

RUHR-UNIVERSITÄT BOCHUM

Security Analysis of the Bitstream Encryption Scheme of Altera FPGAs

Pawel Swierczynski

Master's Thesis. December 6, 2012.
Chair for Embedded Security – Prof. Dr.-Ing. Christof Paar
Advisor: Amir Moradi, David Oswald

Abstract

Altera provides custom logic solutions and is, besides Xilinx, one of the biggest vendors in their sector. Altera's Field Programmable Gate Arrays (FPGAs) are SRAM-based devices and thus volatile, which implies that they load their configuration from a configuration device or a flash memory at each new power-up. The FPGA designs are given in the form of a bitstream. In order to protect such a configuration design from being intercepted and thus cloned or modified, a solution called *design security* is offered. It is a feature based on the Advanced Encryption Standard (AES) encryption, and is available for the low-cost Cyclone III LS FPGAs, for the midrange FPGAs Aria II, and especially for the high-end FPGAs Stratix II, Stratix III, Stratix IV, and Stratix V. The design security is offered in two versions: A non-volatile variant that stores a one-time programmable AES key or a volatile solution based on a backup battery, allowing to re-program the AES key or to erase it.

The utilized AES engine is embedded on the FPGA as an additional unit. Its task is to decrypt previously encrypted configuration designs while they are downloaded from an external source. Stratix II and Stratix II GX FPGAs use AES-128, while all other solutions provide AES-256. From a mathematical point of view, algorithms like AES or 3DES are highly secure. However, recently, it was 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 thesis, we present the first successful side-channel attack on the bitstream encryption of the Altera Stratix II FPGA, which uses the non-volatile solution. For this, we reverse-engineered the details of the proprietary and unpublished Stratix II bitstream encryption scheme (and that of Stratix III) from the Quartus II software. Based on this information, we present how we obtained the full 128-bit AES key of a Stratix II by means of side-channel analysis with 30,000 measurements, which can be acquired in less than three hours.

The complete unencrypted configuration bitstream of a Stratix II that is (seemingly) protected by the design security 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 Intellectual Property (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. Moreover, we outline a potential problem due to the Initialization Vectors (IVs) that are used in a disadvantageous way by the encryption engine.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

PAWEL SWIERCZYNSKI

Contents

1	Acronyms	v
2	Introduction	vii
2.1	Motivation	vii
2.2	Related Work	viii
2.3	Contribution	viii
2.4	Organization of this Thesis	1
3	Reverse-Engineering – Design Security Scheme of Stratix II	3
3.1	Preliminaries	3
3.2	RBF File Format	4
3.3	AES 128-Bit Key Derivation	6
3.3.1	Worked Example	9
3.3.2	Discussing the Security of the Key Derivation Function	9
3.4	Analyzing Unknown DLL Files	9
3.4.1	Finding Important Function Names	10
3.4.2	Explicit Calling of Functions	12
3.5	Utilized AES Encryption Mode	16
3.5.1	Potential Problem of the Initial Vector Computation	18
3.5.2	Decryption and Cloning of an Encrypted RBF File	21
4	Reverse-Engineering – Design Security Scheme of Stratix III	23
4.1	RBF File Structures	23
4.2	AES 256-Bit Key Derivation	24
4.2.1	Worked Example	25
4.3	Utilized AES Encryption Modes	26
4.3.1	Dumping Initial Vectors with IDA Pro	28
5	Side-Channel Profiling of Stratix II	33
5.1	Measurement Setup	33
5.2	Extending the Sasebo-B Board for PS Support	36
5.3	Difference between Unencrypted and Encrypted Bitstream	38
5.4	Correlation Power Analysis	42
5.4.1	Pearson’s Correlation Coefficient	42
5.4.2	Utilized Prediction Models	44
5.5	Locating the FPGA Configuration	49
5.6	Locating the AES Encryption	51

6	Side-Channel Key Extraction of Stratix II	53
6.1	Digital Pre-Processing	53
6.2	Hypothetical Architecture	53
6.3	Results with Digital Pre-Processing	54
6.4	Results without Digital Pre-Processing	57
7	Conclusion	65
7.1	Summary	65
7.2	Future Work	66
	List of Figures	69
	List of Tables	71
	List of Algorithms	72
	List of Listings	75
A	Appendix	77
A.1	IDC Scripting	77
A.1.1	Setting Breakpoints	77
A.1.2	Determining Cursor Offset from Segment	78
A.1.3	Patching IV Bytes for IV Coding Rules	78
A.2	C++ and C Programs	79
A.2.1	Decryption of an Encrypted Stratix II RBF File	79
A.2.2	Header of Unencrypted and Encrypted Stratix III RBF Files	81
A.2.3	First Ten Dumped IVs of Stratix III for Monday, 2012-10-15 23:30:51	82
A.2.4	First Ten Encrypted Bitstream Blocks of Stratix III for Monday, 2012-10-15 23:30:51	82
A.2.5	First Ten Unencrypted Bitstream Blocks of Stratix III	83
A.2.6	Decryption of an Encrypted Stratix III RBF File	83
A.2.7	Welford CPA	85
A.2.8	Prediction Models	89
A.2.9	μ C Configuration of the First Three 16-byte Bitstream Blocks	93
A.2.10	Matlab Scripts	95
A.2.11	Peak Extraction	96
	Bibliography	99

1 Acronyms

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
CPA	Correlation Power Analysis
DUT	Device Under Test
DFT	Discrete Fourier Transform
DFA	Differential Frequency Analysis
SCA	Side-Channel Analysis
FPGA	Field Programmable Gate Array
μC	Microcontroller
SCA	Side-Channel Analysis
PCB	Printed Circuit Board
DSO	Digital Storage Oscilloscope
SNR	Signal-to-Noise Ratio
IV	Initialization Vector
HW	Hamming Weight
HD	Hamming Distance
CBC	Cipher Block Chaining
CFB	Cipher Feedback Mode
CTR	Counter
UART	Universal Asynchronous Receiver Transmitter
JTAG	Joint Test Action Group
HDL	Hardware Description Language
PS	Passive Serial
AS	Active Serial
IP	Intellectual Property

3DES Triple-DES

NVM Non-Volatile Memory

DLL Dynamic Link Library

COFF Common Object File Format

LSB Least Significant Bit

DSP Digital Signal Processing

PC Personal Computer

LFSR Linear Feedback Shift Register

PCB Printed Circuit Board

2 Introduction

Ubiquitous computing has become reality and has begun 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 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 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]. In order to prevent these and other forms of abuse, it is often highly desirable to introduce security mechanisms into embedded systems to prevent reverse-engineering and manipulation of designs.

2.1 Motivation

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 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.

2.2 Related Work

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 [Kru04, Tse05]. Recent findings reported in [MBKP11] and [MKP12] show the vulnerability of these schemes to state-of-the-art Side-Channel Analysis (SCA). Indeed, it was 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 [SW12]. 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 [EKM⁺, KJJ99].

2.3 Contribution

In this thesis, we analyze the design security protection mechanism of Altera's Stratix II and Stratix III FPGAs. We perform a black-box analysis of these mostly undocumented targets, i.e., show how the design security feature is exactly implemented in the Quartus II application. Using the IDA Pro disassembler, we present how the actual AES encryption keys are computed for which one has to specify two keys called KEY1 and KEY2 during the generation of encrypted configuration designs. We thus illustrate the according key derivation functions for Stratix II and Stratix III FPGAs and provide worked examples for the resulting AES keys. After revealing the utilized block mode, we decrypt an encrypted configuration design and show how it can be cloned for Stratix II, once one has obtained the corresponding AES key. 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. Our results show the vulnerability of the design security feature of Altera's Stratix II FPGAs to side-channel attacks, leading to a complete key-recovery from the embedded encryption module due to the leak via the power consumption while the cryptographic operations are performed.

2.4 Organization of this Thesis

The remainder of this thesis is organized as follows. In Chapter 3, we describe the steps needed to reverse-engineer the Quartus II application in order to figure out the details of the design security scheme of Stratix II FPGAs. We reveal the unknown file structure and the secret key derivation function which is responsible for deriving the actual 128-bit AES key used by the FPGA to decrypt encrypted configuration designs. Moreover, we explain how we discovered the utilized AES block mode. Chapter 3 also discusses a potential security problem with the IVs that are used for encryption. Finally, the chapter ends with an explanation on how to decrypt an encrypted configuration design and how to create a clone once one has obtained the secret key (e.g., by means of SCA). Chapter 4 deals with analyzing the utilized design security scheme of Stratix III FPGAs, whose security scheme is based on 256-bit instead of 128-bit keys as it is the case for Stratix II. Furthermore, Chapter 4 outlines how the utilized IVs can be dumped in a debugger. The IVs are needed for the decryption of an encrypted configuration design. The details of our side-channel attacks are presented in Chapter 5 and Chapter 6. We illustrate how the full 128-bit AES key can be extracted from the power consumption of the FPGA. Finally, in Chapter 7, we conclude, summing up our research results.

3 Reverse-Engineering – Design Security Scheme of Stratix II

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 chapter, we thus illustrate the method we used to reveal the essential information, including the proprietary algorithms used for the key derivation and the encryption scheme.

3.1 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 3.1.

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 5.1.

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

Table 3.1: Bitstream file formats generated by Quartus II.

For transferring the bitstream to the FPGA, Altera provides several different configuration schemes [Str07, p.131-132]. Table 3.2 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, the configuration clock signal is controlled by the configuration device.

Regarding the actual realization of the bitstream encryption, relatively little information is known. In the public documents [AN309], 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

Mode	Bitstream Encryption
Fast Passive Parallel (FPP)	Supported
Active Serial (AS)	Supported
Passive Serial (PS)	Supported
Passive Parallel Asynch. (PPA)	Not supported
JTAG	Not supported

Table 3.2: Configuration modes for the Stratix II.

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.

3.2 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 zeroes, 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 batch script, cf. Listing 3.1, 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 IV for the bitstream encryption.

Listing 3.1: Freezing windows time.

```

1 echo on
2 :BEGIN
3 date 29.10.2012
4 time 01.00.00
5 ping -n 2 localhost >nul
6 GOTO BEGIN

```

To gain further insight into the internals of the file format, we used the reverse-engineering tool Hex-Rays IDA Pro [IDA]. 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 3.1 (for the specific FPGA fabric EP2S15F484C5N).

unencrypted.rbf	encrypted.rbf	
Fixed Pre-Header	Fixed Pre-Header	File Header
33 Bytes	33 Bytes	
Coded Header with IV=0xFF..FF(64 Bit)	Coded Header with Random IV (64 Bit)	
40 Bytes	40 Bytes	
CRC16 Modbus over Coded Header	CRC16 Modbus over Coded Header	File Body
2 Bytes	2 Bytes	
Fixed Bodypart	Fixed Bodypart	
21,050 Bytes	21,050 Bytes	
Unencrypted Bitstream	Encrypted Bitstream	
569,068 Bytes	569,085 Bytes	

Figure 3.1: Structure of an unencrypted and an encrypted Stratix II RBF file.

Both the unencrypted and encrypted RBF file starts 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. Algorithm 3.2.1 describes how an attacker can proceed to figure out the distribution of the IV bits in the header of the encrypted RBF file.

Algorithm 3.2.1: Retrieve IV Coding Rules

- 1: Load the Quartus binary into the IDA debugger and setup the design security tool.
 - 2: Set a breakpoint in the assembly code at the point where the 64-bit IV is copied into memory.
 - 3: **for** i=63 downto 0 **do**
 - 4: Setup filename to encryptedDesignbit_i.rbf.
 - 5: Start encryption process. Wait until the breakpoint is reached.
 - 6: Modify the given 64-IV in the stack to 0xFF...FF and clear the i'th bit.
 - 7: Disable the breakpoint. Finish the encryption. Re-enable the breakpoint.
 - 8: **end for**
 - 9: Create an encryptedDesignbit_FF...FF.rbf file using 0xFF...FF as 64-bit IV.
 - 10: **for** i=63 downto 0 **do**
 - 11: Determine bit position of IV bit i from the difference between the encryptedDesignbit_i.rbf and encryptedDesignbit_FF...FF.rbf file.
 - 12: **end for**
-

The results of Algorithm 3.2.1 reveal that changing one IV bit affects the outcome of one header byte. Hence, the IV bits and their encoded positions can be fully extracted. Table 3.3 shows the corresponding 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

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 3.3: Mapping between the IV bits and the header bytes.

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] 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 569,068 bytes. For the encrypted bitstream, 17 additional bytes are given. 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.

3.3 AES 128-Bit 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 3.3.2.

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 3.2, 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.

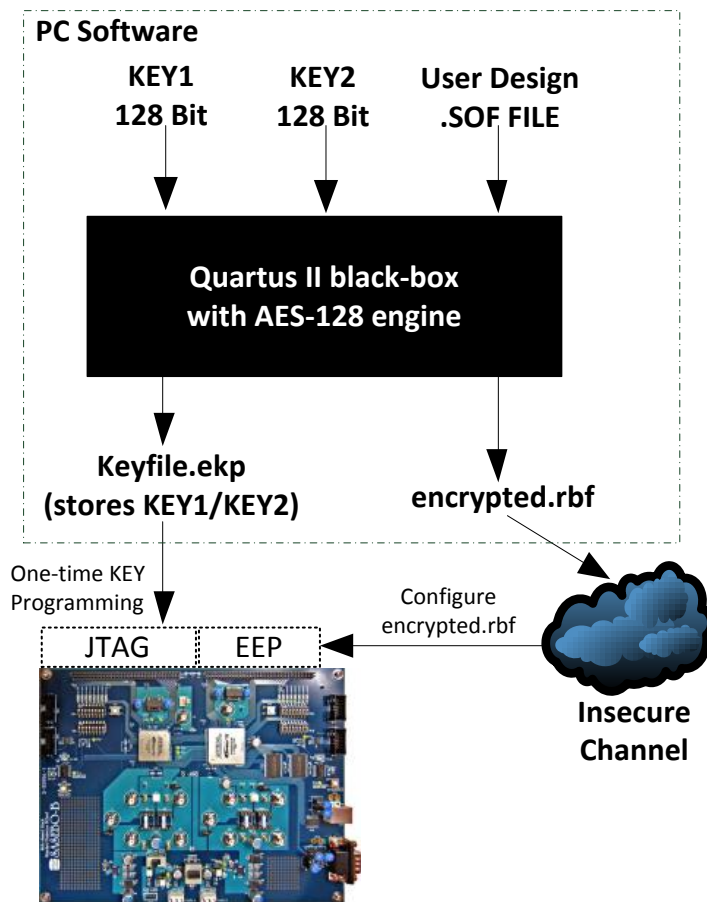


Figure 3.2: Quartus II black-box generating encrypted Stratix II bitstreams.

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 Dynamic Link Library (DLL) file `pgm_pgio_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. With the help of a program called *Dependency Walker* [Ste10], we displayed the DLL routine names. In Table 3.4, we sum up the most important routines. Note that the function names are given as “decorated” string.

Entry point	Function name
0x0001170	?key_init@PGM_AES@@AAEHQAIQBEH@Z
0x00020B0	?make_encrypted_bitstream@PGM_AES@@AAEXPAE0H0@Z
0x0001F90	?make_key@PGM_AES@@AAE?AW4PGM_AES_ERROR_CODE@@...
0x0002170	?do_something@PGM_AES@@QAE?AW4PGM_AES_ERROR_CODE@@...@Z
0x00010B0	?encrypt@AESencrypt@@QBHQBEQAE@Z

Table 3.4: DLL function names of `pgm_pgio_nv_aes.dll`.

Fig. 3.3 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 Fig. 3.3. Note that due to the available debugging information, all function names are exactly those chosen by the Altera developers.

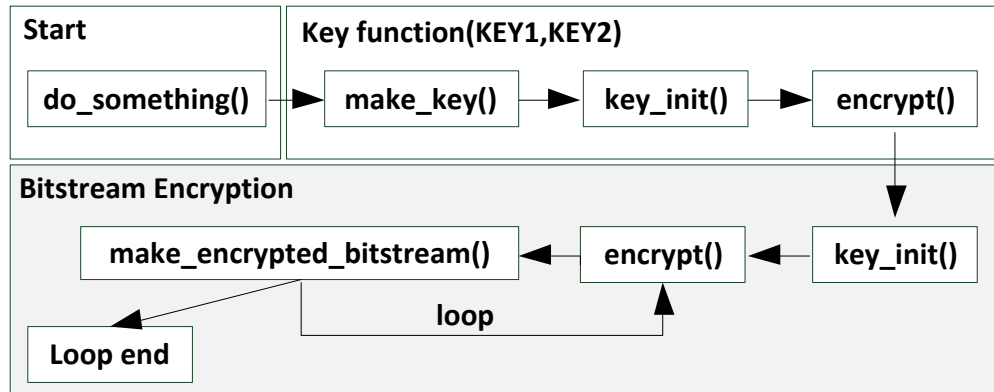


Figure 3.3: Quartus II call sequence during the bitstream encryption.

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

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

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

3.3.1 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 = AES128_{KEY1}(KEY2) (big endian)

0x2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C

3.3.2 Discussing the 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.

3.4 Analyzing Unknown DLL Files

The Quartus application consists of several executable¹ and DLL² files. Since DLL files embed several functions that can be invoked by the main executable, it is important to filter the relevant ones and to understand their corresponding functionality. The source code of a DLL file cannot be viewed in plain and hence, it appears as a black-box, but the names (e.g., as depicted in Table 3.4) and number of arguments of these functions are available and ease further analysis. For debugging the DLL internals, one needs to analyze the function's behavior, e.g., using the the IDA debugger.

¹For example: quartus.exe, quartus_pgm.exe, or quartus_map.exe

²For instance: pgm_pgmio.dll, pgm_common.dll, QtCore4.dll, or pgm_pgmio_nv_aes.dll

3.4.1 Finding Important Function Names

Often a program consists of several DLL files and each DLL file contains thousands of unknown functions. Analyzing all the given function names can be very time consuming and therefore, we recommend to use a script that filters important function names. We created a windows batch script (cf. Listing 3.2) that uses a program called *dumpbin* which comes with Microsoft Visual Studio [Mica]. Dumpbin is a program that is able to extract information of binary files. One can examine Common Object File Format (COFF) object files, standard COFF libraries, executable files, or DLL files.

Listing 3.2: Batch script displaying a filtered set of DLL function names.

```

1 set dumpbinPath=C:"Program Files (x86)"\Microsoft Visual Studio 10.0\VC\bin\amd64\dumpbin.exe
2 set dumpbinParas=/exports
3 set dllfilePath=C:\Users\it\AlteraBitStream\dllfiles
4 set dllfile=pgm_pgmio.dll
5 set storagePath=c:\Users\it\AlteraBitstream\
6 set keywords=(aes key encrypt decrypt enc dec rand^
7             IV seconds ecb cbc ctr ecb 56 64 128 192 256^
8             xor bitstream merge bitslice endian pad^
9             generate mask high low cnt counter nv^
10            aes 3des des crc secure security get set key1^
11            key2 para start run make block store receive^
12            encode decode obfuscate scramble permute^
13            first invert inv perm recover cipher algorithm^
14            hash real restore calc create crypto field bit^
15            input output calc copy concatenate size shift code pass^
16            sum checksum select flip reverse check verify file^
17            enable disable round triple two single mode write read^
18            load config cut clear header volatile data file append concatenate^
19            rbf)
20 mkdir %storagePath%\%dllfile%
21 for %%n in %keywords% do (%dumpbinPath% %dumpbinParas% %dllfilePath%\%dllfile% | grep -i "%%n" > %
    storagePath%\%dllfile%\%dllfile%_%%n.txt)

```

In the first line of Listing 3.2, we specify the path of the dumpbin executable, while we setup the utilized parameters in the second line.

For this example, we use the `/exports` option. It is responsible for displaying all definitions of an specified executable file or a DLL file. The next two lines specify the path of the DLL file target, which is in our case the `pgm_pgmio.dll` file.

However, before we continue to explain the script, we first illustrate the usage of the `export` parameter for a DLL file, for which the source code is available. Our artificial DLL sample file is called `setKey_xor_DllFile.dll`. It provides two external functions, namely a `setKey()` and a `doxor()` function. Listing 3.3 outlines the corresponding header file (`setKey_xor_DllFile.h`) that exports the mentioned DLL functions, while Listing 3.4 (`setKey_xor_DllFile.cpp`) defines its behavior. Compiling both files produces our desired `setKey_xor_DllFile.dll`.

Listing 3.3: C++ header file exporting two DLL functions.

```

1 // setKey_xor_DllFile.h
2 #include <iostream>
3 using namespace std;
4
5 namespace AES_namespace
6 {
7     class AES
8     {
9     public:
10         // Exports the setKey DLL function, key1/key2=input,result=output
11         static __declspec(dllexport) void setKey(unsigned char *key1, unsigned char *key2, unsigned
            char *result);
12         // Exports the doxor DLL function
13         static __declspec(dllexport) void doxor(unsigned char *key1, unsigned char *key2, unsigned
            char *result);
14     };
15 }

```

Listing 3.4: C++ file defining the behavior of the exported functions of Listing 3.3.

```

1 // setKey_xor_DllFile.cpp
2 #include "setKey_xor_DllFile.h"
3
4 namespace AES_namespace
5 {
6     // Defines the setKey behavior
7     void AES::setKey(unsigned char *key1, unsigned char *key2, unsigned char *result) {
8         for (int i = 0; i <= 31; i++) {
9             if (i <= 15) { result[i] = key1[i]; } else { result[i] = key2[i-16]; }
10         }
11     }
12     // Defines the doxor behavior
13     void AES::doxor(unsigned char *key1, unsigned char *key2, unsigned char *result) {
14         for (int i = 0; i <= 31; i++) {
15             result[i] = key1[i] ^ key2[i];
16         }
17     }
18 }

```

Imagine that we only possess the `setKey_xor_DllFile.dll` file instead of the code of Listing 3.3 and Listing 3.4. Then, we are able to readout only the function names and the corresponding entry points with the `dumpbin` tool by using the command (including the `/exports` parameter) that is given at the top of Listing 3.5. Knowing the exact function name, we can call the function explicitly. As it is visible in the output (at code lines 11-12) of Listing 3.5, we can extract the names of export functions, the number of arguments, and the argument types.

Listing 3.5: Extracting information of our sample DLL file.

```

1 // Utilized command:
2 dumpbin.exe /exports setKey_xor_DllFile.dll
3
4 // Corresponding output of the above executed command:
5 Dump of file Create_DllFile.dll
6
7 File Type: DLL
8

```

```

9 | Section contains the following exports for Create_DllFile.dll
10 | ...
11 | 1 0 00001040 ?doxor@AES@AES_namespace@@SAXPAE00@Z = ?doxor@AES@AES_namespace@@SAXPAE00@Z (
    |     public: static void __cdecl AES_namespace::AES::doxor(unsigned char *, unsigned char *,
    |     unsigned char *)
12 | 2 1 00001000 ?setKey@AES@AES_namespace@@SAXPAE00@Z = ?setKey@AES@AES_namespace@@SAXPAE00@Z (
    |     public: static void __cdecl AES_namespace::AES::setKey(unsigned char *, unsigned char *,
    |     unsigned char *)
13 | ...

```

Obviously, the labels of the arguments are not available (we only know that a pointer `*` is used) and thus it is more difficult to guess the meaning of a variable. This implies that one has to make assumptions whether a function argument denotes an input or output variable. Calling the `doxor()` function of Listing 3.3 is for instance possible when one specifies (using an additional program) the complete string `?doxor@AES@AES_namespace@@SAXPAE00@Z` as output in Listing 3.5 (line 11). We detail the process of calling an external function explicitly in Section 3.4.2.

Knowing how dumpbin can be used, we return to Listing 3.2. At the code lines 6-19, we set several keywords that serve for filtering relevant function names. The next command (at code line 20) simply creates a folder using the name of the examined DLL file, i.e., `pgm_pgmio.dll`. The last command iterates in a loop: For each specified keyword, it executes the `"dumpbin.exe /exports pgm_pgmio.dll"` command; then, the output of the dumpbin tool is piped to a tool that is called *grep* [gre]. Grep is a unix tool (also available for windows) that allows to search for matches to a specified regular expression. Each keyword is used once as a case insensitive (parameter *i* of *grep*) regular expression pattern. Therefore, only the lines of the dumpbin output (i.e. function names) that include the specified keyword are displayed in the windows console.

In the last step, the corresponding *grep* outputs (filtered function names) are stored in a text file, i.e., for the first keyword `aes`, the script creates a text file called `pgm_pgmio.dll_aes.txt` that contains only function names including `aes` as substring. Using this script, we located several relevant function names, e.g., a function that is called `make_encrypted_bitstream()`. The behavior of this specific function is outlined in Section 3.4.2

3.4.2 Explicit Calling of Functions

DLL files can be linked by an executable in one of two ways. The first approach is to link a DLL implicitly. This is known as static load or load-time dynamic linking [Micb]. An executable must obtain additional files from the DLL provider in order to be able to link a DLL file implicitly. Besides the actual DLL, a header file (`.h`) and an import library file (`.LIB`) must be provided.

The import library and the DLL file are usually generated simultaneously by the linker. However, the DLL header file has to import functions instead of exporting (used when generating DLL files) them. Thus, the provider publishes a header file that uses the `__declspec(dllimport)` directive instead of `__declspec(dllexport)`, cf. Listing 3.3. The corresponding header file can be then imported (besides the import library file and DLL file) by a main executable. To sum it up, this method requires two additional files

The previously described method is not suitable for calling unknown DLL functions (as we intend to do), because we neither possess a library nor a header file for the Quartus DLL files. For reverse-engineering the corresponding behavior of a DLL function, one has to use the second approach that is known as explicit linking. Explicit linking differs from implicit linking in that it loads the DLL files during the runtime of an executable and no additional files are required except the DLL itself.

The `?doxor@AES...@` function simply performs an XOR operation on the first specified input arrays (`array1[32]` and `array2[32]`) and finally stores the corresponding result into the `result[32]` variable, as defined in Listing 3.4. Note that if a method body of a DLL function is not known, an attacker has to perform further experiments with the function arguments to study the outputs. When the DLL module is not required anymore, it can be unloaded with the help of the `FreeLibrary()` function (at code line 54).

```
#include <windows.h>
```

[illegible]

```

21 // Third argument of the function (doxor...) to be called
22 unsigned char result[32]; for(int i=0; i <= 31; i++) { result[i]=0; }
23
24 // A handle of the setKey_xor_DllFile.dll file is assigned to the variable hDLL
25 // V1: Own artificial DLL file
26 HINSTANCE hDLL = LoadLibrary (L"setKey_xor_DllFile.dll");
27
28 // V2: Quartus DLL file
29 // HINSTANCE hDLL = LoadLibrary (L"pgm_pgmio_nv_aes.dll");
30
31 if (hDLL != NULL) {
32     cout << "DLL file loaded successfully." << endl;
33
34     // Create DLL function pointer
35     MYDLL myDllFunc;
36
37     // V1: Get address of the import function doxor() (Own DLL file)
38     myDllFunc = (MYDLL) GetProcAddress(hDLL, "?doxor@AES@AES_namespace@@SAXPAE00@Z");
39
40     // V2: Get address of the import function make_encrypted_bitstream() (Quartus DLL file)
41     // myDllFunc = (MYDLL) GetProcAddress(hDLL, "?make_encrypted_bitstream@PGM_AES@@AAEXPAEOH0@Z");
42
43     if(!myDllFunc)
44         cout << "Could not load DLL function routine." << endl;
45     else
46         // Execute function that is located at address 'myDllFunc'
47         myDllFunc(array1, array2, result); // V1
48         // int intA=0; myDllFunc(array1, array2, intA, result); // V2
49 }
50 else {
51     cout << "Could not load DLL file." << endl;
52 }
53 // Free library
54 FreeLibrary (hDLL);
55 return 0;
56 }

```

Using the described method of executing DLL functions explicitly, an attacker can, for example, figure out what the function `make_encrypted_bitstream()` of the Quartus DLL file `pgm_pgmio_nv_aes.dll` does. This function is used several times during the bitstream encryption as depicted in Figure 3.3. First, we execute the script given in Listing 3.2, but specify the `pgm_pgmio_nv_aes.dll` instead of the `pgm_pgmio.dll` file in order to obtain the corresponding “decorated” string of the function. One of the resulting text files is called `pgm_pgmio_nv_aes.dll_make.txt`. It contains three resulting function names because three lines of the dumpbin output matched the expression “make” (remember: it is one of several search terms (keywords) of Listing 3.2). In more detail, we obtain the decorated DLL function name

`?make_encrypted_bitstream@PGM_AES@@AAEXPAEOH0@Z`

and hence, we can specify this string at code line 41 of Listing 3.6. Furthermore, the corresponding DLL file name `pgm_pgmio_nv_aes.dll` has to be specified at line 29.

As can be seen in Listing 3.5, the dumpbin tool additionally outputs an undecorated string for each function name. To be more precise, the decorated function name

?make_encrypted_bitstream@PGM_AES@@AAEXPAE0H0@Z (coming from the Quartus DLL file) can be alternatively displayed as the following (undecorated) string:

```
void PGM_AES::make_encrypted_bitstream(unsigned char *, unsigned char *, int, unsigned char *).
```

Three pointers of the type `unsigned char` and an integer variable are passed as an argument to the `make_encrypted_bitstream()` function. We slightly adapted the code of Listing 3.6 in such way that code line 8 is used instead of code line 7 to obtain the correct calling convention of the `make_encrypted_bitstream()` function. In addition to that, code line 47 is replaced by code line 48. By doing so, the unknown Quartus DLL function can be executed without any errors, since the function is called with the correct (amount of) parameters. We executed our modified program and thus called the function `myDLLfunc(array1, array2, intA, result)` several times explicitly (using the `array1`, `array2`, and `result` variable of Listing 3.6). Trying different values for the `intA` parameter, we sum up the observed results in Table 3.5. Note that, in our case, the assignment `myDLLfunc=make_encrypted_bitstream()` applies, cf. code line 47 with 41.

