Side-Channel Attacks on the Bitstream Encryption Mechanism of Altera Stratix II

Facilitating Black-Box Analysis using Software Reverse-Engineering

Amir Moradi, David Oswald, Christof Paar, Pawel Swierczynski
Horst Görtz Institute for IT-Security
Ruhr University Bochum, Germany
firstname.lastname@rub.de

ABSTRACT

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 used to configure the FPGA. From a mathematical point of view, the employed encryption algorithms, e.g., AES or 3DES, are highly secure. However, recently it has been shown that the bitstream encryption feature of several FPGA product lines is susceptible to side-channel attacks that monitor 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 FPGA. To this end, we reverse-engineered the details of the proprietary and unpublished Stratix II bitstream encryption scheme from the Quartus II software. Using this knowledge, we demonstrate that the full 128-bit AES key of a Stratix II can be recovered by means of side-channel analysis with 30,000 measurements, which can be acquired in less than three hours. The complete bitstream of a Stratix II that is (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 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.

Keywords

Side-channel attack, bitstream encryption, AES, Altera, Stratix II, hardware security, reverse-engineering

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 [1]. 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, FPGAs close the gap between powerful but inflexible Application Specific Integrated Circuits (ASICs) and highly flexible but performance-limited microcontroller (μC) solutions. FPGAs combine some advantages of software (fast development, low non-recurring engineering costs) with those of hardware (performance, relative power efficiency). These advantages have made FPGAs an important fixture 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 Advanced Encryption Standard (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 Triple-DES (3DES) or AES in Cipher Block Chaining (CBC) mode [12, 21]. Recent findings reported in [14] and [15] 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 [20].

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 bit, are guessed, and the according hypotheses are verified. This process is repeated until the whole key has been revealed [9, 11].

In this work, we analyze the bitstream protection mechanism of Altera's Stratix II FPGA families called *design security*. 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 family, is given. 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 Altera's Stratix II FPGAs to power analysis 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. Also, basic security problems of the according scheme are illustrated. The details of our side-channel attacks are presented in Section 3 and Section 4. Finally, in Section 5, we conclude, summing up our research results.

2. REVERSE-ENGINEERING – DESIGN SECURITY SCHEME

For a side-channel analysis, 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 1.

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 3.1.

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

Table 1: Bitstream file formats generated by Quartus II

For transferring the bitstream to the FPGA, Altera provides several different configuration schemes [4, p.131-132]. Table 2 gives an overview on the different available schemes. For our purposes, we used the Passive Serial (PS) configuration scheme, because it supports bitstream encryption and moreover, because the configuration clock signal is controlled by the configuration device.

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

Table 2: Configuration modes for the Stratix II

Regarding the actual realization of the bitstream encryption, relatively little information is known. In the public documents [5] 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

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 [2]. Amongst others, this program allows analyzing 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).

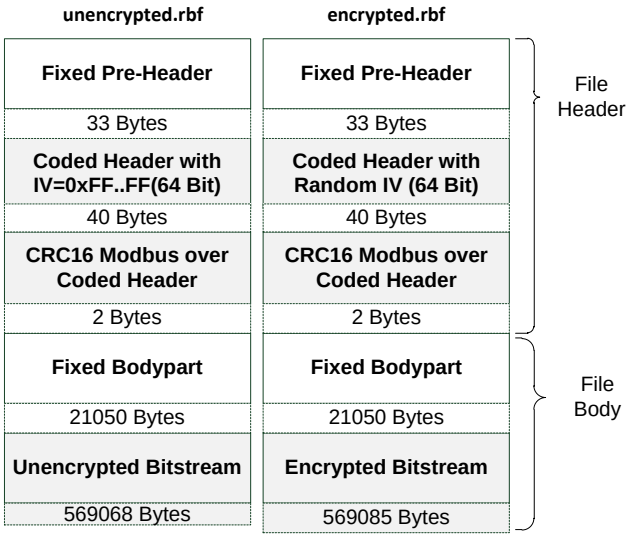

Figure 1: Structure of an unencrypted and an encrypted .RBF file

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.

Table 3 shows the resulting IV bit positions. The notation $Y_{\text{bit}X}$ 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.

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

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}

Table 3: Mapping between the IV bits and the header bytes

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 [3] computed over the preceding 40 header bytes for integrity check purposes.

The body starts with a 21050-byte block that is equal for both encrypted and unencrypted files. This block is followed by the actual bitstream (in encrypted or unencrypted form). 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

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.

