Bitstream-based Attacks against Reconfigurable Hardware

Dissertation

---

zur Erlangung des Grades eines Doktor-Ingenieurs
der Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

---

by Pawel Swierczynski
Bochum, July 2017

To my beloved family.

Pawel Swierczynski
Place of birth: Tychy, Poland
Author's contact information:
pawel.swierczynski@rub.de
www.emsec.rub.de

Thesis Advisor:	**Prof. Dr.-Ing. Christof Paar** Ruhr-Universität Bochum, Germany
Secondary Referee:	**Prof. Russell Tessier** University of Massachusetts Amherst, MA, USA
Thesis submitted:	July 18, 2017
Thesis defense:	September 29, 2017
Last revision:	July 09, 2018

Abstract

Over the last three decades, Field Programmable Gate Arrays (FPGAs) have developed into sophisticated re-programmable hardware devices and have become a central component in numerous information and communication systems. In 2010, more than four billion FPGA devices were shipped world-wide to various customers. SRAM-based FPGAs are widely used in safety-critical applications including aerospace, health, military equipment, automotive, computer networks, data centers, and many other industries.

Many of these applications are security-sensitive and need cryptographic operations such as random number generation, key establishment, digital signatures as well as encryption to ensure security services such as integrity, authenticity, and confidentiality. Therefore, we see an increased use of FPGAs for security-relevant tasks or in avionic systems such as satellites, the Boeing 787 Dreamliner, and the NASA Mars Rover.

As part of the revelations by Edward Snowden, it became public knowledge that the US intelligence agency, e.g., the National Security Agency (NSA) intercepts communication equipment during shipment in order to install malicious backdoors. Such firmware manipulations were for example carried out on Cisco routers enabling the NSA to entirely monitor targeted data networks. To achieve this, parts of the firmware were first reverse-engineered and then replaced. The manipulation of micro-controllers and software in general, e.g., the x86 or ARM architecture, occurs daily and its methods are well studied and advanced. In contrast, no public reports describing compromises of cryptographic hardware configurations of FPGAs – relying on manipulating bitstreams – have been available. A bitstream file, the encoded circuit of an FPGA, contains a description of how to configure all internal hardware elements during initial power-up. It can describe any conceivable hardware circuit that can execute arbitrary functionality. Given an adversary who obtains unauthorized access to a third-party bitstream describing a cryptographic circuit, e.g., through eavesdropping the data configuration bus or reading out the external non-volatile memory chip, the research question arises whether and specifically how a bitstream manipulation can lead to key leakage or Trojan insertion. Since the bitstream file format is proprietary and official extraction tools are unavailable, at first glance manipulating a cryptographic hardware configuration seems to be practically infeasible. This is one of various reasons why no meaningful manipulations have been presented to date.

This hurdle is also present even if the entire human-readable representation of a hardware configuration is available. A reason for this is that an adversary needs to reverse-engineer complex hardware structures that can consist of hundreds of thousands of interconnections, look-up tables, and flip-flops, appearing as unstructured mass of gates.

In general, to protect against Intellectual Property (IP) cloning and potential bitstream manipulations, the market leaders Xilinx and Altera offer a bitstream encryption feature that is supposed to ensure confidentiality and integrity. A problem with those schemes is that the bitstream encryption key of most deployed FPGAs can be disclosed for example by means of side-channel attacks. A leaked bitstream encryption key leads to the complete loss of confidentiality

and integrity making IP cloning a straightforward task and malicious bitstream manipulations a practical threat. In particular the latter issue is a (mostly) unattended research topic.

To close this research gap, this thesis demonstrates that non-invasive and targeted bitstream manipulations are indeed feasible and a powerful tool to compromise the security of cryptographic implementations. In this work, the first successful manipulations of third-party bitstreams are presented, which render the security of cryptographic block ciphers useless. More precisely, the detection and manipulation methods described in this thesis allow for either extracting secret keys or weakening the cryptographic strength of the AES block cipher. To highlight the practical relevance and applicability of bitstream manipulation attacks, the first real-world FPGA Trojan insertion is presented by targeting a commercially available embedded device through tampering with its AES module in the revealed bitstream. The device-under-attack is a FIPS-140-2 level 2 certified high-security USB flash drive from Kingston.

Targeted bitstream manipulation attacks, particularly those relying on prior detection of relevant cryptographic primitives, already offer a high success rate, but are not guaranteed to succeed. To further demonstrate the feasibility of a more generic AES key recovery technique, we present a novel and efficient fault injection approach on third-party bitstreams, which succeeds even if the targeted bitstreams are encrypted. This technique provides a higher success rate for recovering the cryptographic key of various AES IP cores, requiring no prior localization of the AES module within the bitstream. The attack works regardless of the underlying hardware architecture. Consequently, it can avoid the use of expensive laser setups or the reverse-engineering of the underlying hardware design, which may be as complex as reverse-engineering an Application Specific Integrated Circuit (ASIC).

In summary, in this dissertation we evaluate to what extent non-invasive bitstream manipulations can compromise system security relying on SRAM-based FPGAs.

Keywords.

Cryptography, bitstream encryption, Altera Stratix III, Spartan 6, Xilinx Virtex 5, FPGA security, reconfigurable hardware, bitstream manipulation, reverse-engineering, hardware Trojans, key recovery, fault injection, AES, IP protection, Kingston, real-world attack, countermeasures.

Kurzfassung

Bitstreambasierte Angriffe gegen rekonfigurierbare Hardware

In den letzten drei Jahrzehnten haben sich Field Programmable Gate Arrays (FPGAs) zu fortgeschrittenen re-programmierbaren Hardwarebausteinen entwickelt und wurden zu elementaren Komponenten für zahlreiche Informations- und Kommunikationssysteme. Im Jahr 2010 wurden weltweit mehr als vier Milliarden solcher Systeme ausgeliefert. SRAM-basierte FPGAs werden weitgehend in Anwendungen wie der Luft- und Raumfahrt, dem Gesundheitswesen, dem Militärbereich, der Automobilindustrie und in Computernetzwerken sowie Datenzentren genutzt.

Viele dieser Anwendungen sind sicherheitskritisch und benötigen deshalb kryptographische Operationen beispielsweise zur Generierung von Zufallszahlen, zum Schlüsselaustausch, zur Generierung von digitalen Signaturen oder zur Verschlüsselung von Daten. Dies ermöglicht die Sicherheitsanforderungen wie Integrität, Authentizität, und Vertraulichkeit zu erfüllen. Deshalb sehen wir in der Praxis einen erhöhten Einsatz von FPGAs, die sicherheitsrelevante Aufgaben übernehmen und in Avionik Systemen wie Satelliten, der Boeing 787 (Dreamliner) oder dem Mars Rover der NASA genutzt werden.

Ein Teil der Enthüllungen durch Edward Snowden hat gezeigt, dass der US-amerikanische Geheimdienst National Security Agency (NSA) verschiedene Übertragungsgeräte während der Warensendung abfängt, um Hintertüren einzubauen. Beispielsweise führten Firmwaremanipulationen an Cisco-Routern zu einer uneingeschränkten Überwachung gezielter Datennetzwerke durch die NSA. Konkret wurde die Funktionsweise der Firmware rekonstruiert und anschließend Teile ersetzt. Während Softwaremanipulationen in weit verbreiteten Architekturen wie x86 oder ARM bereits gut erforscht sind, gab es bislang keine Dokumentationen, welche Angriffe auf kryptographische Hardwarekonfigurationen von FPGAs bzw. die zugehörige Bitstreamdatei beschreiben. Eine Bitstreamdatei, der kodierte Schaltkreis eines FPGAs, enthält eine Beschreibung der Konfiguration der internen Hardwareelemente, die beim Start geladen wird. Es kann jeden denkbaren digitalen Hardwareschaltkreis beschreiben, der eine beliebige Funktionalität ausführen kann. Angenommen ein Angreifer kommt in Besitz eines Bitstreams der einen kryptographischen Schaltkreis beschreibt, zum Beispiel durch Abhören des Konfigurationsdatenbusses oder durch Auslesen des externen nicht-flüchtigen Speichers. Dann stellt sich die Forschungsfrage, ob und speziell wie eine Manipulation dieses Bitstreams zur Schlüsselextraktion oder dem Einfügen eines Trojaners führen kann. Da das Dateiformat des Bitstreams allerdings proprietär ist und keine offiziellen Extraktionswerkzeuge zur Verfügung gestellt werden, scheint die Manipulation von kryptographischen Hardwarekonfigurationen in der Praxis auf den ersten Blick schwierig zu sein. Dies ist einer der Gründe, weshalb bisher keine praktischen Angriffe auf Hardwarekonfigurationen vorgestellt wurden.

Diese Schwierigkeit stellt sich gleichermaßen, falls eine menschenlesbare Repräsentation gegeben ist. Ein Grund hierfür ist, dass ein Angreifer komplexe Hardwarestrukturen rekonstruieren muss, welche aus hunderttausenden elektrischen Leitungen, Look-up-Tabellen und Flip-Flops bestehen können, die als unstrukturiertes Konstrukt von Gattern erscheinen.

Um sich im Allgemeinem gegen das Klonen von Bitstreams bzw. vor Manipulationen zu schützen, bieten die Marktführer Xilinx und Altera einen Verschlüsselungsmechanismus für Bitstreams an, der Vertraulichkeit und Integrität sichern soll. Ein Problem dieser Gegenmaßnahmen ist jedoch, dass der kryptographische Schlüssel beispielsweise durch Seitenkanal-Angriffe extrahiert werden kann. Die Bestimmung des geheimen Schlüssels von der Bitstreamverschlüsselung führt zum kompletten Verlust der Vertraulichkeit und Integrität. Dies ermöglicht ein Kopieren des Bitstreams und lässt bösartige Manipulationen zur realen Bedrohung werden. Insbesondere die bösartige Manipulation von Bitstreams ist ein kaum untersuchtes Forschungsfeld.

Um diese Forschungslücke zu schließen, zeigt diese Dissertation auf, dass nicht-invasive und gezielte Bitstreammanipulationen praktisch durchführbar sind und ein mächtiges Werkzeug darstellen um die Sicherheit von kryptographischen Implementierungen zu brechen. In dieser Arbeit werden somit die ersten erfolgreichen Manipulationen von Bitstreams gezeigt, welche die Sicherheitseigenschaften von Blockchiffren wirkungslos machen. Konkret beschreibt diese Arbeit Methoden zur Detektion und Manipulation, die es entweder erlauben den geheimen Schlüssel zu extrahieren oder erzwingen, dass die kryptographische Stärke der AES Blockchiffre geschwächt wird. Um die praktische Relevanz und Machbarkeit von Bitstreammanipulationsangriffen zu bestätigen, präsentieren wir den ersten injizierten FPGA Trojaner. Dazu wird das AES-Modul im Bitstream eines kommerziell verfügbaren eingebetteten Gerätes manipuliert. Dabei handelt es sich um einen FIPS-140-2 Level 2 zertifizierten Hochsicherheits-USB-Stick der Firma Kingston.

Gezielte Bitstreammanipulationen, insbesondere diejenigen, welche eine vorherige Detektion von relevanten kryptographischen Primitiven erfordern, weisen eine bereits sehr gute Erfolgsquote auf, haben bei speziellen AES Implementierungen allerdings keine Erfolgsgarantie. Um eine weitere generische Methode zur Schlüsselextraktion bzgl. AES aufzuzeigen, präsentieren wir eine neuartige und effiziente Vorgehensweise, die erfolgreich Fehlerinjektionsangriffe – auch auf verschlüsselte Bitstreams – durchführt. Diese Technik führt zu einer noch höheren Erfolgsrate bzgl. der Schlüsselextraktion diverser AES Implementierungen, unabhängig einer vorherigen Lokalisation des AES im Bitstream. Der Angriff funktioniert zudem völlig unabhängig von der zugrunde liegenden Hardwarearchitektur. Folglich kann der Gebrauch von teuren Laserequipments oder der Rekonstruktionsprozess bezüglich der zugrunde liegenden Hardware-Konfiguration vermieden werden, welche ähnlich komplex wie die Rekonstruktion eines Application Specific Integrated Circuits (ASICs) sein kann.

Zusammenfassend kann gesagt werden, dass diese Arbeit auswertet zu welchem Grad nicht-invasive Bitstreammanipulationen die Systemsicherheit von SRAM-basierten FPGAs aushebeln können.

Schlagworte.

Kryptographie, Bitstreamverschlüsselung, Altera Stratix III, Spartan 6, Xilinx Virtex 5, FPGA-Sicherheit, Rekonfigurierbare Hardware, Bitstream-Manipulation, Reverse-engineering, Hardware-Trojaner, Schlüsselextraktion, Fehlerinjektion, AES, IP-Schutz, Kingston, Praktische Angriffe, Gegenmaßnahmen.

Acknowledgments

Finally! It is done! I'm happy that I can write a couple of thankful words. I actually consider myself as a lucky person who had the opportunity to be part of the Chair of Embedded Security (EMSEC), which is lead by my supervisor Christof Paar. At this point, I especially would like to thank Christof for his great support and positive attitude towards me and our various research projects. I really enjoyed to work with Christof, who is not only a very great teacher (without any doubts all people knowing him will confirm this without hesitation), but also a great motivator and very supportive person. Thank you for approximately 5 great years!

I'm also very thankful to Irmgard Kühn and Horst Edelmann for helping me with all the administrative and technical issues, which made my work life much easier. Thanks!

It must be fairly said that the results of this thesis would not have been possible without collaboration with many of my (external) co-authors. Hence, a huge thanks to all of you!

One person on my list regarding "thanking people separately in the acknowledgments" is Georg Becker, who also greatly guided me to stay on the right track regarding my research. It was particularly fun for me to work with Georg on the BiFI project (not to be confused with BiFi). Huge thanks!

Next, I would like to thank one of my best friends Ilya Ozerov, who tremendously helped me during my studies. Thanks for the various discussions, explanations, learning sessions, for showing me Peppone, and for Wiesbaden. Also, thanks for the various cinema visits, jogging sessions at the Kemnader sea, as well as Rust, DayZ, Portal 1/2 gaming sessions, which was all fun stuff except maybe some of the first or longer lasting jogging sessions ;-)!

Now, I would like to particularly thank Marc Fyrbiak and Philipp Koppe with whom I had a great time at the University of Massachusetts and during our research collaboration. It was kind of funny to observe Philipp's style of letting "coffee bounce" while driving a car. I still do not know how it could not affect Marc's caffeine circulation, but I do not want to get off the point: thank you guys for the great work and funny moments during our America visits and the common time at our chair!

At this point, I also would like to thank Professor Russell Tessier, who welcomed me as a guest at the University of Massachusetts and for being my secondary referee.

The next person on my list is Amir Moradi. I'm happy for the many times he has helped me in solving technical issues, for sharing his (side-channel) knowledge with me and for the nice support during my Master thesis. Also, thanks for the various Hattingen invitations! I'm a bit sorry for my "difficult-to-pronounce and difficult-to-write" surname, but luckily at least Thunderbird solves the latter issue by offering an autocomplete feature :).

Many thanks go to David Oswald, who greatly supervised me prior to my time as a researcher at EMSEC and for convincing me to do a security analysis on the crypto accelerators of Java cards as a Bachelor thesis, which later on resulted into a student assistant job and this somewhat resulted in doing my Master thesis at EMSEC. This in return was one contributing factor for joining Christof's group. Hence, thank you!

Also, I want to thank Falk Schellenberg, who has helped me with some of my car issues and who was driving me home for a couple of times when I was in need. Thanks to Max Hoffmann for organizing all the social events and thanks to Tobias Schneider for the various funny discussions about research and UbiCrypt. Another thanks goes to Pascal Sasdrich for sharing his ECC and tiny AES cores that were helpful for one of my research projects.

I also particularly want to thank Steffen Becker, Maik Ender, and Shahram Rasoolzadeh (in alphabetic order) for the additional time they have spent with me to practice my defense and for their valuable feedback. I should not forget to mention the many interesting chats or playing the game ”Magnet-werfen“ ;-)!

Some thanks are also owed to Ingo von Maurich and Thomas Pöppelmann for sharing their office with me for the first couple of my working months and for explaining all the organisational stuff.

Finally, I want to thank all who proof-read parts of this thesis (again in alphabetic order): Steffen Becker, Erik Boss, Liam Collins, Maik Ender, Christian Kison, Tobias Schneider, Ilya Ozerov, and Felix Wegener. I really appreciate your help! Thanks guys!

Table of Contents

Imprint v
Abstract v
Kurzfassung viii
Acknowledgements xi

I Preliminaries **1**

1 Introduction **3**
1.1 Motivation 3
1.2 Related Work 4
1.2.1 Bitstream Encryption Scheme Security 4
1.2.2 Bitstream Reverse-engineering and Manipulation 6
1.2.3 Malicious Bitstream Manipulations 8
1.3 Contribution and Organization of this Thesis 8

2 Technical Background **11**
2.1 Xilinx FPGAs 11
2.1.1 Hardware Resources 12
2.1.2 Design Flow for Bitstream Generation 14
2.2 System and Adversary Model 15
2.2.1 Practical Hurdles 16
2.3 AES Basics 17

II FPGA Security **19**

3 Bitstream Encryption **21**
3.1 Motivation 21
3.2 The Design Security Feature of Stratix III FPGAs 22
3.3 Required Reverse-Engineering Steps for Stratix III FPGAs 23
3.4 Required Side-Channel Steps for Stratix III FPGAs 24
3.5 Conclusion 27

4 Targeted Bitstream Manipulation Attacks Against Reconfigurable Hardware **29**
4.1 Motivation 29
4.2 Attack Idea - Substitution of S-boxes of Block Ciphers 31

4.3 Bitstream Encoding of Xilinx FPGAs . . . 32
4.3.1 Extracting the LUT Encoding from a Bitstream . . . 32
4.3.2 Extracting the BRAM Encoding from a Bitstream . . . 35
4.4 Exploiting Boolean Functions in FPGA Bitstreams . . . 36
4.4.1 Detection of DES S-boxes . . . 37
4.4.2 Results of DES S-box Detection . . . 38
4.4.3 Manipulating DES S-boxes . . . 39
4.4.4 Detection of AES S-boxes . . . 43
4.4.5 Results of AES S-box Detection . . . 47
4.4.6 Manipulating AES S-boxes . . . 48
4.5 Mitigating S-box Substitution Attacks . . . 53
4.5.1 Built-In Self-Test . . . 53
4.5.2 Decomposition of Larger Circuits into Smaller Ones . . . 53
4.5.3 Proposal for Partial Self-configuration Countermeasure Scheme . . . 54
4.6 Conclusion . . . 59

5 Real-World FPGA Trojan Insertion into a Commercial High-Security Encryption Device 61
5.1 Motivation . . . 61
5.2 Proceeding of Inserting an FPGA Trojan . . . 62
5.2.1 Attack Scenario: Interdiction . . . 62
5.3 Real-World Target Device . . . 63
5.3.1 Initial Steps and Authentication Process . . . 63
5.3.2 Physical Attack — Revealing the FPGA Bitstream . . . 64
5.3.3 Overview and Component Details . . . 65
5.3.4 Unlinking FPGA Trojan from the Authentication Process . . . 66
5.3.5 Modifying Bitstream vs. Replacing Entire Bitstream . . . 67
5.3.6 Manipulation – Master vs. Slave . . . 68
5.4 Building the FPGA Trojan . . . 68
5.4.1 Analysis of the Extracted Bitstream . . . 68
5.4.2 Modifying the Third-Party FPGA Design . . . 69
5.5 ARM Code Modification . . . 70
5.5.1 Utilized Self-tests . . . 70
5.5.2 Disabling Self-tests to Modify ARM Code and FPGA Bitstream . . . 70
5.5.3 Separating Key Derivation and FPGA AES IP-Core . . . 71
5.5.4 Recording XTS-AES Parameters . . . 72
5.6 XTS-AES Manipulation and Plaintext Recovery . . . 72
5.6.1 Manipulation of AES-XTS . . . 73
5.7 Summary of Security Problems . . . 74
5.8 Conclusion . . . 75

6 Bitstream Fault Injections (BiFI) - Automated Fault Attacks Against SRAM-based FPGAs and AES 77
6.1 Related Work . . . 78
6.2 Motivation and Contribution . . . 78

6.3 Background . . . 79
6.4 Attack Idea . . . 80
6.4.1 Manipulations Rules . . . 80
6.4.2 Key Recovery . . . 81
6.5 Experimental Setup and Results . . . 83
6.5.1 Results without Enabled Bitstream Encryption Scheme . . . 84
6.5.2 Experimental Setup with Enabled Bitstream Encryption Scheme . . . 86
6.5.3 Setup and Results with Enabled Bitstream Encryption . . . 88
6.5.4 Testing the Bitstream Encryption Vulnerability of Xilinx FPGAs . . . 89
6.5.5 Discussion on Altera Bitstream Encryption Scheme . . . 89
6.6 Analysis . . . 91
6.6.1 Evaluation of Observed Faults . . . 91
6.7 Discussions and Countermeasures . . . 96
6.7.1 Impact on Other Fault Attack Types . . . 97
6.8 Conclusion . . . 98

III Conclusion **99**

7 Conclusion **101**
7.1 Impact of Bitstream Encryption Vulnerabilities . . . 101
7.2 Impact of Bitstream Manipulations . . . 102
7.3 Future Directions . . . 103

IV Appendix **105**

Bibliography **107**

List of Abbreviations **113**

List of Figures **115**

List of Tables **119**

List of Algorithms **121**

About the Author **123**

Publications and Academic Activities **124**

Part I

Preliminaries

Chapter 1

Introduction

In this chapter, the motivation and related work for the research conducted in this thesis are portrayed. Finally, we outline the structure and highlight the contribution of this thesis.