Variable	Hexadecimal value
array1	0x00 10 20 30 40 50 60 70 80 90 A0 B0 C0 D0 E0 F0 11
array2	0x00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F FF
result if intA=0	0x00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF 00
result if intA=1	0x00 10 21 32 43 54 65 76 87 98 03 AB BC CD DE EF E1 00
result if intA=2	0x00 00 20 31 42 53 64 75 86 97 02 02 AC BD CE DF F1 E1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...	...
result if intA=14	0x00 00 00 00 00 00 00 00 00 00 00 00 00 00 0E 0E 13 12 15 14 17 16 19 18 B1 A1 D1 C1 F1 E1 00 00 00 00 00
result if intA=15	0x00 00 00 00 00 00 00 00 00 00 00 00 00 00 0F 10 13 12 15 14 17 16 19 18 B1 A1 D1 C1 F1 E1 00 00 00 00 00
result if intA=16	0x00 00 00 00 00 00 00 00 00 00 00 00 00 00 11 10 13 12 15 14 17 16 19 18 B1 A1 D1 C1 F1 E1 00 00 00 00 00

Table 3.5: Results of calling the `make_encrypted_bitstream()` function with different parameters.

The results of Table 3.5 reveal that the (previously unknown) `make_encrypted_bitstream()` function simply performs an XOR between the values of the `array1` and `array2` arrays. Moreover, the parameter `intA` shifts the index of the `array1` and `result` array. Listing 3.7 shows a possible implementation of the `make_encrypted_bitstream()` function.

Listing 3.7: Assumed implementation of the `make_encrypted_bitstream()` function.

```
1 // Possible implementation of the make_encrypted_bitstream() function
2 void make_encrypted_bitstream(unsigned char *array1, unsigned char *array2, int intA, unsigned
   char *result) {
3     int i;
4     for(i=0; i <= 15; i++) {
5         result[i+intA] = array1[i+intA] ^ array2[i];
6     }
7 }
```

Since the index of the `array1` and `result` array is shifted, we conclude that these two arrays are large, while `array2` is only supposed to store sixteen bytes. As one can see, we were able to figure out the method body of a simple function without analyzing the corresponding assembly code in the IDA Pro debugger and therefore,

explicit DLL function calling can be a very helpful tool for reverse-engineering a program. In Section 3.5, we further illustrate the utilized AES encryption mode and explain how the `make_encrypted_bitstream()` function is probably used by the Quartus application.

3.5 Utilized AES Encryption Mode

Having already revealed the key derivation scheme in Section 3.3, we focus on the details of the actual AES encryption, i.e., analyze the lower part of Figure 3.3. 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 [NIS01] is applied. Figure 3.4 shows the corresponding block diagram.

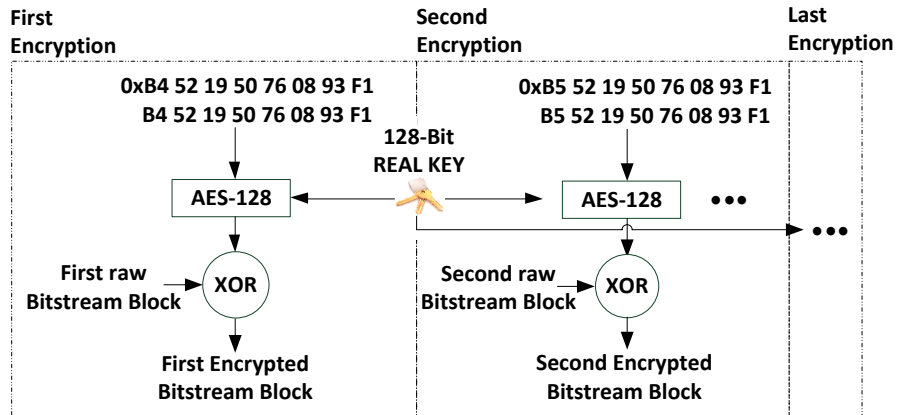


Figure 3.4: AES in CTR mode.

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

outlined in Listing 3.7. Knowing this, together with the function calling order, we assume that the Quartus application uses the `make_encrypted_bitstream()` function as described in Algorithm 3.5.1. Note that the index of an array targets one byte, e.g., `unencRBF[0]` returns the first byte.

Algorithm 3.5.1: Encrypt Bitstream

```

1: BLOCKS  $\leftarrow$  getAmountofBitstreamBlocks()
2: unencRBF[0...16 · BLOCKS]  $\leftarrow$  Derive bitstream from SOF file.
3: encRBF[0...16 · BLOCKS]  $\leftarrow$  0
4: currIV[0...15]  $\leftarrow$  Compute initial IV.
5: for i=0 to BLOCKS-1 do
6:   AES_out[0...15]  $\leftarrow$  AES128real key(currIV[0...15])
7:   make_encrypted_bitstream(unencRBF, AES_out, 16 · i, encRBF)
8:   Increment currIV
9: end for

```

In Section 3.4.2, we concluded that the `array1` and `result` arrays (defined in Listing 3.7) are large, while `array2` only stores sixteen bytes. Hence, it is quite likely that `array1` (1st input of the `make_encrypted_bitstream()` function) stores the full unencrypted bitstream, e.g., in an array that we refer to `unencRBF` (cf. Alg. 3.5.1). The `array2` array (2nd input of the `make_encrypted_bitstream()` function) is the current output (AES_out, cf. Alg. 3.5.1) of the 128-bit AES encryption engine. The `intA` variable (an integer) is then, a multiple of 16 (one block length) to control which bitstream block has to be encrypted. We further assume that the `result` array (4th input of the `make_encrypted_bitstream()` function) is supposed to store the resulting encrypted bitstream block `encRBF`, cf. Alg. 3.5.1. Thus, we assume that the Quartus application uses an algorithm similar to Algorithm 3.5.1.

As mentioned in Section 3.2, 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``B4 52 19 50``76 08 93 F1``B4 52 19 50``76 08 93 F1`.
Timestamp
Fixed bytes
Timestamp
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 (cf. Listing A.4). Given the correct real key and IV, we successfully decrypted the bitstream of an encrypted RBF file. Figure 3.5 summarizes the details of the bitstream encryption process of Stratix II.

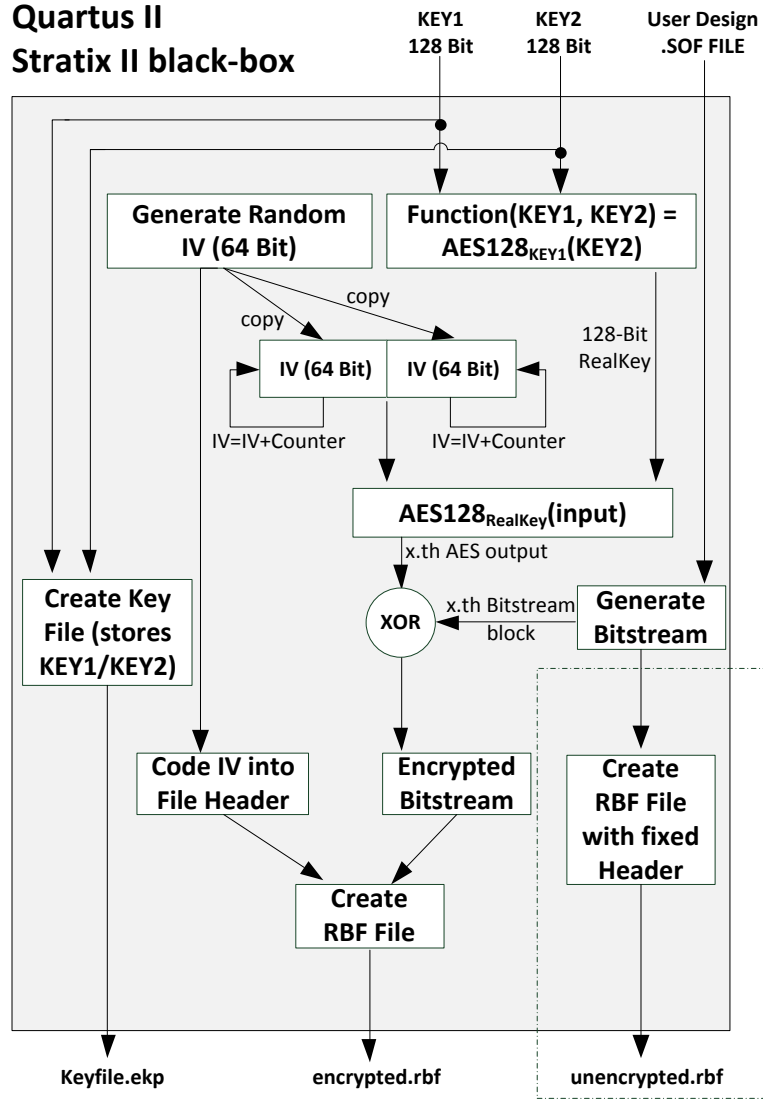


Figure 3.5: Overview of the bitstream encryption process for the Stratix II FPGAs.

3.5.1 Potential Problem of the Initial Vector Computation

Another potential security problem that we noticed is the utilized IV computation scheme. Imagine that a company encrypts a configuration design $unencRBF$ using KEY1 and KEY2. This leads to a specific real key rk_{fpga} . Equation 3.1 defines a 16-byte block that corresponds to the i 'th encrypted block of the first configuration design (also referring to the i 'th IV) using rk_{fpga} as cryptographic AES key.

$$encRBF_{IV_i, rk_{fpga}} := AES128_{rk_{fpga}}(IV + i) \oplus unencRBF_i \quad (3.1)$$

For the case that the same company encrypts a second configuration design (using the same KEY1 and KEY2, and hence, leading to the same real key rk_{fpga}), we obtain Equation 3.2.

$$\widetilde{encRBF}_{\widetilde{IV}_i, rk_{\text{fpga}}} := AES128_{rk_{\text{fpga}}}(\widetilde{IV} + i) \oplus \widetilde{unencRBF}_i \quad (3.2)$$

Since the initial IV depends on the windows timestamp for both configuration designs, obviously, Equation 3.3 applies for the case that the second configuration design is being encrypted with a delay of x seconds.

$$\widetilde{IV} := IV + x \quad (3.3)$$

Using Equation 3.2 and 3.3, we obtain the Equations 3.4-3.5.

$$\widetilde{encRBF}_{\widetilde{IV}_i, rk_{\text{fpga}}} \stackrel{3.2}{=} AES128_{rk_{\text{fpga}}}(\widetilde{IV} + i) \oplus \widetilde{unencRBF}_i \quad (3.4)$$

$$\stackrel{3.3}{=} AES128_{rk_{\text{fpga}}}(IV + i + x) \oplus \widetilde{unencRBF}_i \quad (3.5)$$

Incrementing the IV of Equation 3.1 for x times – as it is performed due to the full encryption of the first configuration design – leads to the IV shape $IV + i + x$. Thus, the x 'th IV of the first encryption matches the first IV ($\widetilde{IV} + i = IV + i + x$) of the second encryption. This also implies that the x 'th pseudorandom key of the first configuration design encryption $AES128_{rk_{\text{fpga}}}(IV + i + x)$ is identical to that of the first pseudorandom key of the second configuration design encryption $AES128_{rk_{\text{fpga}}}(IV + i + x) = AES128_{rk_{\text{fpga}}}(\widetilde{IV} + i)$. Using the same pseudorandom key for two different encryptions can lead to a serious problem because an attacker is able to remove the pseudorandom key by simply XORing both ciphertext blocks that use the same key. To provide an example, Table 3.6 outlines the first five encrypted blocks of $\widetilde{encRBF}_{\widetilde{IV}_i, rk_{\text{fpga}}}$ and $\widetilde{encRBF}_{\widetilde{IV}_i, rk_{\text{fpga}}}$. The second encryption is invoked with a delay of three seconds ($x = 3$). We exemplary perform the XOR between the values of the fourth row. This

Sequence of 1 st encryption	Sequence of 2 nd encryption (delay of 3 seconds)
$encRBF_{IV_0, rk_{\text{fpga}}}$	-
$encRBF_{IV_1, rk_{\text{fpga}}}$	-
$encRBF_{IV_2, rk_{\text{fpga}}}$	-
$encRBF_{IV_3, rk_{\text{fpga}}}$	$\widetilde{encRBF}_{\widetilde{IV}_0, rk_{\text{fpga}}} = \widetilde{encRBF}_{IV_3, rk_{\text{fpga}}}$
$encRBF_{IV_4, rk_{\text{fpga}}}$	$\widetilde{encRBF}_{\widetilde{IV}_1, rk_{\text{fpga}}} = \widetilde{encRBF}_{IV_4, rk_{\text{fpga}}}$
$encRBF_{IV_5, rk_{\text{fpga}}}$	$\widetilde{encRBF}_{\widetilde{IV}_2, rk_{\text{fpga}}} = \widetilde{encRBF}_{IV_5, rk_{\text{fpga}}}$

Table 3.6: IV sequences for two encryptions invoked at different points in time.

leads to Equation 3.71.

$$\begin{aligned}
& \text{encRBF}_{IV_3, rk_{\text{fpga}}} \oplus \widetilde{\text{encRBF}}_{IV_0, rk_{\text{fpga}}} \quad (3.71) \\
& \stackrel{3.1}{=} (\text{AES128}_{rk_{\text{fpga}}}(IV + 3) \oplus \text{unencRBF}_3) \oplus \text{encRBF}_{IV_0, rk_{\text{fpga}}} \\
& \stackrel{3.2}{=} (\text{AES128}_{rk_{\text{fpga}}}(IV + 3) \oplus \text{unencRBF}_3) \oplus (\text{AES128}_{rk_{\text{fpga}}}(\widetilde{IV} + 0) \oplus \widetilde{\text{unencRBF}}_0) \\
& \stackrel{3.3}{=} (\text{AES128}_{rk_{\text{fpga}}}(IV + 3) \oplus \text{unencRBF}_3) \oplus (\text{AES128}_{rk_{\text{fpga}}}(IV + 3) \oplus \widetilde{\text{unencRBF}}_0) \\
& = (\text{AES128}_{rk_{\text{fpga}}}(IV + 3) \oplus \text{AES128}_{rk_{\text{fpga}}}(IV + 3)) \oplus (\text{unencRBF}_3 \oplus \widetilde{\text{unencRBF}}_0) \\
& = \text{unencRBF}_3 \oplus \widetilde{\text{unencRBF}}_0
\end{aligned}$$

An attacker is able to retrieve the XOR sum as given in Equation 3.6.

$$\text{unencRBF}_3 \oplus \widetilde{\text{unencRBF}}_0 \quad (3.6)$$

Analogously, the computation can be repeated for the fifth and sixth row that results in Equation 3.7 and 3.8

$$\text{unencRBF}_4 \oplus \widetilde{\text{unencRBF}}_1 \quad (3.7)$$

$$\text{unencRBF}_5 \oplus \widetilde{\text{unencRBF}}_2 \quad (3.8)$$

which finally leads to the generalization of Equation 3.9 for $i \geq x$ (and a delay of x seconds).

$$\text{retrievedBlock}_i := \text{unencRBF}_i \oplus \widetilde{\text{unencRBF}}_{i-x} \quad (3.9)$$

The XOR sum of Equation 3.9 can only be computed if the second configuration design is encrypted with a maximum delay of `#BITSTREAMBLOCKS` seconds, because, then, no overlap of the same utilized IVs occurs between the encrypted configuration designs.

However, one might argue that this is no real threat in this case, since an attacker only possesses the XOR result between two configuration designs blocks, and hence, is not able to reconstruct one of both bitstreams. Nevertheless, in principle, an attacker should never be able to decrypt any information.

Furthermore, imagine that a company encrypts two configuration designs (both within a short time range) for a customer using the same 128-bit keys, then, transfers it via the internet. Later, the customer might decide, e.g., for marketing purposes, to make one of both (unencrypted) bitstreams public. If an attacker captured both encrypted configuration designs in the past, he would be able to decrypt the confidential configuration design (most of the bitstream blocks) with the help of the published bitstream (e.g., unencRBF_i). To do so, he simply computes an XOR operation as illustrated in Equation 3.10.

$$\text{retrievedBlock}_i \oplus \text{unencRBF}_i = \widetilde{\text{unencRBF}}_{i-x} \quad (3.10)$$

The point we want to make is that there might appear some problems due to the bad practice of encrypting two different configuration designs with the same 128-bit keys. Hence, at this point, we recommend to change the Personal Computer (PC) clock when

encrypting the second configuration design with the same key. Moreover, a designer should try to avoid the usage of the unix timestamp as an IV. At least it should not be simply incremented (when using the timestamp) to avoid overlaps. Apparently, the Altera developers are aware of this problem, since they replaced the simple incremental counter (of Stratix II devices) with an unknown random number function for Stratix III devices.

3.5.2 Decryption and Cloning of an Encrypted RBF File

In this section, we provide an algorithm that can create an unencrypted version of an encrypted RBF file, for the case that we already retrieved the real key (e.g., by means of SCA). Algorithm 3.5.2 describes the work flow. Note that the index of an array targets again one byte, e.g., `encRBF[0]` returns the first byte.

Algorithm 3.5.2: Decrypt an encrypted bitstream of Stratix II

```

1: unencRBF[0...590192]  $\leftarrow$  0
2: encRBF[0...590209]  $\leftarrow$  encryptedBitstream.rbf
3: realkey[0...15]  $\leftarrow$  16-byte KEY
4: unencRBF[0...32]  $\leftarrow$  encRBF[0...32]
5: currIV[0...7]  $\leftarrow$  Get 64-bit IV from encRBF[33...72] header using Table 3.3
6: Code IV 0xFF...FF (64-bit) into encRBF[33...72] header using Table 3.3
7: encRBF[40]  $\leftarrow$  0x1A; encRBF[51]  $\leftarrow$  0x1D; encRBF[52]  $\leftarrow$  0x1B; encRBF[55]  $\leftarrow$ 
   0x1D; encRBF[56]  $\leftarrow$  0x1F; encRBF[57]  $\leftarrow$  0x1D; encRBF[58]  $\leftarrow$  0x19
8: encRBF[73...74]  $\leftarrow$  CRC-16(encRBF[33...72])
9: unencRBF[33...74]  $\leftarrow$  encRBF[33...74]
10: unencRBF[75...21124]  $\leftarrow$  encRBF[75...21124]
11: for j=0 to 35565 do
12:   AESout[0...15]  $\leftarrow$  AES128realkey(currIV[0...7].concat(currIV[0...7]))
13:   index  $\leftarrow$  (21125 + j · 16)...(21125 + j · 16 + 15)
14:   unencRBF[index]  $\leftarrow$  (encRBF[index]  $\oplus$  AESout[0...15])
15:   currIV[0] = currIV[0] + 1
16:   if currIV[0]==0 then
17:     currIV[1] = currIV[1] + 1
18:     if currIV[1]==0 then
19:       currIV[2] = currIV[2] + 1
20:       if currIV[2]==0 then
21:         currIV[3] = currIV[3] + 1
22:       end if
23:     end if
24:   end if
25: end for
26: unencRBF[590181...590192]  $\leftarrow$  0xFF...FF
27: unencryptedBitstream.rbf  $\leftarrow$  unencRBF[0...590192]
```

At code line 1, 590,192 bytes are allocated for an array that we refer to as `unencRBF`. It is supposed to store the bytes (which are determined step by step) of the resulting unencrypted RBF file. At code line 2, all bytes of the encrypted RBF file are copied into an array called `encRBF`. Next, the (previously revealed) real key is stored in an array

called `realkey` (at code line 3). Then, the algorithm copies the `Fixed Pre-Header` bytes from the `encRBF` header to the according positions 0...32 of the `unencRBF` array (at code line 4), since these bytes are equal for the unencrypted and encrypted RBF file. At code line 5, the algorithm reads the initial IV used by the Quartus application for encrypting the RBF. The result is stored in an array called `currIV`.

The code lines 6-8 are responsible for creating a correct unencrypted header: The IV `0xFF...FF` bits (the IV shape of the unencrypted RBF file) are coded into the header (at code line 6), the respective control bytes (observed from an unencrypted RBF file header) have to be replaced (at code line 7), and finally, the CRC-16 checksum has to be updated accordingly (at code line 8). The updated `Coded Header` bytes are copied (at code line 9) to the positions 33...74 of the `unencRBF` array, and thus, the header is complete after code line 9 has been executed. Next, the algorithm simply copies the `Fixed Bodypart` bytes from the `encRBF` array into the positions 75...21124 of the `unencRBF` array.

Then, a for-loop iterates over each of 35,565 encrypted 16-byte bitstream blocks. For each encrypted block, the respective IV is encrypted (at code line 12). The algorithm then performs an XOR operation between the encrypted bitstream block (from the RBF file) and the corresponding AES result (at code line 14), and finally stores the decrypted 16-byte block in the corresponding positions of the `unencRBF` array. One loop round finishes after the IV is incremented (at code lines 15-24). Once the decryption process finishes, the last 12 bytes of the `unencRBF` array are filled (or: padded) with `0xFF` values (at code line 26). Finally, all bytes of `unencRBF` are stored in a binary file called `unencryptedBitstream.rbf`.

Following exactly the steps given in Algorithm 3.5.2 leads to an unencrypted RBF file that can be used to directly configure any other EP2S15F484C5N Altera Stratix II FPGA. The hardware design can thus be cloned. Furthermore, the unencrypted raw bitstream can be modified by an attacker. Then, he is theoretically able to insert a hardware trojan into the design. However, this requires to reverse-engineer the bitstream itself, since no documents are publicly available that describe the meaning of the corresponding bitstream bits. Altera keeps this information confidential, claiming that it is very difficult to reverse-engineer the bitstream.

4 Reverse-Engineering – Design Security Scheme of Stratix III

This chapter contains further not publicly available details of Stratix III FPGAs. The Stratix III series is the third generation of Altera FPGAs. According to [Corb], they come with 65-nm process (note that Stratix II FPGAs are based on a 90-nm process) and have a lower static and dynamic power consumption than Stratix II FPGAs. Besides a higher performance, several hardware features are included, e.g., Digital Signal Processing (DSP) blocks or high speed memory interfaces. According to Table 1 of [Cora], Stratix III FPGAs additionally support a volatile design security feature, where a volatile key is re-programmable and erasable. Moreover, the included AES encryptor operates on 256-bit instead of 128-bit. If these devices also do not include any countermeasures with respect to side-channel attacks, the increased bit size *does not* add to the protection level. Again, we are facing a black-box scenario, have no further knowledge on the inner workings of the key derivation scheme, the utilized block mode, or the IVs. Because of that, we had to reverse-engineer the Quartus II application and pick a new target device, namely an EPS3SE50F484C2 FPGA.

The remainder of this chapter is organized as follows: First we compare the RBF file structures for the unencrypted and encrypted RBF file. Next, we focus on revealing the new key derivation scheme and provide an example of how to compute the 256-bit AES real key for a given 256-bit KEY1 and KEY2. Furthermore, we reveal the utilized AES encryption mode. Finally, we show how one can dump IVs directly from the IDA Pro debugger.

4.1 RBF File Structures

Analyzing the unencrypted and encrypted RBF file (the same way as done for an Stratix II FPGA in Sect. 3.2) we found that the file format is basically the same: The file header structure of an Stratix III RBF file is identical to that for Stratix II. Again, an IV and its according bits are distributed (and stored) over the **Coded Header** by applying the same mapping rules as it is the case for Stratix II, cf. Table 3.3. The same CRC-16 computation is applied on the **Coded Header** bytes to check the integrity.

The file body of the Stratix III RBF file is similar to that of an Stratix II RBF file, except the utilized amount of bytes for the **Fixed Bodypart**, the **Unencrypted Bitstream**, and the **Encrypted Bitstream**. While the **Fixed Bodypart** of Stratix III stores 53,598 bytes, the Stratix II **Fixed Bodypart** only includes 21,050 bytes. The unencrypted bitstream of Stratix III consists of 3,182,823 bytes, while Stratix II only stores 569,068 bytes. The corresponding file format is depicted in Fig. 4.1.

unencrypted.rbf	encrypted.rbf	
Fixed Pre-Header	Fixed Pre-Header	File Header
33 Bytes	33 Bytes	
Coded Header with IV=0xFF..FF(64 Bit)	Coded Header with Random IV (64 Bit)	
40 Bytes	40 Bytes	
CRC16 Modbus over Coded Header	CRC16 Modbus over Coded Header	
2 Bytes	2 Bytes	File Body
Fixed Bodypart	Fixed Bodypart	
53,598 Bytes	53,598 Bytes	
Unencrypted Bitstream	Encrypted Bitstream	
3,182,823 Bytes	3,182,840 Bytes	

Figure 4.1: Structure of an unencrypted and an encrypted Stratix III RBF file.

4.2 AES 256-Bit Key Derivation

We further analysed the Quartus II application using IDA Pro and revealed that the key derivation scheme of Stratix III FPGAs is quite similar to that of Stratix II FPGAs. When passing two 256-bit input values for KEY1 and KEY2 (while observing what happens using the debugger), it turned out that the Quartus application first schedules KEY1 (fourteen round keys are derived). This means that KEY1 represents the AES key and KEY2 is the AES input. Since the AES engine operates on 128-bit input/output blocks, the Quartus II application 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 can be defined as follows:

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

where KEY1 and KEY2 are again those specified in the Quartus II application and where the || symbol means concatenation. As described above, the output block length is 128-bit, and thus, the real key is finally (due to the concatenation) 256 bit long.

The real key serves as the cryptographic key for the actual encryption of all bitstream

blocks. Fig. 4.2 illustrates the implementation of the secret key derivation scheme in the Quartus application. However, this implies that the FPGA has to derive the real key in the same manner when KEY1 and KEY2 are programmed by the user.

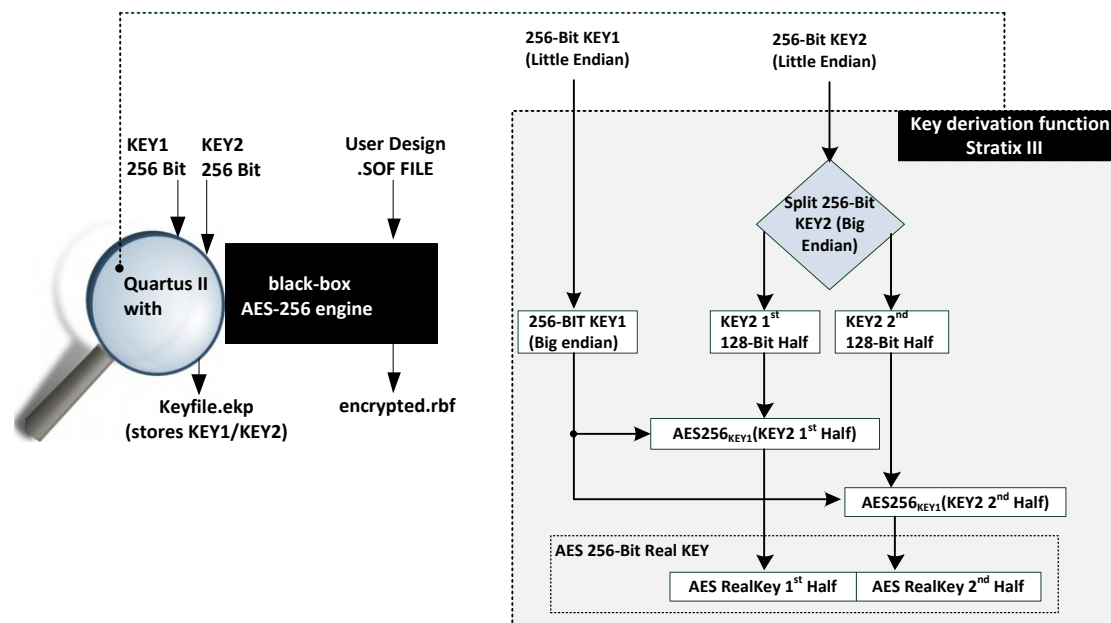


Figure 4.2: Quartus II black-box generating the Stratix III 256-bit real key.

4.2.1 Worked Example

Using the extracted information of the key derivation scheme, we exemplarily provide the inputs and outputs for a chosen KEY1 and KEY2.

KEY1 (Quartus input, little endian)

0x10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27
28 29 2A 2B 2C 2D 2E 2F

KEY2 (Quartus input, little endian)

0x30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47
48 49 4A 4B 4C 4D 4E 4F

Those values lead to the following (separated) KEY2 halves (big endian).

KEY2

1st half (Big endian): 0x4F 4E 4D 4C 4B 4A 49 48 47 46 45 44 43 42 41 40

2nd half (Big endian): 0x3F 3E 3D 3C 3B 3A 39 38 37 36 35 34 33 32 31 30

Encrypting those two 128-bit blocks leads to the following values:

AES256_{KEY1}(KEY2 1st half) (Big endian)

0x57 06 E2 BA C5 F6 67 72 73 0B E8 33 FF 5A 90 C2

AES256_{KEY1}(KEY2 2nd half) (Big endian)

0x7B 6C 4F B9 5D D9 82 A8 98 9B EE CF AA 69 ED 50

Based on these results, we thus obtain the following 256-bit real key:

Real Key (Big endian)

0x57 06 E2 BA C5 F6 67 72 73 0B E8 33 FF 5A 90 C2 7B 6C 4F B9 5D D9 82 A8
98 9B EE CF AA 69 ED 50

Having obtained the real key, we could again perform a known-key attack. Furthermore, by revealing the utilized block mode or the IV structure, we would be able to decrypt the encrypted file. For this purpose, we further disassemble the Quartus II application and study the AES inputs in Sect. 4.3.

