre-processing. Correct key candidate `0x2B`: black curve

number of key candidates is thus $8 \times 2^8 + 4 \times 2^{16} = 264,192$ for which the CPA can be conducted within a few hours using standard hardware.

5.3. Results

Using the described power model, we computed the correlation coefficient for the respective (byte-wise) HD of the AES states. Figure 14 shows the result for the first S-box, i.e., the HD between byte 0 and 4. Evidently, the correct key candidate `0x2B` (black curve) exhibits a maximum correlation of approximately 0.05 after $400,000$ traces, clearly exceeding the "noise level" of $4/\sqrt{\#traces} = 0.006$ [Mangard et al. 2007].

All other (but one) key candidates stay below the noise level. However, a second key candidate `0xAB` (red curve) also results in a significant peak at a different point in time. This is due to the fact that, as explained in Section 2, the first 64-bit half of the plaintext (i.e., the IV) equals the second half. Hence, a second key candidate (from the second 64-bit half) also exhibits a significant correlation. Indeed, the second peak (red one) belongs to the correct key candidate `0xAB` for the corresponding key byte 8 in the second 64-bit half. As expected, due to the serial nature of the hypothetical architecture, the correlation occurs at a later point in time.

We conducted the CPA for all 16 AES S-boxes and obtained a minimal correlation coefficient (determining the required number of traces) of $\rho_{min} = 0.031$ for the fourth S-box. Hence, according to the estimation given in [Mangard et al. 2007], the minimal number of traces to extract the full AES key is approximately $28/\rho_{min}^2 = 29,136$.

Figure 15 depicts the according correlation coefficient for the first S-box when leaving out the DFT pre-processing step. In general, the results are similar to those of Figure 14, however, the observed correlation is halved compared to the CPA with the DFT pre-processing. Overall, we obtained a $\rho_{min} = 0.021$, i.e., 63,492 traces would be needed when leaving out the DFT pre-processing.

Using our current measurement setup, 10,000 traces can be recorded in approximately 55 min. Note that the speed of the data acquisition is currently limited by the μC; thus, this time could be reduced with

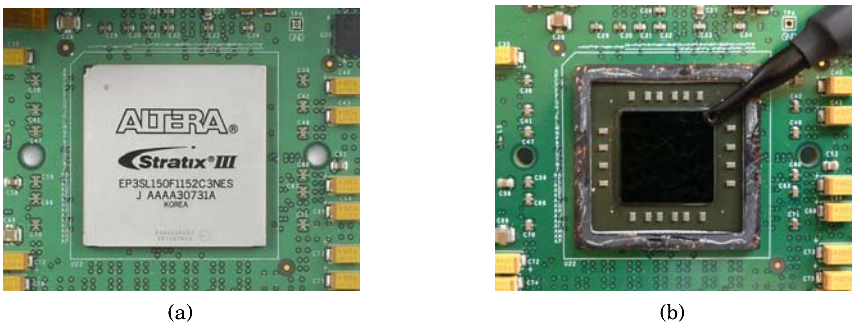


Fig. 16. Stratix III FPGA development kit, a) the original FPGA, b) the decapsulated FPGA with an EM probe at the optimal position

further engineering efforts. Nevertheless, the amount of traces required to perform a full-key recovery can be collected in less than three hours.

6. SIDE-CHANNEL KEY EXTRACTION OF STRATIX III

The platform we selected to examine the side-channel vulnerability of Stratix III FPGAs is a standard development kit [sxi 2008] for the Stratix III EP3SL150F1152 high-performance FPGA. Note that in contrast to the case of Stratix II this development board has not been designed for SCA. Hence, to measure the side-channel signal, one option is to modify the board, e.g., by removing all capacitors buffering the supply voltage and by putting a shunt resistor into the supply line. Another option (which we used) is to measure the side-channel leakage using an electro-magnetic (EM) probe. The FPGA, which is one of Altera's high-density FPGAs, is covered by a metal cap as a heat sink. On the other hand, this cap dampens EM emanations that are observed with a probe on top of the FPGA. Therefore,we removed the cap by means of mechanical tools to directly access the FPGA die.

Another issue was to select an appropriate probe and to localize the best probe position (with low noise level) so that suitable side-channel leakage can be acquired. We experimentally tested several EM probes and numerous different positions. The best result in our experiments was achieved with an H-Field near-field RF-R 3-2 probe made by LANGER EMV-Technik [EMV 2013]. The probe position for which we observed the best side-channel leakage is shown in Fig. 16. Since the amplitude of the signal was still low, we used two Mini-Circuits ZFL-1000LN+ amplifiers [min 2013] connected in series to obtain EM signals filling the input range of the DSO. The used DSO is the same as for Stratix II, i.e., a LeCroy WavePro 715Zi, however, we performed the measurements at a higher sampling rate of 10 GS/s and a bandwidth of 1 GHz.