Contents of this Chapter

1.1 Motivation **3**
1.2 Related Work **4**
1.3 Contribution and Organization of this Thesis **8**

1.1 Motivation

A Static Random Access Memory (SRAM)-based FPGA is an Integrated Circuit (IC) offering a large resource of programmable hardware elements that are distributed over a two-dimensional array. In contrast to an ASIC, these structures are not hard-wired and can be programmed after chip fabrication. Hence, they have the ability to (remotely) update their hardware functionality at any time.

A so-called bitstream, which is used to configure the desired functionality into the volatile SRAM-cells of the FPGA, encodes a low-level description of a previously designed high-level hardware layout. With the help of the proprietary FPGA vendor's toolchain, a designed high-level hardware layout is mapped and routed to the actual hardware resources of the FPGA. The corresponding description is stored in a netlist file, which we refer to as hardware configuration. In a final step, a hardware configuration is encoded and stored as a binary bitstream file.

As FPGAs are increasingly used to execute cryptographic algorithms, it is particularly important to evaluate, whether the manipulation of hardware configurations, describing cryptographic circuits, could lead to a security breach.

One practical hurdle for demonstrating such a security breach is due to the complexity of a hardware configuration. Even if the entire hardware configuration would be given to an attacker, which is usually not the case, it is time-consuming to analyze and (maliciously) manipulate the circuit, since it appears as an unstructured mass of gates to the attacker.

Conducting meaningful manipulations is even harder if an attacker only possesses the binary bitstream file since this file format is proprietary. This establishes an additional barrier for an attacker because the hardware configuration also appears obscured and cannot be algorithmically analyzed unless the file format can be (partially) interpreted. There are no official extraction

tools allowing to interpret a bitstream, analyze the corresponding hardware configuration, or to modify it.

Due to the volatile nature of an SRAM-based FPGA, the bitstream file is usually stored in a dedicated non-volatile memory chip like a flash or EEPROM. Commonly, it is integrated on the same Printed Circuit Board (PCB) that contains the FPGA. Both hardware components are connected through a configuration data bus. Once the PCB is powered, the bitstream is configured consecutively into the SRAM-cells of an FPGA by following a configuration protocol. This way, all combinatorial as well as sequential hardware elements are initialized to form the final circuit. Note that the cells have to be reconfigured after each power down.

On one hand, such a configuration mechanism can allow fixing critical security vulnerabilities through updating a flawed hardware design. On the other hand it represents a security risk. In practice, the major issue is IP theft. This is due to the fact that it is a straightforward task for an adversary to either eavesdrop on the configuration data bus during power-up or to directly read out the plain bitstream from the non-volatile memory, and thus it is easy to clone a competitor's entire hardware design.

While cloning a bitstream is a relatively easy task, a targeted and meaningful manipulation of an AES core within a third-party bitstream, potentially resulting in a security breach, has never been demonstrated to date. Despite more than 20 years of research related to FPGAs and cryptography, relatively little is known about bitstream manipulations with the goal of key extraction or breaking cryptographic primitives. We argue that this is mostly due to its proprietary file format and the underlying complexity of the built hardware circuity. From the view of a security engineer, it is important to study bitstream manipulation attacks to evaluate whether cryptographic algorithms can be executed securely on this type of devices.

To thwart any IP theft issues or bitstream manipulation attempts, the main two FPGA vendors, Xilinx and Altera, introduced a countermeasure which is called bitstream encryption or design security. In the next section, we provide an overview of research that examined the security of these schemes.

1.2 Related Work

In this section, we present state-of-the-art bitstream encryption scheme attacks, provide information about bitstream file format reverse-engineering approaches, and introduce bitstream manipulation strategies.

1.2.1 Bitstream Encryption Scheme Security

Any attempt of tampering a bitstream could be easily mitigated by an encrypted and authenticated bitstream. In practice, this is usually realized as follows. In an initial step, the customer defines a secret bitstream encryption key k with the help of the software-based toolchain of the FPGA vendor. Then, the bitstream encryption key k is programmed into the dedicated hardware decryption module of the FPGA. During this initialization step, the bitstream is encrypted with a symmetric cipher such as AES and can therefore be stored securely in the flash chip. This mitigates the possibility of an attacker analyzing its contents in case the flash chip is read out or the configuration data is wiretapped. The overall goal is to ensure that an attacker

will neither gain any information about the bitstream file nor that he can clone the design into another blank FPGA.

Once set up, during each power-up of the system the previously encrypted bitstream is transferred to the dedicated hardware decryption module of the FPGA, then decrypted, verified, and finally configured. In an ideal world, the secret bitstream encryption key k cannot be recovered as it is not accessible by an attacker. As indicated before, this could prevent any bitstream manipulation attempt and mitigate reverse-engineering of an FPGA design.

Nevertheless, the schemes of both major vendors, namely Xilinx and Altera, have demonstrated security weaknesses allowing for leakage of the secret bitstream encryption key k by utilizing side-channel analysis. This countermeasure can be broken in approximately one day, cf. the works of Moradi *et al.* [MBKP11, MKP12, MOPS13, MS16]. In these attacks, the power consumption can be exploited during the decryption process to reveal the secret bitstream encryption key k. Subsequently, the encrypted bitstream can be decrypted, maliciously manipulated to change its hardware configuration, and finally be re-encrypted so it can be processed by the FPGA again, bypassing the bitstream authentication as well as bitstream integrity. Table 1.1 shows all FPGA devices whose bitstream encryption schemes are known to be vulnerable to side-channel attacks. As can be seen, Xilinx Virtex 2 up to Xilinx Virtex 6, the entire Xilinx 7 series, Xilinx Spartan 6 FPGAs, Altera Stratix II and Altera Stratix III FPGAs are affected. Hence, the bitstream encryption schemes of all Xilinx FPGAs except for the latest and expensive high-security devices (Kintex and Virtex Ultrascale [MS16]) can be practically attacked.

Device Family	**Introduced**	**Bitstream Enc./Auth.**	**Known Vulnerability**	**No integrity/authenticity**
Xilinx Spartan 3	2005	not supported	–	√
Xilinx Spartan 6	2009	AES-256/HMAC	[MS16]	√
Xilinx Virtex II	2001	3-DES/no	[MBKP11]	√
Xilinx Virtex 4	2005	AES-256/no	[MKP12]	√
Xilinx Virtex 5	2006	AES-256/no	[MKP12])	√
Xilinx Virtex 6	2009	AES-256/HMAC	[MS16])	√
Xilinx 7 series	2010	AES-256/HMAC	[MS16])	√
Xilinx UltraSCALE	2014	AES-256 GCM/RSA-2048	no research reports so far	unclear
Xilinx UltraSCALE$^+$	2015	AES-256 GCM/RSA-4096	no research reports so far	unclear
Altera Stratix II	2004	AES-128	[MOPS13]	√
Altera Stratix III	2006	AES-256	[SMOP14]	√
Altera Stratix IV	2008	AES-256	no research reports so far	unclear
Altera Stratix V	2010	AES-256	no research reports so far	unclear
Altera Stratix 10	2013	AES-256, SHA-256, PUF	no research reports so far	unclear
Microsemi FPGAs	–	AES-256, SHA-256	no research reports so far	unclear

Table 1.1: List of Xilinx FPGA families and Altera FPGA devices, which are vulnerable to side-channel attacks. No side-channel attacks for the UltraSCALE and UltraSCALE$^+$ family have been reported so far. Note that the Xilinx 7 series includes the Kintex, Artix, and Virtex families

Another important reason why malicious bitstream manipulations need to be explored is the fact that most currently-deployed FPGAs do not even support bitstream encryption or authentication.

According to Altera's annual business report from 2014 [Alt15] the peak production of an FPGA is roughly 6 years after introduction and the FPGAs are sold for more than 15 years. According to the annual reports of both Xilinx and Altera, around 50% of the revenue actually

comes from older FPGA families which, as indicated before, do not have bitstream authentication [Alt15, Xil15]. It seems likely that it will take some time until FPGAs with bitstream authentication are widely used in practice. Hence, if one can demonstrate successful bitstream manipulations, this also shows that those attacks are applicable to a large share of FPGAs currently used in practical applications.

Note that whenever side-channel attacks or similar ones, e.g., relying on laser fault injections, fail to extract the secret key of a cryptographic hardware configuration, bitstream manipulations are a legitimate alternative attack vector to compromise the system security in a non-invasive manner and hence should be considered a practical threat. In order to successfully conduct *targeted* bitstream manipulations, e.g., leading to a Trojan, we expect that a (partial) reverse-engineering of the bitstream file format is required. Therefore, in the next section we provide an overview of works that have attempted to reveal the bitstream encoding.

1.2.2 Bitstream Reverse-engineering and Manipulation

The proprietary bitstream format obfuscates the encoding of an FPGA configuration and the FPGA vendors do only barely or incompletely support the parsing of bitstream files to a human-readable hardware configuration. Therefore, there are only limited capabilities for analyzing or manipulating a given third-party hardware configuration. For this reason, the reverse-engineering and partial manipulation of the proprietary bitstream structure of FPGAs has been the focus of several works, which we outline in the following.

Xilinx FPGAs

In 1999, Guccione *et al.* [GLS11] introduced a tool called *JBits*. It operates on bitstreams generated by Xilinx tools. This tool allows the user to replace hardware configurations for Look-Up Tables (LUTs), Flip Flops (FFs), and routing. It can modify existing circuits of Xilinx XC4000 and Virtex families, but a limitation of this tool (and for many other tools) is that it supports only a small subset of Xilinx FPGAs. In 2002, two tools (JPG [RS02] and PARBITS [HLK02]) were presented enabling partial bitstream generation, e.g., by using *JBITS* or intermediate file formats. Unfortunately, the support for *JBITS* is discontinued and publicly unavailable.

In 2006, it was demonstrated by Ziener *et al.* [ZAT06] that reverse-engineering the encoding of the entire look-up table configuration of an FPGA bitstream is feasible for Virtex II FPGAs. Later, in 2008, Note and Rannaud [NR08] showed how to reverse-engineer the bitstream file format by cross-correlating data from the binary bitstream file and its corresponding netlist, which was used to generate the bitstream. Their work is meant to be seen as proof-of-concept that bitstream reverse-engineering is feasible. In particular, their tool is able to extract a textual configuration from the bitstreams that are used to configure Xilinx Virtex 2, Virtex 4 LXT and Virtex 5 LXT FPGAs. The authors claimed that reverse-engineering the bitstream file format is an easy task without providing any success rate of actually translating bitstreams back to a netlist.

Later, in 2010, Lavin *et al.* [LPL+10] introduced an open-source Java-based tool called *RapidSmith*. It is a library for low-level manipulations of partially placed-and-routed hardware configurations requiring intermediate file formats in case given circuits need to be modified. *RapidSmith* can also parse, manipulate, and export bitstreams for Xilinx Virtex 4, Virtex 5 and Virtex 6 families, but it seemingly cannot parse the configuration of frame internals such as

LUTs, Block Random Access Memories (BRAMs) blocks, or FF contents. An open-source tool with a similar feature set called *Torc* was published in 2011 by Steiner *et al.* [SWS+11]. It supports a broader class of Xilinx FPGAs (Virtex family, Spartan 3E, Spartan 6, and Spartan 6L) and is based on C++.

Another contribution in this field was presented by Benz *et al.* [BSH12] in 2012. The authors proposed a toolchain, called *Bil*, for bitstream reverse-engineering by correlating binary bitstream data with data extracted from the netlist. Benz *et al.* stated that the reverse-engineering of all hardware elements of an Xilinx FPGA, including routing information, is considerably harder than assumed. According to their analysis, it is rather difficult to reverse-engineer all hardware elements in an automated manner, hence they questioned the claim of Note and Rannaud.

Another work was presented by Ding *et al.* in 2013 [DWZZ13]. The authors provided detailed evaluations showing that the binary bitstream files can almost entirely be translated back to its netlist or graphical representation. According to their findings, most parts can be translated back as their results vary between 87% and 99% in terms of routing resources.

In early 2017, as research progressed, Pham *et al.* [PHK17] published a more advanced tool called *BITMAN*. This tool is able to parse, analyze, and manipulate Xilinx bitstreams, including newer FPGA families such as Zynq-7000 and Kintex Ultrascale. Unlike the previous tools, it can replace larger modules in a complex hardware configuration. Furthermore, a high-level Application Programming Interface (API) provides functions to replace entire FPGA regions, to re-route (clock) wires, and to manipulate LUT and BRAM configurations.

Altera FPGAs

Jean-Baptiste Note also published the source code for reverse-engineering an Altera bitstream as part of a project called *debit* [Not08], although this was not an official scientific publication and the target FPGA models remain unknown. Note that the *debit* project is discontinued and the Altera bitstream reverse-engineering code is incomplete.

In 2016, Jean-Francois Nguyen demonstrated the practical feasibility of reverse-engineering Altera bitstreams by explaining how to exploit the non-determinism of the place-and-route tool to extract the bits corresponding to each logic cell of an MAX-V Complex Programmable Logic Device (CPLD), cf. [Ngu16]. Even though CPLDs are less complex than FPGAs, we expect that his approach applies to Altera FPGA bitstream reverse-engineering as well.

Lattice iCE40 FPGAs

In 2015, Wolf and Lasser reverse-engineered the Lattice iCE40 FPGA, which is for example used in HTC Vive's Virtual Reality system and many other practical applications. The authors have published their results as documentation [WL] and provide fully working parsing tools along with the corresponding source code. As opposed to the reverse-engineering of other FPGAs bitstreams, this project seems to be the first complete and publicly available one. Hence, it enables third-parties to develop open source synthesis tools for generating bitstreams. In turn, this allows third-party bitstreams to be analyzed and manipulated at a very low level without the need to re-run the entire synthesis process.

1.2.3 Malicious Bitstream Manipulations

In 2013, Chakraborty *et al.* [CSPN13] demonstrated how to accelerate the aging process of an FPGA by merging a ring-oscillator circuitry into an existing bitstream. This kind of Trojan is described as a "Type 1 Trojan", i.e., it does not tamper the relevant parts of a cryptographic algorithm or access control mechanism, but rather addresses unoccupied hardware elements. The disjunctively added ring-oscillator Trojan circuitry performs redundant circuit switching to increase power usage, thereby leading to an increase of the operating temperature of the FPGA device, and as a consequence, to a reduced lifetime.

Another related malicious bitstream manipulation work was done by Aldaya *et al.* In early 2015 [A. 15], they demonstrated a key recovery attack for all AES key sizes by tampering AES T-boxes, which are stored in the BRAM of Xilinx FPGAs. It is a ciphertext-only attack and it showed that various FPGA-based AES implementations can be compromised this way.

After presenting the related work, we further would like to pinpoint the interested reader to another large body of research addressing various aspects of cryptography and FPGAs, which can be found in [Dri08].

1.3 Contribution and Organization of this Thesis

We summarize the contribution of this thesis and outline the structure of this work. This thesis is based on joint work with Georg Becker, Philipp Koppe, Marc Fyrbiak, Amir Moradi, Russell Tessier, and Christof Paar.

- Chapter 2 provides the necessary background information regarding Xilinx FPGAs and introduces notations for AES-128. Additionally, the adversary and system model are described, which are valid for almost any conducted experiment within the scope of this thesis.

- Chapter 3 demonstrates that the design security feature of Stratix III is vulnerable to reverse-engineering and side-channel attacks. Hence, our contribution is to raise awareness for security engineers that the bitstream encryption scheme of Stratix III is weak enabling cloning and manipulation of a seemingly protected IP. One take away message is that Stratix III FPGAs should be used carefully in real practical applications requiring confidentiality and integrity of the bitstream. This side project is a follow-up work of [MOPS13]. We published the corresponding results in [SMOP14].

- Chapter 4 introduces the first successful malicious and targeted hardware configuration attacks on Xilinx FPGAs by altering third-party DES and AES bitstreams leading to a security breach. To this end, a new approach is presented for algorithmically detecting and replacing distributed DES and AES S-boxes *directly* in the bitstream. Further, it is explained how to manipulate the hardware configuration in such a way that *i*) the computed DES or AES ciphertexts become cryptographically weak or that *ii*) the 128-bit AES key can be extracted directly after power-up of the system.

  The main contribution of this thesis is to introduce a new kind of research field, namely malicious bitstream manipulations, and to prove that many Xilinx FPGAs are not well suited for executing cryptographic algorithms securely (unless secure authentication schemes are

provided) by conducting real practical attacks at the bitstream level. Hence, we raise awareness that appropriate difficult-to-patch countermeasures should be researched and integrated, e.g., as part of the hardware configuration. Further, it highlights that considering laser-fault injection or side-channel attacks as the only practical threats is not sufficient to achieve a desired security level. The findings of this chapter were published in [SFKP15].

- Section 4.5 of Chapter 4 introduces a new countermeasure for AES cores that are supposed to provide an increased resistance against the attack vectors presented in Chapter 4. Hence, the contribution to this project is to present a new countermeasure concept for impeding targeted bitstream manipulations, to implement the required software and hardware part, and to evaluate the resulting hardware costs. The results of this chapter were published in [SFP+15].

- Chapter 5 highlights the practical relevance of the proposed bitstream manipulation of Chapter 4. The contribution of this work is to demonstrate the first real-world FPGA Trojan insertion into a commercial high-security and FIPS-140-2 level 2 USB flash drive from Kingston. Therefore, it highlights that the introduced bitstream manipulation attacks are indeed practically relevant exhibiting a real threat. Our findings were published in [SFK+16].

- Chapter 6 presents our most recent bitstream attack strategy, called Bitstream Fault Injection (BiFI), where it is demonstrated that even automatic and random bitstream manipulations on third-party AES hardware configurations lead to permanent exploitable faults. This allows to compromise a variety of implementations leading to key recovery including the countermeasure-protected AES core of Section 4.5. It turned out that this new kind of attack also works for a Xilinx Virtex 5 FPGA with an enabled bitstream encryption scheme. Thus, our contribution here is to raise security awareness by demonstrating a more generic key recovery approach and by finding that a weaker attacker is required than indicated by the results of Chapter 4. Particularly, by proving that there is no need *i*) to reverse-engineer the hardware configuration, *ii*) to reverse-engineer the LUT encoding (even though easing the attack) of the bitstream, and *iii*) to conduct a side-channel attack on the bitstream encryption scheme to break the underlying cryptographic hardware configuration. The corresponding results were made public in [SBMP17].

- Chapter 7 evaluates the impact of this conducted research and proposes future directions with respect to FPGAs.

Chapter 2

Technical Background

In this chapter, we provide background information regarding Xilinx FPGAs, which is necessary to fully comprehend the bitstream manipulation attacks presented within this thesis. Additionally, we introduce the underlying system and attacker model. Finally, we present notations for the AES-128 block cipher, which we consistently use throughout all chapters.

Contents of this Chapter

2.1 Xilinx FPGAs **11**
2.2 System and Adversary Model **15**
2.3 AES Basics **17**

2.1 Xilinx FPGAs

An FPGA can implement arbitrary functionality such as simple XOR and AND gates. Also, more complex combinatorial functions can be realized. Its complexity is only limited by the number of available resources offered by an FPGA device. Hence, an FPGA is highly flexible and is suited for running special-purpose applications. The internal hardware architecture of an FPGA varies between devices of different vendors and even the ones from the same family of a vendor. Xilinx FPGAs achieve a high customizability by integrating thousands of building

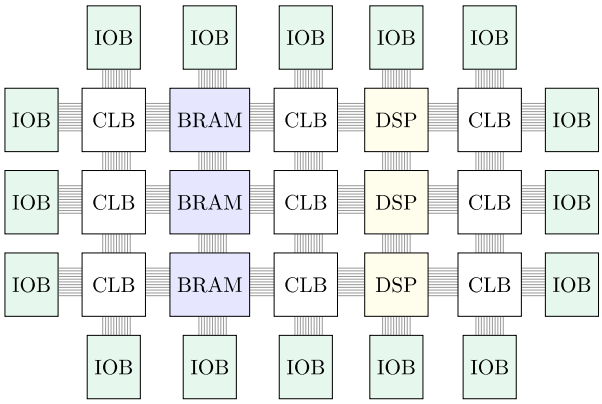


Figure 2.1: Building blocks of Xilinx FPGAs

blocks such as Configurable Logik Blocks (CLBs), BRAMs, Digital Signal Processings (DSPs),

and Input Output Blocks (IOBs). These are connected to a reconfigurable routing fabric, cf. Fig. 2.1.

2.1.1 Hardware Resources

A simplified overview of a Xilinx Spartan 6 CLB is presented in Fig. 2.2. All of these CLBs implement the logic function of a desired circuit. A CLB consists of so-called slices and a switch-matrix. Depending on the FPGA family, usually two or four slices are part of one CLB. Note that this number may vary for older or newer FPGA generations. Switch-matrices connect the slices to the routing fabric, which in turn can connect external devices through IOBs.