4.3 Utilized AES Encryption Modes

Stepping through the program code, we noticed that the first AES input is equal to the initial IV, which is stored in the RBF file header. Observing that the first AES outcome 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 does not possess the shape of an incremented IV. Instead of this, the second AES input consists of random bytes. This indicates that the AES is not used in the classical counter mode.

We performed further experiments, e.g., manipulated the first unencrypted bitstream block (when loaded into the debugger's register) before the block is XORed with the first AES outcome. We noticed that the second AES input stays untouched when using the same windows timestamp (or: initial IV) and 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 to that, we noticed that the second AES input bytes are not influenced by the choice of KEY1 or KEY2, and thus, the second AES encryption input (and all other) is derived independently from the real key. Moreover, we can exclude that the CBC mode is used, because the first input should be then given as $IV \oplus (1^{\text{st}} \text{ bitstream block})$, which is never the case.

We repeated the same encryption with a delay of one second (resulting to the previous initial IV, but incremented by one), and again traced the according AES inputs, leading to the results of Table 4.1. Note that only the first 8-byte halves are shown since the second 8-byte halves are identical.

When comparing the generated AES input sequences, one can observe that the shape is basically the same. E.g., the first three bytes of the second AES input stay unchanged,

Sequence	Values for Monday, 2012-10-15 23:30:51	Values for Monday, 2012-10-15 23:30:52
Initial IV (AES input 1)	0x8B 80 7C 50 10 9D 73 9C	0x8C 80 7C 50 10 9D 73 9C
AES input 2	0x11 90 0F 0C A4 F3 83 7F	0x11 90 0F 02 AA 73 9C 83
AES input 3	0x02 F2 81 83 76 FE F4 2B	0x02 F2 41 42 77 0E 77 34
AES input 4	0x40 3E 70 D4 CA 9F 77 4D	0x40 3E 48 EC CA E1 87 4E
AES input 5	0xC8 07 8E 5A F9 F3 AE 09	0xC8 07 89 5D 39 FC D0 09

Table 4.1: IV sequences for Stratix III encryption using different points in time.

while each other byte slightly differs. Thus, we assume that the generated random numbers are the result of a Linear Feedback Shift Register (LFSR). Coming back to the analysis of the utilized block mode, we figured out that the generated AES outcome serves as stream cipher key.

By simply XORing the first five AES outcomes (traced using the IDA Pro debugger) with the first five corresponding encrypted bitstream blocks (taken from the resulting encrypted RBF file), we successfully decrypted the bitstream blocks by hand. Fig. 4.3 illustrates how the encryption scheme is used by Stratix III FPGAs.

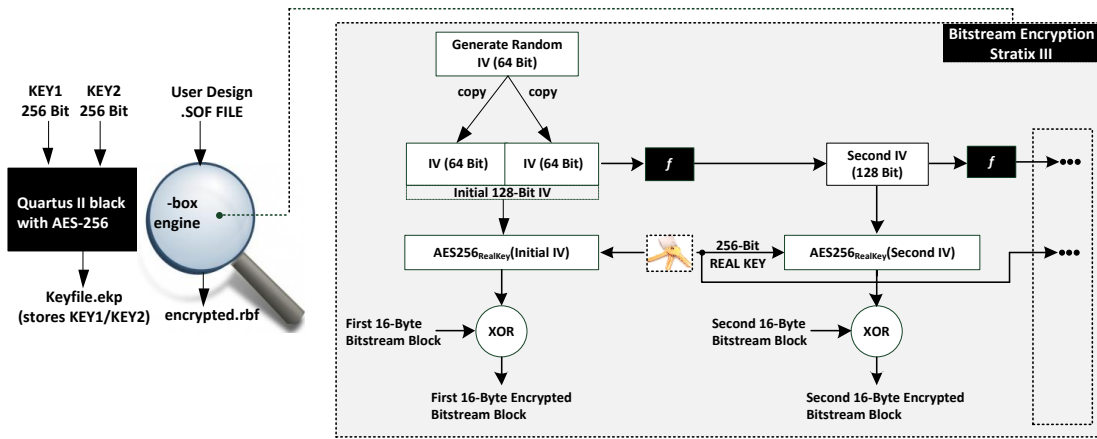


Figure 4.3: Revealed bitstream encryption of Stratix III.

Probably, the idea behind introducing the LFSR is to prevent an attacker (for the case he has already retrieved the real key) from decrypting the encrypted RBF file. An attacker must be able to reconstruct the AES input sequences from the initial IV to be able to compute the respective AES outcomes (stream key). Since the FPGA has to reconstruct these inputs as well as the Quartus II application does, the pseudorandom number function must be deterministic, and thus, Altera again tries to achieve security by obscurity, which does not add to the protection of the system.

Besides this, note that it should be a better solution to simply generate a random number (not based on the timestamp) and then to increment the bytes in the usual way. Then, the probability of getting random numbers of the first design encryption that

overlap to those of the second design encryption stays negligibly small, although both IV sequences are simply incremented. The same applies for the utilized solution of Altera: The probability of overlaps stays negligibly small, but calling a pseudorandom number function each time for generating one random number is less efficient and probably requires (unnecessarily) additional area on the FPGA chip.

Hence, the only advantage of Altera’s modified block mode is that an attacker is forced to spend more time on reverse-engineering the Quartus II application. However, one can even circumvent the reverse-engineering of the application by applying a workaround. As a proof-of-concept, in Section 4.3.1 we further illustrate that it is possible to obtain the AES inputs of any given initial IV with relatively little efforts.

4.3.1 Dumping Initial Vectors with IDA Pro

In this section, we present a workaround of how to dump all AES inputs, which are required to decrypt a bitstream file (once one has revealed the real key). The prerequisite for dumping all AES inputs is that one obtained the encrypted RBF file (e.g., by eavesdropping the communication channel) of an Stratix III FPGA. The initial AES input (initial IV) can be extracted from the RBF file header with the help of Table 3.3 (as we explained in Section 4.1). Knowing the initial IV, one can perform a simple workaround (using the Quartus II application) to obtain all AES inputs. The idea behind this is to load the Quartus II application into the IDA Pro debugger and to dump the AES inputs (setting the same IV) for a random KEY1/KEY2 and a random configuration design using IDA’s scripting language IDC.

IDA’s IDC scripting language provides access to the contents of the IDA Pro environment. E.g., stepping through an assembly code can be automatically performed, or byte values that are located at a specific address in the stack can be displayed in the console or dumped into a file. The manipulation of register values or memory content can be a helpful utility, e.g., for studying the behavior of a function, as IDC allows to analyze the influence of these bytes. For our proof-of-concept, we have written two IDC scripts that we explain in detail in this section. First, we provide an overview of how to exactly dump the AES inputs. The workflow is outlined in Fig. 4.4. The first step is to read out the IV bits according to the mapping rules of Table 3.3. In the next step, we convert the first 32 bit of the IV into a readable windows time format.

I.e., if the IV is given as 0x8B 80 7C 50 10 9D 73 9C, we convert 0x8B 80 7C 50 to Monday, “2012-10-15 23:30:51”. Having figured out the corresponding timestamp, we set and freeze (step 2) the windows clock to the according date (and time) using Listing 3.1. In the third step, we start IDA Pro, load the Quartus executable, and start the debugger. The fourth step is to open the design security tool, setup any random KEY1/KEY2 and specify any configuration design (but for the same FPGA fabric).

In the meanwhile, the IDA Pro debugger loads the `pgm_pgmio_aes_nv.dll` segment that is now accessible in the IDA Pro environment. Next, we can execute the IDC script

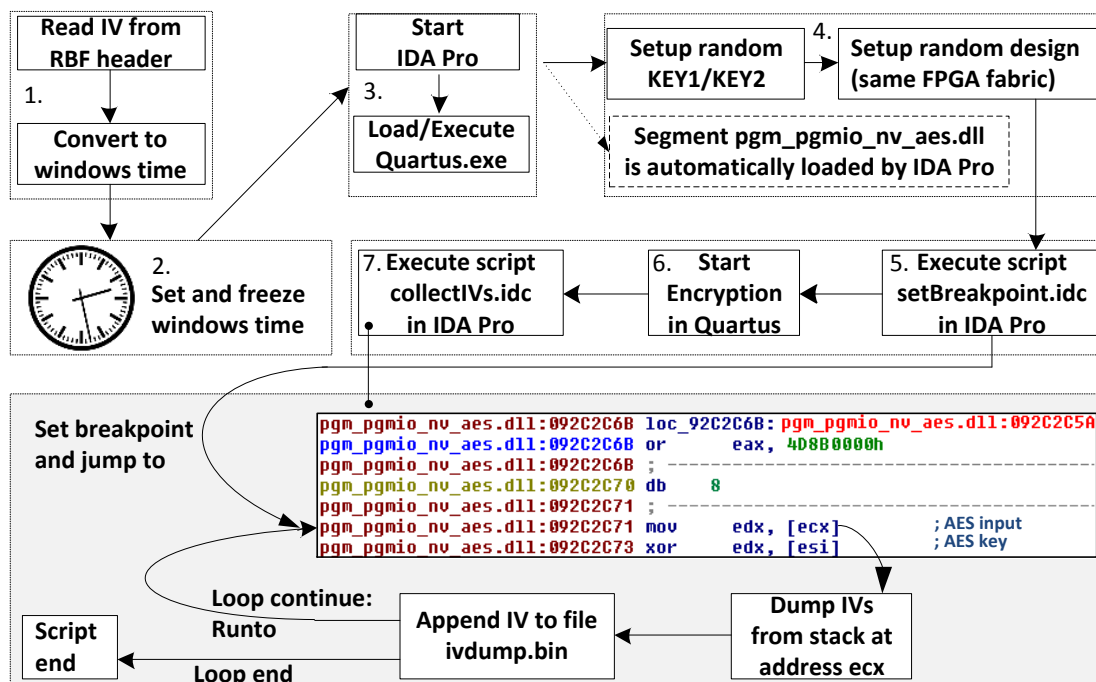


Figure 4.4: Dumping IVs with IDA Pro for any initial vector.

given in Listing 4.1. It is used for setting a breakpoint at a specific assembly instruction (here: instruction that copies AES input bytes to register `edx`) that comes with the `pgm pgmio nv aes.dll` segment.

Since the segments of a DLL file are loaded dynamically into the IDA memory, the address of an DLL assembly instruction can vary when repeatedly loading the Quartus executable into IDA Pro. This implies that one cannot provide a fixed address in order to target the same instruction using the same breakpoint. Instead of this, one can determine the (dynamic) address of the `pgm_pgmio_nv_aes.dll` segment and add the corresponding instruction offset (which is fixed) for being able to target the correct assembly instruction automatically.

In Listing 4.1, we first include an `idc.idc` file that provides fundamental access to certain pre-defined functions. E.g., the function `FirstSeg()` returns the start address of the first segment loaded by the IDA Pro debugger.

Listing 4.1: Setting a breakpoint using IDC scripting.

```
1 #include <idc.idc>
2 static main() {
3     auto x, address, offset2AESin, segname;
4     segname      = "pgm_pgmmio_nv_aes.dll";
5     offset2AESin = 0x2C71;
6 }
```

```

7   for(address = FirstSeg(); address != BADADDR; address = NextSeg(address))
8       if (segname == SegName(address)) break;
9   x = address + offset2AESin; // Compute address of AES input
10  AddBpt(x); Jump(x);        // Add breakpoint and move cursor to address x
11 }

```

Next, we define several variables (`x`, `segname`, `offset2AESin`, and `address`). At code line 4, we specify the segment's name that matches in our case with the string `pgm_pgmio_nv_aes.dll`. The variable `offset2AESin` specifies the offset `0x2C71` that has to be added to the segment's start address to target the assembly instruction responsible for copying the AES inputs to the `edx` register. This offset has to be determined only once. The for-loop of Listing 4.1 determines the start address of the `pgm_pgmio_nv_aes.dll` segment.

Once the for-loop has obtained this address, we add the corresponding offset and store the result in the variable `x` at code line 9. In the last step, we set a breakpoint at the previously determined address `x` (using `AddBpt(x)`) and then jump to the instruction located at address `x`. The breakpoint is set (cf. Fig. 4.4) to the following assembly instruction:

pgm_pgmio_nv_aes.dll:092C2C71 mov edx, [ecx] ; AES input

The `mov` command copies the first four bytes of the stack that are located at address `ecx` (where sixteen AES input bytes are stored) to the 4-byte register `edx`. By executing the IDC script of Listing 4.1, we set a breakpoint that is only reached when we start the encryption process. Because of that, the next step (6.) is to start the encryption in the Quartus application. By doing so, all assembly instructions are executed until the above assembly instruction is reached. At this point in time, one can observe that the 1st KEY2 half (cf. Sect. 4.2.1) is located in the stack at address `ecx`, which is the first AES input. The 256-bit KEY1 (AES key) can be observed in the stack at address `esi` due to the assembly code line that is given in the following (cf. Fig. 4.4):

pgm_pgmio_nv_aes.dll:092C2C73 xor edx, [esi] ; AES key

The key bytes are directly XORed with the previously loaded AES input bytes. The corresponding result is re-written to the `edx` register. By pressing **F9**, the debugger continues the execution of the assembly code and again stops at the same assembly instruction since the application runs in a loop. Again, KEY1 can be observed in the stack at address `esi`, while now the 2nd half of KEY2 (cf. Sect. 4.2.1) is at address `ecx`. Pressing again **F9**, the first IV can be inspected in the stack (located at address `ecx`), while one can observe the real key at address `esi`. However, we avoid to press **F9** (which is equal to the execution of one loop iteration). Instead of this, we execute the IDC script that is given in Listing 4.2 directly after starting the encryption.

Listing 4.2: Dumping AES inputs using IDC scripting.

```

1 #include <idc.idc>
2 static main() {
3     auto fp, filename; // File variables
4     auto x, offset2AESin, segname; // Segment variables
5     auto loop, ivs2dump; // Loop variables
6     auto firsthalf, secondhalf; // Variables for storing 4-byte halves
7     auto b0,b1,b2,b3,b4,b5,b6,b7; // Variables for byte0-byte7
8     auto address; // Address variable
9
10    segname = "pgm_pgmio_nv_aes.dll"; // Specify segment name
11    filename = "IVdump.bin"; // Filename for AES sequences storage
12    offset2AESin = 0x2C71; // Provide segment offset to AES input
13
14    // Derive exact position of LineCursor to stop at
15    for(address = FirstSeg(); address != BADADDR; address = NextSeg(address)){
16        if (segname == SegName(address)) break;
17    }
18    x = address + offset2AESin;
19
20    // Delete breakpoint at address x (AES Input) and execute all assembly instructions until
21    // address x is reached
22    DelBpt(x);RunTo(x);
23
24    // Ensure that the stepping is throttled, else incorrect values might be stored in the file
25    GetDebuggerEvent(WFNE_SUSP, 1); Wait();
26
27    // Open binary file
28    fp=fopen(filename,"wb");
29    if(fp==0) {Message("ERROR: File could not be created!\n"); return;}
30
31    ivs2dump = 198.927; // How many IVs (AES inputs) should be dumped?
32    // For each AES input
33    for (loop=1; loop <= encryptions; loop++) {
34        // Execute all assembly instructions until address x is reached
35        RunTo(x);
36
37        // Ensure that the stepping is throttled, else incorrect values might be stored in the file
38        GetDebuggerEvent(WFNE_SUSP, 1); Wait();
39
40        // Obtain first 4 bytes that are located at address ecx, ecx+1, ecx+2, ecx+3
41        b0=Byte(ECX); b1=Byte(ECX+1); b2=Byte(ECX+2); b3=Byte(ECX+3);
42        firsthalf = ((b0 & 0xFF) << 24 | ((b1 & 0xFF) << 16) | ((b2 & 0xFF) << 8) | ((b3 & 0xFF));
43
44        // Obtain next 4 bytes that are located at address ecx+4, ecx+5, ecx+6, ecx+7
45        b4=Byte(ECX+4); b5=Byte(ECX+5); b6=Byte(ECX+6); b7=Byte(ECX+7);
46        secondhalf = ((b4 & 0xFF) << 24 | ((b5 & 0xFF) << 16) | ((b6 & 0xFF) << 8) | ((b7 & 0xFF));
47
48        writelong(fp, firsthalf ,1); // write first 4 bytes
49        writelong(fp, secondhalf ,1); // write second 4 bytes
50        writelong(fp, firsthalf ,1); // write third 4 bytes
51        writelong(fp, secondhalf ,1); // write fourth 4 bytes
52    }
53    // Close file pointer of binary file
54    fclose(fp);
55 }

```

First, we define several variables at code lines 3-8. Then, at code lines 15-18, we again compute the address of the assembly instruction that copies the first four AES bytes. Code line 21 is responsible for deleting the previously set breakpoint (set in Listing 4.1)

using the `DelBpt(x)` command. In addition to that, we invoke the `RunTo(x)` command that executes all assembly commands until address `x` is reached. In more detail, after the execution of code line 21, one can dump the 2nd half of KEY2 (AES input for the key derivation). This also implies that if the script again executes the `RunTo(x)` command, the first IV can be dumped.

At code line 24 (and 37), we throttle the execution speed of the debugger using the `GetDebuggerEvent()` and `Wait()` function, otherwise, it might be the case that the debugging process is not synchronized, leading to incorrect values. Then, at code line 27f., we create a binary file called `IVdump.bin`. At code line 30, we specify the number of AES inputs that should be dumped, e.g., we want to dump all 198,927 IVs.

Finally, a for-loop (at code lines 32-51) is executed 198,927 times. In the beginning of the for-loop, we execute the `RunTo(x)` command, and thus, the sixteen bytes of one IV block can be dumped (and due to the loop all following 16-byte IV blocks) by executing code lines 40-45. Note that the `Byte(ecx)` function returns the byte value located on the stack at address `ecx`. The first four bytes of one IV are stored in a variable called `firsthalf`, while the next four bytes are stored in a variable labeled as `secondhalf`. In the last step of the for-loop, we append the sixteen IV bytes to the `IVdump.bin` file using the `writelong()` function. The for-loop is also sketched as a gray box in Fig. 4.4.

Following the steps mentioned above, one can dump all AES inputs in approximately three hours, which is not critical for a practical attack, because this process does not require any physical access to the FPGA. To further simplify the attack, another approach is to determine the position in the stack where the initial 64-bit IV sequence is stored and to directly modify those bytes so that it is not necessary to set and freeze the windows time anymore.

This type of attack is another example that security by obscurity does not add to the protection level of the system, since one can use the Quartus application itself as an oracle that provides all required AES inputs. Note that once an attacker has obtained the real key and the AES inputs are given, it is straightforward to decrypt (cf. Listing A.2.6 in the appendix) the encrypted RBF file, and thus to clone the configuration design. With further reverse-engineering efforts, one could also disassemble the random number function that is applied for deriving the AES inputs. Once one has figured out how the random number function is implemented, the decryption process of an RBF file should take a few seconds only.

5 Side-Channel Profiling of Stratix II

With the knowledge of the bitstream encryption process presented in Chapter 3, we are able to analyze the Stratix II from a side-channel point of view. To this end, in this chapter we first describe the measurement setup and scenario. Then, as a prerequisite to the according key extraction attack (Section 6), 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*.

5.1 Measurement Setup

Our DUT, a Stratix II FPGA (EP2S15F484C5N), is soldered onto a SASEBO-B board [AIS08] specifically designed for SCA purposes. The SASEBO-B board provides a JTAG port that allows one-time programming 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 3.3.1.

We directly configure the DUT using the passive serial mode. For this purpose, we built an adapter that is conformant to [Str07, p.599]. We developed a custom programmer based on an ATmega256 μ C. 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 μ C 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 [Str07, 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 Fig. 5.1 and 5.2. 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 μ C. The measurement process was triggered using a dedicated μ C pin providing a rising edge shortly before the first bitstream block is sent.

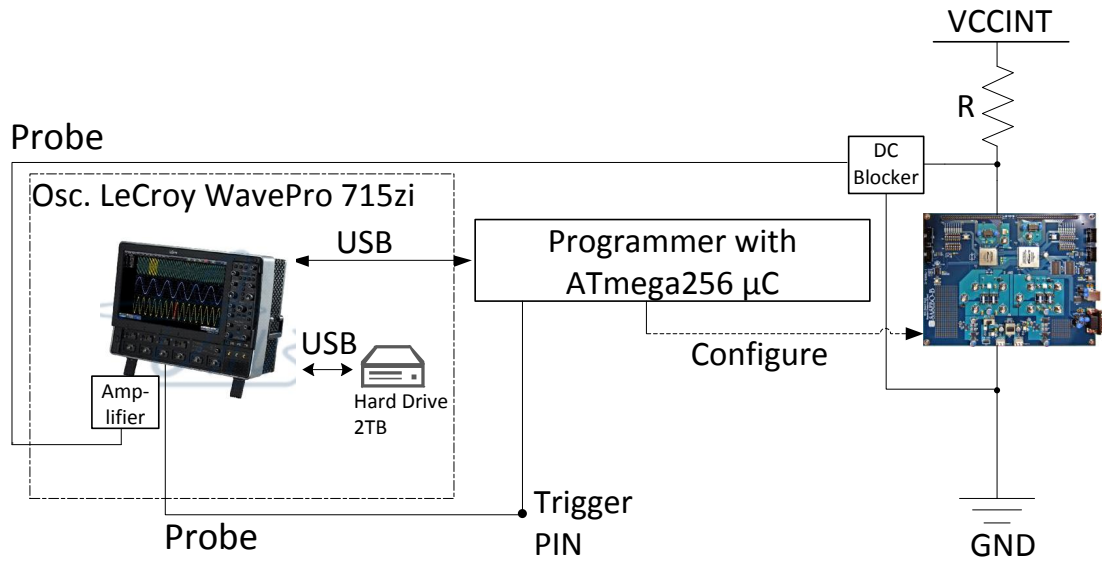


Figure 5.1: Measurement setup for SCA schematic.

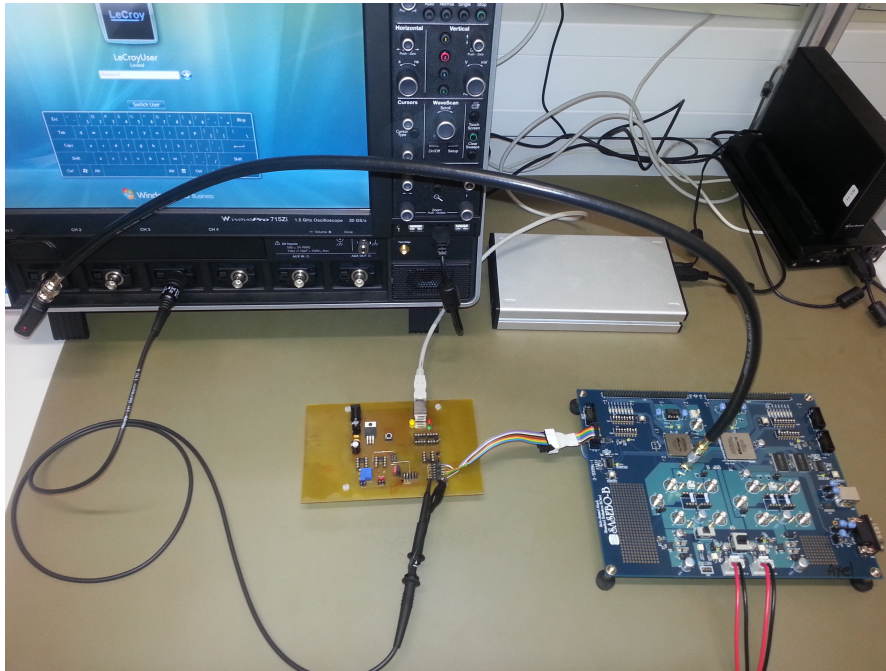


Figure 5.2: Measurement setup for SCA in the real world.

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 recorded. The corresponding steps are described in more detail in Algorithm 5.1.1.

Algorithm 5.1.1: Measurement Setup

```

1: for i=1 to numberOfTraces do
2:   [ $\mu$ C] Perform DUT reset
3:   [ $\mu$ C] Transfer Fixed Pre-Header (33-bytes) to DUT
4:   [PC] myIV[0..7]  $\leftarrow$  rand
5:   [PC] myHeader[]  $\leftarrow$  Get Coded Header from RBF file
6:   [PC] Code myIV[] into myHeader[] (Table 3.3)
7:   [PC] Compute CRC-16 over Coded Header
8:   [PC] Send Coded Header with CRC-16 (42 bytes) to  $\mu$ C
9:   [ $\mu$ C] Transfer Coded Header (42 bytes) to DUT
10:  [ $\mu$ C] Transfer Fixed Bodypart (21,050 bytes) to DUT
11:  [PC] Bitstream[0..47]  $\leftarrow$  rand
12:  [PC] Send Bitstream[] (48 bytes) to  $\mu$ C
13:  [ $\mu$ C] Set trigger. Transfer bitstream (48 bytes) to DUT
14:  [DSO] Record power trace of the DUT
15:  [PC] Store trace  $i$ 
16:  [PC] Store myIV[]
17: end for

```

For each power trace that the DSO records, the μ C first brings the DUT into reset mode and then initiates a new configuration process. At code line 3, the μ C transfers the **Fixed Pre-Header** bytes directly to the DUT. These bytes are stored in the ATmega256 flash to avoid Universal Asynchronous Receiver Transmitter (UART) communication between the PC and μ C, because exchanging data in this way significantly slows down the whole measurement process. Thus, we also store all 21,050 **Fixed Bodypart** bytes in the flash of the μ C. At code line 4, we generate a random 64-bit IV sequence (random AES challenges are a prerequisite for power analysis that is then coded into the **Coded Header** of an encrypted RBF file (at code lines 5-6).

To ensure the validity of the modified **Coded Header**, the PC program updates both CRC-16 header bytes accordingly (at code line 7), and finally transfers the complete (and valid) **Coded Header** – carrying the previously generated IV – to the μ C (at code line 8), which then forwards the **Coded Header** bytes to the DUT (at code line 9). In the next step, all 21,050 **Fixed Bodypart** bytes are directly transferred from the ATmega256 flash to the DUT. At code line 11, the PC creates 48 random bitstream bytes that are sent to the μ C (at code line 12). The μ C first provides a trigger signal for the DSO and then immediately starts to transfer all 48 bitstream bytes to the DUT (at code line 13). The DSO detects the rising edge of the provided trigger, and thus, records the power consumption for the correct point in time (at code line 14). In the last step of the for-loop (code lines 15-16), the PC stores the previously recorded power trace and the utilized 64-bit IV sequence.

5.2 Extending the Sasebo-B Board for PS Support

The Sasebo-B board comes with two ports enabling two configuration modes for the DUT. The first port provides support for the JTAG protocol, while the other port (labeled as **EEP**) is designed for the Active Serial (AS) mode. The AS protocol is a mode, in which the DUT provides the clock during configuration. The right side of Fig. 5.3 (labeled as “Sasebo B-Board”) illustrates the corresponding wiring of the Stratix II pins: e.g., **DCLK** is wired with pin 1 and **nCONFIG** is wired with pin 5. These pins are thus connected with the EEP port of the Sasebo-B board.

This wiring is fixed on the Printed Circuit Board (PCB) and cannot be changed. Since we need the PS mode (FPGA clock provided by the configuration device) instead of the AS mode, the pin mapping has to be changed with the help of a custom adapter. We built an adapter that remaps several pins as depicted in the middle part (labeled as “Adapter”) of Fig. 5.3. Plugging a cable into the adapter (ATmega256 μ C \rightarrow Adapter) and switching the dip switches (not drawn in Fig. 5.3) from **MSEL[0:3]=1011** to **MSEL[0:3]=0100** enables the passive serial mode. The **MSEL** signals tell the DUT which configuration mode (cf. Table 3.2) is used.

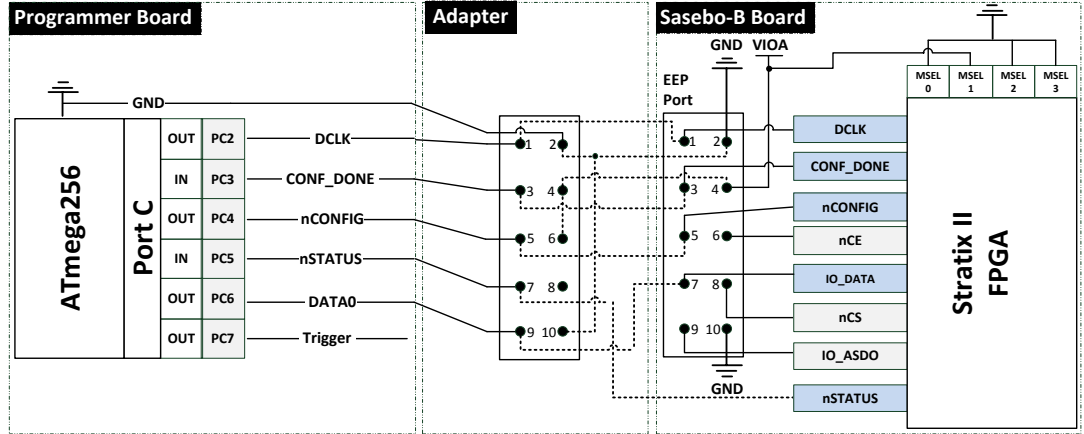


Figure 5.3: Passive serial adapter wiring.

In addition to Fig. 5.3, Table 5.1 gives an overview of the signals that were originally used for the AS mode and additionally shows which PS signals (left part of Fig. 5.3 labeled as “Programmer Board”) are provided by our ATmega256 μ C. Two pins (PC3 and PC5) of port C of the ATmega256 μ C are set as input signals (**CONF_DONE** and **nSTATUS**) for receiving information from the DUT, while four other pins are configured as output for transferring data to the DUT (PC2, PC4, and PC6) and providing the measurement trigger (PC7). The concrete meaning of these signals is explained later in this section.