The measurement scenario is different compared to the case of Stratix II: Due to the counter mode used in Stratix II, most of the AES plaintext bytes stay unchanged (in one power-up). This prevents side-channel attacks to effectively recover all key bytes from a single power-up. Therefore, as stated in the Section 5, we had to perform the measurement for Stratix II by repeating the following scenario: i) choose a random IV, ii) power-up the Stratix II FPGA, and iii) measure a few traces corresponding to the first few encryptions being performed by the FPGA. In contrast, as stated in Section 3.3, the counter mode is not used by Stratix IIIFPGAs. Instead, two consecutive AES plaintext blocks differ completely, and thus each byte is essentially randomized. Therefore, the aforementioned measurement process is not required here, and one can measure the side-channel leakage during a single power-up of a Stratix III FPGA (configured by the original encrypted bitstream). The corresponding IV can be extracted from the bitstream header, and – with the knowledge provided in Section 3 – all plaintexts internally passed to the encryption module can be computed.

The remaining task was to check whether the internal architecture we found for the Stratix II AES module is identical for Stratix III. Since AES-256 is used in Stratix III, at least the key schedule and the number of rounds compared to AES-128 of Stratix II are different. As the first step, we worked in a known-key scenario and used the model that worked best for Stratix II. In other words, we computed the correlation coefficient between the EM traces measured during one full power-up of the Stratix III, i.e., 365,000 traces, and the HD of consecutive bytes in each row of the AES state after `ShiftRows` in the first round. The result shown in Fig. 17 clearly indicates the correctness of this power model and our guess for the internal architecture.

In order to mount an attack, one has to first recover all 16 key bytes of the first round, which form the first half of the 256-bit key. Due to the structure of AES-256, the second half of the key is used as the round key in the second round. Therefore, after having recovered the first round key, the input of the second `AddRoundKey` for each plaintext can be computed. Hence, the attack is extended to the second round by guessing 16

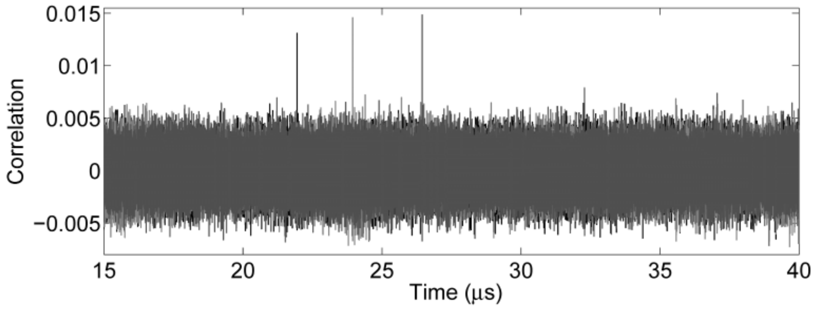


Fig. 17. Correlation coefficient for the HD of the row-wise consecutive `ShiftRows` bytes using 365,000 traces measured during one power-up of the Stratix III

additional key bytes (second part of the key). These bytes can be recovered using the same power model and the same hypothesis for the architecture. Due to a higher noise level in EM measurements compared to power traces, it might be the case that the traces measured during one power-up are not sufficient for a successful side-channel key recovery. In this case, the measurement process can be repeated for more power-ups – using the same encrypted bitstream – until the required number of traces has been collected. Similar to the Stratix II, the quality of the traces could also be improved by applying filters. We should mention that we have examined a complete key-recovery attack on different encrypted bitstream of Stratix III; in the worst case, we required the traces of 5 power-ups to fully recover the 256 bits of the key.

7. IMPLICATIONS AND FUTURE WORK

Having reverse-engineered the relevant functions of the Quartus II program, all details of the bitstream encryption, including the proprietary algorithms of the design security scheme of the Stratix II and Stratix III FPGA families are revealed. Using this knowledge, a side-channel adversary can mount a successful key recovery attack on the dedicated decryption hardware. As a consequence of our attacks, cloning of products employing either Altera Stratix II or Stratix III FPGAs for which the bitstream encryption feature is enabled becomes possible. Moreover, an attacker cannot only extract and reverse-engineer the bitstream, but might also modify it or create a completely new bitstream that would be accepted by the device. This fact is especially relevant in military applications, but could also have a major impact in other cases, e.g., for surveillance and Trojan hardware scenarios. Furthermore, an unencrypted bitstream allows an adversary to read out secret keys from security modules or to recover classified security primitives.