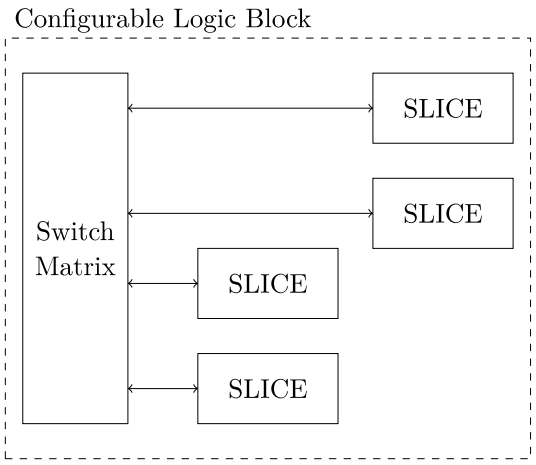


Figure 2.2: Exemplary CLB content

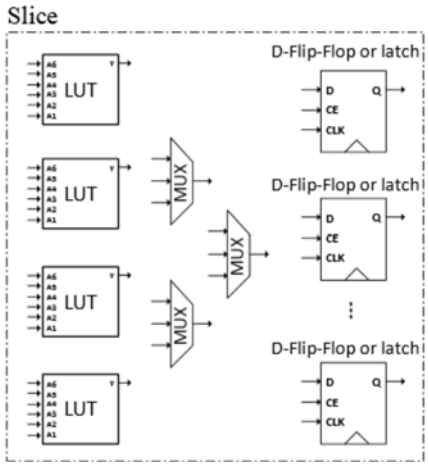


Figure 2.3: Overview of one slice

Figure 2.4: Simplified Spartan 6 FPGA architecture

Each single slice is positioned on the FPGA grid and labeled with a (X,Y)-coordinate. According to the Xilinx device library of the ATHENa project [GKA+10], the number of slices ranges from 600 to 305400 slices. Note that other Xilinx FPGAs may exist exceeding the discussed thresholds. Slices contain further programmable low-level hardware elements such as LUTs implementing Boolean functions, dedicated multiplexers selecting one or more input signals and forwarding the selection to a single output line, D-FFs implementing registers, or latches for realizing level-sensitive registers, cf. Figure 2.3.

All these elements form larger parts of the final hardware configuration. Besides LUTs and registers, several other hardware elements exist such as clock trees or dedicated DSPs allowing area-efficient implementations of complex high-performance arithmetic operations. Since we target LUT and BRAM primitives during our bitstream manipulation attacks, we now describe them in more detail.

Look-up tables (LUTs) are the main logic element. As mentioned before, they implement Boolean functions, and hence, are used for different tasks like storing constants, processing input signals, saving/loading data from flip-flops, copying signals from one location to another, controlling data buses or writing/loading BRAM contents. Xilinx FPGAs instantiate k-input,

1-output LUTs[1] with $k \in \{4, 6\}$. A 2^k-bit LUT can implement any k-input Boolean function $f(x)$ simply by storing its corresponding truth table. It is encoded as a sequence of bits, which we refer to as LUT content, (truth table) pattern, or as 2^k-bit truth table T implementing $T[x] = y = f(x)$ with $x \in \{0,1\}^k$ and $y \in \{0,1\}$, cf. Table 2.1 and Table 2.2. Note that one k-input, 1-output LUT can describe $2^{(2^k)}$ different Boolean functions.

6-input						**1-output**
$\mathbf{x_5}$	$\mathbf{x_4}$	$\mathbf{x_3}$	$\mathbf{x_2}$	$\mathbf{x_1}$	$\mathbf{x_0}$	$\mathbf{T[x]} = y = f(x)$
0	0	0	0	0	0	a_0
0	0	0	0	0	1	a_1
...	...	...	...	...	...	...
1	1	1	1	1	1	a_{63}

Table 2.1: General shape of a 6-input, 1-output look-up table, e.g., used by Xilinx Spartan 3 FPGAs

4-input				**1-output**
$\mathbf{x_3}$	$\mathbf{x_2}$	$\mathbf{x_1}$	$\mathbf{x_0}$	$\mathbf{T[x]} = y = f(x)$
0	0	0	0	a_0
0	0	0	1	a_1
...	...	...	...	...
1	1	1	1	a_{15}

Table 2.2: General shape of a 4-input, 1-output look-up table, e.g., used by Xilinx Spartan 6 FPGAs

There are $k!$ possible combinations in order to connect k arbitrary input signals to one k-input,1-output LUT. Since in Chapter 4 we require the terminology input permutation for successfully detecting relevant cryptographic Boolean functions that are implemented in LUTs, we consider a toy example of two different LUT configurations. Fig. 2.5 shows two possible ways of connecting 6 input signals $\{x_0, x_1, ..., x_5\}$ to the same 64-bit LUT, which both implement the same Boolean function $f(x)$. The order of connecting a 6-input signal to a LUT is referred

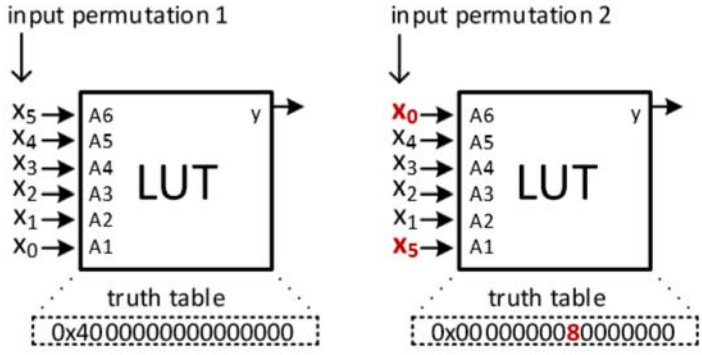


Figure 2.5: Example for configuring a truth table of one LUT using two different input permutations $(x_5, x_4, x_3, x_2, x_1, x_0)$ and $(x_0, x_4, x_3, x_2, x_1, x_5)$

to as input permutation. We for example consider that a LUT is supposed to implement the following Boolean function $f(x)$:

$$f(x_5, x_4, x_3, x_2, x_1, x_0) = \begin{cases} 1 & \text{if } (x_5, x_4, x_3, x_2, x_1, x_0) = (0,0,0,0,0,1) \\ 0 & \text{if } (x_5, x_4, x_3, x_2, x_1, x_0) \neq (0,0,0,0,0,1) \end{cases}$$

[1]For the sake of simplicity our explanations only consider 6-input,1-output LUTs even though Spartan 6 FPGAs for example can instantiate 6-input,2-output LUTs by sharing the LUT inputs.

In case the first input permutation $(x_5, x_4, x_3, x_2, x_1, x_0)$ is for example determined by the synthesizer (cf. left part of Fig. 2.5), the correct truth table content for implementing $f(x)$ is $(a_0, a_1, \ldots, a_{63})_2 = (0, 1, 0, \ldots, 0)_2 = 0\text{x}4000000000000000_{16}$. Given that a different input permutation, e.g., $(x_0, x_4, x_3, x_2, x_1, x_5)$ is determined (cf. right part of Fig. 2.5), now the correct truth table content for implementing the same Boolean function $f(x)$ is $(a_0, a_1, \ldots, a_{63})_2 = 0\text{x}0000000080000000_{16}$. As can be seen, patterns vary. Hence, an attacker, who is interested in finding specific patterns, needs to keep this in mind.

Block RAMs (BRAMs) are programmable Random Access Memory (RAM) blocks that can store up to 18Kb of data for Xilinx Spartan 6 devices. Block RAMs are placed in columns on the FPGA grid, also labeled with a (X,Y)-coordinate. The number of available BRAM columns depends on the size of the FPGA device. For Spartan 6 FPGAs, the number of 18Kb BRAMs varies between 12 and 268 available units, cf. [Xil]. A BRAM supports two modes: it can either be configured as two independent 9Kb RAMs or used as one 18Kb RAM. Each RAM block can be configured to be either accessible through two ports or it may be initialized as single-port. It supports different data width options such as 1-bit, 2-bit, or even 32-bit data widths, and hence, offers various flexible modes of operation.

2.1.2 Design Flow for Bitstream Generation

Figure 2.6 shows the concept of translating an FPGA design into a bitstream. As can be seen, the first step is to specify a hardware circuit with the help of Hardware Description Language (HDL) code. Once a circuit is specified, the developer initiates the synthesis and implementation process by using the vendor's toolchain. In case the developer does not use macros to exactly specify the usage of a hardware element and its location, the Xilinx toolchain will automatically derive a plan of how to map and route all hardware elements to the available resources on the FPGA grid to form the desired circuitry.

The developer neither exactly knows how all hardware elements will be instantiated nor can he easily revert or understand the final hardware configuration. Note that even a change of one single hardware element in the HDL code can lead to a complete change of the final hardware configuration.

As indicated in Section 1.1, the hardware configuration contains a detailed description of how to configure all hardware elements during power-up of the system. Particularly, it stores information about truth tables, block RAM initialization, multiplexer configurations, routing, and so forth. Finally, the hardware configuration is encoded as a bitstream file by a proprietary encoding tool. In the field, this file is the only accessible one to an attacker, who wants to compromise the FPGA configuration of an embedded device. Having detailed the design flow process, we specify our system and attacker model in the next section.

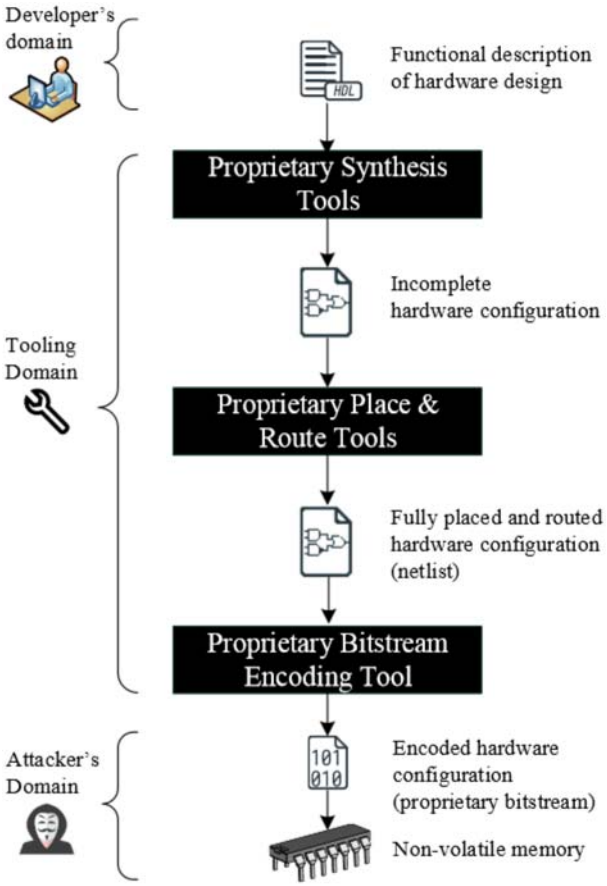


Figure 2.6: Simplified design flow of Xilinx FPGAs, which translates a high-level hardware layout to a low-level hardware configuration

2.2 System and Adversary Model

As explained in Section 1.1, FPGAs employ volatile memory, thus they require an external (usually untrusted) storage, e.g., a flash or EEPROM chip to store the bitstream file. It needs to be loaded upon power-up of the FPGA, cf. Fig. 2.7.

We assume a target device using a non-volatile memory chip that is integrated on the same PCB as the FPGA. The memory chip contains an unknown third-party bitstream describing a cryptographic circuit. Therefore, our adversary does not possess any high-level implementation information regarding the circuit such as the corresponding human-readable HDL source code or the design's low-level netlist. We further assume that the cryptographic key is not directly accessible to the adversary. For example, it can be encoded or obfuscated in external memory, stored in a secure on-chip memory, hard-coded in the design/bitstream, or generated internally by a Physically Unclonable Function (PUF).

Like most implementation attacks relying on side channels or fault injections, our attack requires physical access to the target device containing the FPGA. We assume that the adversary has read and write access to the external memory storing a bitstream. Our adversary can arbitrarily manipulate the bitstream, replace the original bitstream by a malicious one, and

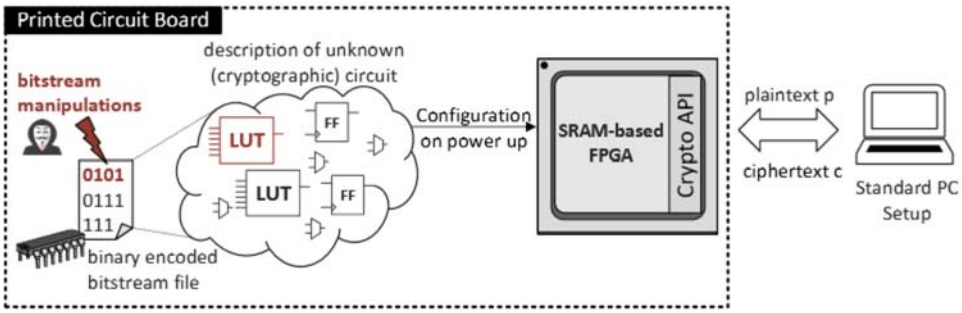


Figure 2.7: Overview of system model. A proprietary bitstream file implements an unknown circuit (e.g., AES), which configures an FPGA once it is powered-up. After this phase, a control circuit provides an interface to the encryption application. In practical applications, the bitstream and FPGA are integrated on the same PCB

consequently he is able to observe the altered behavior of the FPGA. Hence, we assume that our adversary can query the cryptographic implementation with a chosen plaintext and collect the corresponding ciphertext.

2.2.1 Practical Hurdles

Since the bitstream encoding is proprietary, one cannot directly analyze the low-level hardware configuration of any building block of a Xilinx FPGA. Hence, an attacker at least needs to partially translate the bitstream encoding to be able to analyze the hardware configuration. Otherwise, no targeted manipulations are possible. To provide a first impression of the bitstream

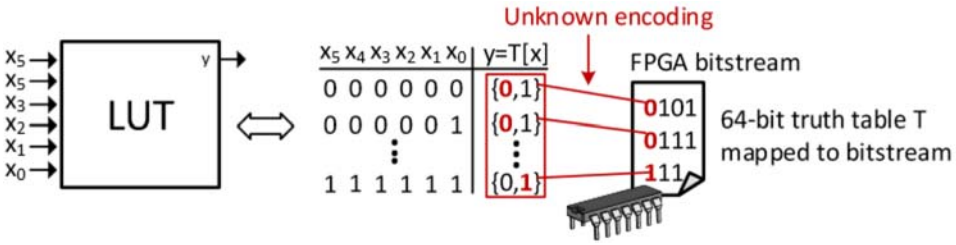


Figure 2.8: On the left, a 6-to-1 LUT with 6 input bits and 1 output bit is depicted. The LUT is a truth table T with 64 entries that are stored in the bitstream

encoding and its relation to storing the description of a hardware element, we shortly present one found example: the 64 bits of each LUT are mapped to fixed positions within the bitstream file according to specific previously unknown rules. This is depicted in Fig. 2.8. The raw bitstream file has a fixed length, and hence, the positions to encode one specific LUT always remain the same even though completely different hardware configurations were generated. The process of reverse-engineering the bitstream encoding has been shown multiple times so far. Since we did not have access to any extraction tool for our targeted FPGAs, we needed to partially learn the bitstream encoding to be able to explore the feasibility of bitstream manipulation attacks.

2.3 AES Basics

In this section, we shortly introduce AES focusing on AES-128. Figure 2.9 shows an overview of the AES-{128,192,256} encryption scheme for the three different key sizes 128, 192, and 256 bit leading to the execution of 10, 12, or 14 rounds, respectively [NIS01a]. AES operates on 128-bit

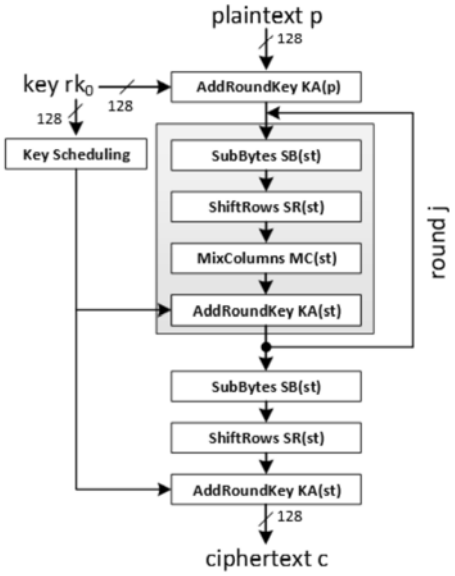


Figure 2.9: Overview of the AES encryption algorithm

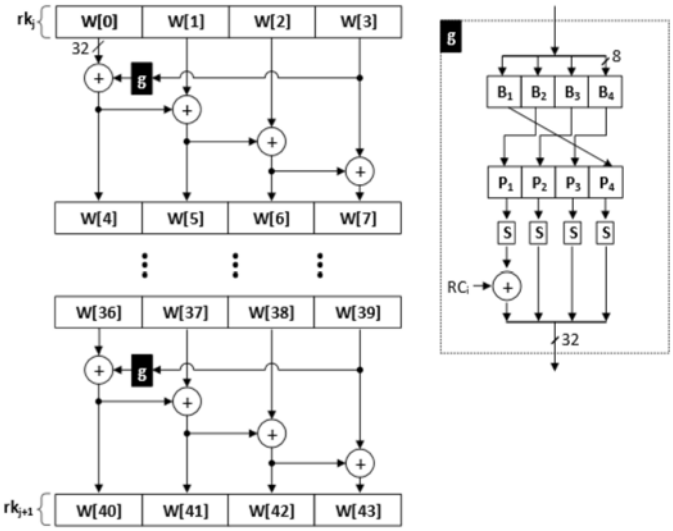


Figure 2.10: Key schedule of AES-128

blocks, independent of the key size. It encrypts a 128-bit plaintext p with a given key rk_0. Each AES round consists of the operations *SubBytes* (SB), *ShiftRows* (SR), *MixColumns* (MC), and *AddRoundkey* (KA), which are executed consecutively. The *SubBytes* step processes sixteen intermediate bytes by using a secure and constant S-box. For round-based implementations, it is common to use multiple S-boxes such that each input byte can be processed in parallel. In addition to that, for a round-based implementation, the key schedule step also needs to process four S-box instances. The key schedule algorithm of AES-128 is depicted in Fig. 2.10. As some of the presented attacks require addressing specific parts of the AES-128 block cipher, we hence introduce and use the following notations throughout this thesis.

- p, k, c: 16-byte plaintext, key, ciphertext with $c = AES128_k(p)$.
- $\tilde{c}$: 16-byte faulty ciphertext resulting from a manipulated AES module that we refer to as $\tilde{c} = \widetilde{AES128}_k(p)$.
- rk_j : 16-byte j^{th} round key being used at round $j \in \{0, 1, \ldots, 10\}$ with rk_1 being the first, rk_{10} the last round key, and rk_0 the initial key k.
- $SB(st), SR(st), MC(st), KA(st)$: SubBytes, ShiftRows, MixColumns, and Keyadd operations on the current 16-byte state st. Analogously, $SB^{-1}(st), SR^{-1}(st)$, $MC^{-1}(st)$, and $KA^{-1}(st)$ represent their inverse functions.

- $S(x)$ refers to one call of the 8-input,8-output S-box of AES.

To mark each of the possible AES states with a label, we use the following definitions.

- ka_j: 16-byte state at start of round j
- sb_j: 16-byte state after SubBytes operation at round j.
- sr_j: 16-byte state after ShiftRows operation at round j.
- $mc_{j'}$: 16-byte state after MixColumns operation at round $j' \in \{1, 2, ..., 9\}$.
- 0^{128}: A string of 128 bits set to zeros.

Note that there is no MixColumns operation during the last round, i.e., $j = 10$ for AES-128.

Before demonstrating our bitstream-based attacks against Xilinx FPGAs, we first present our follow-up work of [MOPS13], where we prove that the bitstream encryption scheme of Altera Stratix III FPGAs is vulnerable to side-channel attacks. Our motivation is to raise security awareness.

Part II

FPGA Security

Chapter 3

Bitstream Encryption

The work of Moradi et al. [MOPS13] showed that the bitstream encryption scheme of Altera Stratix II FPGAs can be circumvented through reverse-engineering Altera's toolchain and performing a side-channel attack, where the power-consumption of the device is exploited to disclose the secret bitstream encryption key k. The contribution of this chapter is to prove that the attack on Stratix II can also be applied to Stratix III FPGAs requiring additional engineering efforts. For complete details of Stratix II FPGAs, which are helpful for understanding the adapted attack on Stratix III FPGAs, the reader is referred to [MOPS13]. Note that Amir Moradi was the main project leader, who also conducted most of the side-channel steps, which we present for the sake of completeness.