Active Serial			Passive Serial			
Programmer Signal	EEP Pin	Board/Pin	Programmer Signal	Adapter Pin	EEP PIN	Board/FPGA Pin
DCLK	1	DCLK (D19)	DCLK (PC2)	1		DCLK (D19)
GND	2	GND	GND	2		GND
CONF_DONE	3	CONF_DONE (C20)	CONF_DONE (PC3)	3		CONF_DONE (C20)
–	4	VIOA	–	4		VIOA
nCONFIG	5	nCONFIG (W18)	nCONFIG (PC4)	5		nCONFIG (W18)
nCE	6	nCE (A21)	–	6: directly wired with 4	–	nCE (A21)
IO_DATA	7	IO_DATA (E13)	nSTATUS (PC5)	7: directly wired with nSTATUS	–	nSTATUS (B20)
nCS	8	nCS (D11)	–	not connected	8	nCS (D11)
IO_ASDO	9	IO_ASDO (G12)	DATA0 (PC6)	9	7	IO_DATA (E13)
GND	10	–	GND	10: directly wired with 2	–	–

Table 5.1: Mapping signals for AS and PS support.

As one can see in Table 5.1, the PS mode works with two signals less than AS. The first five pins (DCLK, GND, CONF_DONE, VIOA, and nCONFIG) are identical for both configuration modes and hence, the adapter pins 1–5 are simply connected to the respective EEP pins 1–5. When using the AS configuration mode, the DCLK pin of the DUT is used as output pin. While using the PS mode, the DCLK pin is configured as an input pin. In the active serial mode, the EEP pin 6 is connected to a pin called nCE. In contrast, when using the PS mode, the adapter pin 6 is connected to the VCIO voltage supply.

Due to the adapter wiring, the sixth EEP pin (nCE) is not connected with any signal and hence unused since it is not relevant in the PS mode. The adapter pin 7 (with nSTATUS signal) is wired directly with the nSTATUS pin of the DUT (for PS only), while adapter pin 8 is not connected with any pin of the DUT. The EEP pin 8 (that is connected to the nCS pin) is ignored for the same reason due to which the nCE pin is not taken into account. The adapter pin 9 (with signal DATA0) is wired to the EEP pin 7 and therefore connected to the IO_DATA pin of the DUT. The last pin is connected to the ground path, as demanded in the configuration handbook of Altera. Using the described adapter, we now focus on the signals and the workflow of the PS scheme.

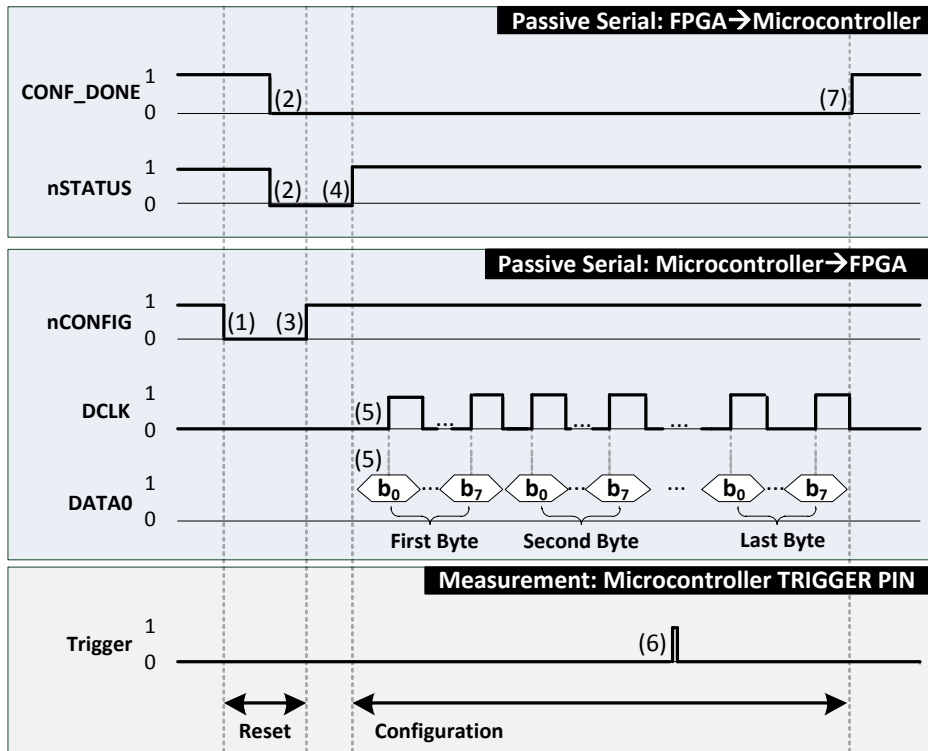


Figure 5.4: Passive serial protocol.

Fig. 5.4 illustrates the according protocol. It is divided into two main parts. The first part deals with bringing the DUT into the *reset* state, while the second part is responsible for the actual *configuration* of the bitstream. Note that the whole protocol is also used for encrypted bitstreams and that there are no differences, except the data that

has to be send to the DUT. The protocol starts by pulling the `nCONFIG` signal low (1), because this signal is used for resetting the DUT. The DUT recognizes the low level of the `nCONFIG` signal and reacts setting its output signals (`CONF_DONE` and `nSTATUS`) low (2). A low level of the `CONF_DONE` signal indicates that the configuration of the DUT is not complete, while a low voltage level of the `nSTATUS` signal tells the μC that the DUT is preparing for receiving the bitstream.

During this time, the μC pulls the `nCONFIG` signal high and ends the reset state (3). When the DUT is ready for receiving the bitstream, it pulls the `nSTATUS` signal high (4). From that point in time, the μC is able to send the actual bitstream. The bytes of the bitstream file are transferred bitwise with the Least Significant Bit (LSB) first (5). In more detail, the value $0x82_{(16)} = 1000010_{(2)}$ is transferred in the order $0 \rightarrow 1 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 1$. The μC clocks the data into the DUT by outputting the current data bit over the `DATA0` signal on a rising edge of the `DCLK` signal.

We pull the trigger pin high and instantly down to a low level (6). This is done shortly before the first encrypted bitstream byte is being sent to the DUT. The oscilloscope is then able to trigger at the correct point in time, for which we measure the power consumption of the DUT. Once the last bitstream block has been transferred, the DUT informs the μC that the process of configuration is finished setting the `CONF_DONE` signal high.

5.3 Difference between Unencrypted and Encrypted Bitstream

Using our measurement script, we recorded 10,000 power traces to locate differences in the power consumption for the time range that includes the transmission of 48 fixed, encrypted bitstream bytes. The FPGA decryption unit 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 5.5 illustrates the corresponding mean traces.

As visible in Figure 5.5, 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.

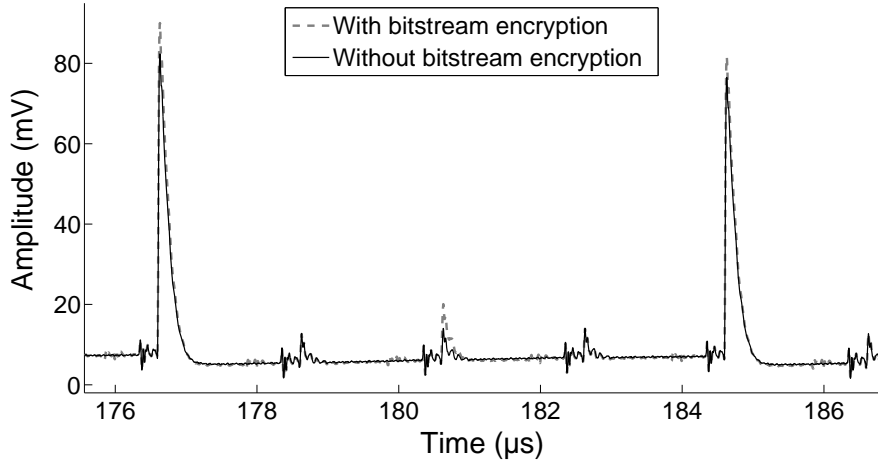
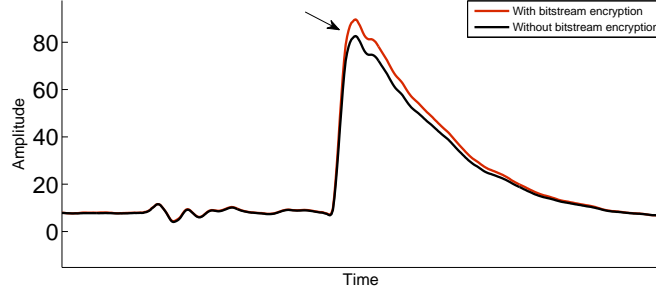
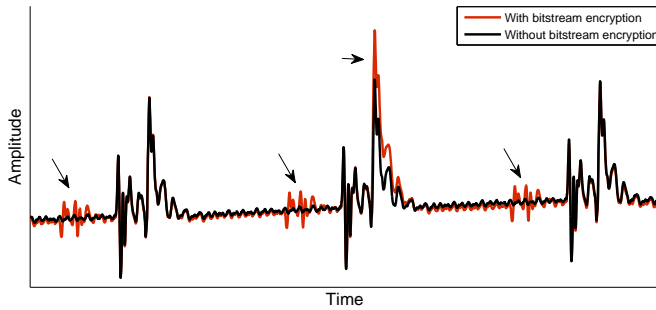


Figure 5.5: Average power consumption (10k traces) while sending an unencrypted (solid) and an encrypted (dashed) bitstream. Zoom on one byte.

Fig. 5.6a shows a zoom on the first bit of Fig. 5.5. As one can see, the higher (dashed, red) power trace (belonging to the encrypted bitstream) consumes more energy compared unencrypted bitstream (solid, black). Next, in Fig. 5.6b, the arrows indicate the points of differences during the transfer of bit 1-3. It seems that the additional three peaks (given in Fig. 5.6b), belong to the processing of the AES encryption, as they do not appear in the unencrypted version.



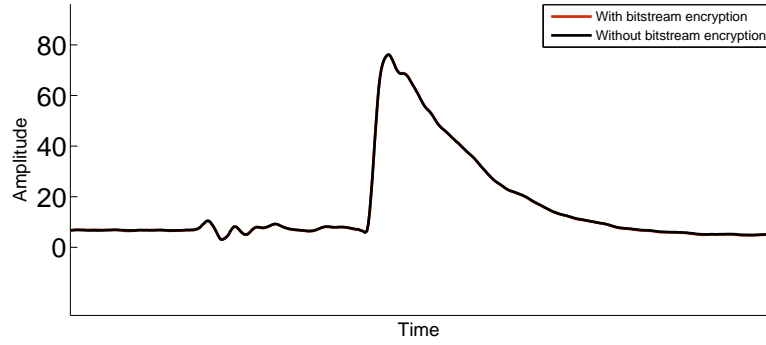
(a) While sending the first bit.



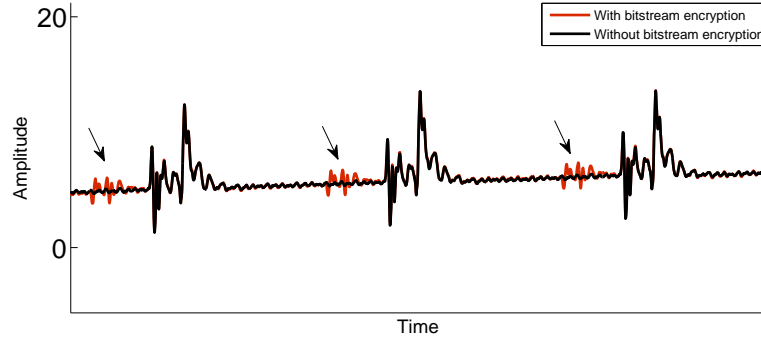
(b) While sending bit 1-3.

Figure 5.6: Zoom on average power consumption (10k trace) while sending the first half of one bitstream byte.

Similarly, Fig. 5.7a and Fig. 5.7b illustrate the average power consumption while sending the second half of a byte. It turns out that there is no relevant difference given for certain bits (here: bit 4 of Fig. 5.7a). The reason for this might be due to the usage of an own clock for the cryptographic engine, and hence, the engine might process data independently of the provided clock (DCLK) of the μ C. This may explain that there is no difference given for certain points in time (AES unit idles) while the μ C continues to transfer the data to the DUT.



(a) While sending the fourth bit.



(b) While sending bit 5-7.

Figure 5.7: Zoom on average power consumption (10k trace) while sending the second half of one bitstream byte.

To illustrate further differences, Fig. 5.8 shows (once again) the average power consumption for both scenarios, but now, the transfer of the first sixteen bitstream bytes (instead of one byte) are drawn. In addition to that, the dashed (and red) vertical lines mark the points in time (for the first five bytes) where a difference occurs.

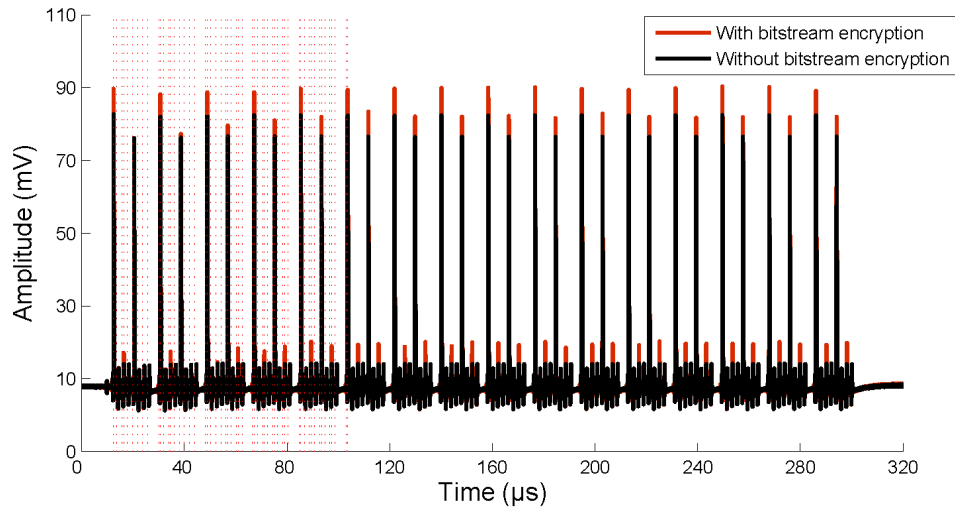


Figure 5.8: Difference between unencrypted and encrypted bitstream while sending 16 bytes to the DUT.

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 as indicated by Fig. 5.9.

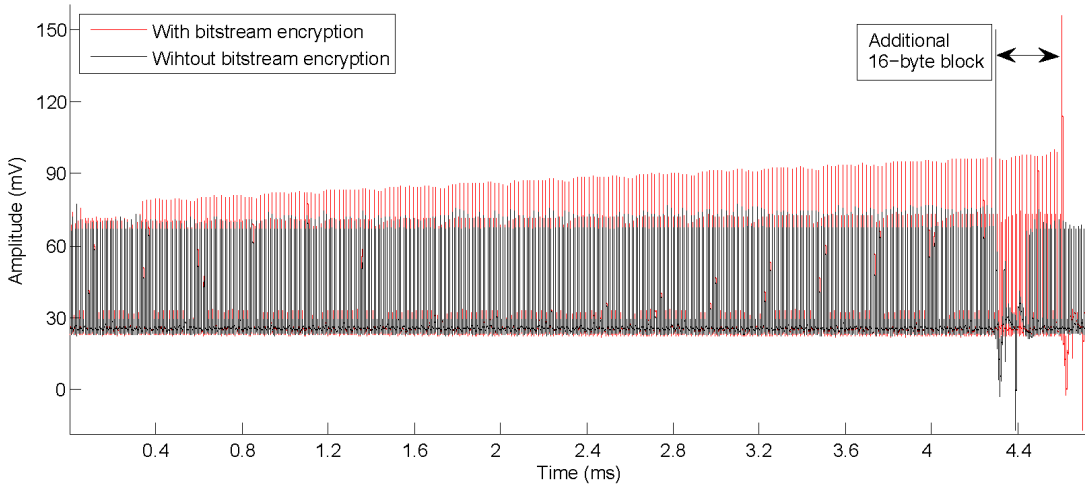


Figure 5.9: Difference between unencrypted and encrypted bitstream while sending seventeen 16-byte blocks to the DUT.

We found that while sending seventeen unencrypted 16-byte bitstream blocks (black curve with lower power consumption), the power consumption strongly increases at ~ 4.3 ms (for a very short time interval) from ~ 90 mV to ~ 150 mV. Sending seventeen encrypted 16-byte blocks leads to the same behavior, but the peak is approximately

delayed by the time that is usually required to transfer one 16-byte block. This indicates that the configuration of the unencrypted bitstream is performed shortly after the data transfer, while the configuration process of the encrypted bitstream is delayed due to the decryption.

5.4 Correlation Power Analysis

For a Correlation Power Analysis (CPA), [BCO04], we use Pearson's correlation coefficient and several prediction models in order to extract the real key of the DUT.

5.4.1 Pearson's Correlation Coefficient

In the side-channel community, it is common to use the Pearson correlation coefficient as a measure for the dependency between two data sets (e.g., X and Y) that yields a value in the interval $[-1, 1]$. A correlation value of -1 represents a perfect negative linear dependency between X and Y , while $+1$ stands for a perfect positive linear dependency. To distinguish key guesses (e.g., for a key byte of an S-box), one determines the dependency of power traces (X) to predicted (key dependent) intermediate values (Y) of the utilized algorithm.

A correlation value very close to zero indicates that there is no dependency given. This is, e.g., the case for wrong key (or: state) guesses, since it implies predicted intermediate values that do not match with the processed intermediate values of the DUT. Hence, no data dependency is given with the measured power consumption.

In the following, we refer to a power trace that stores Q sampled data points as $\vec{t}(q)$, where q targets one sampled data point ($q \in \{1, \dots, Q\}$). In addition to that, we use the notation $\vec{t}_{x_i}(q)$ for the i 'th ($i \in \{1, \dots, N\}$) power trace $\vec{t}(q)$, for which the i 'th 16-byte (AES) challenge $\vec{x}_i(u)$ (with $u \in \{0, \dots, 15\}$) is used as 16-byte input block of a DUT. The correlation coefficient is computed for each data point q (over N traces), yielding a correlation trace $\Delta_{CPA}(K_p, q)$ for each key candidate guess ($K_p := p$) $\in \{0, \dots, P-1\}$, where P refers to the number of possible key guesses. The set $\{0, \dots, P-1\}$, e.g., applies for guessing one key byte ($P = 2^8$) of one AES S-box or, to give another example, for guessing $P = 2^{16}$ key candidates (two AES key bytes at once). Note that the range of key values might differ for different algorithms. Consider Equation 5.1 with the intermediate values $\vec{x}_i(u)$, the subkey candidate K_p , and the resulting intermediate value v_{ip} .

$$v_{ip} := f(\vec{x}_i(u), K_p) \quad (5.1)$$

v_{ip} is then the result of a function f that might be, for instance, the resulting S-box output of one key byte guess K_p and the first challenge byte $\vec{x}_i(u=0)$ such that one obtains the intermediate value given in Equation 5.2.

$$v_{ip} = f'(\vec{x}_i(u), K_p) = \text{SBOX}(\vec{x}_i(0) \oplus K_p) \quad (5.2)$$

Another commonly targeted intermediate value is presented in Equation 5.3.

$$v_{ip} = \tilde{f}(\vec{x}_i(u), K_p) = (\vec{x}_i(0) \oplus K_p) \oplus \text{SBOX}(\vec{x}_i(0) \oplus K_p) \quad (5.3)$$

The intermediate value v_{ip} is mapped to a value h_{ip} by a function g . We refer to the result as a prediction (see Equation 5.4).

$$h_{ip} := g(v_{ip}) \quad (5.4)$$

Commonly, one uses the Hamming Weight (HW) for g . The HW counts how many "1" bits are set within a binary string. E.g., the HW of the binary string $00101100_{(2)}$ is three. This model assumes that the more bits are set (e.g., in a register), the more energy is consumed by a DUT. This means a HW of zero is equal to a low power consumption (during the processing of the corresponding value), while a HW value of "eight" represents a high power consumption. In the case of computing the HW of $f'(\vec{x}_i(u), K_p)$, we predict that a DUT performs a bitwise processing of the AES S-box and stores the result into an 8-bit register. It is also very common to use the Hamming Distance (HD) model. This prediction model counts the number of toggling bits between two binary strings, e.g., $\text{HD}(01000001_{(2)}, 01000010_{(2)})$ yields the value "two", because bit_0 toggles from 1 to 0 and bit_1 toggles from 0 to 1. The number of toggling bits can be determined with Equation 5.5.

$$\text{HD}(A, B) := \text{HW}(A \oplus B) \quad (5.5)$$

Similar to the HW computation of Equation 5.2, computing the HW of $\tilde{f}(\vec{x}_i(u), K_p)$ is identical to the computation of the HD of $(\vec{x}_i(0) \oplus K_p)$ and $\text{SBOX}(\vec{x}_i(0) \oplus K_p)$ as described by Equation 5.6– 5.8.

$$h_{ip} = \text{HW}(\tilde{f}(\vec{x}_i(u), K_p)) \quad (5.6)$$

$$= \text{HW}((\vec{x}_i(0) \oplus K_p) \oplus \text{SBOX}(\vec{x}_i(0) \oplus K_p)) \quad (5.7)$$

$$= \text{HD}((\vec{x}_i(0) \oplus K_p), \text{SBOX}(\vec{x}_i(0) \oplus K_p)) \quad (5.8)$$

This HD model assumes that the previous state A of a register is replaced by a new state B . State A could be an S-box input, while state B might be the corresponding output, as it is the case in Equation 5.8. However, depending on the hardware architecture or the software implementation of a DUT, several prediction models can be applied to test whether a data dependency is visible in the correlation trace. Some models work better than other. Thus, we first recommend to exhaustively test several prediction models for the case that a known key is given. This is possible for our DUT since we successfully reverse-engineered the real key computation. We present our utilized prediction models in Section 5.4.2. As shown in Equation 5.9, the correlation trace $\Delta_{CPA}(K_p, q)$ (using the prediction model h_{ip}) is computed as

$$\Delta_{CPA}(K_p, q) = \frac{\frac{1}{N-1} \cdot \sum_{i=1}^N (\vec{t}_{x_i}(q) - m_{\vec{t}(q)}) \cdot (h_{ip} - m_{f(K_p)})}{\sqrt{\sigma_{\vec{t}(q)}^2 \cdot \sigma_{f(K_p)}^2}} \quad (5.9)$$

with

$$\sigma_{\vec{t}(q)}^2 = \frac{1}{N-1} \sum_{i=1}^N (\vec{t}_{x_i}(q) - m_{\vec{t}(q)})^2 \quad (5.10)$$

$$\sigma_{f(K_p)}^2 = \frac{1}{N-1} \sum_{i=1}^N \left(h_{ip} - m_{f(K_p)} \right)^2 \quad (5.11)$$

denoting the respective sample variances and

$$m_{\vec{t}(q)} = \frac{1}{N} \sum_{i=1}^N \vec{t}_{x_i}(q) \quad (5.12)$$

$$m_{f(K_p)} = \frac{1}{N} \sum_{i=1}^N h_{ip} \quad (5.13)$$

denoting the respective sample means. Plotting the correlation curves $\Delta_{CPA}(K_p, q)$ for the correct key candidate K_p over all sampled points q leads to a positive (or negative) correlation peak. This should not be the case for wrong key guesses and thus, the correct key candidate is distinguishable from incorrect ones.

5.4.2 Utilized Prediction Models

To be able to use several prediction models, we first compute the full AES state for each measurement/challenge and store the corresponding intermediate values in the variables defined in Table 5.2. Note that $\vec{x}_i(0 : 3)$ targets the first four plaintext bytes. To be more precise, u targets one (or more) bytes, i targets the i 'th AES encryption (used during the i 'th measurement), while j accesses the j 'th AES round within one AES encryption.

Variable	Description	State byte	AES round
$\vec{x}_i(u)$	i 'th AES Plaintext state	$u \in \{0, \dots, 15\}$	–
$\vec{ka}_{i, \text{round}_j}(u)$	i 'th AES AddRoundKey state	$u \in \{0, \dots, 15\}$	$j \in \{0, \dots, 10\}$
$\vec{sbox}_{i, \text{round}_j}(u)$	i 'th AES SubBytes state	$u \in \{0, \dots, 15\}$	$j \in \{1, \dots, 10\}$
$\vec{sr}_{i, \text{round}_j}(u)$	i 'th AES ShiftRows state	$u \in \{0, \dots, 15\}$	$j \in \{1, \dots, 10\}$
$\vec{mc}_{i, \text{round}_j}(u)$	i 'th AES MixColumns state	$u \in \{0, \dots, 15\}$	$j \in \{1, \dots, 9\}$
$\vec{y}_i(u)$	i 'th AES Ciphertext	$u \in \{0, \dots, 15\}$	–

Table 5.2: Computing AES states for each AES input.

For $u \in \{0, \dots, 15\}$ and $j \in \{1, 2\}$, we compute the following 8-bit predictions:

- $\text{HW}(\vec{x}_i(u))$
- $\text{HW}(\vec{y}_i(u))$
- $\text{HW}(\vec{ka}_{i, \text{round}_j}(u))$
- $\text{HW}(\vec{sbox}_{i, \text{round}_j}(u))$
- $\text{HW}(\vec{mc}_{i, \text{round}_j}(u))$
- $\text{HD}(\vec{ka}_{i, \text{round}_j}(u), \vec{sbox}_{i, \text{round}_j}(u))$

The according 8-bit predictions are shown in Fig. 5.10.

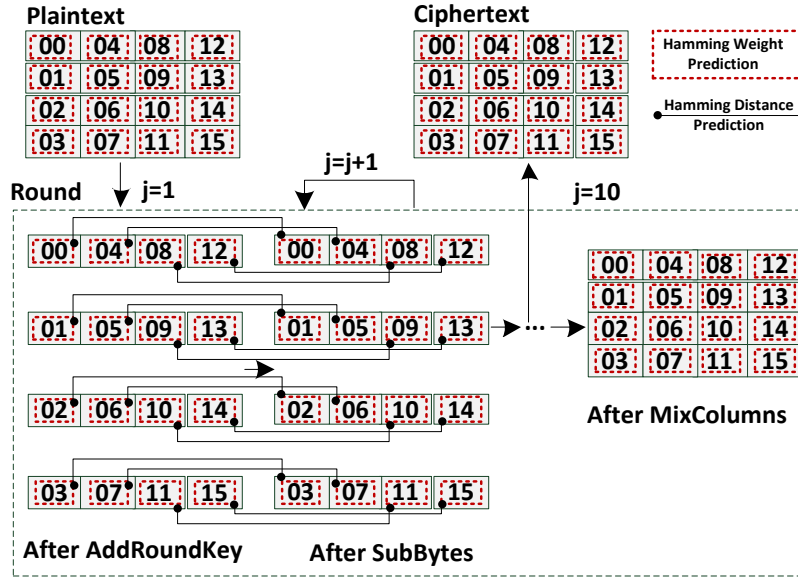


Figure 5.10: Utilized 8-bit prediction models.

For $u \in \{0, 2, 4, 6, 8, 10, 12, 14\}$ and $j \in \{1, 2\}$, we compute the following 16-bit predictions:

- $\text{HW}(\vec{x}_i(u : u + 1))$
- $\text{HW}(\vec{y}_i(u : u + 1))$
- $\text{HW}(\vec{k}a_{i, \text{round}_j}(u : u + 1))$
- $\text{HW}(\vec{s}box_{i, \text{round}_j}(u : u + 1))$
- $\text{HW}(\vec{m}c_{i, \text{round}_j}(u : u + 1))$
- $\text{HD}(\vec{k}a_{i, \text{round}_j}(u : u + 1), \vec{s}box_{i, \text{round}_j}(u : u + 1))$

The corresponding 16-bit predictions are outlined in Fig. 5.11.

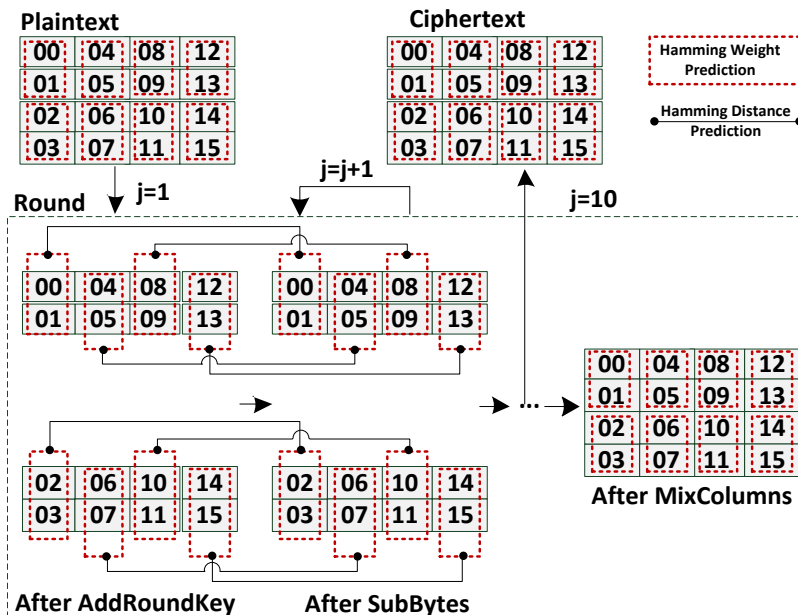


Figure 5.11: Utilized 16-bit prediction models.