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_pgio_nv_aes.dll`. Apparently,

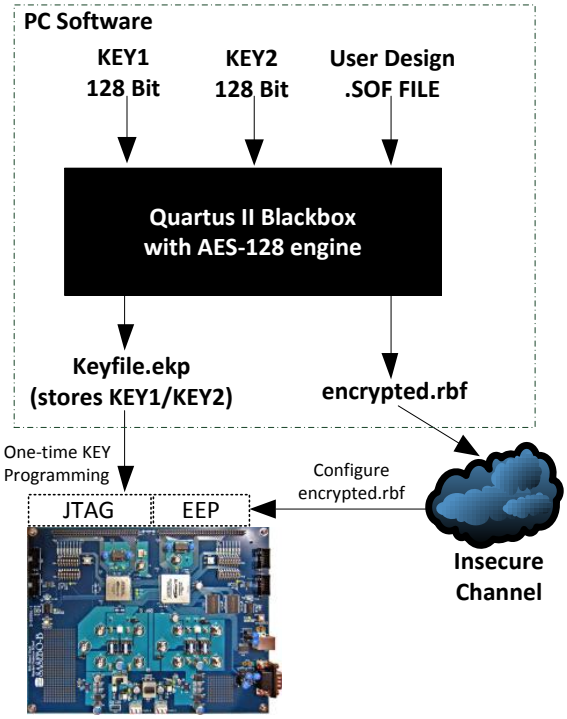

Figure 2: Quartus II black-box generating encrypted Stratix II bitstreams

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.

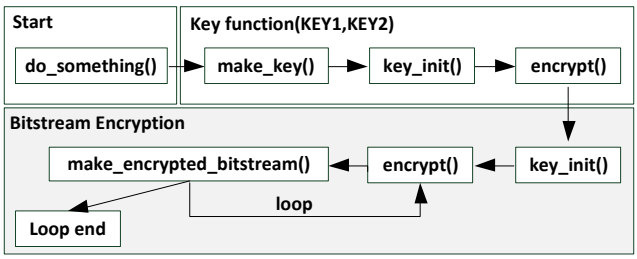

Figure 3: Quartus II call sequence during the bitstream encryption

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 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 *Worked Example*

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

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

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

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 exemplary 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 [17] is applied. Figure 4 shows the corresponding block diagram.

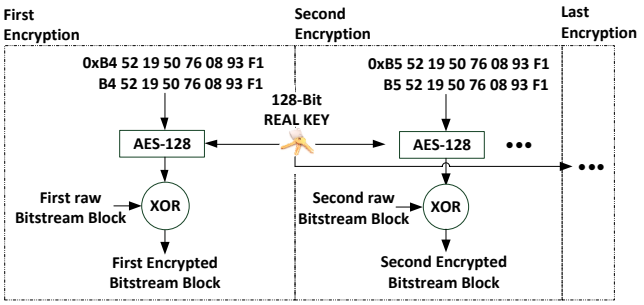

Figure 4: AES in CTR mode

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:

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

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.

3. SIDE-CHANNEL PROFILING

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 4), 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 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*.

3.1 Measurement Setup

Our DUT, a Stratix II FPGA (EP2S15F484C5N), is soldered onto a SASEBO-B board [6] 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

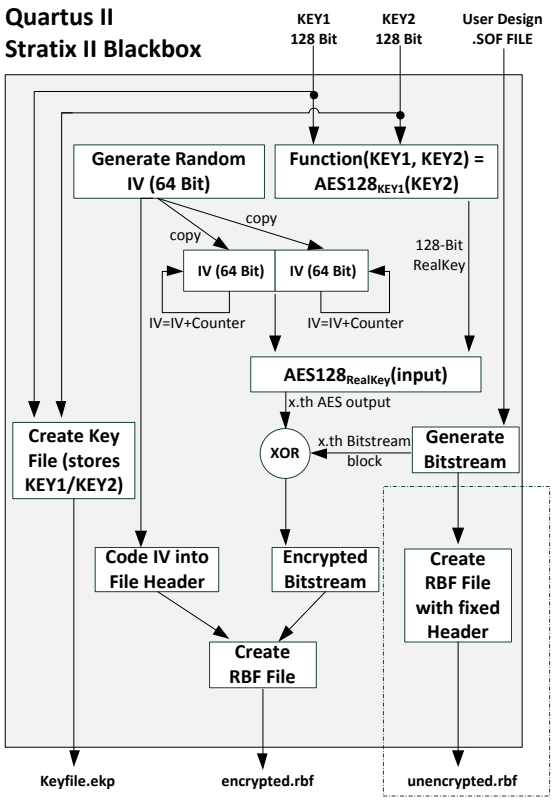

Figure 5: Overview of the bitstream encryption process for the Stratix II FPGAs

mode. For this purpose, we built an adapter that is conformant to [4, 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.