Contents of this Chapter

3.1 Motivation **21**
3.2 The Design Security Feature of Stratix III FPGAs **22**
3.3 Required Reverse-Engineering Steps for Stratix III FPGAs **23**
3.4 Required Side-Channel Steps for Stratix III FPGAs **24**
3.5 Conclusion **27**

3.1 Motivation

The content of a user's bitstream file is usually the result of major investments in manpower and development costs. To protect this valuable IP against theft or cloning, the major FPGA vendors introduced bitstream encryption. Altera's bitstream encryption scheme (which is called design security) is making use of the AES. In 2013, Moradi *et al.* [MOPS13] demonstrated that the bitstream encryption scheme of Altera Stratix II FPGAs is insecure, since it is vulnerable to side-channel attacks leading to key exposure. Because public documents indicated that newer Stratix families including Stratix III FPGAs use AES-256 instead of AES-128, our motivation was to examine whether the newer devices are still vulnerable to side-channel attacks. Once such vulnerabilities can be demonstrated, consecutively, besides raising awareness that IP cloning is possible, we also learn that an adversary is likely able to carry out (malicious) bitstream modifications, which is the major research topic of this thesis.

3.2 The Design Security Feature of Stratix III FPGAs

The Stratix III series is the third generation of Altera Stratix FPGAs. According to [Cor12], Stratix III is manufactured using a 65 nm technology, while Stratix II FPGAs are based on 90 nm technology and therefore come with a lower static and dynamic power consumption than Stratix II FPGAs. The platform we selected to examine the side-channel vulnerability of Stratix III FPGAs is a standard development kit [Alt08] for the Stratix III EP3SL150F1152 high-performance FPGA, cf. Fig. 3.1.

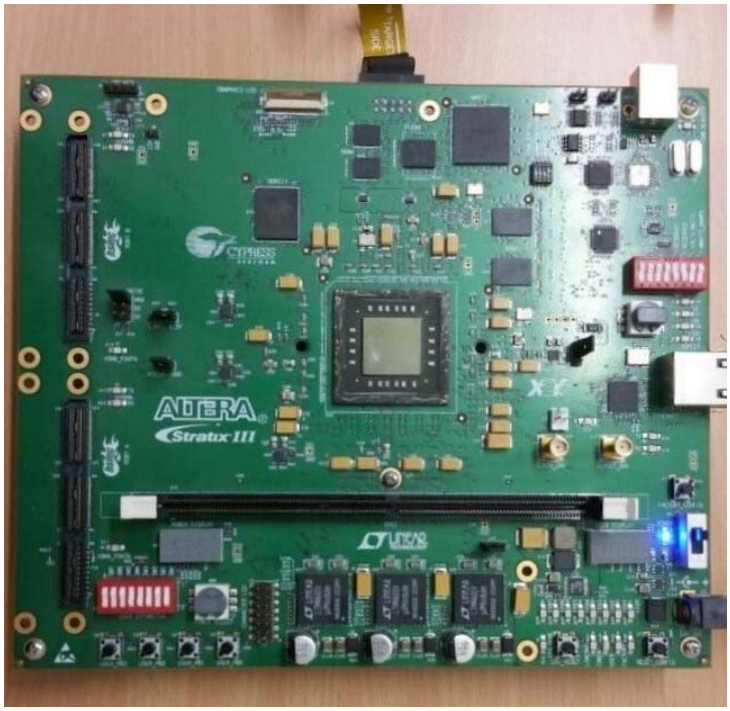


Figure 3.1: Device under attack - Official development board containing a Stratix III FPGA

To be able to conduct a successful attack, one needs to *i*) reveal the work-flow of the bitstream encryption scheme through reverse-engineering the vendor's software and *ii*) observe side-channel characteristics that leak the bitstream encryption key k during the decryption process.

One difference between Stratix II and Stratix III FPGAs is that the newer series features a hardware-based decryption module which uses AES-256 instead of AES-128 to decrypt and configure encrypted bitstreams. Similar to the required reverse-engineering steps of the Quartus application for attacking Stratix II FPGAs, we needed to reverse-engineer the inner workings[1] of the key derivation function, the utilized encryption mode[2], or the resulting AES inputs and

[1]Note that the reverse-engineering of the key derivation function and parts of the utilized encryption mode were already done during my Master thesis, but the proprietary AES input function f was unknown. Hence, reverse-engineering f (enabling the side-channel part) and adapting the passive serial configuration protocol for Stratix III was my main contribution to this project.

[2]The utilized mode of operation turned out to be similar to the AES in counter mode and therefore only uses the AES encryption to generate a key stream for the actual encryption.

the corresponding AES input update function that we refer to as f. During the initial setup phase, a bitstream encryption key k needs to be implicitly programmed through providing two 256-bit sequences called KEY_1 and KEY_2 into the FPGA's decryption hardware module. Obviously, the same bitstream encryption key k must be used in the vendor's tool to generate an appropriate encrypted bitstream. By doing so, the encrypted bitstream can be securely stored in external memory. In an ideal world, an attacker should never obtain any access to the bitstream encryption key k. Our intention is to extract the bitstream encryption key k by means of a side-channel attack.

The main prerequisite to conduct a side-channel attack for the Stratix III FPGA is to know the inputs which are processed by the AES encryption module during bitstream decryption. If the inputs remain unknown, it is difficult to create key guesses and to derive appropriate intermediate hypotheses. The hypotheses are needed for an attacker who can then try to correlate them with either the measured power consumption or the Electro-Magnetic (EM) emanation of the target device by using Pearson's correlation coefficient, cf. [Wik]. Hence, the necessary reverse-engineering steps for f are outlined in the next section.

3.3 Required Reverse-Engineering Steps for Stratix III FPGAs

In order to obtain the remaining required details, we continued to reverse-engineer the Quartus II application and revealed the necessary work-flow of the entire design security scheme for the Stratix III FPGA (fabric EP3SC150). Further note that when knowing the AES input update function f, all AES inputs required to decrypt a bitstream file can be reconstructed given the initial Initialization Vector (IV) (extractable from the bitstream file header with the help of the reverse-engineered encoding table which is described in [MOPS13]) and the bitstream encryption key k (once obtained by the attacker).

Regarding Stratix II FPGAs, the usual counter mode of AES increments its inputs and is then encrypted to produce a key stream which is XORed with the plain bitstream, cf. [MOPS13]. In contrast to that, we noticed that Stratix III FPGAs use a proprietary function f to derive an updated AES input, which in turn is again encrypted to produce a random keystream output. For reverse-engineering f, we observed the Quartus II application in the debugger of IDA Pro. Figure 3.2 gives a simplified overview of the executed functions from which we learned the work-flow of the bitstream encryption process in software making it possible to conclude the decryption process of the hardware-based bitstream decryption module. As depicted in Figure 3.2, an initially unknown function `sub_10007310(3)` (implementing f) is executed to generate the next input for the AES execution. The integer value "3" is passed as an argument to this function. When observing the memory locations being read by this function, it turned out that the bytes located at address p_3 are repeatedly accessed. We found that the value at address p_3 is updated three times (as specified by the argument "3") in a loop and that the result is directly related to the next AES input. Subsequently, we analyzed the assembly instructions of f and found that the update is performed as depicted in Figure 3.3.

It turned out that f (`sub_10007310(3)`) implements a 64-bit Linear Feedback Shift Register (LFSR). If the Least Significant Bit (LSB) of the first byte IV_1 is set to "1", the value `0x20 24 00 10 10 00 00 01` is XORed to the current 64-bit state and the whole LFSR is rotated to the right by one bit. If the LSB is set to "0", the XOR operation is skipped. This process is repeated three times yielding the updated 64-bit value. Once this corresponding

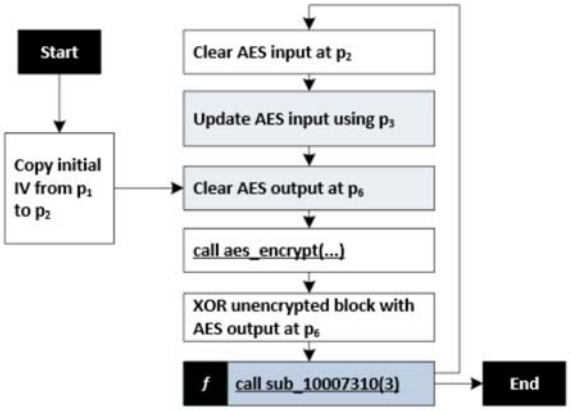


Figure 3.2: Observed execution order of the encryption module of Stratix III FPGAs

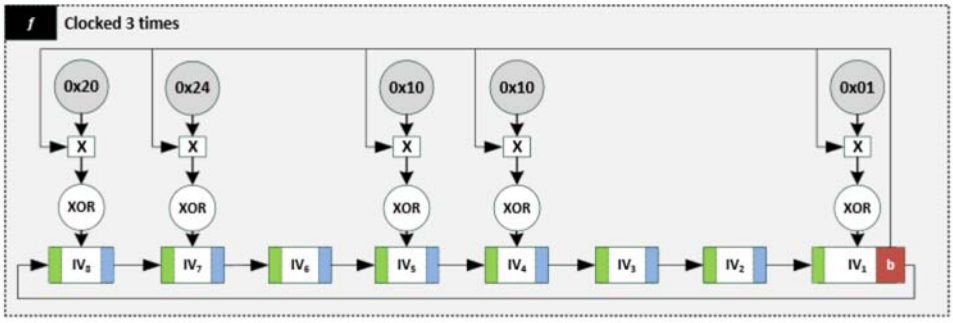


Figure 3.3: Overview of the function f responsible for updating AES inputs

64-bit value is derived, the 64-bit value is duplicated and appended to itself forming the updated 128-bit AES input. While the update mechanism of the 64-bit state is different when compared to Stratix II FPGAs, the same duplication step of the 64-bit state can also be observed in [MOPS13]. Algorithm 1 gives a possible implementation of the function f. With a `C` implementation of f, it took less than one second to compute all IVs from the initial IV which is stored in the file header.

Once this information is revealed, an attacker is able *i*) to conduct side-channel experiments for recovering the bitstream encryption key k and later on he is capable of *ii*) decrypting the entire encrypted bitstream file making IP cloning or manipulation possible. Therefore, in the next section, we provide the required side-channel attack steps.

3.4 Required Side-Channel Steps for Stratix III FPGAs

With the knowledge of computational steps of the AES inputs presented in the previous section, we are able to analyze the Stratix III from a side-channel point of view. Hence, in this section we describe the required steps for a successful key recovery.

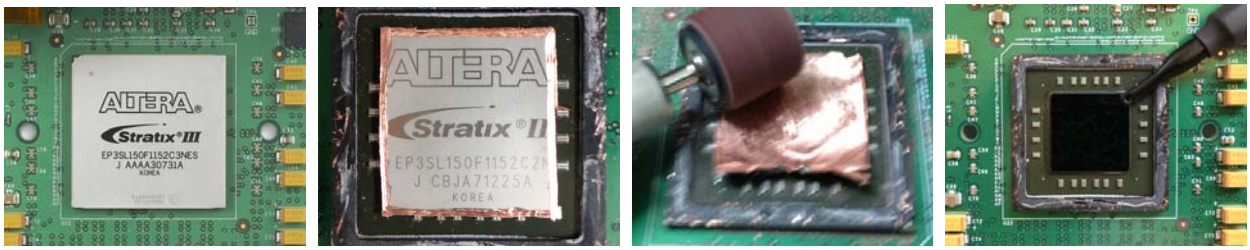
In contrast to the case of Stratix II FPGAs, our targeted development board has not been designed for Side-Channel Analysis (SCA), and hence, does not provide appropriate measurement

Algorithm 1 Pseudo-code for AES input update function f

Input: $IV = IV_8 \,||\, IV_7 \,||\, IV_6 \,||\, IV_5 \,||\, IV_4 \,||\, IV_3 \,||\, IV_2 \,||\, IV_1$, IV_i one byte
Output: Updated IV

for $i = 1 \ldots 3$ **do**
  if $\mathrm{LSB}(IV_1) = 1$ **then**
    $IV \leftarrow IV \oplus 0x2024001010000001$
  $IV \leftarrow \mathrm{rotate\_right}_1(IV)$

points that are well suited for recording the power consumption of the targeted FPGA. Hence, we decided to perform an EM-based side-channel attack. The side-channel leakage is recorded by an EM probe. The FPGA, which is one of Altera's high-density FPGAs, is covered by a metal cap as a heat sink. This cap dampens EM emanations that are observed with a probe on top of the FPGA. Therefore, we removed the cap by means of mechanical tools to directly access the FPGA die, cf. Figures 3.4.(a-c). Another issue was to select an appropriate probe and to localize the best probe position (with low noise level) so that suitable side-channel leakage can be acquired. We experimentally tested several EM probes and numerous positions. The best result in our experiments was achieved with an H-Field near-field RF-R 3-2 probe made by LANGER EMV-Technik [ET13]. Note that we also needed to modify parts of the PCB to support the passive serial configuration mode. By soldering additional wires to the FPGA's configuration pins, we acquired an additional port allowing us to implement and perform the configuration process on our own.



(a) (b) (c) (d)

Figure 3.4: Stratix III FPGA development kit, a) the original FPGA, b) and c) removing the metal cap of the FPGA, d) the decapsulated FPGA with an EM probe at the optimal position

The probe position for which we observed the best side-channel leakage is shown in Fig. 3.4.(d). Since the amplitude of the signal was still low, we used two Mini-Circuits ZFL-1000LN+ amplifiers [MC13] connected in series to obtain EM signals filling the input range of the Digital Storage Oscilloscope (DSO). The used DSO is the same as for Stratix II, i.e., a LeCroy WavePro 715Zi, however, we performed the measurements at a higher sampling rate of 10 GS/s and a bandwidth of 1 GHz.

The measurement scenario is different compared to the case of Stratix II: due to the counter mode used in Stratix II, most of the AES input bytes remain unchanged during one power-up

of the system. This prevents side-channel attacks to effectively recover all key bytes from a single (or few) power-up(s). To this end, we had to perform the measurement for Stratix II by repeating the following scenario: *i*) choose a random IV, *ii*) power-up the Stratix II FPGA, and *iii*) measure a few traces corresponding to the first few encryptions being performed by the FPGA.

In contrast, as stated in Section 3.3, the counter mode is not used by Stratix III FPGAs. Instead, due to the update function f two consecutive AES input blocks differ completely, and thus each byte is essentially randomized. Therefore, the aforementioned measurement process is not required here, and one can collect measurements during a single power-up of a Stratix III FPGA (configured by the original encrypted bitstream).

From the obtained Stratix II results (cf. [MOPS13]), we assumed that the FPGA's hardware decryption circuit is basically the same even though AES-256 is used instead of AES-128. Therefore, we did not see any necessity to adapt the hypotheses used for the Stratix II FPGA. Hence, the remaining task was to check whether the internal architecture we discovered for the Stratix II AES module is indeed identical for Stratix III.

As the first step, we worked in a known-key scenario (256-bit AES key) and tried the model that worked best for Stratix II. In other words, we computed the correlation coefficient between the EM traces measured during one full power-up of the Stratix III, i.e., 365,000 traces, and the Hamming Distance (HD) of consecutive bytes in each row of the AES state after `ShiftRows` in the first round. For more information, we refer to [MOPS13]. The result shown in Fig. 3.5 clearly indicates the correctness of this power model and our guess for the internal architecture.

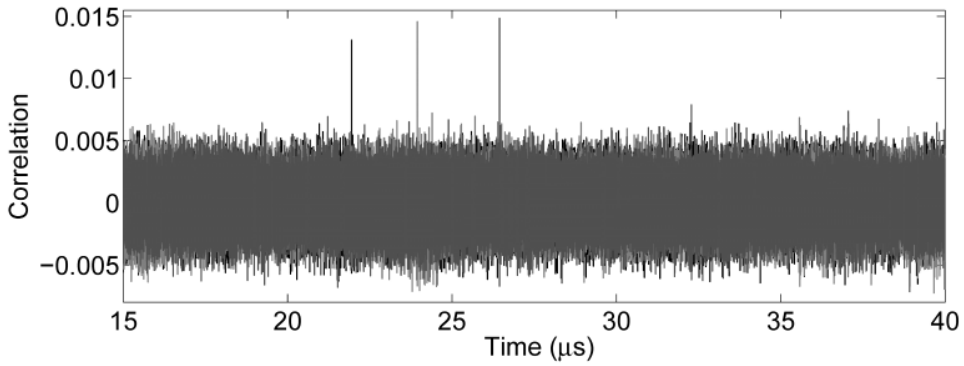


Figure 3.5: Correlation coefficient for the HD of the row-wise consecutive *ShiftRows* bytes using 365,000 traces measured during one power-up of the Stratix III

In order to mount an attack, one has to first recover all 16 key bytes of the first round, which form the first half of the 256-bit key. Due to the structure of AES-256, the second half of the key is used as the round key in the second round. Therefore, after having recovered the first round key, the input of the second `AddRoundKey` for each plaintext can be computed. Hence, the attack is extended to the second round by guessing 16 additional key bytes (second part of the key). These bytes can be recovered using the same power model and the same hypothesis for the architecture. Due to a higher noise level in EM measurements compared to power traces, it might be the case that the traces measured during one power-up are not sufficient for a successful side-channel key recovery.

In this case, the measurement process can be repeated for more power-ups – using the same encrypted bitstream – until the required number of traces has been collected. Similar to the Stratix II, the quality of the traces could also be improved by applying filters. We should mention that we have examined a complete key-recovery attack on different encrypted bitstreams of Stratix III; in the worst case, we required the traces of 5 power-ups to fully recover the 256 bits of the key.

3.5 Conclusion

Having shown the practical feasibility for attacking Stratix III FPGAs, we would like to depict the main differences in Table 3.1 and then conclude our results.

	Altera Stratix II FPGA	**Altera Stratix III FPGA**
Platform	Side-channel Sasebo-B Board	Official Development Board
Process	90nm	65nm
Bitstream Encryption Algorithm	$AES128_k$(bitstream)	$AES256_k$(bitstream)
Block Cipher Mode of Operation	Counter (CTR)	Proprietary derivation function f based on CTR
Algorithm for Key Derivation	$k = AES128_{KEY_1}(KEY_2)$	$k = AES256_{KEY_1}(KEY_2)$
Applied Side-channel Attack	Correlation Power Analysis (CPA) based on power consumption	CPA based on electromagnetic emanation
Required Power-ups	One for each power trace	A few ones
Measurement Duration	A maximum of a few hours.	
Off-line Computational Power	A maximum of a few hours.	

Table 3.1: Required differences between Stratix II and Stratix III FPGAs when performing reverse-engineering and a side-channel attack

Both bitstream encryption schemes do differ, but can be similarly attacked through conducting a side-channel attack. Measuring the power consumption or the electromagnetic emanation of the target device leads to the leakage of the bitstream encryption key k.

Our previous attack on Stratix II FPGAs indicated that SCA countermeasures have been likely ignored during the development phase. Those issues have not been addressed in the newer Stratix III devices. The replaced AES update function f does not add further security to the scheme. It should be noted that recent product families like Stratix V or Aria II probably deploy the same bitstream decryption module. Therefore, it can be assumed that the shown attack is adaptable to those ones. To sum up, both schemes do not provide the desired confidentiality and integrity making IP cloning and (malicious) bitstream manipulations possible.

Since only little is known about the feasibility of bitstream manipulations of third-party designs that execute cryptographic functions, we from now on focus on attacking unknown encoded hardware configurations of Xilinx FPGAs. We introduce our new attack strategies in the next chapters.

Chapter 4

Targeted Bitstream Manipulation Attacks Against Reconfigurable Hardware

As explained in Section 1.2.1, the majority of bitstream encryption schemes, offered by the two market leaders Xilinx and Altera, can be circumvented through side-channel attacks. However, no prior research reports of malicious manipulations of third-party bitstreams, which encode a cryptographic circuit, exist. In this chapter, we present the first malicious bitstream manipulations, which undermine the security of DES and AES cores. The examined cores are executed by SRAM-based Xilinx FPGAs, where the majority was found to be vulnerable to our proposed attack. Our manipulations either lead to weakened encryption schemes or to key recovery. We further evaluate the feasibility of our discovered attack vector by examining 3 different DES and 16 different third-party AES cores.

Contents of this Chapter

4.1 Motivation **29**
4.2 Attack Idea - Substitution of S-boxes of Block Ciphers **31**
4.3 Bitstream Encoding of Xilinx FPGAs **32**
4.4 Exploiting Boolean Functions in FPGA Bitstreams **36**
4.5 Mitigating S-box Substitution Attacks **53**
4.6 Conclusion **59**

4.1 Motivation

In general, bitstream manipulations are feasible for most commercially available Xilinx FPGAs due to the lack of (secure) encryption, integrity, and authentication mechanisms, cf. Section 1.2.1. Even though it can be assumed that the unencrypted bitstream is known to an adversary, whose goal is to compromise the implementation, two major problems remain: first, as explained in Section 1.1, the bitstreams of all commercial FPGAs make use of proprietary file formats. Thus, the attacker has to overcome an obfuscation hurdle. It is not documented how the bits of a proprietary Xilinx bitstream file encode a hardware configuration. Therefore, a tool must exist or has to be developed to partially reverse-engineer the file format in order to be able to analyze the encoded circuit. Second, an attacker has to identify, locate, detect

and meaningfully manipulate the relevant hardware primitives of a complex and potentially unknown hardware circuit.