For $u \in \{0, 4, 8, 12\}$ and $j \in \{1, 2\}$, we compute the following 32-bit predictions:

- $\text{HW}(\vec{x}_i(u : u + 3))$
- $\text{HW}(\vec{y}_i(u : u + 3))$
- $\text{HW}(\vec{ka}_{i, \text{round}_j}(u : u + 3))$
- $\text{HW}(\vec{sbox}_{i, \text{round}_j}(u : u + 3))$
- $\text{HW}(\vec{mc}_{i, \text{round}_j}(u : u + 3))$
- $\text{HD}(\vec{ka}_{i, \text{round}_j}(u : u + 3), \vec{sbox}_{i, \text{round}_j}(u : u + 3))$

The corresponding 32-bit guesses are illustrated in Fig. 5.12.

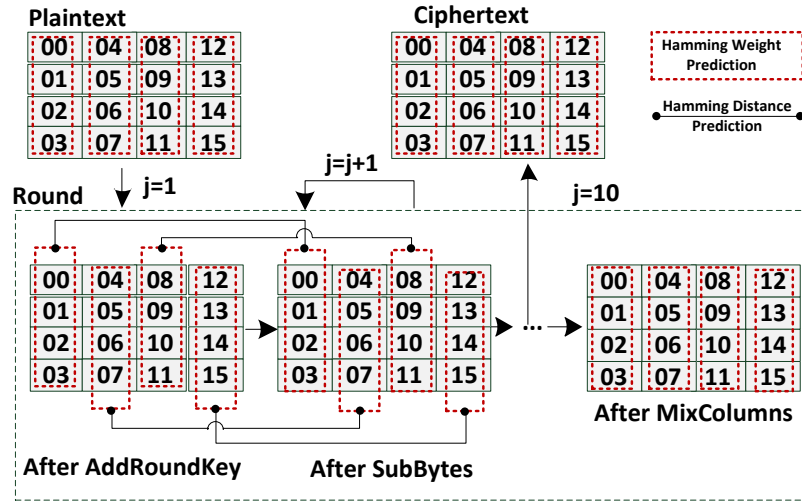


Figure 5.12: Utilized 32-bit prediction models.

For $u \in \{0, 8\}$ and $j \in \{1, 2\}$, we compute the following 64-bit predictions:

- $\text{HW}(\vec{x}_i(u : u + 7))$
- $\text{HW}(\vec{y}_i(u : u + 7))$
- $\text{HW}(\vec{ka}_{i, \text{round}_j}(u : u + 7))$
- $\text{HW}(\vec{sbox}_{i, \text{round}_j}(u : u + 7))$
- $\text{HW}(\vec{mc}_{i, \text{round}_j}(u : u + 7))$
- $\text{HD}(\vec{ka}_{i, \text{round}_j}(u : u + 7), \vec{sbox}_{i, \text{round}_j}(u : u + 7))$

The 64-bit predictions are shown in Fig. 5.13

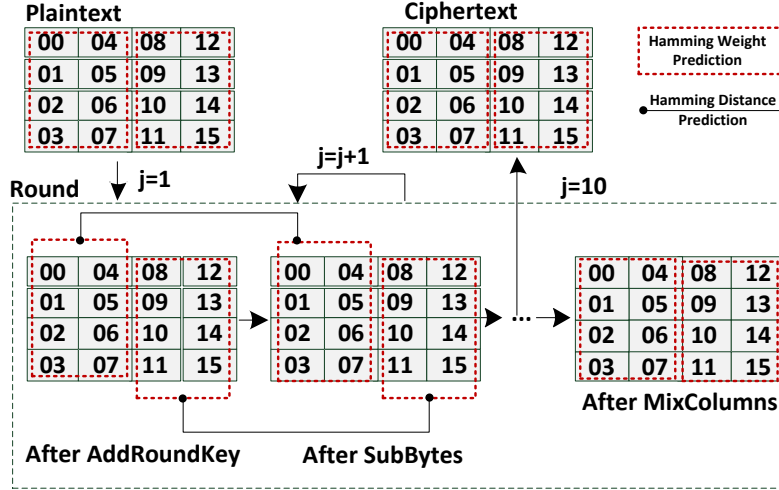


Figure 5.13: Utilized 64-bit prediction models.

For $j \in \{1, 2\}$, we compute the following 128-bit predictions:

- $\text{HW}(\vec{x}_i(0 : 15))$
- $\text{HW}(\vec{y}_i(0 : 15))$
- $\text{HW}(\vec{ka}_{i, \text{round}_j}(0 : 15))$
- $\text{HW}(\vec{sbox}_{i, \text{round}_j}(0 : 15))$
- $\text{HW}(\vec{mc}_{i, \text{round}_j}(0 : 15))$
- $\text{HD}(\vec{ka}_{i, \text{round}_j}(0 : 15), \vec{sbox}_{i, \text{round}_j}(0 : 15))$

The resulting 128-bit predictions are outlined in Fig. 5.14.

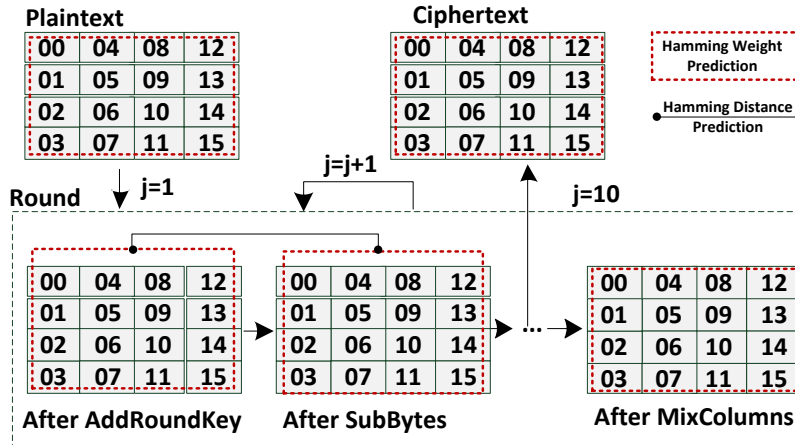


Figure 5.14: Utilized 128-bit prediction models.

In addition to that, we use the "bit model". This prediction takes only one bit into account, e.g., one bit of one byte of one AES S-box output. The function $\text{GETBIT}_b(X)$ yields bit b of a given value X , e.g., $\text{GETBIT}_{b=0}(1110)$ returns the value 0, while $\text{GETBIT}_{b=1}(1110)$ yields 1. We have used the following bit models for this thesis:

- $b \in \{0, \dots, 7\} : \text{GETBIT}_b(\vec{x}_i(0))$
- $b \in \{0, \dots, 7\} : \text{GETBIT}_b(\vec{mc}_{i,\text{round}_2}(0))$
- $b \in \{0, \dots, 7\} : \text{GETBIT}_b(\vec{y}_i(0))$
- $b \in \{0, \dots, 127\} : \text{GETBIT}_b(\vec{sbox}_{i,\text{round}_1}(0 : 15))$

This means that we compute 128 correlation curves for the first round of the **SubBytes** state (for each bit), while for the **MixColumns**, plaintext, and ciphertext state, we only compute 8 correlation curves. To also cover round-based implementations of the AES, we compute the following predictions:

- $\text{HD}(\vec{ka}_{i,\text{round}_1}(0 : 15), \vec{ka}_{i,\text{round}_2}(0 : 15))$
- $\text{HD}(\vec{sbox}_{i,\text{round}_1}(0 : 15), \vec{sbox}_{i,\text{round}_2}(0 : 15))$
- $\text{HD}(\vec{mc}_{i,\text{round}_1}(0 : 15), \vec{mc}_{i,\text{round}_2}(0 : 15))$

Moreover, for detecting any columnwise processing of one 4-byte **ShiftRows** state that toggles to one 4-byte **MixColumns** state we predict:

$$u \in \{0, 4, 8, 12\} : \text{HD}(\vec{sr}_{i,\text{round}_1}(u : u + 3), \vec{mc}_{i,\text{round}_1}(u : u + 3))$$

Finally, for the case that the AES implementation shifts each 4-byte **ShiftRows** state (and thus overwrites the previous **ShiftRows** state), we utilize:

$$u \in \{0, 4, 8\} : \text{HD}(\vec{sr}_{i,\text{round}_1}(u : u + 3), \vec{sr}_{i,\text{round}_1}(u + 4 : u + 7))$$

To also cover a byte-wise shifting of the **ShiftRows** state, we compute:

$$u \in \{0, \dots, 11\} : \text{HD}(\vec{sr}_{i,\text{round}_1}(u), \vec{sr}_{i,\text{round}_1}(u + 4))$$

All the described prediction models were used to locate the AES engine in the power consumption of the DUT. Having located the correct point in time, we predict key candidates (in the attack step) instead of predicting states for which the key has to be known (in the profiling step).

5.5 Locating the FPGA Configuration

In this section, we demonstrate that the points in time of the configuration of the DUT can be found by means of CPA. First note that the gray, dotted lines of Fig. 5.15a and Fig. 5.15b indicate a new 16-byte block being transferred to the DUT. Furthermore, we compute several correlation curves that are shown in Fig. 5.15a and Fig. 5.15b. First, we send 48 random bytes to the DUT, and vary the IV (and thus the AES input) to record 1000 power traces. Then, we compute the following predictions of the first AES outcome (first encrypted 16-byte IV block)

$$\text{HD}(\vec{y}_i(u), \vec{y}_i(u+1)) \text{ for } u \in \{0, \dots, 14\}$$

and hence guess that the AES outcome is shifted bitwise into a register. Due to the random data sent to the DUT, the DUT internally computes (trying to decrypt the data)

$$\vec{\text{conf}}_i(0 : 47) := \vec{y}_i(0 : 47) \oplus \vec{r}_i(0 : 47)$$

whereas r_i refers to the transferred (random) data and $\vec{\text{conf}}_i(0 : 47)$ represents the result of the decryption. The DUT then (wrongly) configures its internals using the $\vec{\text{conf}}_i(0 : 47)$ bytes. As a result, we obtain the correlation given in Fig. 5.15a. As one can see, no significant correlation occurs and hence, we conclude that the AES outcome is probably not shifted bitwise into a register, since the corresponding prediction is uncorrelated to the power traces.

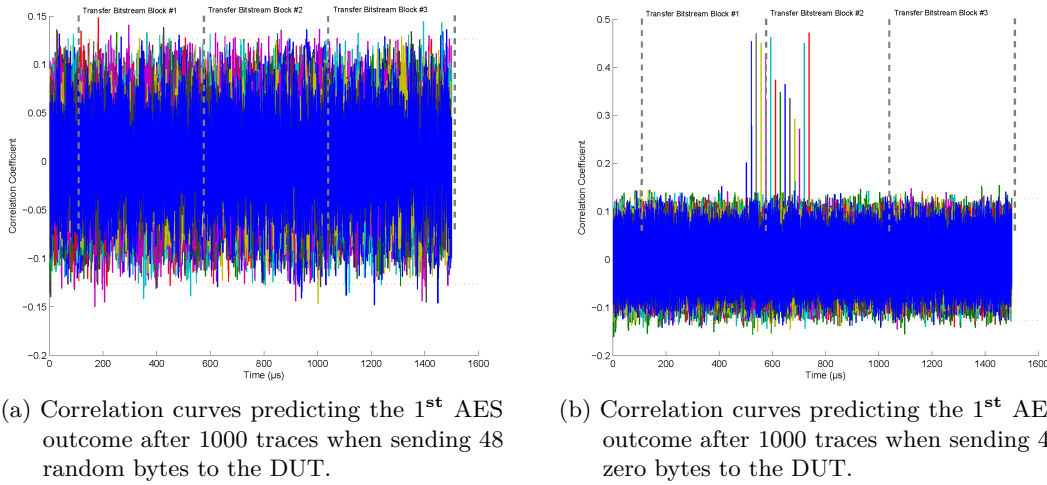


Figure 5.15: Difference between sending 48 random bytes or 48 zero bytes to the DUT.

We repeated the experiment, but this time sending 48 zero bytes to the DUT. Again, we computed the same predictions using the new set of power traces. Note that then, the FPGA internally computes:

$$\vec{\text{conf}}_i(0 : 47) = \vec{y}_i(0 : 47) \oplus 0x00\dots00 = \vec{y}_i(0 : 47)$$

This means that the DUT configures the (incorrect) AES outcome bytes. We obtain the results presented in Fig. 5.15b. We get several curves that are correlated to our predictions at approximately 550 μs and 800 μs (shortly before the transfer of the 2nd 16-byte block starts).

From these observations, we conclude that in general, the decrypted bytes are shifted byte-wise into a register. Note that we do not obtain a correlation for each utilized prediction model, e.g., $\text{HD}(\vec{y}_i(14), \vec{y}_i(15))$ is uncorrelated to the power traces. This might be due to the reason that some bytes are not involved or processed differently. We further assume that the points in time of the given correlations are also the points in time of configuring the internals of the DUT.

Next, we use the same prediction models (but this time we guess the 2nd AES outcome which is equal to encrypting the first incremented IV), using again the new set of power traces. It turns out that we obtain a very similar result compared to the previous correlation curves, as shown in Fig. 5.16.

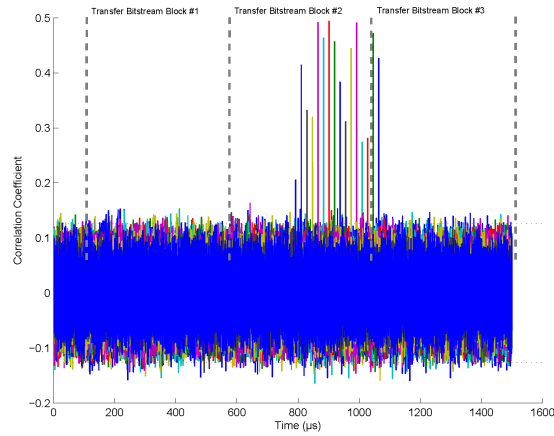


Figure 5.16: Correlation curves predicting the 2nd AES outcome after 1000 traces when sending 48 zero bytes to the DUT.

This time, the leakage appears earlier, compared to the 1st AES outcome. This implies that the final XOR-operation (for decrypting a bitstream block) is not necessarily performed directly after the transfer of one 16-byte block. Again, most of the HD models correlate with our power traces. Thus the configuration process can be made visible with the help of CPA. These results suggest that the FPGA internally processes a decrypted block as depicted in Fig. 5.17.

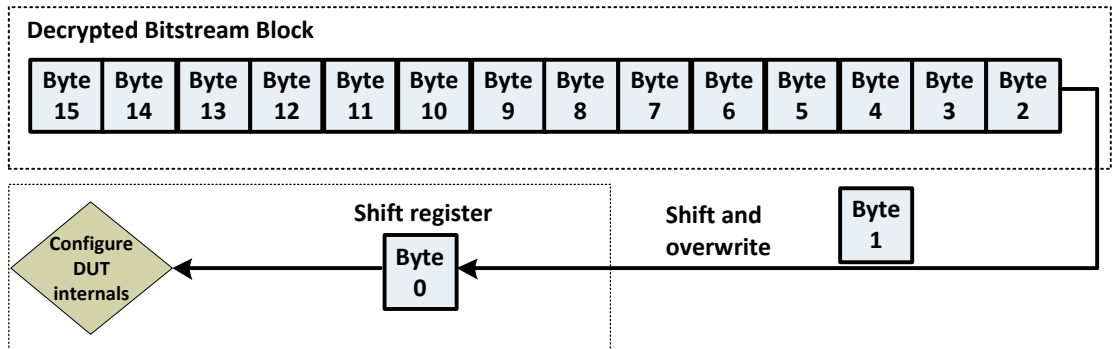


Figure 5.17: Hypothetical DUT configuration process.

5.6 Locating the AES Encryption

To locate (and verify) the correct time instance of the first AES encryption, we use the set of 840,000 recorded power traces to which Algorithm 5.1.1 was applied. Then, for our profiling, we used the known key to compute all intermediate AES values for each IV challenge/trace.

As we described in Sect. 5.4.2, we computed the correlation curves of about 220 different prediction models, e.g., each S-box bit of the first AES round, several HD models with different predicted register sizes, and several 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 5.18 shows nine of the correlation curves for the states after each AES round. To be more precise, we computed the following models of Section 5.4.2: $\text{HW}(\vec{ka}_{i,\text{round}_j}(0 : 15))$ for $j \in \{1, \dots, 9\}$.

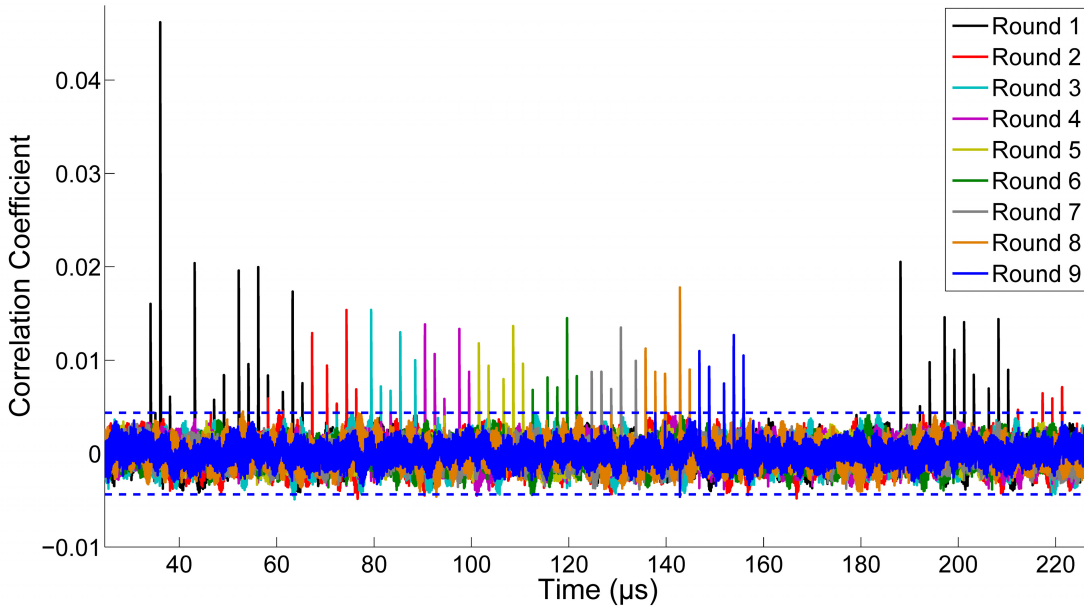


Figure 5.18: Correlation coefficient for one full AddRoundKey 128-bit state (one curve for each round). Utilized models: 1st curve \leftrightarrow HW of round 1, 2nd curve \leftrightarrow HW of round 2, etc.

The first correlation curve (black) that exhibits a peak up to an approximate value of 0.05 between 30 and 65 μs 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 5.18, 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 5.18) 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) thus 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.

6 Side-Channel Key Extraction of Stratix II

As shown in Chapter 5, 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.

6.1 Digital Pre-Processing

As commonly encountered in SCA, the effect of the AES encryption on the overall power consumption is rather small (cf. Chapter 5). 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 [BPT10]. 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., $1.5 \mu\text{s}$ 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 [GTC05] 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 [OP11, PHF08].

6.2 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, cf. Section 5.4.2.

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 explained in Section 5.6. 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. 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 6.1.

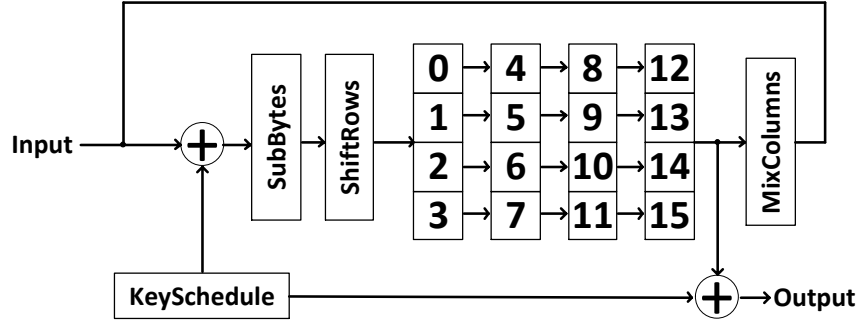


Figure 6.1: Hypothetical architecture of the AES implementation.

For the key extraction in Section 6.3, 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. To be more precise, we used following models in the profiling phase:

$$u \in \{0, \dots, 11\} : \text{HD}(\vec{s}r_{i,\text{round}_1}(u), \vec{s}r_{i,\text{round}_1}(u + 4))$$

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.

6.3 Results with Digital Pre-Processing

Using the described power model, we computed the correlation coefficient for the respective (byte-wise) HD of the AES states. Figure 6.2 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 $4/\sqrt{\#traces} = 0.006$ [MOP07].

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 3, the first 64-bit half of the plaintext (i.e., the IV) equals the second half. Hence, a second key candidate (from the

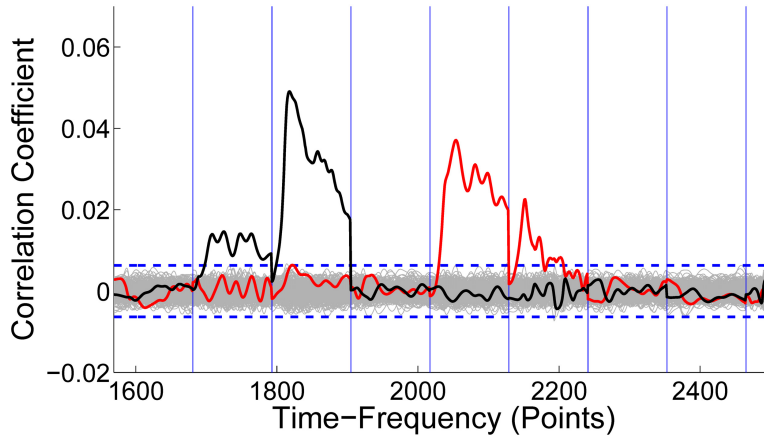


Figure 6.2: Correlation coefficient for the first S-box after 400,000 traces using DFT pre-processing. Correct key candidate 0x2B: black curve.

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 [MOP07], the minimal number of traces to extract the full AES key is approximately $2^8/\rho_{min}^2 = 29,136$.

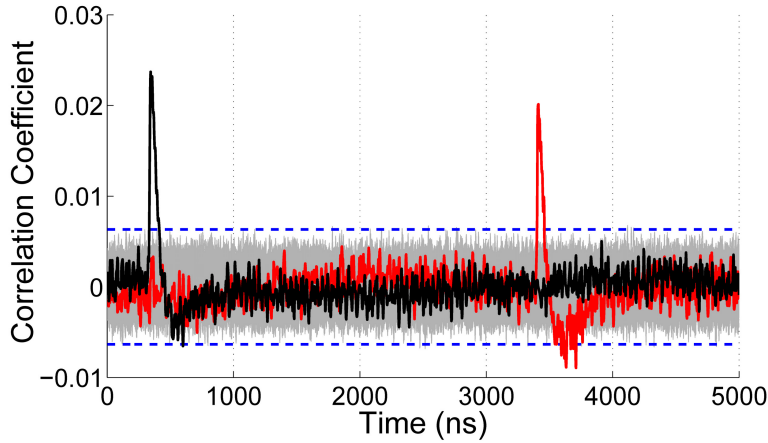


Figure 6.3: Correlation coefficient for the first S-box after 400,000 traces without DFT pre-processing. Correct key candidate 0x2B: black curve.

Figure 6.3 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 6.2,

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.

Moreover, we are able to recover the full key using the bitmodel prediction. Again, we tested filtering of the raw traces and reduced the bandwidth of the signal, this time by applying a lowpass and bandpass filter given in Fig. 6.4a and Fig. 6.4b.

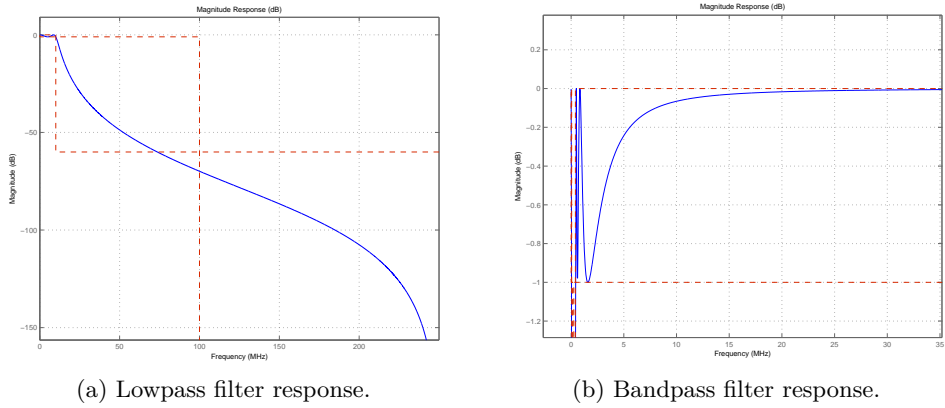


Figure 6.4: Applied filter on raw measurement traces.

With the help of both filters, we were able to remove a high-frequency component of the signal, and hence extract lower frequencies. From a side-channel point of view, the HF signal adds disturbing noise.

Fig. 6.5a, Fig. 6.5b, and Fig. 6.5c show five raw traces, five traces after applying the lowpass filter of Fig. 6.5b, and five traces after applying both the lowpass filter of Fig. 6.5b and the bandpass filter of Fig. 6.4b.

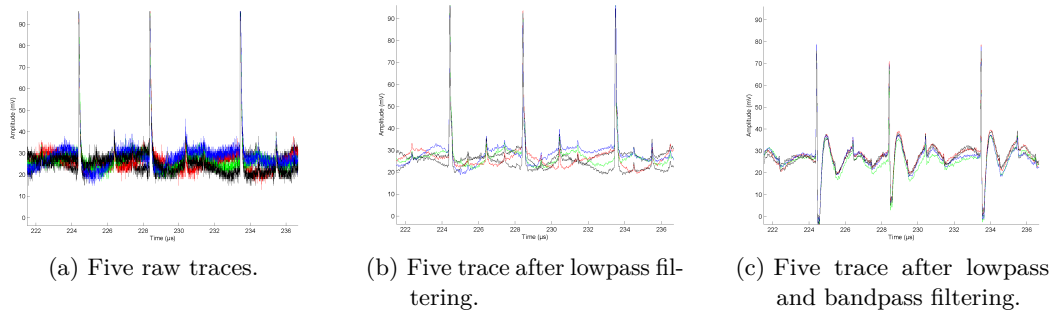


Figure 6.5: Influence of pre-processing (5 traces).

After filtering all traces using the above described filters we computed – for each of

the sixteen AES state bytes (first S-box round) – 2^8 key candidates. The corresponding 256 correlation coefficient curves are based on predicting only one bit of the non-linear S-box output. The attack is repeated for each key byte separately. Hence, in total, we only test $16 \times 256 = 2^4 \times 2^8 = 2^{12} = 4096$ key candidates.

Fig. 6.6a, Fig. 6.6b, Fig. 6.6c, and Fig. 6.6d exemplary illustrate the recovery of four key bytes. The correct key candidates belongs to the red curve with the highest peak.

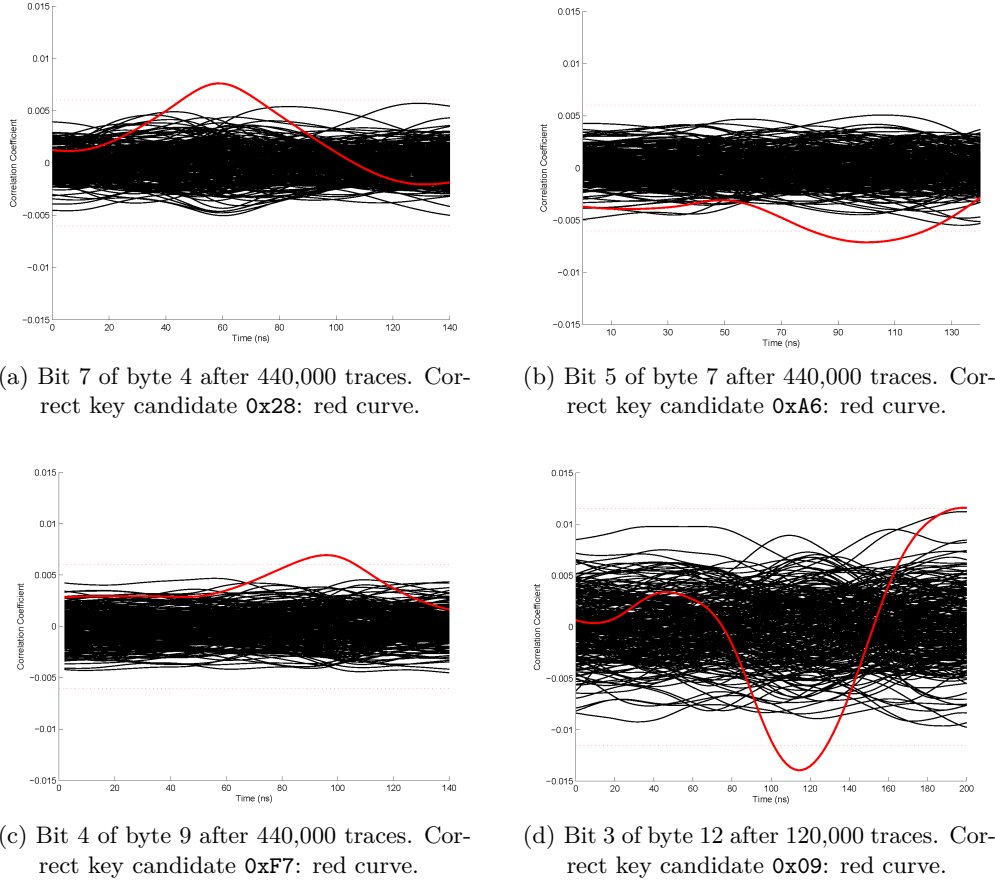


Figure 6.6: Bit model after first S-box for byte 4, 7, 9, and 12.