Since the Stratix II family belongs to an older generation of Altera FPGAs, the fact that SCA countermeasures have been ignored during the development appears likely. However, our findings show that this issue has not been addressed in the newer generation as Stratix III. Note that recent families like Stratix V or Arria II probably also feature an only slightly different scheme for bitstream encryption compared to Stratix III. Therefore, with the knowledge we obtained by reverse-engineering the bitstream encryption scheme and performing the attack on Stratix III, analyzing the security of the more recent Altera FPGAs from an SCA point of view is interesting for future work.

REFERENCES

2007. *Stratix II Device Handbook, Volume 1*. Technical Report. Altera. http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf.

2008. Stratix III FPGA Development Kit. http://www.altera.com/products/devkits/altera/kit-siii-host.html. (2008).

2009. *AN 341: Using the Design Security Feature in Stratix II and Stratix II GX Devices*. Technical Report. Altera. http://www.altera.com/literature/an/an341.pdf.

2011. Defense Science Board. http://www.acq.osd.mil/dsb/. (2011).

2012. Hex-Rays SA. (2012). http://www.hex-rays.com.

2012. On-line CRC calculation and free library. (2012). http://www.lammertbies.nl/comm/info/crc-calculation.html.

2013. LANGER EMV-Technik, near-field probes. http://www.langer-emv.de/en/products/disturbance-emission/near-field-probes/rf-1/devices-data. (2013).

2013. Mini-Circuits, Amplifier Data Sheet. http://www.minicircuits.com/pdfs/ZFL-1000LN+.pdf. (2013).

AIST. 2008. *Side-channel Attack Standard Evaluation Board SASEBO-B Specification*. http://www.risec.aist.go.jp/project/sasebo/download/SASEBO-B_Spec_Ver1.0_English.pdf.

Alessandro Barenghi, Gerardo Pelosi, and Yannick Teglia. 2010. Improving First Order Differential Power Attacks through Digital Signal Processing. In *Security of Information and Networks - SIN 2010*. ACM, 124–133.

Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation Power Analysis with a Leakage Model. In *CHES 2004 (LNCS)*, Vol. 3156. Springer, 16–29.

Altera Corporation. 2012a. Design Security . (2012). http://www.altera.com/products/devices/stratix-fpgas/about/security/stx-design-security.html.

Altera Corporation. 2012b. Stratix III FPGA: Lowest Power, Highest Performance 65-nm FPGA. (2012). http://www.altera.com/devices/fpga/stratix-fpgas/stratix-iii/st3-index.jsp.

Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani. On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme.. In *CRYPTO 2008 (LNCS)*, Vol. 5157. Springer, 203–220.

C.H. Gebotys, C.C. Tiu, and X. Chen. 2005. A countermeasure for EM attack of a wireless PDA. In *ITCC 2005*, Vol. 1. IEEE Computer Society, 544–549.

Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *CRYPTO 99 (LNCS)*, Vol. 1666. Springer, 388–397.

Ralf Krueger. 2004. *Application Note XAPP766: Using High Security Features in Virtex-II Series FPGAs*. Technical Report. Xilinx. http://www.xilinx.com/support/documentation/application_notes/xapp766.pdf.

Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer.

Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. 2011. On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs. In *CCS 2011*. ACM, 111–124.

Amir Moradi, Markus Kasper, and Christof Paar. 2012. 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 (LNCS)*, Vol. 7178. Springer, 1–18.

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

NIST. 2001. *Recommendation for Block 2001 Edition Cipher Modes of Operation*. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

David Oswald and Christof Paar. 2011. Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World. In *CHES 2011 (LNCS)*, Vol. 6917. Springer, 207–222.

Thomas Plos, Michael Hutter, and Martin Feldhofer. 2008. Evaluation of Side-Channel Preprocessing Techniques on Cryptographic-Enabled HF and UHF RFID-Tag Prototypes. In *RFIDSec 2008*. 114–127.

Sergei Skorobogatov and Christopher Woods. 2012. In the blink of an eye: There goes your AES key. Cryptology ePrint Archive, Report 2012/296. (2012). http://eprint.iacr.org/.

Chen Wei Tseng. 2005. *Lock Your Designs with the Virtex-4 Security Solution*. XCell Journal. Xilinx.