Since an integrated circuit is split up into thousands of interconnected hardware primitives, which are mapped to the FPGA elements, the analysis and recovery of the implemented functionality is not a straightforward task. To give an example, Fig. 4.1 shows an enlarged segment

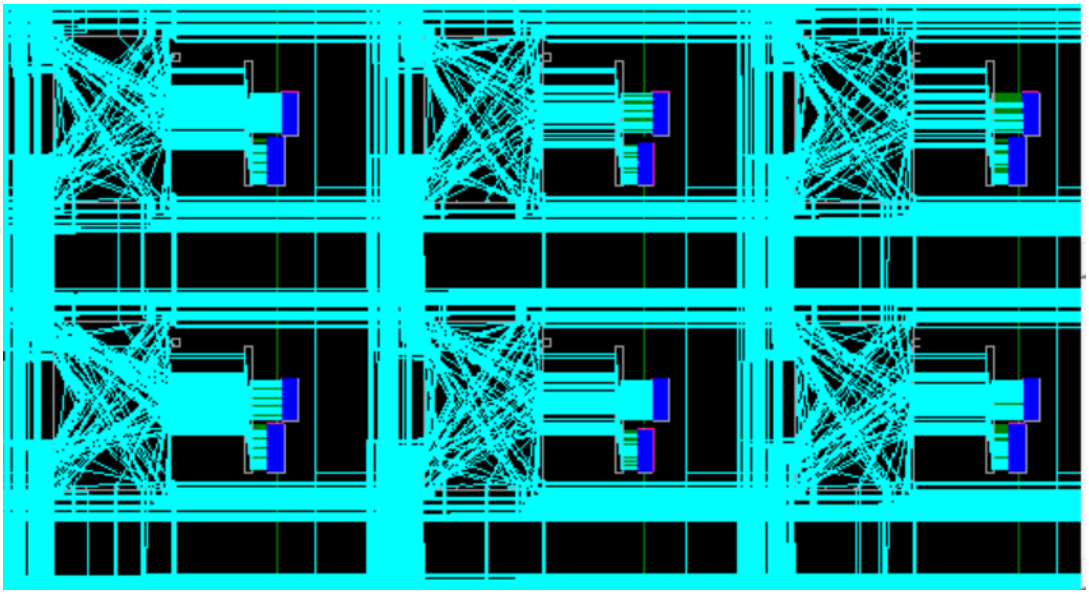


Figure 4.1: A fully mapped and routed hardware configuration showing a more specific part of the FPGA grid with occupied hardware resources implementing an AES design. If the corresponding intermediate file format is given in a practical attack scenario, the hardware configuration could be viewed with the help of Xilinx's FPGA editor. It shows the switch-matrix, routing, and distribution of utilized slices. Note that such a hardware circuit is encoded by the proprietary bitstream file. Therefore, an attacker does not possess this representation

(switch-boxes, slices, and interconnections) of a fully mapped and routed hardware configuration implementing an AES circuit. Under the given circumstances, an attacker cannot simply distinguish between this circuit and other non-cryptographic circuits, especially if the bitstream encoding is unknown. Therefore, it is beneficial for him to identify the work-flow of relevant sub-circuits that are crucial for securely executing cryptographic operations. As we demonstrate later, algorithmic analysis of the low-level hardware elements can facilitate meaningful hardware configuration manipulations.

As explained in Section 1, there are no official tools being capable of parsing, analyzing, and manipulating hardware configurations. Hence, it seems to be infeasible to meaningfully manipulate a cryptographic core if only the corresponding proprietary bitstream (obscured hardware configuration) is given with the goal of undermining the security of an embedded device.

However, our investigations proved these assumptions wrong for many of the examined implementations, since we discovered an attack vector that can potentially be used to undermine the security of practical applications relying on FPGAs. Our findings show that an adversary

neither needs to reverse-engineer the entire bitstream file format nor does he need to understand the complete functionality of a given third-party hardware configuration.

To demonstrate the feasibility of our approach, we utilize the SP601 Evaluation Kit featuring a Spartan 6 FPGA (model XC6SLX16). Our target device offers 2278 slices, 9112 look-up tables, 18 224 flip-flops, and 32 BRAMs with 18Kb of storage each. Our setup is depicted in Fig. 4.2. This FPGA can be configured using the Joint Test Action Group (JTAG) interface.

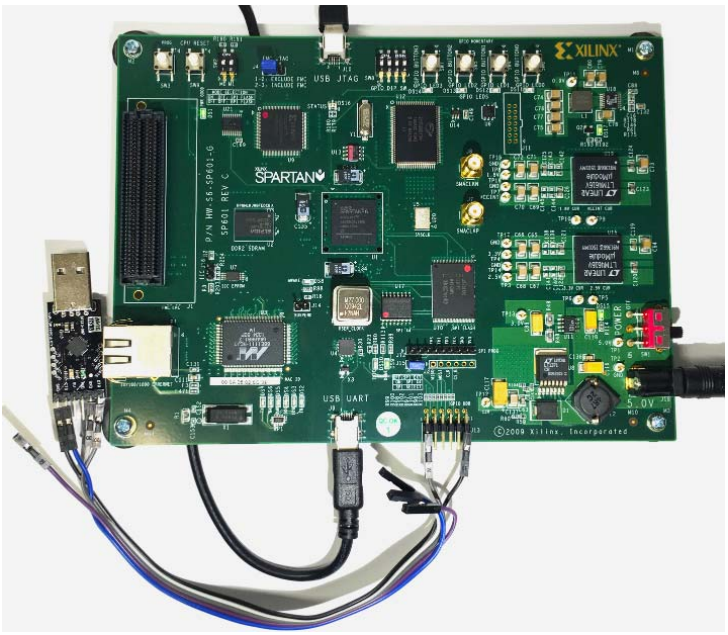


Figure 4.2: SP601 evaluation kit featuring a Xilinx Spartan 6 FPGA serving as target device for our proof-of-concept bitstream manipulation attack

4.2 Attack Idea - Substitution of S-boxes of Block Ciphers

Many modern block ciphers such as AES rely on using cryptographically strong functions. Here, S-boxes are one of the most important primitives of a symmetric block cipher, as they are the only non-linear part ensuring strong cryptographic properties such as confusion mitigating crypt-analytical attacks. If the S-boxes of an AES implementation can be replaced by a linear function, the manipulated algorithm becomes cryptographically weak, i.e., the corresponding ciphertexts can be decrypted with one or a few known plaintext-ciphertext pairs. Our attack relies on manipulating the S-boxes directly in a third-party bitstream, subsequently leading to a weak cryptographic algorithm or key leakage. One main strength of this attack procedure is that there is no need to take internal routing information into consideration.

To undermine the practical relevance, we target DES, AES, and 3DES which are the most widely used block ciphers in current and legacy applications. Note that the attack is limited to scenarios in which encryption and decryption are computed by the same device, e.g., USB flash drives, solid-state disks, or encrypted cloud storage. The manipulations can also be used in systems, where all involved devices can be altered. In case of attacking AES-128, fixing all S-

box outputs to zero leads to key leakage as explained in Section 4.4.6. A key recovery is feasible if the target device automatically loads and processes the non-accessible cryptographic key.

4.3 Bitstream Encoding of Xilinx FPGAs

LUTs constitute an important hardware primitive in Xilinx FPGAs. They are mainly responsible for implementing the combinatorial logic of a hardware configuration. When combining LUTs with multiplexers, an FPGA can implement more complex combinatorial logic functions. FPGAs use thousands of LUTs that either implement logic functions or serve as distributed RAM. embedded in a slice, cf. Figure 4.6.

Since LUTs implement the primary functionality of an FPGA design, they are promising targets for an attacker who intends to maliciously change the functionality of a third-party bitstream. As many practical applications rely on Xilinx FPGAs, it is also quite important to analyze the LUTs contents in terms of security. As indicated above, in the real world an attacker usually only possesses the bitstream of a hardware configuration. Section 4.3.1 provides the necessary information on how to derive all LUT bit positions enabling the manipulation. Note that the LUT content bits are distributed over the bitstream following predefined, but unknown patterns. We successfully obtained the bitstream encoding of two different FPGA devices with different hardware architectures that are based on 4-input,1-output and 6-input,1-output LUT principle. Note that the described approach can be applied for any Xilinx FPGA. Additionally, we explain how to discover the bitstream encoding that describes how the BRAM content is configured.

4.3.1 Extracting the LUT Encoding from a Bitstream

As an example, we consider a Xilinx Spartan 6 FPGA using a 6-input,1-output architecture. It is part of the SP601 development board, cf. Section 4.1. It comes with the following hardware architecture properties:

- Two slices are interconnected to a switch-matrix and can form an 8-bit data bus each.
- Four 6-input,1-output LUTs are embedded in one slice with the ability to store 4×64 bits and to implement an $8 \mapsto 1$ Boolean function.
- Three dedicated multiplexers within one slice being capable of combining LUT outputs.

To extract the bitstream encoding of all LUTs, initially an attacker has to conduct a learning phase. The approach of deriving the LUT contents from a bitstream relies on generating appropriate hardware configurations specifying the rules of reconfiguring the hardware for the given FPGA target. The hardware configuration can be used to manually configure any LUT with an arbitrary 6-input, 1-output Boolean function. Listing 4.1 provides an example by showing the configuration of four LUTs that are embedded in one slice. Note that the presented partial hardware configuration uses a fictional syntax for the sake of simplicity.

```
FPGA design "minimal_lut_implementation",
instance "slice_X_Y",
config {
 content of LUT1 = {0x0000000000000000} // 64 inputs output a logical '0' on evaluation
 content of LUT2 = {0xFFFFFFFFFFFFFFFF} // 64 inputs output a logical '1' on evaluation
 content of LUT3 = {0x0000000000000001} // Only input x = 63 outputs a logical '1' on evaluation
 content of LUT4 = {0x8000000000000000} // Only input x = 0 outputs a logical '1' on evaluation
}
```

Listing 4.1: Hardware configuration example for specifying LUT contents

As further illustrated by Listing 4.1, each LUT of one slice can be configured by specifying a 64-bit LUT content representing a Boolean function. An attacker configures two different Boolean functions for exactly one LUT. Thus, he has to create two different hardware configurations. Both are used to let the Xilinx tools generate two slightly different bitstreams. In the next step, the generated bitstreams can be compared to extract the bitstream encoding. It should be noted that for each input value one output bit is stored in the bitstream. The first hardware configuration sets a LUT content to always output a logical zero (`0x0000000000000000`) for all 64 input values (6-input,1-output architecture). All 64 outputs bits together specify the LUT content. In this case, 64 "0"-bits, which is the resulting LUT content of the currently discussed Boolean function, are stored in the generated (final) bitstream. Analogously, in a 4-input,1-output architecture (16 input values), only sixteen "0"-bits are stored for one LUT content in the bitstream.

Now, a second hardware configuration, only differing in the specified LUT content, is generated. Instead of fixing the truth table to 64 zeros, the Boolean function is chosen in such a way that it always outputs a logical one regardless of the input value (`0xFFFFFFFFFFFFFFFF`). Again, the corresponding bitstream is generated. This leads to the storage of 64 "1"-bits in the bitstream.

When comparing both bitstreams, one can observe that exactly 64 bits toggle from "0" to "1", while all other bits remain unchanged. Therefore, one can easily determine and store the bitstream encoding of all 64 bits that are related to one LUT, but obviously the correct order of these 64 bits stays unclear. It is important to know the correct order to be able to reconstruct the correct Boolean function. Thus, an attacker has to extend the previous approach: now, the idea is to additionally create 64 bitstreams from 64 slightly different hardware configurations.

Each hardware configuration is chosen to set an appropriate value (cf. Table 4.1) for the same LUT such that only one bit of the LUT content is set, while all other 63 bits are cleared. All 64 generated bitstreams can be compared with the bitstream, whose LUT content bits are all cleared, because then only one bit toggles.

To be more precise, each LUT content bit is recovered separately by observing the toggling positions, and thus, the correct order can be revealed. In a 6-input,1-output architecture, one should generate 65 bitstreams for each LUT, while for a 4-input,1-output architecture only 17 bitstream generations are appropriate. This approach has to be repeated for all given LUTs of the underlying FPGA in order to be able to extract all LUT contents from a third-party bitstream.

Note that the bits of one LUT are not necessarily stored next to each other in the bitstream. Instead, they are distributed in the bitstream file by following specific offsets rules. For example, the first bit of one LUT content can be stored in the bitstream at position (Byte Y, Bit 0), while

Generation of	Content of exactly one LUT	Meaning
Bitstream 1	0x0000000000000001	Only input 0 outputs a 1
Bitstream 2	0x0000000000000002	Only input 1 outputs a 1
Bitstream 3	0x0000000000000004	Only input 2 outputs a 1
...	...	...
Bitstream 63	0x4000000000000000	Only input 62 outputs a 1
Bitstream 64	0x8000000000000000	Only input 63 outputs a 1
Bitstream 65	0x0000000000000000	Each input outputs a 0

Table 4.1: Generating 65 bitstreams for one LUT

the second bit may be located at position (Byte $Y-8$, Bit 5). We were able to practically verify the correctness of our extracted bitstream encoding for any single LUT. This can be done by setting a random configuration for any LUT (in a hardware configuration describing all LUTs) and by creating the corresponding bitstream. Then, the LUT contents can be parsed from the bitstream and compared to the LUT contents of the previously generated human-readable hardware configuration. Algorithm 2 illustrates this straightforward but time-consuming approach in more detail. It can be used for k-input, 1-output based LUTs with $k \in \{4, 6\}$.

Algorithm 2 LUT encoding extraction for any k-input,1-output Xilinx FPGA with $k \in \{4, 6\}$

Input: k-input,1-output FPGA device file (report file) containing coordinate information of all available LUTs on the FPGA grid with $k \in \{4, 6\}$
Output: Bitstream encoding *lut_encoding* of all LUTs allowing to dump all LUT contents from any third-party bitstream belonging to one specific FPGA model.

---

configure_lut_content(a,b) creates a hardware configuration so that the LUT content is set to b for the a^{th} LUT on the FPGA grid.
bitstream_lut_zero: Temporary reference bitstream file with zeroized LUT content.
bitstream_tmp: Temporary bitstream file with LUT content for which only one bit is set.

1: **for** *lut_index* = 0 to NUM_OF_LUTS - 1 **do**
2: Create minimal hardware configuration using configure_lut_content(*lut_index*, 0)
3: Generate bitstream *bitstream_lut_zero* from previously generated hardware configuration.
4: **for** *bit* = 0 to $2^k - 1$ **do**
5: *lut_content* = 2^{bit}
6: Create minimal hardware config. using configure_lut_content(*lut_index*, *lut_content*)
7: Generate bitstream *bitstream_tmp* from previously generated hardware configuration.
8: Compare *bitstream_lut_zero* and *bitstream_tmp* and derive toggled bit position *pos*
9: Store toggled position *lut_encoding*[*lut_index*][*bit*] = *pos*
10: **return** *lut_encoding*

A more sophisticated and considerably faster method is to learn the offset patterns of one or several LUTs that can be applied to all other LUTs. For a mid-sized FPGA the computation time is approximately 1-2 days, whereas the straightforward approach needs much longer. It

must be highlighted that the learning phase has to be performed only once per FPGA device and can be applied on any third-party bitstream fitting to the FPGA device.

4.3.2 Extracting the BRAM Encoding from a Bitstream

A common implementation strategy for storing data in a hardware configuration is to use the dedicated BRAM of the FPGA. Hence, we describe how the corresponding bitstream encoding of the BRAM can be obtained. Knowing this encoding, critical data like cryptographic keys or S-boxes can be potentially extracted from the bitstream, since one obtains the plain representation of the BRAM content.

Suppose that a fixed AES-{128,192,256} key with its corresponding subkeys has been placed in the BRAM. An attacker then may easily verify the presence of these subkeys by finding XOR-dependencies. This can be done with a tool called *aesfindkey* written by Haldermann *et al.* [HSH+09].

For the reverse-engineering process, we need to create a VHDL file in order to derive the appropriate low-level hardware configuration description that again serves for learning the bitstream encoding. A simplified VHDL code example, realizing an array of zeros, is depicted in Code Listing 4.2. When using this VHDL code, the BRAM of the FPGA is filled with the specified bytes of the given signal *rom_array*. The corresponding low-level hardware configuration of this design is generated to serve as input for Algorithm 3.

```
architecture rtl of BRAM_zero is
...
type rom_array is array (0 to NUM_OF_BYTES) of
std_logic_vector(7 downto 0);
signal ROM : rom_array := (
               X"00", X"00", X"00", X"00",
               ...
               X"00", X"00", X"00", X"00"
);
...
process(clk)
...
if(rising_edge(clk)) then
               data <= ROM(conv_integer(addr));
end if;
end process;
```

Listing 4.2: AES S-box instantiation in the BRAM

The idea of obtaining the bitstream encoding of the BRAM content is similar to the approach of extracting the encoding of the LUT contents. Again, an attacker can create various low-level hardware configurations, for which he changes all memory values bitwise. For each change, the bitstream is generated and the corresponding toggling bits are observed. Algorithm 3 shows a generic approach for extracting the bitstream encoding. Having obtained the encoding, we verified the correctness for several Xilinx FPGAs belonging to different families. Note that there are certain setups for the memory layout that can be chosen by the user. We could verify that the contents of the BRAM can be extracted – regardless of the chosen memory layout.

Algorithm 3 BRAM encoding extraction for Xilinx FPGAs

Input: Low-level hardware configuration *low_conf* observed from the synthesis/implementation step of translating multiple instances of the VHDL code in Listing 4.2
Output: Bitstream encoding *BRAM_encoding* of all BRAM instances allowing to dump all BRAM contents from any third-party bitstream belonging to one specific FPGA model.

---

configure_bram_content(a,b) creates a hardware configuration so that the BRAM content is set to b for the a^{th} BRAM block on the FPGA grid.
bitstream_bram_zero: Temporary reference bitstream file with zeroized BRAM block.
bitstream_tmp: Temporary bitstream file with BRAM block for which only one bit is set.

1: Generate bitstream *bitstream_bram_zero* from previously synthesized/implemented hardware configuration *low_conf*.
2: **for** *bram_block* = 0 to NUM_OF_BRAM_BLOCKS - 1 **do**
3: **for** *bit* = 0 to NUM_OF_BITS_PER_BRAM_BLOCK - 1 **do**
4: Copy *low_conf* and apply configure_bram_content(*bram_block*, 2^{bit})
5: Generate bitstream *bitstream_tmp* from previously adapted hardware configuration.
6: Compare *bitstream_bram_zero* and *bitstream_tmp* and derive toggled bit position *pos*
7: Store toggled position *BRAM_encoding*[*bram_block*][*bit*] = *pos*
8: **return** *bram_encoding*

Note that Algorithm 3 has to be executed only once per FPGA device. With the help of the recovered bitstream encoding (describing the contents of the BRAM), an attacker is able to extract and modify the contents of a bitstream file.

While the intention of this chapter is not to provide detailed insights into the proprietary bitstream file format, it illustrates the feasibility of the approach. In summary, this approach is generic and applicable, using the standard FPGA design flow. The next sections explain the detection of DES and AES S-boxes whose precomputed output values are commonly distributed over various LUTs on the FPGA grid. Once the detection is successful, an attacker can maliciously manipulate a third-party bitstream. Therefore, we evaluate the feasibility of detecting S-boxes in a third-party hardware configuration for which we assume that only the corresponding third-party bitstream is given.

4.4 Exploiting Boolean Functions in FPGA Bitstreams

The difficulty of detecting S-boxes in third-party hardware configurations varies with the type of S-boxes that are used by a block cipher. We found that whenever precomputed S-box tables are used by a hardware implementation, we are able to fully detect them. If the FPGA's architecture uses y-input,1-output LUTs (2^y bits of available memory each) and x-input,1-output Boolean functions (2^x bits of required memory) need to be placed and routed, two cases may occur:

Case 1 $x = y$**:** If x is equal to y, then the whole Boolean function can be implemented in exactly one LUT. The LUT contents can be analyzed. It is thus straightforward to detect

single x-input,1-output Boolean functions. This is the case for S-boxes of the DES algorithm in a 6-input,1-output architecture ($x = y = 6$). We therefore introduce a pattern search algorithm in Section 4.4.1 that is able to detect all DES S-box instances.

Case 2 $x > y$**:** If x is larger than y, then it is a more challenging task to find x-input,1-output Boolean functions within an FPGA design. Due to the dimensions, one Boolean function must be split up into $\frac{2^x}{2^y}$ LUTs that have to be combined by $\frac{2^x}{2^y} - 1$ multiplexers. We have developed a search strategy for Boolean functions that exceed the common 16-bit (4-input,1-output) and 64-bit (6-input,1-output) memory limitations of one LUT. This technique is described for the AES in Section 4.4.4. AES uses 8-input,8-output (eight 8-input,1-output Boolean functions) S-boxes and thus $\frac{2^8}{2^6} = 4$ LUTs can implement one S-box column within one slice of a Spartan 6 FPGA.