Fig. 6.6a illustrates the recovery of key byte 4 (0x28) after 440,000 traces, while Fig. 6.6d outlines that key byte 12 (0x09) can be revealed after 120,000 traces. For the majority of key bytes, (almost) any S-box output bit is suitable as prediction to make the correct key candidate distinguishable from incorrect ones.

6.4 Results without Digital Pre-Processing

It is even possible to recover key bytes (using the bit model) without applying any pre-processing techniques. Similar to Section 6.3, we provide examples of recovering four

key bytes. The according correlation curves are given in Fig. 6.7a, Fig. 6.7b, Fig. 6.7c, and Fig. 6.7d.

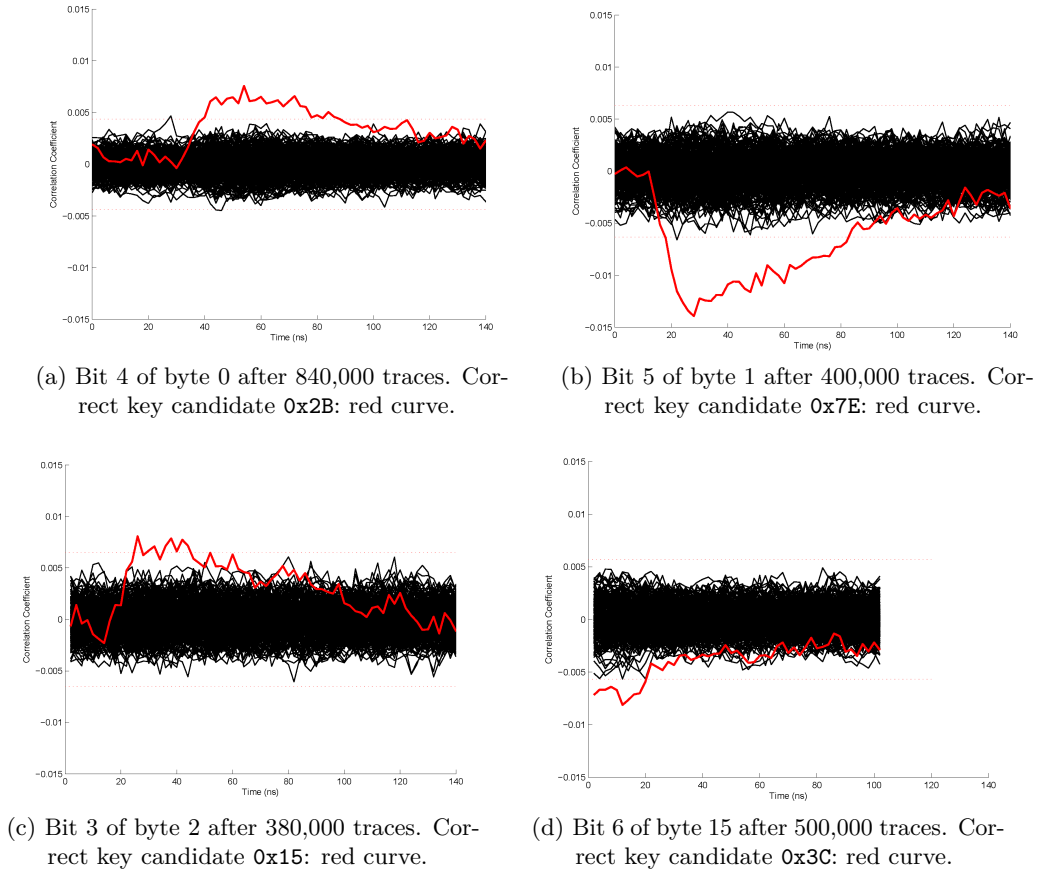


Figure 6.7: Bit model after first S-box for byte 0, 1, 2, and 15.

Note that if one has twelve of sixteen key bytes it is possible to find the remaining four key bytes using a standard PC, since $2^{32} = 4,294,967,296$ key candidates can be tested in a quick time. For such an attack, it is essential to have a valid testvector (plaintext \leftrightarrow ciphertext block) to be able to verify the correctness of a decrypted block. In our case, such a testvector is initially not available, but we could create an additional unencrypted random configuration design (for the same FPGA fabric) and examine which bitstream blocks are fixed amongst two (or more) configuration designs.

It would also be conceivable to count the number of zeroes or ones of a decrypted 16-byte block. If the corresponding counter exceeds a certain threshold, one can assume that the encrypted block was decrypted successfully. This is due to the fact that the bytes of an unencrypted bitstream are mostly set to zero or one, while the bytes of a wrongly decrypted block are distributed randomly.

We also analysed the key bytes 5, 6, 10, and 11. The results are depicted in Fig. 6.8a, Fig. 6.8b, Fig. 6.8c, and Fig. 6.8d.

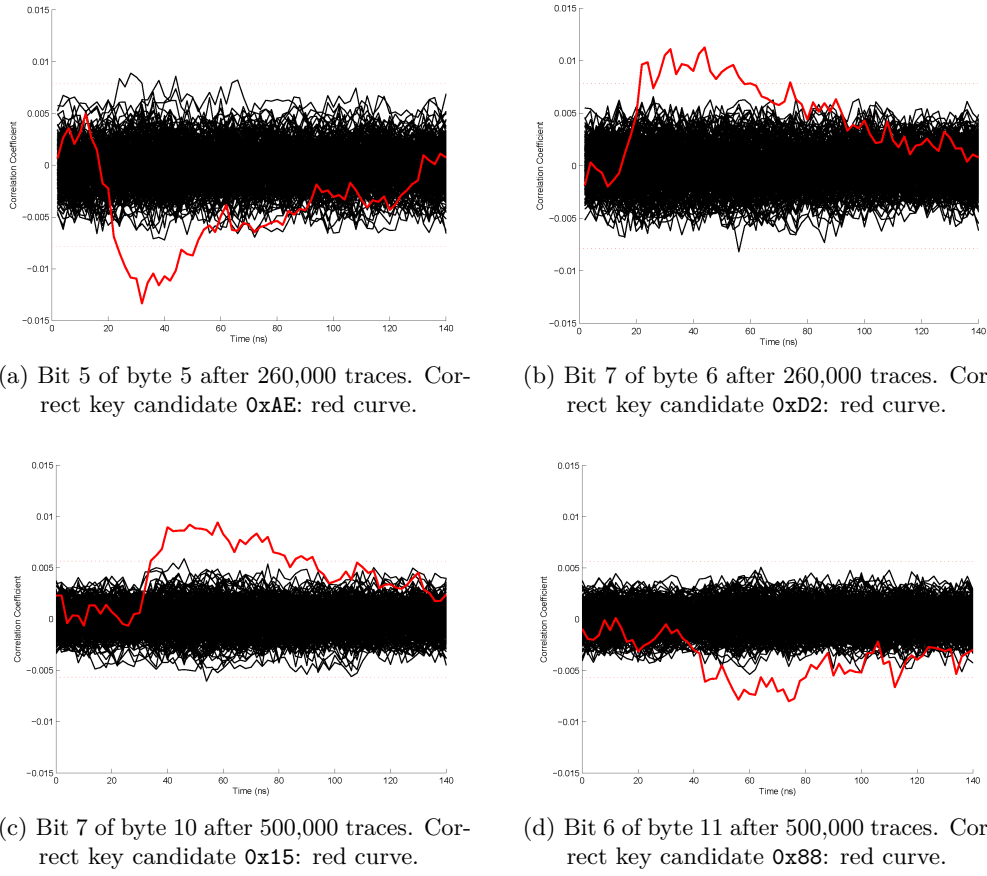


Figure 6.8: Bit model after first S-box for byte 5, 6, 10, and 11.

For revealing the key bytes 5, 6, 10, and 11, we computed the correlation coefficient traces after 260,000 (or 500,000 traces). Once again, we use the formula $\frac{28}{p^2}$ to estimate the minimum number of required traces. The corresponding results are presented in Table 6.1.

Keybyte	Used Traces	Maxium correlation value	Required Traces ($\frac{28}{p^2}$)
0	260,000	-0.01334	157,342
6	260,000	0.01126	220,841
10	500,000	0.009414	315,943
11	500,000	-0.007981	439,585

Table 6.1: Minimum amount of required traces for performing a successful attack

It turns out that theoretically less than the given amount of traces can be used to successfully perform a CPA attack. E.g., for revealing key byte 5, approximately 157,000 traces are necessary. To practically verify this, we updated (and stored) the correlation coefficient for each of 256 key candidates of exactly one time instance (where the leakage occurs), after each trace using the Welford algorithm [KOP09]. Fig. 6.9 illustrates the corresponding result.

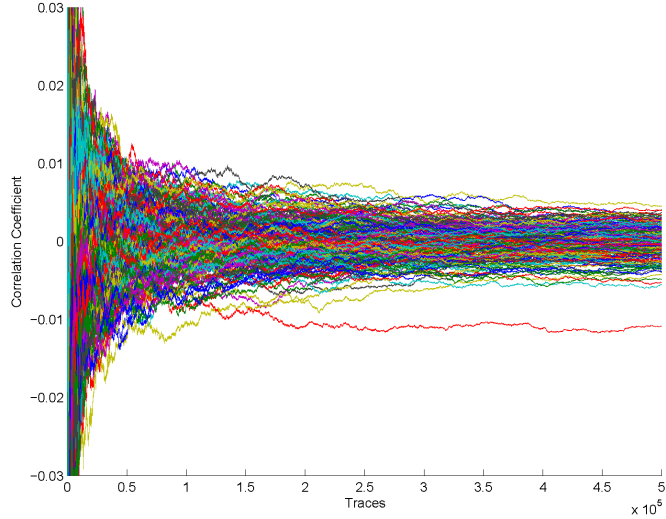


Figure 6.9: Correlation coefficient after each unfiltered trace, one time instance for each of 256 key candidates using the bit model (bit 5 of byte 5 of the AES S-box output).

As one can see in Fig. 6.9, only one (correct) key candidate (0xAB) starts to clearly differentiate from all other 255 (wrong) key candidates after approximately 155,000 traces, which is very close to the theoretical value (157,342). Furthermore, we repeated the same computation for the filtered set of traces. The according result is given in Figure 6.10.

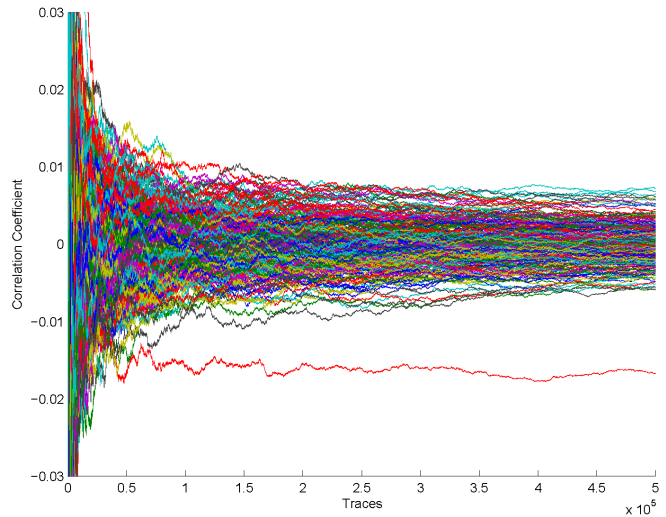


Figure 6.10: Correlation coefficient after each filtered trace, one time instance for each of 256 key candidates, using the bit model (bit 5 of byte 5 of the AES S-box output).

Fig. 6.10 shows that the attack can be improved using filtering as pre-processing, because only $\sim 60,000$ traces are required to distinguish the correct key candidate from incorrect ones. This is an improvement by a factor of 2.58. Hence, filtering power traces can speed up the key recovery.

Note that the bit model requires more traces, but less key candidates have to be tested, compared to the HD prediction model introduced in Section 6.2. Fig. 6.11 shows the points in time where the above shown bit model leakages occur. One arrow points at the leakage of one key byte for one point in time.

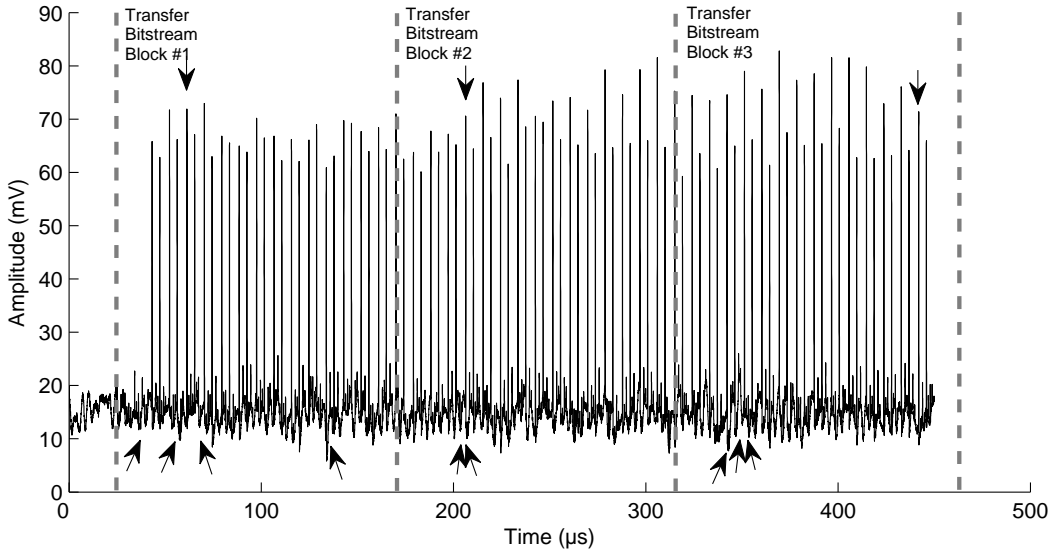


Figure 6.11: Occurrences of leakage for all bit model correlation curves.

Five key bytes can be revealed due to the measured power consumption that corresponds to the transfer of the first bitstream block, while three further key bytes (and four further key bytes) are the result of the second encryption (and the third encryption, respectively). It turns out that the incremented IV is helpful for a side-channel attack, since it is more likely that an attacker can find a leakage, e.g., during the second encryption (but not during the first encryption), although the prediction was initially designed for the first encryption.

Apart from the **ShiftRows** HD model of Section 6.2 and the bit model of this section, we found other models that correlate with our traces. All predictions refer to the first AES encryption, and the corresponding correlations are given in the Figures 6.12a- 6.12i.

Note that the second and third AES encryption also leak for certain prediction models, although the models were designed for the first AES encryption. This comes due to the incrementation of the IV and the similarity of the state bytes of the first two AES rounds amongst all three AES encryptions as already explained in Sect. 6.3.

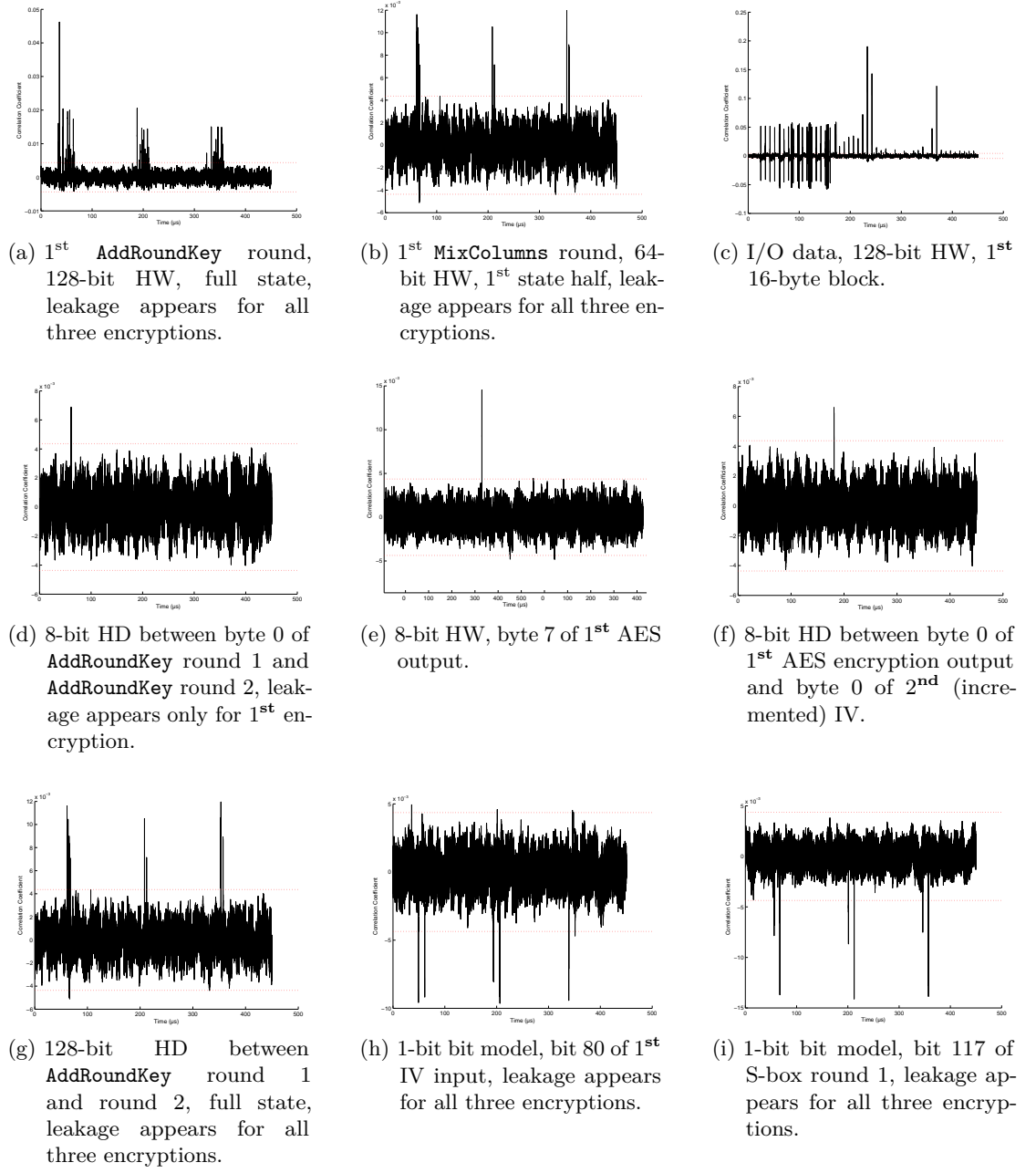


Figure 6.12: Correlating prediction models after 840,000 traces.

Finally, we present further HW models for the IV state of the 1st AES encryption using different bit widths, i.e., 8, 16, 32, 64, and 128 bit.

The results show that the majority of predictions are correlated to the power consumption. For the case of guessing the first 8 bit of the utilized IV, we obtained an uncorrelated curve that does not exceed the theoretical noise level (dotted vertical lines) as depicted in Fig. 6.13. When guessing 16 bit of the IV state, the prediction model starts to weakly correlate with the power consumption and the theoretical noise threshold is briefly exceeded. Increasing the predicted state size to 32 bit leads to a higher correlation of approximately 0.004 and hence, it indicates that (at least) more than sixteen bits are processed by the DUT at once.

Applying a 64-bit and 128-bit model, we obtained a clear correlation with a maximum of approximately 0.02. There is no visible difference given between the 64-bit and 128-bit correlation curve, as the first 64-bit half of the IV matches with the second 64-half, and hence, the returned HW values are in principal equal except a constant factor of two (when guessing the 128-bit state) that does not affect the correlation.

To sum up, basically, the shapes of all correlation curves are rather similar and predicting the full IV state leads to the strongest correlation. Thus, it seems that the DUT processes 128 bit (or 64 bit) simultaneously.

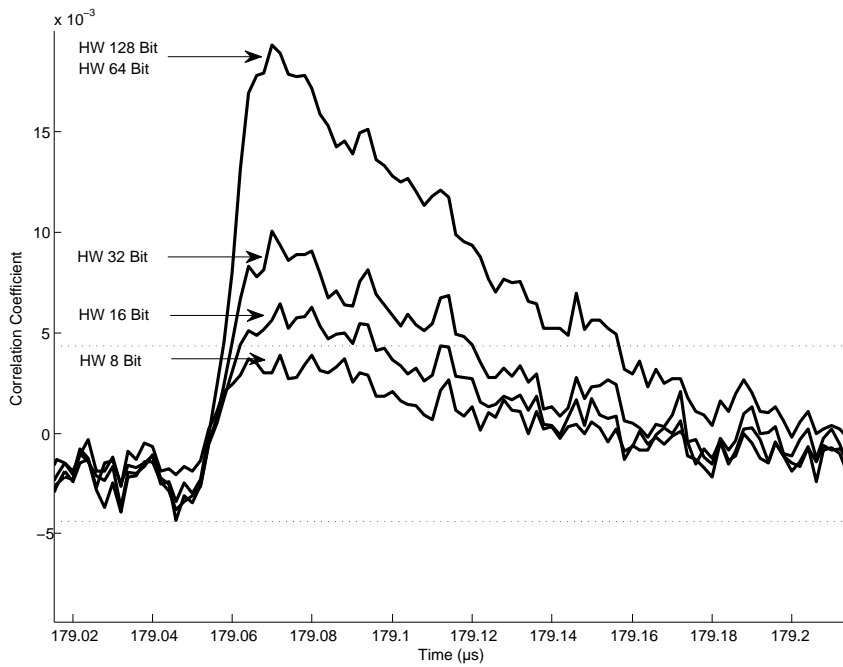


Figure 6.13: Correlation coefficient for the initial IV after 840,000 traces without applying pre-processing.

7 Conclusion

In this chapter, we summarize, discuss, and provide an overview of our research results. Moreover, we describe the methods that were used to reveal severe security vulnerabilities of Altera FPGAs. In addition to that, we discuss further approaches that may be useful for future work.

7.1 Summary

In this thesis, we performed a security analysis of Altera Stratix II FPGAs. The security evaluation is based on reverse-engineering the Quartus II black-box software and applying a CPA attack to reveal the (secret) 128-bit real key that is used by an internal AES engine to decrypt encrypted configuration designs. We revealed the (undocumented) file structures, the differences between an unencrypted and encrypted bitstream file, and reverse-engineered Quartus II to find out how bitstream encryption works.

We found the (unknown) key derivation function that processes two 128-bit keys (KEY1 and KEY2) deriving the actual AES 128-bit key (real key) of the FPGA. The real key is used for the AES encryption of the IVs. From here, all relevant details for decrypting the encrypted RBF file were given and hence we implemented a program that is able to decrypt encrypted RBF files.

We picked a suitable configuration mode (PS) for the Stratix II FPGA, implemented the according protocol for our own ATmega256 μ C, and recorded the power consumption over the V_{CCINT} power supply line. Because of that, we found differences in the power consumption between the processing of an unencrypted and encrypted RBF file. In addition to that, we wrote an evaluation framework for guessing several AES implementations and performed a known-key profiling. By doing so, we obtained a significant correlation using 400,000 traces. Based on this, we improved the results applying different pre-processing techniques like filtering, peak extraction, or DFA, and figured out which power model works best.

Having localized the leaking points in time, we demonstrated that it is possible to recover the full 128-bit AES real key by means of side-channel attacks in a few hours. Thus the cloning of a configuration design becomes possible in practice. The vulnerabilities that involve Stratix II FPGAs are the following:

- The Quartus II application itself is unprotected. This implies that it is possible to reverse-engineer the software.
- Security by obscurity: Hiding the IV bits in the header *does not* add to the protection of the system. The IV could be stored directly in the RBF file.

- Using the timestamp as a part of the IV, which is then incremented for the next AES input, is a potential security problem.
- The KEY1 and KEY2 derivation function does not prevent from cloning. Once an attacker has obtained the real key, he is able to decrypt the real key picking any KEY1 that results in a KEY2, forming one of 2^{128} tuples leading to the same (desired) real key when programmed into another blank Stratix II FPGA. The new FPGA accepts the (previously intercepted) encrypted configuration design (that was basically encrypted with the revealed real key). Instead of this, a secure key derivation function has to be one-way.
- It is very likely that no countermeasures are included against side-channel attacks.

After having demonstrated that the design security feature of Stratix II can be broken by SCA, we analysed the Stratix III series. Again, we reverse-engineered the Quartus II application. We found that the utilized block mode is similar to that of Stratix II. In addition to that, we discovered how the 256-bit real key (consisting of two AES output parts) is derived, and that the IV byte sequences are not incremented.

Instead of this, a pseudorandom number generator is used to derive all AES inputs, which are encrypted, and used as stream cipher key. We illustrated a workaround how one can dump all IVs (with relatively little efforts) by loading the Quartus II application in the IDA Pro debugger and using the IDA environment.

As a last step, we finally implemented a C++ program that is able to decrypt encrypted Stratix III (an EPS3SE50F484C2 FPGA) configuration designs, once we provide the correct 256-bit AES real key and the corresponding IV sequences.

7.2 Future Work

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 for Stratix II and Stratix III FPGAs. Using this knowledge, a side-channel adversary can mount a successful key recovery attack on Stratix II devices. 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 generation of Altera FPGAs, the fact that SCA countermeasures have been ignored during the development appears likely. However, it is an interesting aspect whether Stratix III, Stratix IV, Stratix V, or Arria II include any SCA countermeasures. 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. We already created a basis for an SCA attack regarding Stratix III FPGAs, because we reverse-engineered all relevant functions. For the case that it is also possible to obtain the real key from Stratix III FPGAs – by means of SCA – revealing the pseudorandom number function (for deriving the AES inputs) becomes interesting. Having reverse-engineered the pseudorandom number function, an attacker is able to speed-up the decryption process of a RBF file to few seconds instead of approximately three hours.