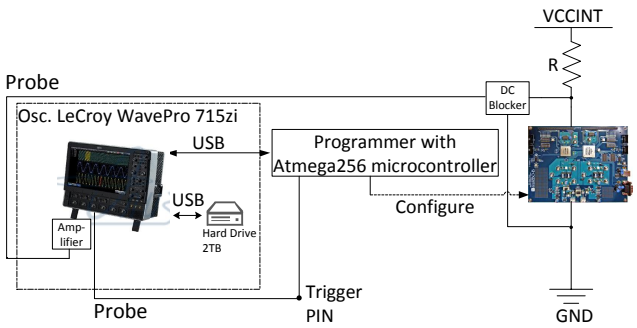

Figure 6: Measurement setup for SCA

According to [4, 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 6. 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 1.

Algorithm 1 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 3)
- [PC] Compute CRC-16 over coded header
- [PC] Send coded header with CRC-16 (42 bytes) to μC
- [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 μC
- [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

3.2 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 decryptor hence each time has the same input. 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 7 illustrates the corresponding mean traces.

As it is clearly visible in Figure 7, 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

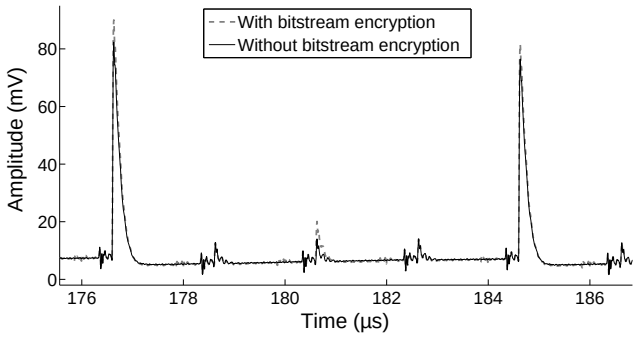

Figure 7: Average power consumption (10k traces) while sending an unencrypted (solid) and an encrypted (dashed) bitstream. Zoom on one byte.

second encrypted bitstream block, the DUT computes the XOR of the first AES output with the encrypted bitstream and configures the corresponding FPGA blocks.

3.3 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 1. 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), [8], 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 8 shows nine of the correlation curves for the states after each AES round.

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

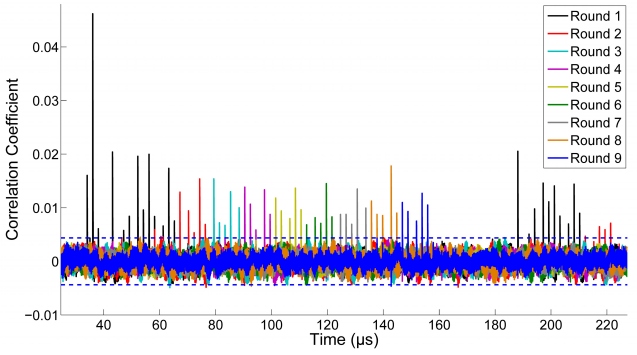

Figure 8: Correlation coefficient for one full AddRoundKey 128-bit state (one curve for each round). Utilized models: 1st curve ↔ HW of round 1, 2nd curve ↔ HW of round 2, etc.

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 8, 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 8) appears for both the first and second AES encryption. Even the states after round 2 of 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.

4. SIDE-CHANNEL KEY EXTRACTION

As shown in Section 3, 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.

4.1 Digital Pre-Processing

As commonly encountered in SCA, the effect of the AES encryption on the overall power consumption is rather small (cf. Section 3). 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 [7]. 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 μs 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 [10] 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 [18, 19].

4.2 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 3.3. 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 [16]. 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 9.

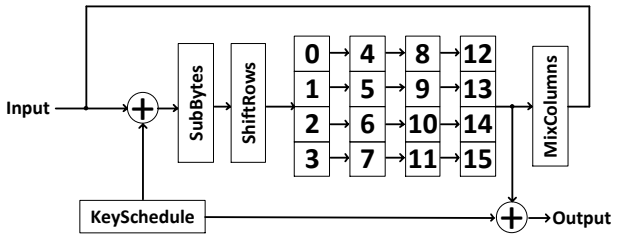

Figure 9: Hypothetical architecture of the AES implementation.

For the key extraction in Section 4.3, 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 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.

4.3 Results

Using the described power model, we computed the correlation coefficient for the respective (byte-wise) HD of the AES states. Figure 10 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 $4/\sqrt{\#traces} = 0.006$ [13].

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

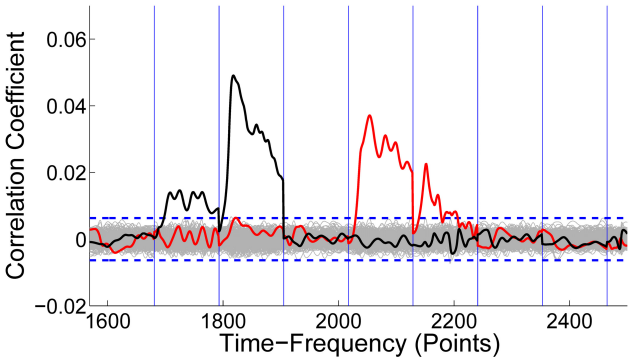

Figure 10: Correlation coefficient for the first S-box after 400k traces using DFT pre-processing. Correct key candidate `0x2B`: black curve.