4.4.1 Detection of DES S-boxes

This section covers the detection of DES S-boxes from a third-party bitstream that configures our target device. The DES algorithm is briefly recapped in Section 4.4.3. DES uses eight different predefined 6-input,4-output S-boxes. Since our target device provides 6-input,1-output LUTs (64 bits of memory), one DES S-box column[1] can fit into one 64-bit LUT. Therefore, one complete DES S-box (4x64-bit columns) can be implemented by four 64-bit LUTs. Hence, a round-based DES implementation requires 32 LUTs for all eight S-boxes. A general 6-input,4-output S-box is illustrated in Table 4.2. Note that each column, which we refer to as LUT_0, LUT_1, LUT_2, and LUT_3, stores a unique 64-bit sequence describing a Boolean equation. These might be the fixed bit-sequences of a DES S-box. Each value a_i, b_i, c_i, and d_i with $i \in \{0, ..., 63\}$ stores exactly one bit.

Input values						**Output columns**			
$\mathbf{x_5}$	$\mathbf{x_4}$	$\mathbf{x_3}$	$\mathbf{x_2}$	$\mathbf{x_1}$	$\mathbf{x_0}$	$\mathbf{LUT_0}$	$\mathbf{LUT_1}$	$\mathbf{LUT_2}$	$\mathbf{LUT_3}$
0	0	0	0	0	0	a_0	b_0	c_0	d_0
0	0	0	0	0	1	a_1	b_1	c_1	d_1
...	...	...	...	...	...	...	...	...	...
1	1	1	1	1	1	a_{63}	b_{63}	c_{63}	d_{63}

Table 4.2: General shape of a 6-input,4-output S-box

To give an example, the four patterns of the first DES S-box are as follows.

- $LUT_0 = (a_{63}, a_{62}, ..., a_0)_2 =$ `0x869D497A86E67619`
- $LUT_1 = (b_{63}, b_{62}, ..., b_0)_2 =$ `0xB0C7871B497826BD`
- $LUT_2 = (c_{63}, c_{62}, ..., c_0)_2 =$ `0x27E9D492609F1F29`
- $LUT_3 = (d_{63}, d_{62}, ..., d_0)_2 =$ `0x917BE9066F81B478`

[1] It is equal to the LUT content describing one output bit of the S-box. A DES S-box column can be understood as $LUT_0 = (a_{63}, a_{62}, \ldots, a_0)_2$, cf. Table 4.2.

Note that each 64-bit pattern is unique for all 32 DES S-box columns. As demonstrated before, an adversary can learn the bitstream encoding for all LUTs, thus he can now extract all LUT contents from a third-party bitstream and analyze the corresponding patterns. The key idea is that an attacker can try to match all specific DES S-box patterns with all extracted LUT contents. If there is a match, one S-box column is successfully detected.

Due to the uniqueness of all 64 bits forming a DES S-box column, the likelihood of matching a false positive, i.e., falsely matching one DES S-box column pattern with an unrelated Boolean function is negligible. This is because there are $2^{64} = 18\ 446\ 744\ 073\ 709\ 551\ 716$ different possible patterns. In contrast to that, an attacker only needs to test 32 unique reference search patterns in the best case or $23040 = 720 \cdot 32$ in the worst case. During our investigations it turned out to be insufficient to only match the 32 possible DES patterns with all extracted LUT contents. The reason is that the truth table bits of a LUT (for example LUT_0) are usually permuted due to permuted input signals, which we also refer to as input permutation as briefly introduced in Section 2.1.1. Hence, a different input permutation leads to different permuted truth table patterns within the bitstream. Therefore, a basic pattern matching fails to detect all DES S-boxes.

The synthesizer always aims at finding an optimal routing path. For this purpose, the router needs to permute the input signals of most LUTs, i.e., the input signals of a DES S-box. Therefore, the previously discussed pattern matching approach needs to be extended. The LUT content (extracted from a bitstream) has to be considered as a permutation of the search pattern. The permutation can be computed by a function that we refer to as $\pi_p(\cdot)$. Our further analysis showed that $\pi_p(\cdot)$ is computed by the synthesizer as described by Algorithm 4.

Hence, instead of matching one DES S-box column pattern, one needs to test $720 = 6!$ permuted patterns. One may think that an attacker needs further knowledge of the FPGA's routing to obtain the given unknown input permutation, but this is not necessary, since our search algorithm can reveal it. In summary, an attacker needs to precompute all possible permutations for all given DES patterns which are in total $23040 = 32 \cdot 6!$ and has to compare them with all extracted LUT contents. The corresponding DES pattern search algorithm is depicted in Algorithm 5.

4.4.2 Results of DES S-box Detection

Table 4.3 illustrates our ability to locate all DES S-boxes from three tested FPGA implementations for which we verified that all S-box instances were successfully found. Besides the exact

Impl.	Architecture	Found 6-input,1-output LUTs	Detection rate
#1	Round-based	32	100 %
#2	Round-based	32	100 %
#3	Unrolled (16 rounds)	$16 \cdot 32$	100 %

Table 4.3: Overview of evaluated DES implementations

location of the LUTs on the FPGA's grid, we obtained the exact permutation order of the corresponding input pins (without any knowledge of the routing) for every single S-box column. The obtained knowledge is extremely useful for an attacker, e.g., if EM-based side-channel attacks are used. Knowing the exact location an attacker can try to locate the best probe position for the measurement while a target device performs its cryptographic operations.

Algorithm 4 Computation of $\pi_p(\cdot)$ of the p^{th} permutation with $p \in \{0, 1, \ldots, 719\}$)

Inputs: A 64-bit reference LUT pattern that we refer to as $LUT[x] \in \{0, 1\}$ with $x \in \{0, 1, \ldots, 63\}$, Permutation index p
Output: The 64-bit pattern of the p^{th} permutation of a reference LUT pattern

---

A = get_bit(*value*, i) returns the i^{th} bit of *value* to A
set_bit(*value*, i) sets the i^{th} bit of *value*

```
1:  permutations[0] = (0,1,2,3,4,5,6)
2:  permutations[1] = (0,1,2,3,4,6,5)
3:                    ...
4:  permutations[719] = (6,5,4,3,2,1,0)
5:  // Process each input value of LUT individually
6:  for input_index = 0 to 63 do
7:      // Only if a bit is set a re-ordering must be performed
8:      if LUT[input_index] == 1 then
9:          // Obtain permuted input position to be set for the permuted pattern
10:         input_index_perm = 0
11:         for input_column = 0 to 5 do
12:             tmp = get_bit(input_index, permutations[p][input_column])
13:             set_bit(input_index_perm, tmp)
14:         // Set bit in permuted 64-bit LUT pattern with previously determined input position
15:         LUT_perm[input_index_perm] = 1
16: Return LUT_perm
```

The bitstream can also expose information about the utilized architecture of the design. Knowing the architecture can indicate whether an implementation is round-based, unrolled, or other cryptographic instances are running in parallel. Note that one can also easily identify the S-boxes of a 3DES architecture.

A slightly adapted version of Algorithm 5 can also be applied to a 4-input,1-output LUT FPGA architecture. In this case, we evaluated whether one can also detect the corresponding 4-bit-to-4 bit S-boxes of the lightweight cipher PRESENT [BKL+07]. Again, we could identify all S-box instances in the bitstream. As long as a Boolean function is known to an attacker and as long as its patterns do not overlap with the patterns of other Boolean functions, he is able to successfully find it in the bitstream. In case the S-boxes can be altered by attacker, this represents a security risk from a cryptographic perspective, since the S-boxes are usually the only non-linear function of a block cipher. In the next section, we describe how to weaken the DES algorithm through replacing its S-boxes in the bitstream.

4.4.3 Manipulating DES S-boxes

Since the DES and especially the 3DES algorithms are still widely used, e.g., in financial systems and SSL/TLS applications, both algorithms represent an attractive target to be weakened in FPGA bitstreams. Again, to undermine the security of DES, an attacker should be able to

Algorithm 5 Detection of all 8 DES S-boxes being distributed over 32 6-input,1-output LUTs

Inputs: 3rd-party bitstream *bs*, Bitstream encoding *lut_encoding*
Output: Flagged LUTs on the FPGA grid that implement DES S-box columns

---

$S_p(x)$ represent the DES S-boxes with $p \in \{0, 1, \ldots, 7\}$
$S_p^j(x)$ denotes to the j'th output bit (column) of $S_p(x)$ with $j \in \{0, 1, 2, 3\}$
$\pi_i(\cdot)$ denotes the i'th permutation out of 720
get_lut_content$(\cdot)$ extracts all 64-bit contents of all LUTs of a bitstream using *lut_encoding*

1: //Generate DES search patterns
2: **for** *sbox* = 0 to 7 **do**
3: **for** *column* = 0 to 3 **do**
4: DES_pattern[*sbox*][*column*] = $S_{sbox}^{column}(63)||S_{sbox}^{column}(62)||\ldots||S_{sbox}^{column}(0)$
5: //Dump all LUTs from bitstream
6: lut_content[NUM_OF_LUTS]← get_lut_content(*bs*)
7:
8: //Search for DES pattern
9: **for** *lut_index* = 0 to NUM_OF_LUTS - 1 **do**
10: pattern_of_interest = lut_content[*lut_index*]
11: **for** *p_index* = 0 to 719 **do**
12: **for** *sbox* = 0 to 7 **do**
13: **for** *column* = 0 to 3 **do**
14: **if** ($pattern\_of\_interest == \pi_{p\_index}(DES\_pattern[sbox][column])$) **then**
15: flag_LUT_as_DES_sbox_column(*lut_index, sbox, column*)

directly modify the bits of the bitstream related to the DES S-boxes. As demonstrated in Section 4.3.1 and Section 4.3.2, we can easily locate these bits. Figure 4.3 shows the general Feistel structure of the DES algorithm. The DES algorithm processes a 64-bit plaintext using a 56-bit main key [NIS99]. Sixteen subkeys are derived from the main key by following a fixed scheduling plan. The basic properties of diffusion and confusion are realized by the f-function. Each of the 16 round keys is processed by this function. Figure 4.4 shows the internal structure of the DES f-function. As can be seen, all eight S-boxes process an intermediate value that has been previously XOR-ed with a subkey.

The goal is to directly modify all DES S-boxes in a way that a modified ciphertext (computed by this modified DES) can be decrypted by an adversary without the need of knowing the secret key. This holds for all plaintext blocks being encrypted by the modified algorithm. The idea of DES S-box alteration is discussed by Kerins *et al.* [KK06], which we briefly present here. If all S-boxes $(S_0, S_1, ..., S_7)$ can be modified in such a way that they always output a zero – regardless of all 64 possible input values – an attacker has successfully performed a malicious alteration of the DES algorithm. To be more precise, the following modification should be applied to the DES S-boxes implemented in the FPGA bitstream:

$$\text{Substitute } S_p(x) \text{ by } S_p^{zero}(x) = 0, \quad \forall x \in \{0, 1, \ldots, 63\}, \forall p \in \{0, 1, \ldots, 7\}$$

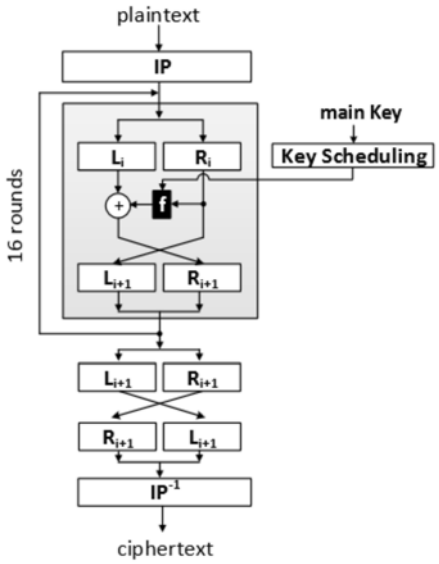


Figure 4.3: Overview of the DES encryption algorithm

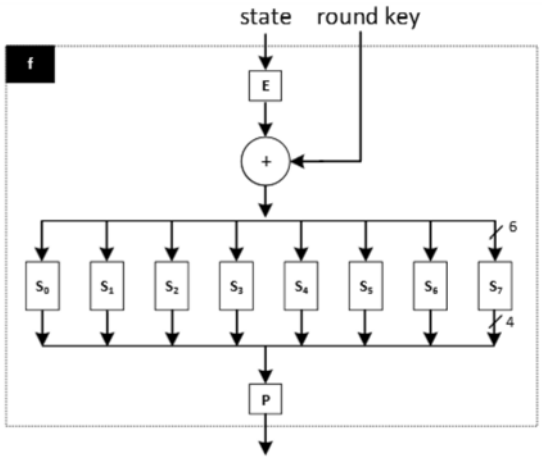


Figure 4.4: DES round function f

Due to the proposed manipulation, the whole DES algorithm turns into a key-less permutation. The modified DES is visible in Figure 4.5. In a normal operating f-function, the S-box outputs

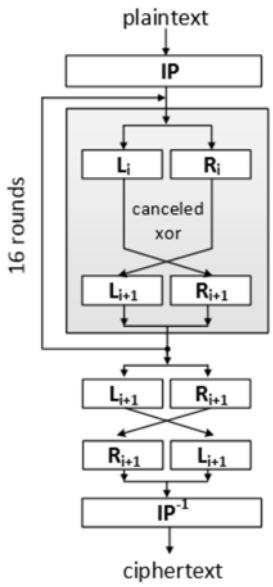


Figure 4.5: Modified DES with canceled f-function

(32 bits) are permuted according to the mapping rules of permutation P, cf. Figure 4.4. The evaluated result of P is concurrently the output of the function f. Since in the modified version

all S-boxes outputs are zero, the output of the permutation P is also completely zero. Hence, the output of the function f is zero, independent of the input and subkey. Because a zero output of f is XOR-ed with the left state L_i, it remains unchanged as XOR-ing a value with zero is equal to the identity function. Thus, the state after IP$(\cdot)$ is not affected when having processed all 16 DES rounds. This is because the number of swaps is even. In the end, a final swap is performed, which is followed by a permutation denoted by IP$^{-1}(\cdot)$.

The following two equations compare the computation steps of a normal DES encryption with the one of a modified $\widetilde{\text{DES}}$-encryption using $S_j^{zero}(i)$. The modified encryption only applies three permutations on the plaintext (denoted by p) that can be easily inverted by an attacker.

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

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

An attacker has to perform the following computation to obtain the plaintext from any weakly encrypted ciphertext $\widetilde{c}$:

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

This attacks works likewise for a manipulated 3DES encryption that is computed as follows [NIS99]:

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

A plaintext from the modified 3DES with $S_j^{zero}(i)$ can also be easily recovered by computing:

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

In a round-based DES implementation (cf. Table 4.3), an attacker only has to modify eight S-boxes (or: 32 decomposed LUTs in a 6-input,1-output architecture) within the bitstream to significantly weaken the DES algorithm[2]. The S-box changes were directly applied on the bitstreams and we verified that the alteration of the design was successful. The presented attack of the DES algorithm works for both a LUT-based implementation and for an implementation based on BRAM. Due to the fact that the DES algorithm does not exhibit any inverse S-boxes, the decryption of faulty ciphertexts will reveal the expected plaintexts. This severe bitstream manipulation may remain undetected in applications such as data storage, where encryption and decryption are performed by the same device, or if all ciphers in a given system are modified in this way. Possible countermeasures include self-tests or integrity checks.

[2]Potentially more S-box instances have to be modified in the bitstream, depending on the design architecture, e.g., in an unrolled implementation.

4.4.4 Detection of AES S-boxes

In the following, the detection of decomposed AES S-boxes, which are realized in a 6-input,1-output architecture, is considered. At first the attacker computes the 8-input,8-output AES S-box. The decomposition is necessary, because the degree x of the Boolean function is higher than the number of LUT inputs y. Thus, Case 2 of Section 4.4 holds here. Each of the 8 AES S-box columns represents an 8-input,1-output Boolean function and is stored as a 256-bit value denoted by $(a_{255}, a_{254}, ..., a_0)$. Such an AES S-box column is similar to the bit-string of Table 4.2 but is larger (256 bits instead of 64 bits). The AES S-box columns have to be split up into 4 LUTs each, because one LUT of the FPGA can only store 64 bits. All LUTs are denoted by LUT_i with $i \in \{0, 1, 2, 3\}$ that need to be multiplexed with 2 input bits and hence the hardware layout requires two multiplexer stages, which are available in one slice . They are denoted by μ_1 and μ_2. The purpose of these two input bits is to select the correct LUT_i once an 8-input value is processed during one clock cycle. Chosing the correct set for computing the correct output value of one input is realized with the help of 3 multiplexers, cf. Fig. 4.6.

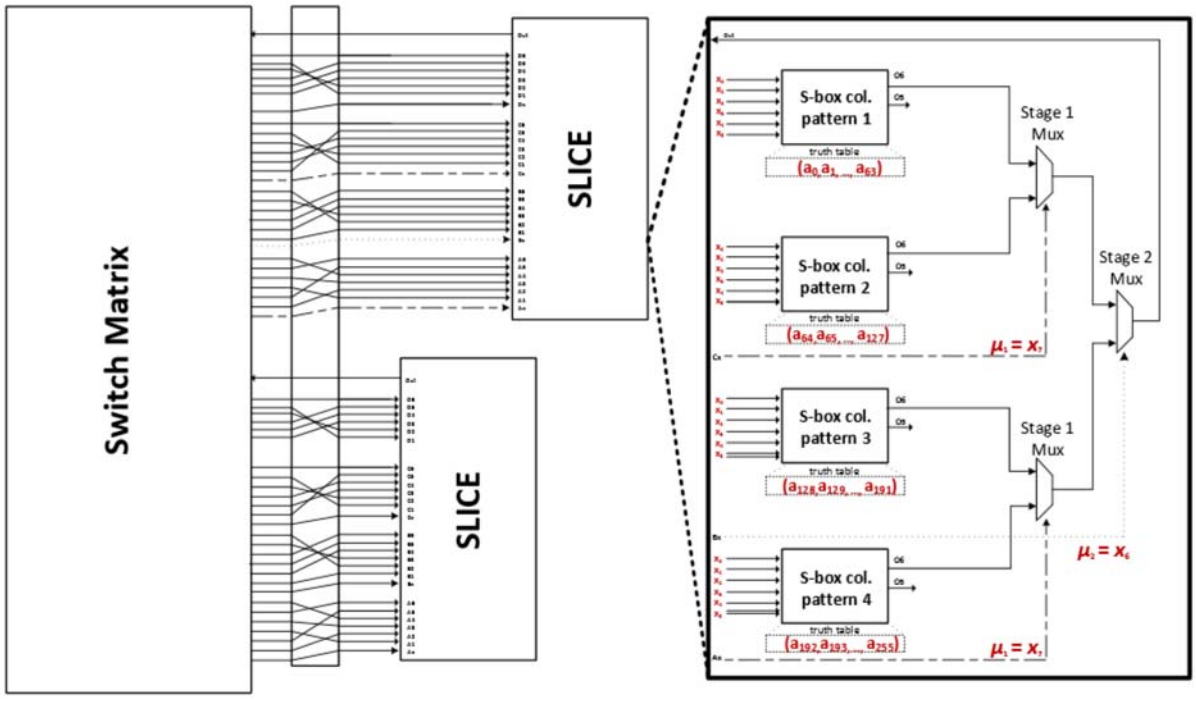


Figure 4.6: Simplified overview of a slice of a Spartan 6 FPGA realizing an 8-input,1-output Boolean function (256 bits of memory) with four 6-input,1-output LUTs (64 bits of memory each). Our example implements one S-box output column of AES. Eight of those instances are needed to implement one AES S-box

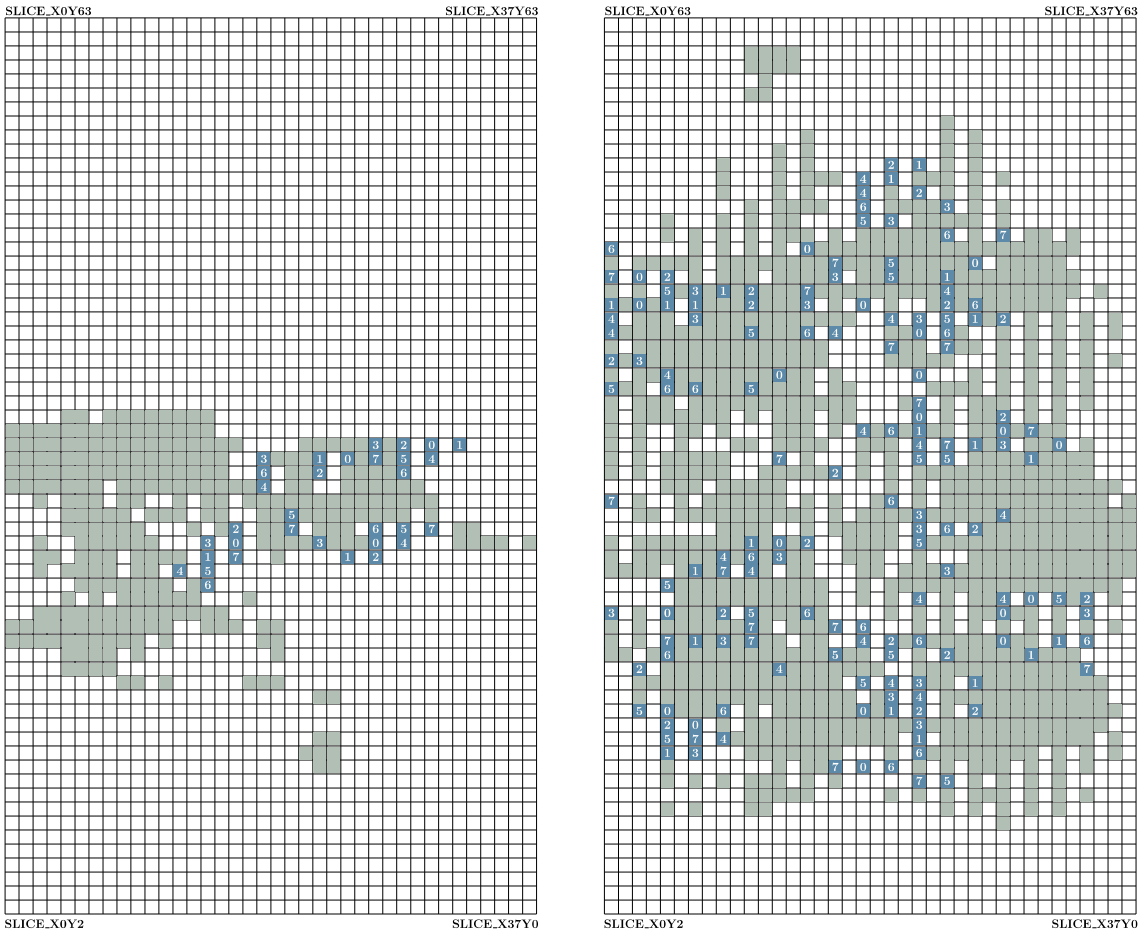
The synthesizer has to chose 64 bits from $(a_{255}, a_{254}, ..., a_0)$ and assigns them to one LUT which implements the first part of one AES S-box column. Once assigned, the next 64 bits are assigned to another LUT. In total, this is performed 4 times. An AES S-box column has eight input bits that we denote by (x_7, x_6, ..., x_0). We pick the most straightforward example that

can occur to explain how the synthesizer distributes the 256 bit values $(a_{255}, a_{254}, ..., a_0)$ to four 64-bit LUTs. One of the most simple configurations is as follows.