List of Figures

3.1	Structure of an unencrypted and an encrypted Stratix II RBF file.	5
3.2	Quartus II black-box generating encrypted Stratix II bitstreams.	7
3.3	Quartus II call sequence during the bitstream encryption.	8
3.4	AES in CTR mode.	16
3.5	Overview of the bitstream encryption process for the Stratix II FPGAs. .	18
4.1	Structure of an unencrypted and an encrypted Stratix III RBF file.	24
4.2	Quartus II black-box generating the Stratix III 256-bit real key.	25
4.3	Revealed bitstream encryption of Stratix III.	27
4.4	Dumping IVs with IDA Pro for any initial vector.	29
5.1	Measurement setup for SCA schematic.	34
5.2	Measurement setup for SCA in the real world.	34
5.3	Passive serial adapter wiring.	36
5.4	Passive serial protocol.	37
5.5	Average power consumption (10k traces) while sending an unencrypted (solid) and an encrypted (dashed) bitstream. Zoom on one byte.	39
5.6	Zoom on average power consumption (10k trace) while sending the first half of one bitstream byte.	39
5.7	Zoom on average power consumption (10k trace) while sending the second half of one bitstream byte.	40
5.8	Difference between unencrypted and encrypted bitstream while sending 16 bytes to the DUT.	41
5.9	Difference between unencrypted and encrypted bitstream while sending seventeen 16-byte blocks to the DUT.	41
5.10	Utilized 8-bit prediction models.	45
5.11	Utilized 16-bit prediction models.	45
5.12	Utilized 32-bit prediction models.	46
5.13	Utilized 64-bit prediction models.	47
5.14	Utilized 128-bit prediction models.	47
5.15	Difference between sending 48 random bytes or 48 zero bytes to the DUT. .	49
5.16	Correlation curves predicting the 2 nd AES outcome after 1000 traces when sending 48 zero bytes to the DUT.	50
5.17	Hypothetical DUT configuration process.	50
5.18	Correlation coefficient for one full AddRoundKey 128-bit state (one curve for each round). Utilized models: 1 st curve \leftrightarrow HW of round 1, 2 nd curve \leftrightarrow HW of round 2, etc.	51

6.1	Hypothetical architecture of the AES implementation.	54
6.2	Correlation coefficient for the first S-box after 400,000 traces using DFT pre-processing. Correct key candidate 0x2B: black curve.	55
6.3	Correlation coefficient for the first S-box after 400,000 traces without DFT pre-processing. Correct key candidate 0x2B: black curve.	55
6.4	Applied filter on raw measurement traces.	56
6.5	Influence of pre-processing (5 traces).	56
6.6	Bit model after first S-box for byte 4, 7, 9, and 12.	57
6.7	Bit model after first S-box for byte 0, 1, 2, and 15.	58
6.8	Bit model after first S-box for byte 5, 6, 10, and 11.	59
6.9	Correlation coefficient after each unfiltered trace, one time instance for each of 256 key candidates using the bit model (bit 5 of byte 5 of the AES S-box output).	60
6.10	Correlation coefficient after each filtered trace, one time instance for each of 256 key candidates, using the bit model (bit 5 of byte 5 of the AES S-box output).	60
6.11	Occurrences of leakage for all bit model correlation curves.	61
6.12	Correlating prediction models after 840,000 traces.	62
6.13	Correlation coefficient for the initial IV after 840,000 traces without ap- plying pre-processing.	63
A.1	Peak extraction example for one power trace.	97

List of Tables

3.1	Bitstream file formats generated by Quartus II.	3
3.2	Configuration modes for the Stratix II.	4
3.3	Mapping between the IV bits and the header bytes.	6
3.4	DLL function names of <code>pgm_pgio_nv_aes.dll</code>	8
3.5	Results of calling the <code>make_encrypted_bitstream()</code> function with different parameters.	15
3.6	IV sequences for two encryptions invoked at different points in time. . . .	19
4.1	IV sequences for Stratix III encryption using different points in time. . . .	27
5.1	Mapping signals for AS and PS support.	36
5.2	Computing AES states for each AES input.	44
6.1	Minimum amount of required traces for performing a successful attack . .	59

List of Algorithms

3.2.1 Retrieve IV Coding Rules	5
3.5.1 Encrypt Bitstream	17
3.5.2 Decrypt an encrypted bitstream of Stratix II	21
5.1.1 Measurement Setup	35

List of Listings

3.1	Freezing windows time.	4
3.2	Batch script displaying a filtered set of DLL function names.	10
3.3	C++ header file exporting two DLL functions.	11
3.4	C++ file defining the behavior of the exported functions of Listing 3.3. . .	11
3.5	Extracting information of our sample DLL file.	11
3.6	Calling doxor() DLL function explicitly.	13
3.7	Assumed implementation of the make_encrypted_bitstream() function. .	15
4.1	Setting a breakpoint using IDC scripting.	29
4.2	Dumping AES inputs using IDC scripting.	31
A.1	Code example of setting breakpoints for a specific segment.	77
A.2	Code example of how to determine cursor offset from segment.	78
A.3	Code example of how to patch IV bytes.	78
A.4	Code example of decrypting an encrypted Stratix II RBF file.	79
A.5	Comparing Stratix III header.	81
A.6	Example of first ten dumped IVs of Stratix III.	82
A.7	Example of first ten encrypted bitstream blocks of Stratix III.	82
A.8	Example of first ten unencrypted bitstream blocks.	83
A.9	Code example of decrypting an encrypted Stratix III RBF file.	83
A.10	Code example for Welford CPA.	85
A.11	Prediction model examples.	89
A.12	Configuration of the first three 16-byte blocks.	93
A.13	Code example for filtering one unfiltered trace.	95
A.14	Code example for peak extraction.	96

A Appendix

A.1 IDC Scripting

A.1.1 Setting Breakpoints

Listing A.1: Code example of setting breakpoints for a specific segment.

```
1 #include <idc.idc>
2
3 static main() {
4     // Variables
5     auto x, address, myOffset, segname, myChoice;
6     auto offsetToLastSegment, lastSegmentEA;
7
8     // Specify segment name
9     segname = "pgm_pgmio_nv_aes.dll";
10
11     // Offset to last segment
12     offsetToLastSegment = 0x13000;
13
14     // Choose where to set breakpoint
15     myChoice = 0;
16
17     if(myChoice == 0) { myOffset = 0x2C71; } // AES input
18     else if (myChoice == 1) { myOffset = 0x2C73; } // AES key
19     else if (myChoice == 2) { myOffset = 0x3a1b; } // AES output @[EAX] address
20     else if (myChoice == 3) { myOffset = 0x2a20; } // loc_a3112A20 encrypted bytes are copied here
21     else if (myChoice == 4) { myOffset = 0x1075; } // pgm_pgmio_nv_aes_?
22         key256@AESencrypt@@QAEHQBE@Z
23     else if (myChoice == 5) { myOffset = 0x56b0; } // pgm_pgmio_nv_aes_aes_encrypt_key
24     else if (myChoice == 6) { myOffset = 0x2170; } // pgm_pgmio_nv_aes_?do_something
25     else if (myChoice == 7) { myOffset = 0x260c; } // IV processing
26
27     // Determine dynamic address
28     for(address = FirstSeg(); address != BADADDR; address = NextSeg(address))
29         if (segname == SegName(address)) break;
30
31     // Compute final address
32     x = address + myOffset;
33
34     // Jump to x and add breakpoint
35     Jump(x);
36     AddBpt(x);
37
38     // Compute last segment address of pgm_pgmio_nv_aes.dll
39     lastSegmentEA = address + offsetToLastSegment;
40
41     // Convert to readable assembly code
42     AnalyzeArea(address, lastSegmentEA);
43 }
```

A.1.2 Determining Cursor Offset from Segment

Listing A.2: Code example of how to determine cursor offset from segment.

```

1 #include <idc.idc>
2
3 static main() {
4     // Define variables
5     auto address, segname, offsetTillCursorLine;
6
7     // Specify segment name
8     segname = "pgm_pgmio_nv_aes.dll";
9
10    // Determine dynamic address
11    for(address = FirstSeg(); address != BADADDR; address = NextSeg(address))
12        if (segname == SegName(address)) break;
13
14    // Compute offset to current cursor
15    offsetTillCursorLine = ScreenEA()-address;
16
17    // Output offset
18    Message("Offset from pgm_pgmio_nv_aes.dll till cursor is: %08lx", offsetTillCursorLine);
19 }

```

A.1.3 Patching IV Bytes for IV Coding Rules

Listing A.3: Code example of how to patch IV bytes.

```

1 #include <idc.idc>
2
3 static main() {
4     auto patch_byte_at_addr;
5
6     // Specify address of IV bytes
7     patch_byte_at_addr = 0x080A0A54;
8
9     // Patch IV bytes to 0xFF...FF
10    PatchByte(patch_byte_at_addr, 0xFF);
11    PatchByte(patch_byte_at_addr+1, 0xFF);
12    PatchByte(patch_byte_at_addr+2, 0xFF);
13    PatchByte(patch_byte_at_addr+3, 0xFF);
14    PatchByte(patch_byte_at_addr+4, 0xFF);
15    PatchByte(patch_byte_at_addr+5, 0xFF);
16    PatchByte(patch_byte_at_addr+6, 0xFF);
17    PatchByte(patch_byte_at_addr+7, 0xFF);
18
19    /** Patch IV bytes to 0x7F...FF
20    PatchByte(patch_byte_at_addr, 0x7F);
21    PatchByte(patch_byte_at_addr+1, 0xFF);
22    PatchByte(patch_byte_at_addr+2, 0xFF);
23    PatchByte(patch_byte_at_addr+3, 0xFF);
24    PatchByte(patch_byte_at_addr+4, 0xFF);
25    PatchByte(patch_byte_at_addr+5, 0xFF);
26    PatchByte(patch_byte_at_addr+6, 0xFF);
27    PatchByte(patch_byte_at_addr+7, 0xFF);
28    **/
29
30    // ...
31 }

```

```

32  /** Patch IV bytes to 0xFF...FE
33  PatchByte(patch_byte_at_addr, 0xFF);
34  PatchByte(patch_byte_at_addr+1, 0xFF);
35  PatchByte(patch_byte_at_addr+2, 0xFF);
36  PatchByte(patch_byte_at_addr+3, 0xFF);
37  PatchByte(patch_byte_at_addr+4, 0xFF);
38  PatchByte(patch_byte_at_addr+5, 0xFF);
39  PatchByte(patch_byte_at_addr+6, 0xFF);
40  PatchByte(patch_byte_at_addr+7, 0xFE);
41  **/
42 }

```

A.2 C++ and C Programs

A.2.1 Decryption of an Encrypted Stratix II RBF File

Listing A.4: Code example of decrypting an encrypted Stratix II RBF file.

```

1  #include "aes.h"
2  ...
3  // Function for obtaining IV
4  void getIV(FILE *filepoi, unsigned char IVstartpos, unsigned char *resultingIV) {
5
6      // Create space for coded header + checksum
7      unsigned char temp[42];
8
9      // Read coded header + checksum
10     fseek(filepoi, IVstartpos, SEEK_SET);
11     for(int j = 0; j < 42; j++) {
12         temp[j] = fgetc(filepoi);
13     }
14     // Print checksum
15     printf("\nCRC-16 Modus Checksum: 0x%02X %02X\n", temp[41], temp[42]);
16
17     // Get distributed IV bits according to the revealed mapping rules
18     unsigned char bit[64];
19     bit[63]=getbit(temp[16],3); bit[62]=getbit(temp[15],3); bit[61]=getbit(temp[14],3);
20     bit[60]=getbit(temp[13],3); bit[59]=getbit(temp[12],3); bit[58]=getbit(temp[11],3);
21     bit[57]=getbit(temp[10],3); bit[56]=getbit(temp[9],3); bit[55]=getbit(temp[24],3);
22     bit[54]=getbit(temp[23],3); bit[53]=getbit(temp[22],3); bit[52]=getbit(temp[21],3);
23     bit[51]=getbit(temp[20],3); bit[50]=getbit(temp[19],3); bit[49]=getbit(temp[18],3);
24     bit[48]=getbit(temp[17],3); bit[47]=getbit(temp[32],3); bit[46]=getbit(temp[31],3);
25     bit[45]=getbit(temp[30],3); bit[44]=getbit(temp[29],3); bit[43]=getbit(temp[28],3);
26     bit[42]=getbit(temp[27],3); bit[41]=getbit(temp[26],3); bit[40]=getbit(temp[25],3);
27     bit[39]=getbit(temp[0],4); bit[38]=getbit(temp[39],3); bit[37]=getbit(temp[38],3);
28     bit[36]=getbit(temp[37],3); bit[35]=getbit(temp[38],3); bit[34]=getbit(temp[35],3);
29     bit[33]=getbit(temp[34],3); bit[32]=getbit(temp[35],3); bit[31]=getbit(temp[8],4);
30     bit[30]=getbit(temp[7],4); bit[29]=getbit(temp[6],4); bit[28]=getbit(temp[5],4);
31     bit[27]=getbit(temp[6],4); bit[26]=getbit(temp[3],4); bit[25]=getbit(temp[2],4);
32     bit[24]=getbit(temp[1],4); bit[23]=getbit(temp[16],4); bit[22]=getbit(temp[15],4);
33     bit[21]=getbit(temp[14],4); bit[20]=getbit(temp[13],4); bit[19]=getbit(temp[12],4);
34     bit[18]=getbit(temp[11],4); bit[17]=getbit(temp[10],4); bit[16]=getbit(temp[9],4);
35     bit[15]=getbit(temp[24],4); bit[14]=getbit(temp[23],4); bit[13]=getbit(temp[22],4);
36     bit[12]=getbit(temp[21],4); bit[11]=getbit(temp[20],4); bit[10]=getbit(temp[19],4);
37     bit[9]=getbit(temp[18],4); bit[8]=getbit(temp[17],4); bit[7]=getbit(temp[32],4);
38     bit[6]=getbit(temp[31],4); bit[5]=getbit(temp[30],4); bit[4]=getbit(temp[29],4);
39     bit[3]=getbit(temp[28],4); bit[2]=getbit(temp[27],4); bit[1]=getbit(temp[26],4);
40     bit[0]=getbit(temp[25],4);

```

```

41
42 // Create 64-bit IV-Sequence
43 uint64_t receivedIV = 0x0000000000000000;
44 for(int i = 0; i < 64; i++) {
45     receivedIV = receivedIV^((uint64_t)bit[63-i] << (63-i));
46 }
47
48 // Store IV-Sequence bitwise
49 for (int i = 7; i >= 0; i--) {
50     resultingIV[i] = (receivedIV >> 64-(i+1)*8) & 0xFF;
51     resultingIV[8+i] = resultingIV[i];
52 }
53 }
54 int main() {
55     // Provide positions
56     long header_IVstartpos = 33;
57     long bitstream_startpos = 0x5285;
58     long bitstream_endpos = 0x90170+16;
59
60     // Load unencrypted (for comparison) and encrypted (for decryption) file
61     FILE *plaintext_pointer = fopen("unencrypted.rbf", "rb");
62     FILE *ciphertext_pointer = fopen("encrypted.rbf", "rb");
63
64     // If one file is not available stop program
65     if((plaintext_pointer == NULL) || (ciphertext_pointer == NULL)) { exit(0); }
66
67     // Set KEY1
68     unsigned char key1[16] = {
69         0x0F,0x0E,0x0D,0x0C,0x0B,0x0A,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00
70     };
71
72     // Set KEY2
73     unsigned char key2[16] = {
74         0x32,0x00,0x31,0xC9,0xFD,0x4F,0x69,0x8C,0x51,0x9D,0x68,0xC6,0x86,0xA2,0x43,0x7C
75     };
76
77     // Create variables
78     unsigned char realkey[16], IV[16], bitstream_plain_ref[16], bitstream_enc[16], AES_out[16],
79         bitstream_decrypted[16], state[16];
80
81     // Initialize variables to zero
82     for(int i = 0; i < 16; i++) realkey[i]=IV[i]=bitstream_plain_ref[i]=bitstream_enc[i]=AES_out[i]
83         =bitstream_decrypted[i]=state[i]=0;
84
85     // Bytewise little endian -> big endian
86     changeBytewiseEndianness(key1);
87     changeBytewiseEndianness(key2);
88
89     // Setup AES keysize and blocksize
90     int keysize=128;
91     int blocksize=128;
92
93     // Create and setup AES object
94     AES AESobj;
95     AESobj.setDimensions(keysize, blocksize);
96
97     // Key schedule KEY1
98     AESobj.keyschedule(key1);
99
100     // Encrypt KEY2 using KEY1 as key leading to real key
101     AESobj.encryptoneBlock(key2, realkey);

```



```

101 // KeySchedule realkey
102 AESObj.keyschedule(realkey);
103
104 // Read IV from encrypted.rbf file header
105 getIV(ciphertext_pointer, header_IVstartpos, IV);
106
107 int availableBlocks = (bitstream_endpos-bitstream_startpos)/16;
108 for(long i = 0; i <= availableBlocks; i++) {
109     // Reads current 16 byte block: bitstream_enc<->from encryptedRBF; bitstream_plain <-> from
110     // unencryptedRBF
111     setPlainsEncs(bitstream_enc, bitstream_plain_ref, bitstream_startpos+i*16, plaintext_pointer,
112     ciphertext_pointer);
113
114     // If initial IV: do nothing else increment
115     if(i != 0) {
116         // Increment IV (AES input)
117         IV[0] = IV[8] = (IV[0] + 1);
118         if(IV[0] == 0) {
119             IV[1] = IV[9] = IV[1]+1;
120             if(IV[1] == 0) {
121                 IV[2] = IV[10] = IV[2]+1;
122                 if(IV[2] == 0) {
123                     IV[3] = IV[11] = IV[3]+1;
124                 }
125             }
126         }
127     }
128
129     // Encrypt incremented IV and store result in AES_out
130     AESObj.encryptoneBlock(IV, AES_out);
131
132     // Finally decrypt block (like FPGA would do) XOR AES_out with encrypted block from RBF file
133     // Store result in bitstream_decrypted
134     do_xor(AES_out, bitstream_enc, bitstream_decrypted);
135
136     // If bitstream_plain_ref == bitstream_decrypted => decryption successful
137     int res = compare(bitstream_plain_ref, bitstream_decrypted);
138     if(res == 1) {
139         printf("\nDecrypted block %i successfully. ", i);
140     }
141     else {
142         printf("\nFailed to decrypt block %i ", i);getchar();
143     }
144 }
145
146 // Close file pointers
147 fclose(plaintext_pointer);
148 fclose(ciphertext_pointer);
149 }

```

A.2.2 Header of Unencrypted and Encrypted Stratix III RBF Files

Listing A.5: Comparing Stratix III header.

```

1 // Header of unencrypted Stratix III RBF file
2 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
3 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 6A F6
4 F7 F6 F7 F6 F7 F3 FA DB FB DB DB D9 D9 F9 FD F9 F9
5 FD FD FF FF F9 F9 F9 FD F9 FD F9 F9 F9 FD FD F9 EF
6 EB EB FB FF EF 98 19

```

```

7 // Header of encrypted Stratix III RBF file (1)
8 // Timestamp 0x507C808B = Monday, 2012-10-15 23:30:51
9 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
10 FF FF FF FF FF FF FF FF FF FF FF FF FF FF 6A E6
11 E7 E6 E7 E6 F7 E3 E2 CB FB CB D3 D9 D1 E1 E5 F9 F1
12 F1 E1 E3 F3 F1 F5 E9 E5 E1 FD F9 F9 E9 ED F5 F1 E7
13 E3 E3 FB F7 EF AC E0
14
15 // Header of encrypted Stratix III RBF file (2)
16 // Timestamp 0x507C808C = Monday, 2012-10-15 23:30:52
17 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
18 FF FF FF FF FF FF FF FF FF FF FF FF FF FF 6A E6
19 E7 E6 E7 E6 F7 E3 E2 CB F3 C3 DB D9 D1 E1 E5 F9 F1
20 F1 E1 E3 F3 F1 F5 E9 E5 E1 FD F9 F9 E9 ED F5 F1 E7
21 E3 E3 FB F7 EF 23 2A
22
23 // Difference between encrypted header (1) and (2)
24 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
25 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
26 00 00 00 00 00 00 00 00 08 08 08 00 00 00 00 00
27 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
28 00 00 00 00 00 8F CA

```

A.2.3 First Ten Dumped IVs of Stratix III for Monday, 2012-10-15 23:30:51

Listing A.6: Example of first ten dumped IVs of Stratix III.

```

1 8B 80 7C 50 10 9D 73 9C - 8B 80 7C 50 10 9D 73 9C (AES input 1 = initial IV)
2 11 90 0F 0C A4 F3 83 7F - 11 90 0F 0C A4 F3 83 7F (AES input 2)
3 02 F2 81 83 76 FE F4 2B - 02 F2 81 83 76 FE F4 2B (AES input 3)
4 40 3E 70 D4 CA 9F 77 4D - 40 3E 70 D4 CA 9F 77 4D (AES input 4)
5 C8 07 8E 5A F9 F3 AE 09 - C8 07 8E 5A F9 F3 AE 09 (AES input 5)
6 F9 C0 51 2B 7F DE 35 01 - F9 C0 51 2B 7F DE 35 01 (AES input 6)
7 1F 38 6A E7 CD 3B 22 24 - 1F 38 6A E7 CD 3B 22 24 (AES input 7)
8 03 47 ED B2 77 C7 9B F8 - 03 47 ED B2 77 C7 9B F8 (AES input 8)
9 E0 A8 5D F0 E8 F8 1E 73 - E0 A8 5D F0 E8 F8 1E 73 (AES input 9)
10 1C B5 0B 1E 1D DF 63 0E - 1C B5 0B 1E 1D DF 63 0E (AES input 10)

```

A.2.4 First Ten Encrypted Bitstream Blocks of Stratix III for Monday, 2012-10-15 23:30:51

Listing A.7: Example of first ten encrypted bitstream blocks of Stratix III.

```

1 56 D3 E4 7E D6 7B E6 23 CD B0 DA 89 D5 95 68 2A (encrypted bitstream block 1)
2 80 3E C8 0C 06 1B EB 47 37 D5 F5 E1 00 DF 6A C0 (encrypted bitstream block 2)
3 7C FB B5 87 39 0E A3 76 F4 ED B1 27 15 72 56 03 (encrypted bitstream block 3)
4 7B 96 E5 F1 31 6C D8 EC 76 C0 D9 0D 0B 83 1C B7 (encrypted bitstream block 4)
5 07 95 EF 58 D7 B9 5A 55 4A 66 88 77 B1 91 1C 04 (encrypted bitstream block 5)
6 36 0B E3 54 59 D6 E6 85 49 89 98 56 71 E7 56 77 (encrypted bitstream block 6)
7 69 8D C4 6D 94 39 9A B9 6E 4D E8 3F DB 36 44 12 (encrypted bitstream block 7)
8 A2 E9 A5 32 A5 1B 0C 5F 31 43 B2 7A D8 E8 C1 1D (encrypted bitstream block 8)
9 7E BC 6C F3 B7 4B C4 6F 61 72 C2 88 EB 27 88 72 (encrypted bitstream block 9)
10 6A BC 36 C6 DC A5 CD 7C FD E9 E1 5C 8C OD 53 12 (encrypted bitstream block 10)

```

A.2.5 First Ten Unencrypted Bitstream Blocks of Stratix III

Listing A.8: Example of first ten unencrypted bitstream blocks.

```

1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (unencrypted bitstream block 1)
2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (unencrypted bitstream block 2)
3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (unencrypted bitstream block 3)
4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (unencrypted bitstream block 4)
5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (unencrypted bitstream block 5)
6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (unencrypted bitstream block 6)
7 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (unencrypted bitstream block 7)
8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (unencrypted bitstream block 8)
9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (unencrypted bitstream block 9)
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (unencrypted bitstream block 10)

```

A.2.6 Decryption of an Encrypted Stratix III RBF File

Listing A.9: Code example of decrypting an encrypted Stratix III RBF file.

```

1 #include "aes.h"
2 ...
3 int main() {
4     // Provide positions
5     long bitstream_startpos = 0xD1A9;
6     long bitstream_endpos = 0x3162A0;
7
8     // Load unencrypted (for comparison), encrypted (for decryption), and dumped IV (for decryption)
9     // file
10    FILE *plaintext_pointer = fopen("unencrypted.rbf", "rb");
11    FILE *ciphertext_pointer = fopen("encrypted.rbf", "rb");
12    FILE *ivdump = fopen("IVdump.bin", "rb");
13
14    // If one file is not available stop program
15    if((plaintext_pointer==NULL)|| (ciphertext_pointer==NULL)|| ivdump==NULL){exit(0);}
16
17    // Set KEY1
18    unsigned char key1[32] = {
19        0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F,
20        0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F
21    };
22
23    // Set KEY2
24    unsigned char key2[32] = {
25        0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,0x3B,0x3C,0x3D,0x3E,0x3F,
26        0x40,0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F
27    };
28
29    // Create variables
30    unsigned char realkey[32], realkey_firsthalf[16], realkey_secondhalf[16];
31    unsigned char key2_firsthalf[16], key2_secondhalf[16];
32    unsigned char IV[16], AES_out[16], bitstream_decrypted[16], bitstream_plain_ref[16],
33        bitstream_enc[16];
34
35    // Initialize variables to zero
36    for(int i = 0; i < 16; i++) {
37        realkey[i]=realkey_firsthalf[i]=realkey_secondhalf[i]=0;
38        key2_firsthalf[i]=key2_secondhalf[i]=0;
39        IV[i]=AES_out[i]=bitstream_decrypted[i]=bitstream_plain_ref[i]=bitstream_enc[i]=0;
40    }

```

```

39
40 // Bytewise little endian -> big endian
41 changeBytewiseEndianness256(key1);
42 changeBytewiseEndianness256(key2);
43
44 // Split KEY2 into two halves
45 for(int j = 0; j <= 15; j++) {
46     key2_firsthalf[j] = key2[j];
47     key2_secondhalf[j] = key2[j+16];
48 }
49 // Setup AES keysize and blocksize
50 int keysize=256;
51 int blocksize=128;
52
53 // Setup AES object
54 AES AESobj;
55 AESobj.setDimensions(keysize, blocksize);
56
57 // Key schedule KEY1
58 AESobj.keyschedule(key1);
59
60 // Encrypt both KEY2 halves using KEY1 leading to the 256-bit real key
61 AESobj.encryptoneBlock(key2_firsthalf, realkey_firsthalf);
62 AESobj.encryptoneBlock(key2_secondhalf, realkey_secondhalf);
63
64 // Merge real key
65 for(int j = 0; j <= 15; j++) {
66     realkey[j] = realkey_firsthalf[j];
67     realkey[j+16] = realkey_secondhalf[j];
68 }
69
70 // KeySchedule realkey
71 AESobj.keyschedule(realkey);
72
73 int availableBlocks = (bitstream_endpos-bitstream_startpos)/16;
74 for(long i = 0; i <= availableBlocks; i++) {
75
76     // Read dumped IV
77     for(int j=0; j < 16; j++) {
78         IV[j] = fgetc(ivdump);
79     }
80
81     // Reads current 16 byte block: bitstream_enc<->from encryptedRBF; bitstream_plain <-> from
82     // unencryptedRBF
83     setPlainsEncs(bitstream_enc, bitstream_plain_ref, bitstream_startpos+i*16, plaintext_pointer,
84     ciphertext_pointer);
85
86     // Encrypt incremented IV and store result in AES_out
87     AESobj.encryptoneBlock(IV, AES_out);
88
89     // Finally decrypt block (like FPGA would do) XOR AES_out with encrypted block from RBF file
90     // Store result in bitstream_decrypted
91     do_xor(AES_out, bitstream_enc, bitstream_decrypted);
92
93     // If bitstream_plain_ref == bitstream_decrypted => decryption successful
94     int res = compare(bitstream_plain_ref, bitstream_decrypted);
95     if(res == 1) {
96         printf("\nDecrypted block %i successfully. ", i);
97     }
98     else {
99         printf("\nFailed to decrypt block %i ", i);getchar();
100     }

```

```

99     }
100
101     // Close file pointers
102     fclose(plaintext_pointer);
103     fclose(ciphertext_pointer);
104     fclose(ivdump);
105 }

```

A.2.7 Welford CPA

Listing A.10: Code example for Welford CPA.

```

1  ...
2  int main() {
3      /**=====
4      Include setup configuration
5      =====**/
6      #include "attack_setup.h"
7      /**=====
8      Determine parameters
9      =====**/
10     TRACELEN = endPoint - startPoint + 1;
11     KEY_CANDIDATES = endkey - startkey + 1;
12     N = endTrace - startTrace + 1;
13     GROUPS_COUNTED = N / TRACES_PER_GROUP;
14     TRACES_COUNTED = N % TRACES_PER_GROUP;
15
16     /**=====
17     Allocate memory
18     =====**/
19     variance_curve_trace = malloc(TRACELEN*sizeof(double));
20     mean_value_curve_trace = malloc(TRACELEN*sizeof(double));
21     CPA_curve = malloc(TRACELEN*sizeof(double));
22     mean_trace = malloc(TRACELEN*sizeof(double));
23     M2_trace = malloc(TRACELEN*sizeof(double));
24
25     CPA = malloc(KEY_CANDIDATES*MODEL_CANDIDATES*TRACELEN*sizeof(double));
26     denominator = malloc(KEY_CANDIDATES*MODEL_CANDIDATES*TRACELEN*sizeof(double));
27     numerator = malloc(KEY_CANDIDATES*MODEL_CANDIDATES*TRACELEN*sizeof(double));
28     c = malloc(KEY_CANDIDATES*MODEL_CANDIDATES*TRACELEN*sizeof(double));
29
30     mean_key = malloc(MODEL_CANDIDATES*KEY_CANDIDATES*sizeof(double));
31     M2_key = malloc(MODEL_CANDIDATES*KEY_CANDIDATES*sizeof(double));
32
33     #ifdef make1dotPlot4allcandidates
34         double *DOT_CPA = malloc(KEY_CANDIDATES*N*sizeof(double));
35     #endif
36
37     #if BINFILE_PRECISION == 1
38         trace_group = malloc(TRACES_PER_GROUP*TRACELEN*sizeof(double));
39         trace_temp = malloc(TRACES_PER_GROUP*TRACELEN*sizeof(double));
40     #else
41         trace_group = malloc(TRACES_PER_GROUP*TRACELEN*sizeof(signed char));
42         trace_temp = malloc(TRACES_PER_GROUP*TRACELEN*sizeof(signed char));
43     #endif
44
45     int i;
46     for(i = 0; i < TRACELEN; i++)
47         variance_curve_trace[i] = mean_value_curve_trace[i] = CPA_curve[i] = 0;
48     for(i = 0; i < KEY_CANDIDATES*MODEL_CANDIDATES*TRACELEN; i++)

```

```

49     CPA[i] = 0;
50     for(i = 0; i < TRACES_PER_GROUP*TRACELEN; i++)
51         trace_group[i] = trace_temp[i] = 0;
52
53     chall = create_2D_int_array(N+1, 16);
54     resp = create_2D_int_array(N+1, 16);
55     keyadd = create_3D_unsignedint_array(N+1, 10+1, 16);
56     subbytes = create_3D_unsignedint_array(N+1, 10+1, 16);
57     shiftrows = create_3D_unsignedint_array(N+1, 10+1, 16);
58     mixcolumn = create_3D_unsignedint_array(N+1, 9+1, 16);
59
60     /**=====
61     Setup correlation savepath
62     =====**/
63     setupCorrelationSavePath(fi);
64
65     /**=====
66     Output model path
67     =====**/
68     printOutputKonfiguration(testPredictions);
69     printf("\nPress any key to continue\n");getchar();
70
71     /**=====
72     Read challenges into memory
73     =====**/
74     readChallengesResponses();
75     int iNof;
76     int iTrace;
77     int iTime;
78     int iKey;
79     int iModel;
80     double PrediValue;
81
82     /**=====
83     Clear values to 0
84     =====**/
85     for(iTime = 0; iTime < TRACELEN; iTime++) {
86         mean_trace[iTime] = 0;
87         M2_trace[iTime] = 0
88         for(iModel = 0; iModel < MODEL_CANDIDATES; iModel++) {
89             for(iKey = 0; iKey < KEY_CANDIDATES; iKey++) {
90                 unsigned int current_model = MODEL_CANDIDATES*iKey+iModel;
91                 unsigned int current_model_timepoint = TRACELEN*(current_model)+iTime;
92                 mean_key[current_model]=0;
93                 M2_key[current_model]=0;
94                 c[current_model_timepoint]=0;
95                 CPA[current_model_timepoint]=0;
96                 denominator[current_model_timepoint]=0;
97                 numerator[current_model_timepoint]=0;
98             }
99         }
100     }
101     printf("%d groups", GROUPS_COUNTED);
102     printf("\n%d traces per group\n", TRACES_PER_GROUP);
103
104     for(iNof = 0; iNof < GROUPS_COUNTED; iNof++) {
105         readTraceGroup();
106         for(iTrace = 0; iTrace < TRACES_PER_GROUP; iTrace++) {
107             const int current_traceno = TRACES_PER_GROUP*iNof+iTrace;
108             if (current_traceno % 100 == 0)printf("\rN=%d", current_traceno);
109             double Delta_key;
110             for(iModel = 0; iModel < MODEL_CANDIDATES; iModel++) {