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 [13], the minimal number of traces to extract the full AES key is approximately $^{28}/_{\rho^2_{min}} = 29,136$.

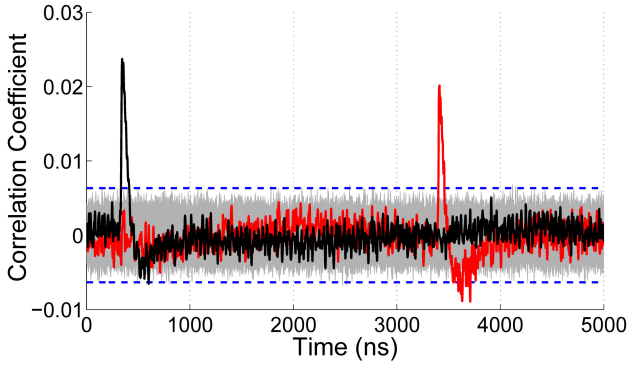

Figure 11: Correlation coefficient for the first S-box after 400k traces without DFT pre-processing. Correct key candidate `0x2B`: black curve.

Figure 11 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 10, 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 further engineering efforts. Nevertheless, the amount of traces required to perform a full-key recovery can be collected in less than three hours.

5. IMPLICATIONS – FUTURE WORKS

After reverse-engineering the relevant functions of the Quartus II program, all details of the bitstream encryption, including the proprietary algorithms of the design security scheme have been 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 Altera Stratix II FPGAs for which the bitstream encryption feature is enabled becomes straightforward. Moreover, an attacker can not only extract and reverse-engineer the bitstream, but might also modify it or create a completely new one that would be accepted by the device. This fact is especially sensitive in military applications, but could also have a major impact in other cases, e.g., 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 generations of Altera FPGAs, the fact that SCA countermeasures have been ignored during the development appears likely. However, recent families like Stratix V or Arria II probably feature an only slightly different scheme for bitstream encryption. At least, these FPGAs are supposed to provide 256-bit security compared to the 128-bit security of Stratix II. Therefore, analyzing the security of the more recent Altera FPGAs from an SCA point of view is interesting for future work.

6. REFERENCES

[1] Defense Science Board. http://www.acq.osd.mil/dsb/.

[2] Hex-Rays SA. http://www.hex-rays.com.

[3] On-line CRC calculation and free library. http://www.lammertbies.nl/comm/info/crc-calculation.html.

[4] Stratix II Device Handbook, Volume 1. Technical report, Altera, 2007. http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf.

[5] AN 341: Using the Design Security Feature in Stratix II and Stratix II GX Devices. Technical report, Altera, 2009. http://www.altera.com/literature/an/an341.pdf.

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

[7] A. Barenghi, G. Pelosi, and Y. Teglia. Improving First Order Differential Power Attacks through Digital Signal Processing. In *Security of Information and Networks - SIN 2010*, pages 124–133. ACM, 2010.

[8] E. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. In *CHES 2004*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004.

[9] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani. On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme. In *CRYPTO 2008*, volume 5157 of *LNCS*, pages 203–220. Springer.

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

[11] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO 99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

[12] R. Krueger. Application Note XAPP766: Using High Security Features in Virtex-II Series FPGAs. Technical report, Xilinx, 2004. http://www.xilinx.com/support/documentation/application_notes/xapp766.pdf.

[13] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[14] A. Moradi, A. Barenghi, T. Kasper, and C. Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from xilinx Virtex-II FPGAs. In *CCS 2011*, pages 111–124. ACM, 2011.

[15] A. Moradi, M. Kasper, and C. Paar. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures - An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism. In *CT-RSA 2012*, volume 7178 of *LNCS*, pages 1–18. Springer, 2012.

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

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

[18] D. Oswald and C. Paar. Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World. In *CHES 2011*, volume 6917 of *LNCS*, pages 207–222. Springer, 2011.

[19] T. Plos, M. Hutter, and M. Feldhofer. Evaluation of Side-Channel Preprocessing Techniques on Cryptographic-Enabled HF and UHF RFID-Tag Prototypes. In *RFIDSec 2008*, pages 114–127, 2008.

[20] S. Skorobogatov and C. Woods. In the blink of an eye: There goes your AES key. Cryptology ePrint Archive, Report 2012/296, 2012. http://eprint.iacr.org/.

[21] C. W. Tseng. Lock Your Designs with the Virtex-4 Security Solution. XCell Journal, Xilinx, Spring 2005.