- $LUT_0 = (a_{255}, ..., a_{192})$ selected by $(\mu_1, \mu_2) = (0, 0)$
- $LUT_1 = (a_{191}, ..., a_{128})$ selected by $(\mu_1, \mu_2) = (0, 1)$
- $LUT_2 = (a_{127}, ..., a_{64})$ selected by $(\mu_1, \mu_2) = (1, 0)$
- $LUT_3 = (a_{63}, ..., a_0)$ selected by $(\mu_1, \mu_2) = (1, 1)$
- The multiplexer configuration is $\mu_1 = x_7$ and $\mu_2 = x_6$
- The remaining input bits (x_5, x_4, ..., x_0) are not permuted

If the multiplexer configuration is chosen differently due to routing optimizations, then the assignment of $(a_{255}, a_{254}, ..., a_0)$ to the lookup-tables LUT_0, LUT_1, LUT_2, and LUT_3 has to be re-organized by the synthesizer. An attacker would observe that the tools proceed as follows: for each AES S-box input value $x \in \{0, 1, ..., 255\}$, for which $(\mu_1, \mu_2) = (0, 0)$ holds, add the corresponding S-box bit value a_x (one bit of $(a_{255}, ..., a_0)$) to the same 64-bit LUT. There are 64 entries out of 256 for which $(\mu_1, \mu_2) = (0, 0)$ holds. Analogously, this is repeated for the multiplexer configurations $(\mu_1, \mu_2) \in \{(0, 1), (1, 0), (1, 1)\}$. Again, the contents of LUT_i can vary due to one out of $6! = 720$ possible input permutations. Also, there are $\binom{8}{2}$ possibilities to pick two multiplexer bits (μ_1, μ_2) from (x_7, x_6,..., x_0). So there are $\binom{8}{2} \cdot 720 \cdot 4 = 80640$ patterns for one AES S-box column. To be able to search for all 8 AES S-box output columns, one needs to generate $8 \cdot 80640 = 645120$ search patterns in total. Algorithm 6 provides the necessary steps for detecting all AES S-boxes from the bitstream. Depending on the synthesizer's choice for generating and distributing the S-box patterns over the FPGA grid (which cannot be predicted beforehand), the case may occur that some S-box search patterns may overlap, i.e., the two generated patterns of the configuration $(\mu_1 = 0, \mu_2 = 0)$ for S-box column 1 and $(\mu_1 = 0, \mu_2 = 1)$ for S-box column 2 may equal. If the search is limited to analyzing LUTs only, it can be concluded that a part of an AES S-box column is detected, but it cannot be clearly assigned to which S-box column it belongs to. Note that this is a necessary information for an S-box substitution attack, where an attacker intends to weaken the AES algorithm. If the goal is key extraction, then this information is not required as explained later.

However, in our approach one S-box column is successfully detected if all 4 LUTs of one slice indicate the implementation of the same S-box column as well as the same multiplexer configuration (μ_1, μ_2). Hence, from an attacker's point of view, it is an advantage that all four LUTs are placed within one slice allowing to conclude the exact input permutation of any single LUT. Algorithm 6 reveals any slice implementing an AES S-box column and particularly to which S-box output column it belongs to. Figure 4.7.a and Figure 4.7.b show two results for two different AES cores that we refer to as D_0 (word-based) and D_{10} (round-based). More precisely, it shows the distribution of all detected AES S-box columns over the entire FPGA grid. As can be seen, we do not only obtain the exact coordinates, but we also learn how many S-box instances are implemented by an AES core. Given this information, we can conclude whether the AES implementation is a byte-based, word-based, round-based, or unrolled design which is valuable information, e.g., if an attacker wants to reverse-engineer a third-party hardware configuration. Furthermore, one main advantage is that we do not need any further knowledge



a) Detection of 4 S-box instances (32x 6-input,1-output S-box columns) for the word-based AES design D_0

b) Detection of 20 S-box instances (160x 6-input,1-output S-box columns) for the round-based AES design D_{10}

Figure 4.7: FPGA grid view of a Xilinx Spartan 6 XC6SLX16, where each cell represents one slice. It shows the results of the proposed 8-input, 8-output S-box detection algorithm for AES hardware configurations. Any cell containing a number shows that the i^{th} (with $i \in \{0,1,\ldots, 7\}$) AES S-box output column is implemented by the corresponding slice, i.e., it is successfully detected. White cells represent unused slices, whereas gray cells denote the usage of arbitrary combinatorial logic within one slice, i.e., at least one out of 4 LUTs is configured

Algorithm 6 Detection of AES S-boxes being distributed over 32 6-input,1-output LUTs

Input: 3rd-party bitstream *bs*, Bitstream encoding *lut_encoding*
Output: Flagged LUTs on the FPGA grid that implement AES S-box columns

---

$S^j(x)$ denotes the j'th output bit (column) of the AES S-box $S(x)$ with $j \in \{0, 1, \ldots, 7\}$
$\pi_i(\cdot)$ denotes the i'th permutation out of 720
get_lut_content($\cdot$) extracts all 64-bit contents of all LUTs of a bitstream using *lut_encoding*

1: //Generate AES search patterns
2: **for** *column* = 0 to 7 **do**
3: **for** $(\kappa_1, \kappa_2) \in \{(\kappa_1, \kappa_2) \in \{0, 1, \ldots, 7\} \times \{0, 1, \ldots, 7\} : \kappa_1 \neq \kappa_2\}$ **do**
4: Set cnt_i to 0 for each $i \in \{0, 1, 2, 3\}$
5: **for** $x = 0$ to 255 **do**
6: $(\mu_1, \mu_2) = (S^{\kappa_1}(x), S^{\kappa_2}(x))$
7: **switch** $((\mu_1, \mu_2))$
8: case (0,0): $\text{AES\_pattern}_0[(\kappa_1, \kappa_2)][column][cnt_0{+}{+}] = S^{column}(x)$
9: case (0,1): $\text{AES\_pattern}_1[(\kappa_1, \kappa_2)][column][cnt_1{+}{+}] = S^{column}(x)$
10: case (1,0): $\text{AES\_pattern}_2[(\kappa_1, \kappa_2)][column][cnt_2{+}{+}] = S^{column}(x)$
11: case (1,1): $\text{AES\_pattern}_3[(\kappa_1, \kappa_2)][column][cnt_3{+}{+}] = S^{column}(x)$
12: **end switch**
13: //Dump all LUTs from bitstream
14: lut_content[NUM_OF_LUTS]← get_lut_content(*bs*)
15:
16: //Flag potential AES S-box column parts
17: **for** *lut_index* = 0 to NUM_OF_LUTS - 1 **do**
18: **for** *column* = 0 to 7 **do**
19: **for** $(\kappa_1, \kappa_2) \in \{(\kappa_1, \kappa_2) \in \{0, 1, \ldots, 7\} \times \{0, 1, \ldots, 7\} : \kappa_1 \neq \kappa_2\}$ **do**
20: **for** *p_index* = 0 to 719 **do**
21: pattern_of_interest = lut_content[*lut_index*]
22: $p_0 = \pi_{p\_index}(\text{AES\_pattern}_0[(\kappa_1, \kappa_2)][column])$
23: $p_1 = \pi_{p\_index}(\text{AES\_pattern}_1[(\kappa_1, \kappa_2)][column])$
24: $p_2 = \pi_{p\_index}(\text{AES\_pattern}_2[(\kappa_1, \kappa_2)][column])$
25: $p_3 = \pi_{p\_index}(\text{AES\_pattern}_3[(\kappa_1, \kappa_2)][column])$
26: **for** i = 0 to 3 **do**
27: **if** pattern_of_interest == p_i **then**
28: flag_LUT_as_AES_sbox_column(*lut_index, column,* (κ_1, κ_2), *p_index*)
29: //Remove false positives
30: **for** *slice_index* = 0 to NUM_OF_LUTS/4 - 1 **do**
31: Get any flagged information for all 4 LUTs that correspond to the slice *slice_index*
32: Only keep the flagged information for which the value *column* and the (κ_1, κ_2) configuration are identical for all 4 LUTs

about the routing to be able to detect and manipulate S-boxes. Thus, the reverse-engineering

effort is small. Listing 4.3 shows some partial output of Alg. 6 indicating the information, which can be obtained from a third-party bitstream.

```
Pattern 1 of AES S-box output column 4 detected in SLICE_X12Y24 and LUT_A,
Detected permutation indices (4,3,1,5,0,2)
Detected MUX input wiring (μ1,μ2) = (x0,x1) or (μ1,μ2) = (x1,x0)
Found input permutation (x6,x5,x3,x7,x2,x4)

Pattern 2 of AES S-box output column 4 detected in SLICE_X12Y24 and LUT_B,
Detected permutation indices (0,4,3,2,5,1)
Detected MUX input wiring (μ1,μ2) = (x0,x1) or (μ1,μ2) = (x1,x0)
Found input permutation (x2,x6,x5,x4,x7,x3)

Pattern 3 of AES S-box output column 4 detected in SLICE_X12Y24 and LUT_C,
Detected permutation indices (1,3,4,0,2,5)
Detected MUX input wiring (μ1,μ2) = (x0,x1) or (μ1,μ2) = (x1,x0)
Found input permutation (x3,x5,x6,x2,x4,x7)

Pattern 4 of AES S-box output column 4 detected in SLICE_X12Y24 and LUT_D,
Detected Permutation π = (1,3,4,0,2,5)
Detected MUX input wiring (μ1,μ2) = (x0,x1) or (μ1,μ2) = (x1,x0)
Found input permutation (x3,x5,x6,x2,x4,x7)
```

Listing 4.3: Output of Algorithm 6 for AES design D_0. As can be seen, each detected LUT has the same multiplexer configuration (μ_1, μ_2) allowing to filter false positive patterns

4.4.5 Results of AES S-box Detection

For our evaluation, we examined 16 AES encryption designs $D_0, D_1, ..., D_{15}$, four of which were developed by our group. The other 12 cores have been taken from publicly-available websites, e.g., NSA homepage, OpenCores, GitHub, SASEBO project[3]. Each core is provided by an interface to set the key k and the plaintext p, and to fetch the ciphertext c. We developed an FPGA design template to play the role of an RS-232 interface between a PC and the target AES core, where the key is kept constant by the template. During the integration of any of the target AES cores, we did not modify the AES circuitry, but adopted the template to fit to the sequences required by the underlying AES interface. As an example, some of the AES cores require first to perform a key schedule, while the others process the actual encryption and key schedule in parallel.

Most of the designs (D_0, $D_2 - D_4$, D_6, $D_9 - D_{15}$) operate on a round-based architecture. D_1 is based on a word-based architecture, where 32-bit blocks are processed at each clock cycle. D_7 and D_8 follow a serialized architecture, i.e., with only one instance of the S-Box, where at each clock cycle one byte is processed. Finally, D_5 is an unrolled pipeline design with 10 separately instantiated round functions, which can give one ciphertext per clock cycle if the pipeline is full. Hence, we cover a variety of implementation styles. To be able to analyze all AES cores with Algorithm 6, we generated the corresponding bitstreams to evaluate whether they can be compromised in practice for which we provide further details in Section 4.4.6. The S-box detection results are given in Table 4.4.

[3] D_0 [Mic10], D_1 [Jer11], D_2 [Fek14], D_4 [Hem14], D_5 [Tar13], D_6 [Mot13], D_{10} [NSA99], D_{11}-D_{15} [Aka07]

	Usage of precomputed S-boxes										Usage of S-box Circuit				Countermeasure of Section 4.5	
AES Design	D_0	D_1	D_2	D_4	D_5	D_7	D_8	D_{10}	D_{11}	D_{15}	D_6	D_{12}	D_{13}	D_{14}	D_3	D_9
S-box columns (LUTs)	32	0	32	160	1024	8	8	160	0	32	0	0	0	0	0	32
S-box inst. in LUTs	4	0	4	20	128	1	1	20	0	4	0	0	0	0	0	4
S-box inst. (BRAM)	16	5	16	0	32	0	0	0	20	16	0	0	0	0	0	0
Σ S-box instances	20	5	20	20	160	1	1	20	20	20	0	0	0	0	0	4
Detection rate	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	0%	0%	0%	0%	0%	25%

Table 4.4: Overview of evaluated AES implementations

As can be seen, an adversary would be able to successfully attack all third-party AES cores which use precomputed S-boxes. From these 10 AES cores, 4 (D_4, D_7, D_8, and D_{10}) can be compromised only by detecting LUT-based S-boxes. Regarding the other AES cores, one needs to additionally analyze the BRAM content. Only two AES cores (D_1 and D_{11}) implement all AES S-boxes in the BRAM, while the remaining ones (D_0, D_2, D_5, and D_{15}) make use of both, distributed LUTs and BRAM instances. Note that there are various possibilities to implement an AES S-box in an FPGA design and thus the presented detection method does not always lead to a successful finding. Hence, a limitation of Algorithm 6 is that it cannot detect AES S-boxes, which are implemented as computational on-the-fly circuit. This is the case for the AES cores D_6 and $D_{12} - D_{14}$. Even if an attacker would be able to extract the entire routing information from a bitstream (which is out of the scope of this thesis and attacker model), it remains unclear whether and how the relevant S-box regions can be reliably detected from the hardware configuration: neither can an attacker search for specific patterns nor is he able to predict how the S-box circuit is built and distributed over the FPGA grid by the synthesizer.

Nevertheless, our results indicate that the majority of AES implementations make use of precomputed AES S-boxes in order to achieve a higher performance. Hence, it is likely that an attacker can compromise an AES core in practice. We practically verified the correctness of bitstream manipulation, as we were indeed able to extract the secret key from the manipulated hardware configuration.

After having presented all necessary detection steps and providing the corresponding evaluation for various AES S-boxes, we describe the AES manipulations easing a practical attack.

4.4.6 Manipulating AES S-boxes

In this section, we present further analysis regarding malicious AES manipulations being conducted on FPGA bitstreams. Note that the AES algorithm is briefly introduced in Section 2.3. As described in the previous sections, we are able to detect S-box instances by analyzing the corresponding bitstream of an FPGA. Similar to DES, an attacker may be able to silently weaken the algorithm such that the encryption and decryption are still compatible, but the corresponding ciphertexts are cryptographically weak, cf. Section 4.4.6. Furthermore, a key leakage approach is introduced in Section 4.4.6.

Implanting an AES Trojan by Replacing Distributed Precomputed S-boxes

When setting all AES S-box instances to the identity mapping as described below, the encryption and decryption function turn into a linear bijection. The corresponding altered AES-128 (that we refer to as $\widetilde{c} = \widetilde{\mathrm{AES}}_k(p)$) does weakly encrypt plaintexts and decrypt the weak and modified ciphertexts, hence they are vulnerable to crypt-analytical attacks.

$$\text{Substitute } S(x) \text{ by } S^{id}(x) = x, \quad \forall x \in \{0, 1, \ldots, 255\}$$

Given one known plaintext-ciphertext pair $(p, \widetilde{c})$, an attacker is able to decrypt all other faulty ciphertext blocks, because $\widetilde{\text{AES}}_k(\cdot)$ can be described as:

$$\begin{aligned}\widetilde{c} = \widetilde{\text{AES}}_k(p) &= SR(\ldots MC(SR(p \oplus rk_0) \oplus rk_1)\ldots) \oplus rk_{10} \\ &= SR(\ldots MC(SR(p)\ldots)) \oplus \underbrace{SR(\ldots MC(SR(rk_0) \oplus rk_1)\ldots) \oplus rk_{10}}_{:=\widetilde{K}} \\ &= SR(\ldots MC(SR(p)\ldots)) \oplus \widetilde{K}\end{aligned}$$

The equation above holds, because the $MC(\cdot)$ and the $SR(\cdot)$ functions are linear as described below.

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

It is important to understand that the XOR sum of all processed subkeys is constant and can be expressed by one variable $\widetilde{K}$. In addition, the number of $MC(\cdot)$ and $SR(\cdot)$ operations depends on the utilized AES key size, i.e., 128, 192, or 256 bits.

Once an attacker can obtain one plaintext/ciphertext pair $(p, \widetilde{c})$ of the manipulated AES, he is able to compute the secret $\widetilde{K}$. For this purpose, he simply reconstructs $SR(\ldots MC(SR(p)\ldots))$, and then computes the following:

$$\widetilde{K} = \widetilde{c} \oplus SR(\ldots MC(SR(p)\ldots))$$

With the knowledge of $\widetilde{K}$, an attacker can recover any plaintext x from any faulty ciphertext $\widetilde{y}$. To do so, the adversary has to XOR the value $\widetilde{y}$ with the previously recovered secret $\widetilde{K}$. Afterwards, the $MC(\cdot)$ and $SR(\cdot)$ transformations have to be inverted. Algorithm 7 provides the necessary step to decrypt a weak ciphertext $\widetilde{y}$.

Algorithm 7 Decryption of ciphertexts that were encrypted with $S^{id}(\cdot)$

Input: Ciphertext $\widetilde{y}$ from a modified AES with $S^{id}(\cdot)$, one previously obtained $(p, \widetilde{c})$ pair
Output: Plaintext x corresponding to $\widetilde{y}$

1: $\widetilde{K} \leftarrow \widetilde{c} \oplus SR(\ldots MC(SR(p)\ldots))$ // Calculate $\widetilde{K}$
2: $\widetilde{y} \leftarrow \widetilde{y} \oplus \widetilde{K}$ // Cancel secret $\widetilde{K}$
3: $x \leftarrow SR^{-1}(MC^{-1}(SR^{-1}(\ldots MC^{-1}(SR^{-1}(\widetilde{y})\ldots))))$ // Apply inverted AES operations

Note that this attack works regardless of the key schedule (AES-128, AES-192, and AES-256) once an attacker manages to alter all SubBytes S-boxes, because the secret $\widetilde{K}$ can be canceled in any case. It also does not matter whether any S-box of the key schedule is altered or not. We practically verified the feasibility of this attack by applying the S-box manipulation directly to the bitstream file. The manipulated AES core yielded a weakened ciphertext, which we successfully decrypted with the help of $\widetilde{K}$.

Key Recovery of AES by Clearing Distributed Precomputed S-box Tables

Analogous to the DES modification of Section 4.4.3, all AES S-boxes in the FPGA bitstream can be altered to always output a zero – regardless of the input value. This kind of modification is also presented by Kerins *et al.* [KK06], which we tested for Xilinx FPGAs. The modification is described in the following equation:

$$\text{Substitute } S(x) \text{ by } S^{zero}(x) = 0, \quad \forall x \in \{0, 1, \ldots, 255\}$$

Obviously, after having altered all S-box instances in this manner, the AES algorithm becomes unusable. That is, because any information regarding the plaintext is lost, right after the first SubBytes step has been processed by the modified AES instance, hence the cipher is not bijective anymore. However, such an alteration is still useful for an adversary since the output of the AES is now the last subkey rk_{10} from which the main key rk_0 can be computed and verified by one plaintext-ciphertext pair (p, c). This is done by testing whether the computation $AES^{-1}_{key\ hypothesis}(c)$ outputs the correct plaintext p.

This is for example useful in all scenarios, where the underlying non-accessible main key rk_0 is hard-coded in the FPGA design. Another imaginable scenario is a main key rk_0 which is securely transfered to the FPGA after power-up, e.g., by a Hardware Security Module (HSM) whose data bus cannot be eavesdropped. Then, an adversary can obtain the key by querying the modified AES instance with an arbitrary plaintext.