```

```

111     for(iKey = 0; iKey < KEY_CANDIDATES; iKey++) {
112         unsigned int current_model = MODEL_CANDIDATES*iKey+iModel;
113         /*=====
114         START >> update_m_M2_equations_key start
115         =====*/
116         PrediValue = getPredi(current_traceno, iKey+startkey, testPredictions[iModel]);
117         Delta_key = PrediValue - mean_key[current_model];
118         mean_key[current_model] += Delta_key / (double)((current_traceno+1));
119         M2_key[current_model] += Delta_key * (PrediValue - mean_key[current_model]);
120     }
121 }
122 /*=====
123 START >> update_c start
124 =====*/
125 for(iTime = 0; iTime < TRACELEN; iTime++) {
126     unsigned int current_trace_timepoint = TRACELEN*iTrace+iTime;
127     /*=====
128     START >> update_m_M2_equations_trace start
129     =====*/
130     double Delta_trace;
131     Delta_trace = (double) trace_group[current_trace_timepoint] - mean_trace[iTime];
132     mean_trace[iTime] += Delta_trace / (double)((current_traceno+1));
133     M2_trace[iTime] += Delta_trace * ((double) trace_group[current_trace_timepoint] -
        mean_trace[iTime]);
134
135     for(iModel = 0; iModel < MODEL_CANDIDATES; iModel++) {
136         for(iKey = 0; iKey < KEY_CANDIDATES; iKey++) {
137             unsigned int current_model = MODEL_CANDIDATES*iKey+iModel;
138             unsigned int current_model_timepoint = TRACELEN*(current_model)+iTime;
139             if(current_traceno > 1) {
140                 c[current_model_timepoint] += (getPredi(current_traceno, iKey+startkey,
                    testPredictions[iModel]) * trace_group[current_trace_timepoint] - c[
                    current_model_timepoint])/current_traceno;
141             }
142             else if(current_traceno == 1) {
143                 c[current_model_timepoint] += getPredi(current_traceno, iKey+startkey,
                    testPredictions[iModel]) * trace_group[current_trace_timepoint];
144             }
145             else if(current_traceno == 0) {
146                 c[current_model_timepoint] += getPredi(current_traceno, iKey+startkey,
                    testPredictions[iModel]) * trace_group[current_trace_timepoint];
147             }
148             /*=====
149             START >> update_cpa_coeff start
150             =====*/
151             numerator[current_model_timepoint] = (current_traceno-1)*c[
                current_model_timepoint] - current_traceno* mean_trace[iTime] * mean_key[
                current_model];
152             denominator[current_model_timepoint] = sqrt(M2_key[current_model] * M2_trace[
                iTime]);
153             if ((fabs(denominator[current_model_timepoint]) < 1e-15) || (denominator[
                current_model_timepoint] == 0)) {
154                 CPA[current_model_timepoint] = 0;
155             }
156             else {
157                 CPA[current_model_timepoint] = numerator[current_model_timepoint]/denominator[
                current_model_timepoint];
158             }
159             #ifdef make1dotPlot4allcandidates
160                 if(iTime==0) DOT_CPA[N*current_model+current_traceno] = CPA[
                current_model_timepoint];
161             #endif

```

```

162         }
163     }
164 }
165 }
166 }
167
168 /*=====
169 CPA curve storage
170 =====*/
171 for(iModel = 0; iModel < MODEL_CANDIDATES; iModel++) {
172     for(iKey = 0; iKey < KEY_CANDIDATES; iKey++) {
173         for(iTime = 0; iTime < TRACELEN; iTime++) {
174             CPA_curve[iTime] = CPA[TRACELEN*(MODEL_CANDIDATES*iKey+iModel)+iTime];
175         }
176         IntToString(startkey+iKey, cpa_curve_name);
177         strcat(cpa_curve_name, "key_");
178         strcat(cpa_curve_name, "predictionmodel_");
179         IntToString(testPredictions[iModel], reveal_keybytename);
180         strcat(cpa_curve_name, reveal_keybytename);
181         store_curve_double(CPA_curve, cpa_curve_name, TRACELEN);
182     }
183 }
184
185 #ifdef make1dotPlot4allcandidates
186     IntToString(N, cpa_curve_name);
187     strcat(cpa_curve_name, "predictionmodel_");
188     IntToString(testPredictions[0], reveal_keybytename);
189     strcat(cpa_curve_name, reveal_keybytename);
190     store_curve_double(DOT_CPA, cpa_curve_name, N*KEY_CANDIDATES*MODEL_CANDIDATES);
191 #endif
192
193 /*=====
194 Store mean_vale_curve_trace and variance_curve_trace
195 =====*/
196 store_curve_double(mean_value_curve_trace, "mean_value_curve_trace", TRACELEN);
197 store_curve_double(variance_curve_trace, "variance_curve_trace", TRACELEN);
198
199 /*=====
200 Clear memory
201 =====*/
202 free(variance_curve_trace);
203 free(mean_value_curve_trace);
204 free(CPA_curve);
205 free(CPA);
206 free(denominator);
207 free(mean_trace);
208 free(M2_trace);
209 free(mean_key);
210 free(M2_key);
211 #ifdef make1dotPlot4allcandidates
212     free(DOT_CPA);
213 #endif
214 free(trace_group);
215 free(trace_temp);
216 free(chall);
217 free(resp);
218 free(keyadd);
219 free(subbytes);
220 free(shiftrows);
221 free(mixcolumn);
222
223 /*=====

```



```

224     End of program
225     =====*/
226     printf("\n\nFinished!");
227
228     return 0;
229 }

```

A.2.8 Prediction Models

Listing A.11: Prediction model examples.

```

1  /**=====
2  Returns a prediction
3  =====*/
4  double getPredi(int challno, int p, int predictionModel) {
5      switch(predictionModel) {
6          // Plaintext search Hamming Weight model using byte "plain_byte" (1 byte)
7          case 1: { return getHW[chall[challno][plain_byte]]; } break;
8
9          // Plaintext search Hamming Weight model using byte 1 and 0 (2 bytes)
10         case 2: { return (getHW[chall[challno][1]]+getHW[chall[challno][0]]); } break;
11
12         // Plaintext search Hamming Weight model using byte 0-3 (4 bytes)
13         case 5: {
14             double tmp = 0;
15             for(int byte = 0; byte < 4; byte++)
16                 tmp += getHW[chall[challno][byte]];
17             return tmp;
18         }
19         break;
20
21         // Plaintext search Hamming Weight model using byte 0-7 (8 bytes)
22         case 24: {
23             double tmp = 0;
24             for(int byte = 0; byte < 8; byte++)
25                 tmp += getHW[chall[challno][byte]];
26             return tmp;
27         }
28         break;
29
30         // Plaintext search Hamming Weight model using byte 0-15 (128-bit full state)
31         case 25: {
32             double tmp = 0;
33             for(int byte = 0; byte < 16; byte++)
34                 tmp += getHW[chall[challno][byte]];
35             return tmp;
36         }
37         break;
38
39         // Plaintext search bit model (bit 0 - bit 7) using byte 15
40         case 26: { return getbit7(chall[challno][BYTE15]); } break;
41         case 27: { return getbit6(chall[challno][BYTE15]); } break;
42         case 28: { return getbit5(chall[challno][BYTE15]); } break;
43         case 29: { return getbit4(chall[challno][BYTE15]); } break;
44         case 30: { return getbit3(chall[challno][BYTE15]); } break;
45         case 31: { return getbit2(chall[challno][BYTE15]); } break;
46         case 32: { return getbit1(chall[challno][BYTE15]); } break;
47         case 33: { return getbit0(chall[challno][BYTE15]); } break;
48
49

```

```

50 // Plaintext search Hamming Distance model (1 byte)
51 case 50: { return getHD(chall[challno][1], chall[challno][0]); } break;
52 case 59: { return getHD(chall[challno][2], chall[challno][1]); } break;
53 case 60: { return getHD(chall[challno][3], chall[challno][2]); } break;
54 case 61: { return getHD(chall[challno][4], chall[challno][3]); } break;
55 case 62: { return getHD(chall[challno][5], chall[challno][4]); } break;
56 case 63: { return getHD(chall[challno][6], chall[challno][5]); } break;
57 case 64: { return getHD(chall[challno][7], chall[challno][6]); } break;
58
59 // AES S-box bit model (bit 0 - bit 7) using byte "plain_byte"
60 case 100: { return getbit0(AES_SBOX(chall[challno][plain_byte], p)); } break;
61 case 101: { return getbit1(AES_SBOX(chall[challno][plain_byte], p)); } break;
62 case 102: { return getbit2(AES_SBOX(chall[challno][plain_byte], p)); } break;
63 case 103: { return getbit3(AES_SBOX(chall[challno][plain_byte], p)); } break;
64 case 104: { return getbit4(AES_SBOX(chall[challno][plain_byte], p)); } break;
65 case 105: { return getbit5(AES_SBOX(chall[challno][plain_byte], p)); } break;
66 case 106: { return getbit6(AES_SBOX(chall[challno][plain_byte], p)); } break;
67 case 107: { return getbit7(AES_SBOX(chall[challno][plain_byte], p)); } break;
68
69 // Initial keyadd Hamming Weight model key^plain_byte (1 byte)
70 case 108: { return getHW[xor_lut[chall[challno][plain_byte]][p]]; } break;
71
72 // First S-box round Hamming Weight model of S-box(key^plain_byte) (1 byte)
73 case 109: { return getHW[AES_SBOX(chall[challno][plain_byte], p)]; } break;
74
75 // First round Hamming Distance of key^plain_byte<->S-box(key^plain_byte) (1 byte)
76 case 110: { return getHD(xor_lut[chall[challno][plain_byte]][p], AES_SBOX(chall[challno][
    plain_byte], p)); } break;
77
78 // Search full keyadd state Hamming Weight of round X (128 bit)
79 case 250: {
80     double tmp = 0;
81     for(int byte = 0; byte < 16; byte++)
82         tmp += getHW[keyadd[challno][round_X][byte]];
83     return tmp;
84 }
85 break;
86
87 // Search full S-box state Hamming Weight of round X (128 bit)
88 case 251: {
89     double tmp = 0;
90     for(int byte = 0; byte < 16; byte++)
91         tmp += getHW[subbytes[challno][round_X][byte]];
92     return tmp;
93 }
94 break;
95
96 // Search full MixColumns state Hamming Weight of round X (128 bit)
97 case 253: {
98     double tmp = 0;
99     for(int byte = 0; byte < 16; byte++)
100         tmp += getHW[mixcolumn[challno][round_X][byte]];
101     return tmp;
102 }
103 break;
104
105 // Search byte 15 of Keyadd, S-box, or MixColumns Hamming Weight of round X
106 case 266: { return (getHW[keyadd[challno][round_X][BYTE15]]); } break;
107 case 267: { return (getHW[subbytes[challno][round_X][BYTE15]]); } break;
108 case 269: { return (getHW[mixcolumn[challno][round_X][BYTE15]]); } break;
109
110

```

```

111 // Search toggling Keyadd state Hamming Distance of round X->Y (128-bit)
112 case 290: {
113     double tmp = 0;
114     for(int byte = 0; byte < 16; byte++)
115         tmp += getHD(keyadd[challno][round_X][byte], keyadd[challno][round_Y][byte]);
116     return tmp;
117 }
118 break;
119
120 // Search toggling S-box state Hamming Distance of round X->Y (128-bit)
121 case 291: {
122     double tmp = 0;
123     for(int byte = 0; byte < 16; byte++)
124         tmp += getHD(subbytes[challno][round_X][byte], subbytes[challno][round_Y][byte]);
125     return tmp;
126 }
127 break;
128
129 // Search toggling ShiftRows state Hamming Distance of round X->Y (128-bit)
130 case 292: {
131     double tmp = 0;
132     for(int byte = 0; byte < 16; byte++)
133         tmp += getHD(shiftrows[challno][round_X][byte], shiftrows[challno][round_Y][byte]);
134     return tmp;
135 }
136 break;
137
138 // Search toggling MixColumns state Hamming Distance of round X->Y (128-bit)
139 case 293: {
140     double tmp = 0;
141     for(int byte = 0; byte < 16; byte++)
142         tmp += getHD(mixcolumn[challno][round_X][byte], mixcolumn[challno][round_Y][byte]);
143     return tmp;
144 }
145 break;
146
147 // Search toggling keyadd->Sbox state Hamming Distance of round X (128-bit)
148 case 302: {
149     double tmp = 0;
150     for(int byte = 0; byte < 16; byte++)
151         tmp += getHD(keyadd[challno][round_X][byte], subbytes[challno][round_x][byte]);
152     return tmp;
153 }
154 break;
155
156 // Search toggling keyadd->Sbox state Hamming Distance of round X (64-bit)
157 case 303: {
158     double tmp = 0;
159     for(int byte = 8; byte < 16; byte++)
160         tmp += getHD(keyadd[challno][round_X][byte], subbytes[challno][round_x][byte]);
161     return tmp;
162 }
163 break;
164
165 // Search toggling keyadd->Sbox state Hamming Distance of round X (32-bit)
166 case 304: {
167     double tmp = 0;
168     for(int byte = 12; byte < 16; byte++)
169         tmp += getHD(keyadd[challno][round_X][byte], subbytes[challno][round_x][byte]);
170     return tmp;
171 }
172 break;

```

```

173
174 // Search toggling keyadd->Sbox state Hamming Distance of round X (16-bit)
175 case 305: {
176     double tmp = 0;
177     for(int byte = 15; byte < 16; byte++)
178         tmp += getHD(keyadd[challno][round_X][byte], subbytes[challno][round_x][byte]);
179     return tmp;
180 }
181 break;
182
183 // Search toggling keyadd->Sbox state of byte 15 Hamming Distance of round X (8-bit)
184 case 306: {
185     return (getHD(keyadd[challno][round_X][BYTE15], subbytes[challno][round_X][BYTE15]));
186 }
187 break;
188
189 // Search bit 0 bit model of {Keyadd, S-box, MixColumns} byte 15 of round X
190 case 310: { return getbit0(keyadd[challno][round_X][BYTE15]);} break;
191 case 318: { return getbit0(subbytes[challno][round_X][BYTE15]);} break;
192 case 326: { return getbit0(mixcolumn[challno][round_X][BYTE15]);} break;
193
194 // Search bit {0,1,2,3,4,5,6,7} bit model of S-box byte 15
195 case 540: { return getbit7(AES_SBOX(chall[challno][15], used_key[15]));} break;
196 case 541: { return getbit6(AES_SBOX(chall[challno][15], used_key[15]));} break;
197 case 542: { return getbit5(AES_SBOX(chall[challno][15], used_key[15]));} break;
198 case 543: { return getbit4(AES_SBOX(chall[challno][15], used_key[15]));} break;
199 case 544: { return getbit3(AES_SBOX(chall[challno][15], used_key[15]));} break;
200 case 545: { return getbit2(AES_SBOX(chall[challno][15], used_key[15]));} break;
201 case 546: { return getbit1(AES_SBOX(chall[challno][15], used_key[15]));} break;
202 case 547: { return getbit0(AES_SBOX(chall[challno][15], used_key[15]));} break;
203
204 // Test whether a 32-byte value is processed for shifting the ShiftRows states
205 case 970: {
206     return (
207         getHD(shiftrows[challno][round_X][BYTE7], shiftrows[challno][round_X][BYTE3])
208         + getHD(shiftrows[challno][round_X][BYTE6], shiftrows[challno][round_X][BYTE2])
209         + getHD(shiftrows[challno][round_X][BYTE5], shiftrows[challno][round_X][BYTE1])
210         + getHD(shiftrows[challno][round_X][BYTE4], shiftrows[challno][round_X][BYTE0])
211     );
212 }
213 break;
214
215 // Test whether a 8-byte value is processed for shifting the ShiftRows states
216 case 970: {
217     return (
218         getHD(shiftrows[challno][round_X][BYTE7], shiftrows[challno][round_X][BYTE3])
219     );
220 }
221 break;
222
223 default:{
224     printf("\nModel not found!");
225     exit(1);
226 }
227 }
228
229 return 0;
230 }

```

A.2.9 μ C Configuration of the First Three 16-byte Bitstream Blocks

Listing A.12: Configuration of the first three 16-byte blocks.

```

1 int main(void){
2
3     // Setup signals SIG_DCLK, SIG_NCONFIG, SIG_DATA0, and SIG_TRIGGER as output
4     DDRC=(1<<SIG_DCLK)|(1<<SIG_NCONFIG)|(1<<SIG_DATA0)| (1 << SIG_TRIGGER);
5
6     // Init uART for communication
7     uart_init();
8
9     // Tell PC about it
10    uart_putc('L');
11
12    // A new measurement starts
13    reset:
14
15    // Perform a passive serial initialization
16    PSConfig_init();
17
18    // Configuration can be started
19    // Transfer Pre-Header bytes from microcontroller flash to FPGA
20    fromFlash_PreHeader2FPGA();
21
22    // Request Coded Header carrying random. IV bits
23    uart_putc('H');
24
25    // Forward received Coded Header bytes to FPGA
26    fromhostPC_IVHeader2FPGA();
27
28    // Transfer fixed middle part from microcontroller flash to FPGA
29    fromFlash_Middle2FPGA();
30
31    // Define helper variables
32    char index1,index2;
33    char tempdata;
34    char AllX;
35    int global = 0;
36
37    // Obtain one 16-byte block from PC
38    while(bs_length>16)
39    {
40        // Request one 16-byte block from PC
41        uart_putc('R');
42        AllX=1;
43
44        // Fill the bitstream bytes into array c
45        for(index1=0;index1<16;index1++) {
46            c[index1+global]=uart_getc();
47
48            if (c[index1+global] != 'X'){
49                AllX=0;
50            }
51            global++;
52        }
53
54        // If the PC sends 16 times 'X' => done
55        if (AllX){
56            goto got_all_bytes;
57        }
58        bs_length -= 16;

```

```

59     }
60     got_all_bytes:
61
62     // Trigger once
63     #ifdef trigger_bitstream
64         setBit(PORTC, SIG_TRIGGER);
65         clrBit(PORTC, SIG_TRIGGER);
66     #endif
67
68     // Start to configure bitstream bytes
69     unsigned int test;
70     for(test=0;test<global-17;test++){
71         tempdata=c[test];
72         byte2fpga(tempdata);
73     }
74
75     // New measurement can be done
76     goto reset;
77 }
78
79 void PSConfig_init(void) {
80     int i;
81
82     // Init signal states
83     clrBit(PORTC,SIG_TRIGGER);
84     clrBit(PORTC,SIG_DCLK);
85     clrBit(PORTC,SIG_DATA0);
86     setBit(PORTC,SIG_NCONFIG);
87
88     // Ensure that nConfig was set for a certain time to "1"
89     setBit(PORTC,SIG_NCONFIG);
90     for(i=0;i<10000;i++){__asm__("nop");}
91
92     // Signalize to FPGA that we start a new configuration
93     clrBit(PORTC,SIG_NCONFIG);
94
95     // FPGA gets enough time to pull down CONF_DONE and nSTATUS
96     for(i=0;i<10000;i++){__asm__("nop");}
97
98     // Set nCONFIG back to 1
99     setBit(PORTC,SIG_NCONFIG);
100
101     // Provide nSTATUS enough time to get pulled up
102     for(i=0;i<10000;i++){__asm__("nop");}
103 }
104
105 // Transfer of fixed Pre-Header
106 void fromFlash_PreHeader2FPGA(void) {
107     char index;
108     for (index=0; index < 32; index++) {
109         byte2fpga(0xFF);
110     }
111     byte2fpga(0x00);
112 }
113
114 // Transfer of Coded Header with valid checksum
115 void fromhostPC_IVHeader2FPGA(void) {
116     char temp[42];
117     for (char index=0; index < 42; index++) {
118         if (((index % 16) == 0) && (index != 0))
119         {
120             uart_putc('h');

```

```

121     }
122     temp[index]=uart_getc();
123 }
124
125 for (char index=0; index < 42; index++) {
126     byte2fpga(temp[index]);
127 }
128 }
129
130 // Transfer middle part
131 void fromFlash_Middle2FPGA(void) {
132     char temp;
133     for(unsigned long int i=0; i < middleLen; i++) {
134         temp = pgm_read_byte(&middle[i]);
135         byte2fpga(temp);
136     }
137 }
138
139 // Clock one byte to FPGA
140 void byte2fpga(char tempdata) {
141     char index2;
142     for (index2=0;index2<8;index2++) {
143         if (tempdata & 0x01) {
144             setBit(PORTC,SIG_DATA0);
145         }
146         else {
147             clrBit(PORTC,SIG_DATA0);
148             asm("nop");
149         }
150         clock();
151         tempdata >>= 1;
152     }
153 }

```

A.2.10 Matlab Scripts

Listing A.13: Code example for filtering one unfiltered trace.

```

1 // Specify Megasamples/sec
2 fs = 500E6;
3
4 // Lowpass filter
5 Fpass=1e7;
6 Fstop=1e8;
7 Apass=1;
8 Astop=60;
9 FilterDesign1=fdesign.lowpass(Fpass,Fstop,Apass=1,Astop,fs);
10 normalizefreq(FilterDesign1);
11 FilterHandle1 = design(FilterDesign1,'cheby1');
12 %h2=fvtool(FilterHandle2);
13
14 // Bandpass filter
15 Fpass=50e3;
16 Fstop1=150e3;
17 Fstop2=250e3;
18 Fpass2=400e3;
19 Apass1=1;
20 Astop=60;
21 Apass2=1;
22 FilterDesign2=fdesign.bandstop(Fpass,Fstop1,Fstop2,Fpass2, Apass1,Astop,Apass2,fs); %

```

```

23 normalizefreq(FilterDesign2);
24 FilterHandle2 = design(FilterDesign2,'cheby1');
25 %h2=fvtool(FilterHandle2);
26
27 // Get one trace
28 Trace_unfiltered = getTrace('Traces_1.dat', 'int8');
29
30 // Pass trace through lowpass filter
31 Trace_lowpass=filter(FilterHandle1,Trace_unfiltered);
32
33 // Pass lowpass-filtered trace through bandpass filter
34 Trace_lowpass_bandpass=filter(FilterHandle2,Trace_lowpass);

```

A.2.11 Peak Extraction

Listing A.14: Code example for peak extraction.

```

1 function [maxima] = peaksearch(t1, peaks)
2     windowSize = 500;
3     bigWindowShift = 1600;
4     smallWindowShift = 1300;
5     maxima = zeros(1, peaks);
6
7     windowStart = 1.741E4;
8     windowEnd = 1.766E4;
9
10    for j=1:peaks
11        current_max = -100;
12        for i=windowStart:windowEnd
13            if (t1(i) > current_max)
14                current_max = t1(i);
15                max_at = i;
16            end
17
18        maxima(j) = max_at;
19        if (j < 702)
20            if(mod((j+2),3)==0)
21                windowStart = max_at + bigWindowShift;
22                ; big window
23            elseif (mod(j+2,3)==1)
24                windowStart = windowEnd;
25                ; small window
26            else
27                windowStart = max_at + smallWindowShift;
28                ; middle window
29            end
30            windowEnd = windowStart + windowSize;
31        elseif (702 <= j <= 709)
32            if(j == 702)
33                windowStart = 8.850E5;
34                windowEnd = 8.853E5;
35            elseif(j==703)
36                windowStart = windowEnd;
37                windowEnd = 8.858E5;
38            elseif(j==704)
39                windowStart = 8.887E5;
40                windowEnd = 8.888E5;
41            elseif(j==705)
42                windowStart = windowEnd;
43                windowEnd = 8.894E5;

```



```

44     elseif(j==706)
45         windowStart = windowEnd;
46         windowEnd = 8.8929E5;
47     elseif(j==707)
48         windowStart = 8.924E5;
49         windowEnd = 8.927E5;
50     elseif(j==708)
51         windowStart = windowEnd;
52         windowEnd = 8.929E5;
53     elseif(j==709)
54         windowStart = 8.942E5;
55         windowEnd = 8.945E5;
56     end
57 end
58 if (j > 709)
59     if(mod((j+2),3)==1)
60         windowStart = max_at + bigWindowShift;
61     elseif (mod(j+2,3)==2)
62         windowStart = windowEnd;
63     else
64         windowStart = max_at + smallWindowShift;
65     end
66     windowEnd = windowStart + windowSize;
67 end
68 end
69 end

```

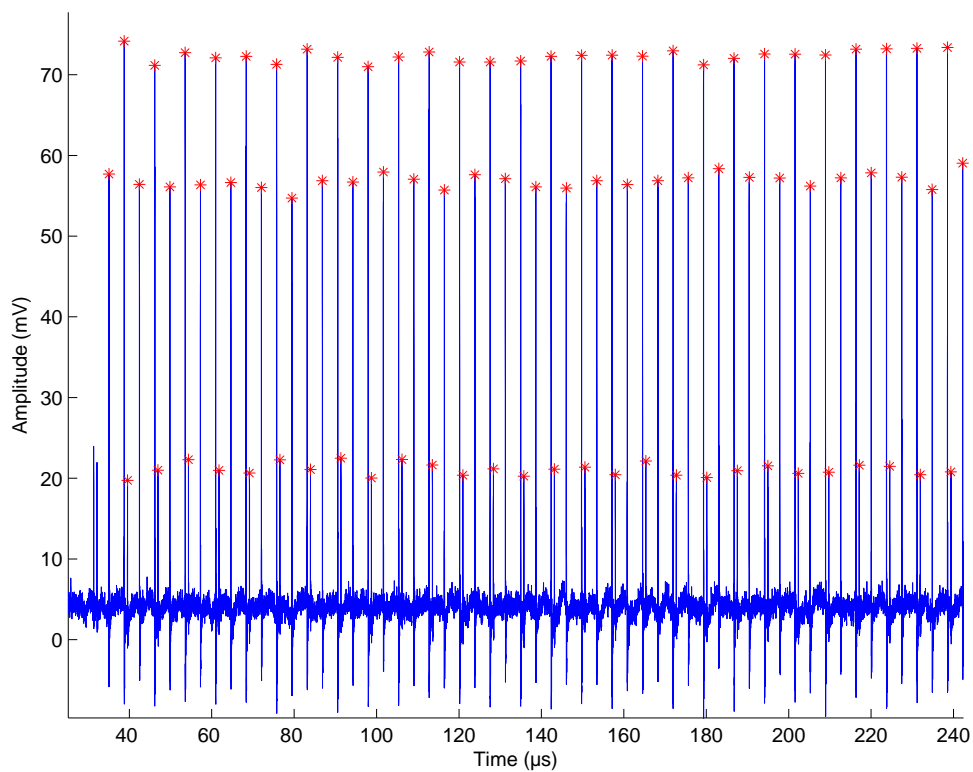


Figure A.1: Peak extraction example for one power trace.

Bibliography

- [AIS08] 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.
- [AN309] 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>.
- [BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In *CHES 2004*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004.
- [BPT10] Alessandro Barenghi, Gerardo Pelosi, and Yannick Teglia. Improving first order differential power attacks through digital signal processing. In *ACM-SIGSAC International Conference on Security of Information and Networks*, pages 124–133. ACM, 2010.
- [Cora] Altera Corporation. Design Security . <http://www.altera.com/products/devices/stratix-fpgas/about/security/stx-design-security.html>.
- [Corb] Altera Corporation. Stratix III FPGA: Lowest Power, Highest Performance 65-nm FPGA. <http://www.altera.com/devices/fpga/stratix-fpgas/stratix-iii/st3-index.jsp>.
- [CRC] On-line CRC calculation and free library. <http://www.lammertbies.nl/comm/info/crc-calculation.html>.
- [DSB] Defense Science Board. <http://www.acq.osd.mil/dsb/>.
- [EKM⁺] 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*, volume 5157 of *LNCS*, pages 203–220. Springer.
- [gre] Grep for Windows. <http://gnuwin32.sourceforge.net/packages/grep.htm>.
- [GTC05] C.H. Gebotys, C.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.

- [IDA] Hex-Rays SA. <http://www.hex-rays.com>.
- [KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO 99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
- [KOP09] Timo Kasper, David Oswald, and Christof Paar. EM Side-Channel Attacks on Commercial Contactless Smartcards using Low-Cost Equipment. In *WISA '09*, to appear in Springer LNCS, 2009.
- [Kru04] Ralf 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.
- [MBKP11] Amir Moradi, Alessandro Barengi, Timo Kasper, and Christof 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.
- [Mica] Microsoft. DUMPBIN Reference. <http://msdn.microsoft.com/en-ca/library/c1h23y6c%28v=vs.80%29.aspx>.
- [Micb] Microsoft. Linking an Executable to a DLL. <http://msdn.microsoft.com/en-us/library/9yd93633.aspx>.
- [Micc] Microsoft. Linking Explicitly. <http://msdn.microsoft.com/en-us/library/784bt7z7.aspx>.
- [MKP12] Amir Moradi, Markus Kasper, and Christof 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.
- [MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
- [NIS01] NIST. *Recommendation for Block 2001 Edition Cipher Modes of Operation*, 2001. <http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>.
- [OP11] David Oswald and Christof 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.
- [PHF08] Thomas Plos, Michael Hutter, and Martin Feldhofer. Evaluation of Side-Channel Preprocessing Techniques on Cryptographic-Enabled HF and UHF RFID-Tag Prototypes. In *RFIDSec 2008*, pages 114–127, 2008.
- [Ste10] Steve P. Miller. Dependency Walker 2.2, August 2010. <http://www.dependencywalker.com/>.

- [Str07] Stratix II Device Handbook, Volume 1. Technical report, Altera, 2007.
http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf.
- [SW12] Sergei Skorobogatov and Christopher Woods. In the blink of an eye: There goes your AES key. Cryptology ePrint Archive, Report 2012/296, 2012.
<http://eprint.iacr.org/>.
- [Tse05] Chen Wei Tseng. Lock Your Designs with the Virtex-4 Security Solution. XCell Journal, Xilinx, Spring 2005.