Since the S-boxes of the key schedule are usually not distinguishable from the SubBytes S-boxes, the previously discussed modification should be conducted on all S-boxes, including those from the key schedule. In the following, we have a look at the potential manipulation cases.

Manipulation Possibilities for AES-128

In the case of AES-128, the main key rk_0 can be fully recovered if an attacker accomplishes specific manipulations to the AES S-boxes which are encoded within the bitstreams. In order to better understand Algorithm 8, the AES-128 key schedule is depicted in Figure 2.10 in Section 2.3. Furthermore, if the attacker is able to collect one faulty ciphertext $\widetilde{c}$ by querying the manipulated AES core with one arbitrary plaintext p, the steps described in Algorithm 8 will lead to key recovery. In order to successfully mount a key recovery attack, two out of three thinkable S-box manipulations are helpful to ensure a successful key recovery:

Manipulation 1 - Only key schedule S-boxes are replaced by $S^{zero}(x)$: After the key schedule finished its computations, all valuable information is lost regarding the AES key. Therefore, even if this manipulation is achieved, the adversary cannot recover rk_0 from the faulty ciphertext $\widetilde{c}$.

Manipulation 2 - Only SubBytes S-boxes are replaced by $S^{zero}(x)$: This may be the case if the function $g(\cdot)$ does not utilize precomputed AES S-boxes for some reason. In this case, Algorithm 8 directly reveals the main key rk_0, once the attacker records the faulty ciphertext $\widetilde{c}$.

Manipulation 3 - SubBytes and key schedule S-boxes are replaced by $S^{zero}(x)$: This is the most likely case. In this case, the g-function only returns the corresponding round constant

Algorithm 8 Reconstruction of the full main key of AES-128

Input: Faulty ciphertext $\widetilde{c}$ from manipulation AES with $S^{zero}(x)$
Output: Fully recovered 128-bit AES main key rk_0.

```
//Process faulty ciphertext c̃ (= last round key)
1: for i = 0 to 3 do
2:     w[43 − i] = ỹ[3 − i]
//Invert the 128-bit key schedule
3: for i = 39 to 0 do
4:     if i % 4 == 0 then
5:         w[i] = w[i + 4] ⊕ g(w[i + 3])
6:     else
7:         w[i] = w[i + 4] ⊕ w[i + 3]
```

RC[i], also padded with three zero bytes. Code line 5 of Algorithm 8 should then be changed to

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

in order to reveal the main key rk_0. Therefore, both variants should be tested when trying to recover the key.

Manipulation Possibilities for AES-192 and AES-256

Compared to AES-128, the algorithm AES-192 and AES-256 only leak the parts of the main key rk_0 under special conditions. We discuss only the two most interesting manipulation cases. For a better understanding, the graphical representation of the AES-192 key schedule function is shown in Figure 4.8.

Manipulation 1 - Only all SubBytes S-boxes are replaced by $S^{zero}(x)$: Not any single byte of the main key rk_0 can be recovered, if this manipulation is accomplished by an attacker. This fact also holds for AES-256. Figure 4.8 reveals why this is the case by showing the computable words (white background) and non-computable words (gray background) when trying to invert the AES operations of $\widetilde{c}$. If the round keys are calculated utilizing normal AES S-boxes, then for instance, $w[42]$ cannot be calculated from the faulty ciphertext $\widetilde{c}$. This is because the output of the last g-processing is unknown to an attacker. Therefore, in the set of $w[36] - w[41]$ only the words $w[38]$ and $w[39]$ are computable. The other intermediate values belonging to the same set cannot be computed, because $w[42], w[46]$, and $w[47]$ are unknown. The last possible word that can be computed is $w[33]$, but it does not reveal any information of the main key rk_0.

Manipulation 2 - All SubBytes and key schedule S-boxes are replaced by $S^{zero}(x)$: This is the preferable case, since the first 128 bits of the AES-192 (AES-256 respectively) main key rk_0 can be derived. The g-function returns the round constant value RC[i], if all S-box outputs yield a zero byte (for every input), cf. function g of Figure 2.10. Hence, $w[42]$ can be derived

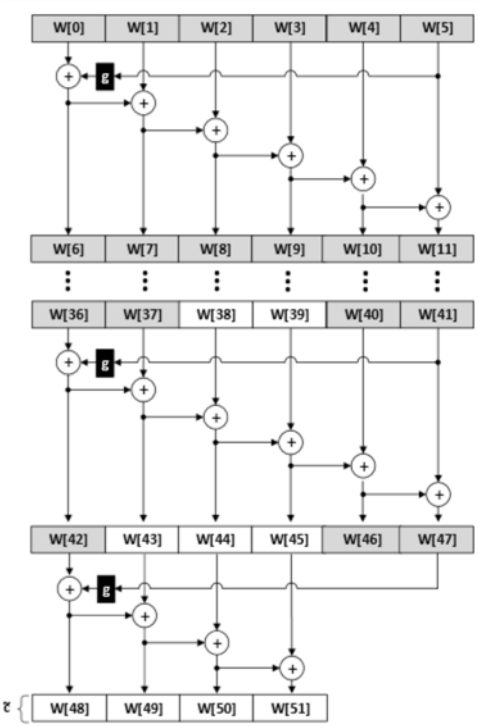


Figure 4.8: Key schedule of AES-192

and all the first *left* 4 words of each subkey of the key schedule step are computable. Even if the *right* part is not known, the first 4 words $w[0] - w[3]$ can be computed, cf. Algorithm 9.

Algorithm 9 Partial key reconstruction of AES-192/256

Input: Ciphertext $\widetilde{y}$ from modified AES with $S^{zero}(x)$
Output: First 128 bit of 192/256 main key

$N_w \leftarrow 51$ for AES-192 ($\leftarrow 59$ for AES-256)
$N_k \leftarrow 6$ for AES-192 ($\leftarrow 8$ for AES-256)

//Load the ciphertext (= last round key)
1: **for** $i = 0$ to 3 **do**
2: $\quad w[N_w - i] = \widetilde{y}[3 - i]$
//Invert the KeySchedule
3: **for** $i = N_w$ to 0 **do**
4: $\quad$ **if** $i \mod N_k \geq 4$ **then**
5: $\quad\quad$ continue
6: $\quad$ **if** $i \mod N_k == 0$ **then**
7: $\quad\quad w[i] = w[i + N_k] \oplus \mathrm{RC}[i]$
8: $\quad$ **else**
9: $\quad\quad w[i] = w[i + N_k] \oplus w[i + N_k - 1]$

To be more precise, the first 128 bits of each subkey can be recovered. Therefore, in this case, an attacker can obtain the first 128 bits of the main key of AES-192 and AES-256. The other words (64 bits) of AES-192 and AES-256 (128 bits) cannot be computed. Having discussed the attack vectors, the next section covers several countermeasures and proposes one concrete one to mitigate the introduced S-box substitution attack.

4.5 Mitigating S-box Substitution Attacks

In this section, we do not only discuss potential countermeasures that may be deployed in order to raise the bar for an adversary, but we also propose a concrete one and evaluate its costs in terms of hardware area. Remember that in our attack model the adversary can only analyze and modify the LUT content and the BRAM content of an FPGA bitstream, i.e., the parts of the hardware in which cryptographic S-boxes are implemented. The countermeasures are based on obfuscation. In general, every obfuscation strategy helps to defeat this kind of modification attack, but if a strategy is known to an attacker, it may be circumvented again. In the following, several ideas and their drawbacks are presented.

4.5.1 Built-In Self-Test

Built-In Self-Tests are a well known concept to test different kinds of faults and circuits. A simple integrated self-test can be used to defeat the attacks presented in this work. For example, one can check if the algorithm outputs the correct ciphertext for a fixed key and plaintext. Such a self-test can be circumvented, however, by a more powerful adversary with the following approaches:

- The integrity value has to be stored in the FPGA bitstream. Thus, an adversary may be able to identify and modify this value.
- The adversary could disable the self-test or modify it in such a way that the test routine marks the test as “passed”.

4.5.2 Decomposition of Larger Circuits into Smaller Ones

Another approach targets decomposed LUTs. They are detectable because of their characteristically non-linear patterns. Security-critical Boolean equations, generating the LUT contents for the S-boxes, should be difficult to distinguish from other linear LUT content patterns to defeat detection and consequently modification of these parts. One possible way to achieve this is to further decompose the LUTs along the Disjunctive Normal Form. For example, in a 6-input,1-output architecture, a 64-bit LUT content may be split-up into 8 LUTs. The output of each LUT can be OR-ed together to compute the original LUT content.
To give an example, assume a Boolean function $f(a, b, c) = ab \vee bc \vee abc$. Suppose that this Boolean function is realized in one LUT. Following the idea described above, this LUT is separated into three LUTs:

$$f_1(a, b, c) = ab$$
$$f_2(a, b, c) = bc$$
$$f_3(a, b, c) = abc$$

The result of every function f_i is then OR-ed. Thus, it should be more difficult to identify f_1, f_2, and f_3, if this function splitting scheme is unknown to an attacker. Nonetheless, the decomposition to multiple LUTs has also a drawback. An adversary can search for the hardware part where the OR-ing of all f_i functions is processed. In a test implementation, we observed that one LUT is used to implement the corresponding OR function, hence an adversary could modify this LUT. For a targeted alteration to an identity mapping, e.g., in the case of AES, the adversary would need to trace back the path to the f_i functions with the help of routing information.

Even when the set of candidates is too large for an adversary, it is likely possible to obtain the correct set of LUTs belonging to the S-box. The attacker's effort depends on the decomposition method and the corresponding parameters. It might be more challenging for an attacker, if the decomposition of the LUT content is chosen randomly for each S-box column.

4.5.3 Proposal for Partial Self-configuration Countermeasure Scheme

To mitigate the presented S-box substitution attack with respect to Xilinx FPGAs, it is particularly important to protect the SubBytes layer against hardware configuration manipulations. The following countermeasure proposal can impede static attacks, which attempt to identify and alter precomputed AES S-boxes in a deterministic manner. Note that no potential round counter, state machine, or data bus manipulations, etc. are covered by this countermeasure.

A very powerful adversary could, of course, manage to inject more sophisticated backdoors, again leading to the loss of confidentiality. However, this task requires considerably more efforts with respect to design reverse-engineering and hardware configuration alteration.

The strategy behind our proposed countermeasure is to significantly increase the reverse-engineering efforts for an attacker (using static analysis) through obfuscating the known S-box patterns. Our *raise the bar* countermeasure hides the precomputed S-box values in a cryptographic manner, thus an attacker is not able to easily detect and modify any S-box value in a predictable fashion.

Key Idea of the S-box Protection: The high-level idea of our approach is to instantiate an initial hardware configuration *i*) with a slightly modified AES core that we refer to $\widetilde{AES}$ and *ii*) a fixed initial key k_1. This variant uses an arbitrary 8-input,8-output S-box instead of the original AES S-box. The same slightly modified AES core is implemented in software, which is used to generate ciphertexts of the actual AES S-boxes using the same key k_1. The resulting ciphertexts are embedded as part of the initial hardware configuration. On boot-up of the initial hardware configuration, the S-box is recovered with the help of the modified AES core using k_1. In the last step, the initial hardware configuration itself updates its random S-box to the correct AES S-box.

To implement the above mentioned workflow, in the first step a random bijective S-box is determined using a software application before the hardware configuration is designed. Once the random S-box is determined, the same software application *decrypts* the correct AES S-box values by using the inverse $\widetilde{AES}$ operation, i.e., $\widetilde{c} = \widetilde{AES}_{k_1}^{-1}(\text{AES S-box})$. In this context, we also refer to $\widetilde{c}$ as ciphertexts. Having obtained $\widetilde{c}$, it is integrated as part of the initial hardware configuration, i.e., the ciphertexts $\widetilde{c}$ are distributed over the available LUT memory. Once $\widetilde{AES}$ is configured after powering the FPGA, the goal is to dynamically replace the random S-box

with the original AES S-box, so that a true AES core can perform the actual encryption of data (using another fixed key k_2). By following this approach, the correct S-box values are only dynamically available and cryptographically hidden for an attacker who only statically analyzes the design and searches for the correct AES S-box in the initial hardware configuration. Note that the ciphertexts $\widetilde{c}$ may be more easily identified if they are stored in the BRAM and not in the FPGA's distributed LUT memory. After FPGA power-up, the hardware configuration starts to recover the correct AES S-box. This is carried out by the initial hardware configuration itself by *encrypting*[4] the ciphertext $\widetilde{c}$, i.e., by computing $\widetilde{AES}_{k_1}(\widetilde{c})$ and hence inverting the conducted steps of the software application. Having obtained the AES S-box in the explained manner, the initial hardware configuration starts to replace the random and bijective S-box by using the dynamic reconfiguration feature of Xilinx FPGAs.

To implement our proposed countermeasure, we make use of dynamic look-up tables that can be changed during runtime of the FPGA, i.e., only by the FPGA configuration itself. Additionally, we implemented an AES design using BRAM to support smaller FPGA devices.

Dynamic LUT reconfiguration is used as a building block for our countermeasure. This feature is supported by several Xilinx FPGA families, in particular we implemented the protection scheme utilizing a Xilinx Spartan 6 device. Since we only provided a rough description of the countermeasure, in the following we present its work-flow in more detail. Our scheme makes use of so-called CFGLUT5 primitives, hence they are first briefly introduced.

Building Block, Xilinx CFGLUT5

Xilinx provides a dynamic LUT primitive called *CFGLUT5* (5-input, 1-output LUT), cf. Figure 4.9 and Figure 4.10.

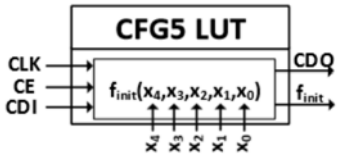


Figure 4.9: Initial LUT with Boolean function f_{init}

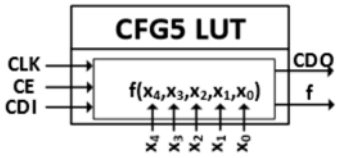


Figure 4.10: Reconfigured LUT with Boolean function f

Fig. 4.9 illustrates the initial state of a CFGLUT5 (directly after FPGA power-up and initial configuration), while Fig. 4.10 depicts the reconfigured state. The f_{init} Boolean function output values are encoded in the FPGA bitstream, whereas the Boolean function output values of f are reconfigured during runtime. Note that only five instead of six input pins are available, since one pin (CE) decides when to update a LUT content.

Reconfiguration of CFGLUT5 Elements

Three input reconfiguration pins are available for this hardware element. The *CDI* pin handles

[4]One main advantage of decrypting the true AES S-boxes for the initial hardware configuration (instead of encrypting as one would expect) is that the hardware configuration only needs to implement the encryption function to be able to recover the true AES S-boxes during power-up of the FPGA. This is an advantage for specific modes of operation such as the counter mode where only the underlying encryption function of the AES core needs to be implemented. Hence, one can avoid the implementation of the decryption function in the hardware configuration to reduce its costs in terms of hardware area.

the new LUT content data or functionality. The *CE* pin needs to be pulled high to activate or keep the reconfiguration process running. The clock is used to configure the current data of the *CDI* pin. The 5-input,1-output Boolean function f is coded by $2^5 = 32$ output values that are being shifted bit-wise for updating the functionality. Two output pins are provided by the CFGLUT5: through the *CDO* pin, the old configuration data can be read out and the other output pin provides the evaluation of the Boolean function f.

Memory Organization

The memory organization of our countermeasure is explained with the example of the AES S-box $S(x)$ even though it initially configures the random S-box that is denoted by $\widetilde{S}(x)$. Regarding the AES block cipher, an 8-input,8-output S-box instance can be implemented using 2048 bits ($= 2^8$ inputs $\cdot$ 8 bits) of memory. The general shape of an 8-input, 8-output AES S-box is depicted in Table 4.5. Note that $a_i, b_i, \ldots, h_i$ with $i \in \{0, 1, \ldots, 255\}$ store one S-box bit value each. The AES S-box call $S(x)$ with $x = (x_7, x_6, \ldots, x_0)$ can also be expressed as 8-bit value

Input values $x = (x_7, x_6, \ldots, x_0)$								**S-box output** $S(x) = S^1(x)\|\|S^2(x)\|\|\ldots\|\|S^8(x)$							
x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	$S^1(x)$	$S^2(x)$	$S^3(x)$	$S^4(x)$	$S^5(x)$	$S^6(x)$	$S^7(x)$	$S^8(x)$
0	0	0	0	0	0	0	0	a_0	b_0	c_0	d_0	e_0	f_0	g_0	h_0
0	0	0	0	0	0	0	1	a_1	b_1	c_1	d_1	e_1	f_1	g_1	h_1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1	1	1	1	1	1	1	1	a_{255}	b_{255}	c_{255}	d_{255}	e_{255}	f_{255}	g_{255}	h_{255}

Table 4.5: General shape of an 8-input,8-output AES S-box.

such that $S(x) = S^1(x)||S^2(x)||\ldots||S^8(x) = a_x||b_x||\ldots||h_x$, where $||$ denotes a concatenation. We further refer to the i^{th} S-box column as $S^i(x)$ with $i \in \{1, 2, \ldots, 8\}$. For example, the second S-box column $S^2(x)$ is described by the bit pattern $b_0||b_1||\ldots||b_{255}$, so that the correct output bit value can be retrieved for a given input x. Since our goal is to design a countermeasure relying on distributing the S-box output values $(a_i, b_i, \ldots, h_i)$ over various CFGLUT5 elements (32 bits of capacity), for each S-box column $S^i(x)$ we need to divide its 256-bit pattern into 32-bit subpatterns. To be more precise, one AES 8-input, 1-output (256 bits of memory) S-box column is divided into 8 individual 5-input, 1-output (32 bits of memory) subfunctions. We further introduce S^i_j with $j \in \{1, 2, \ldots, 8\}$ that describes the j^{th} 32-bit pattern of the i^{th} S-box output column. To give an example, S^2_j describes the 32 bits $b_{32\cdot(j-1)}||b_{32\cdot(j-1)+1}||\ldots||b_{32\cdot j-1}$, while S^8_j targets the $h_{32\cdot(j-1)+1}||h_{32\cdot(j-1)}||\ldots||h_{32\cdot j-1}$ bits. In total, $64 = \frac{2048}{32}$ CFGLUT5 primitives need to be instantiated to store one 8-input,8-output S-box truth table. We propose to connect each of the 64 CFGLUT5 elements together in a consecutive chain and to multiplex each of the 8 CFGLUT5 (implementing one S-box column) instances with three selection bits (x_7, x_6, x_5) allowing to select between $2^3 = 8$ these CFGLUT5 primitives, cf. Fig. 4.11. Furthermore, the remaining five bits $(x_4, x_3, \ldots, x_0)$ picks the appropriate output bit of one $S^i_j(x)$ CFGLUT5 element (not shown in Fig. 4.11). Hence, the shown hardware structure of Fig. 4.11 implements $S(x)$. Note that the scheme makes use of a 1-bit reconfiguration data bus, and therefore, the entire 8-input,8-output S-box can be replaced by using 2048 clock cycles.

Having introduced the basic hardware layout, in the following, each step from system design to FPGA power-up is described to present the concept in its entirety.

Phase I – System Design

During the system design phase, the following steps have to be carried out: first, a random 8-input, 8-output S-box $\widetilde{S}(x)$ and its inverse $\widetilde{S}^{-1}(x)$ have to be generated by a software application. Then, a so-called *reconfiguration key* k_1 has to be chosen and the correct AES S-box values (256 bytes) have to be *decrypted* with $\widetilde{AES}_{k_1}^{-1}$ (for which the original AES S-box was replaced by the random and bijective S-box $\widetilde{S}(x)$). Since 128 bits need to be decrypted at the same time, we propose to perform the *decryption* of $16 = \frac{2048}{128}$ blocks as follows.

$$\widetilde{c_\kappa} := \widetilde{AES}_{k_1}^{-1}(\widetilde{p_\kappa}), \qquad \text{with } \kappa \in \{1, 2, \ldots, 16\}$$

A 128-bit input $\widetilde{p_\kappa}$, which exhibits one part of the original AES S-box $S(x)$, is merged with four 32-bit S-box patterns in the following manner:

$$\widetilde{p_\kappa} := \begin{cases} S_1^{\lceil \frac{\kappa}{2} \rceil} \,||\, S_2^{\lceil \frac{\kappa}{2} \rceil} \,||\, S_3^{\lceil \frac{\kappa}{2} \rceil} \,||\, S_4^{\lceil \frac{\kappa}{2} \rceil} & , \kappa \text{ odd} \\ S_5^{\lceil \frac{\kappa}{2} \rceil} \,||\, S_6^{\lceil \frac{\kappa}{2} \rceil} \,||\, S_7^{\lceil \frac{\kappa}{2} \rceil} \,||\, S_8^{\lceil \frac{\kappa}{2} \rceil} & , \kappa \text{ even} \end{cases}$$

Fig. 4.11 shows how the exact inner workings of the dynamic AES core and particularly that the *decrypted* AES S-box values ($\widetilde{c_k}$) are stored as part of the hardware configuration. Note that in Fig. 4.11 the beginning of a dotted arrow indicates the initial state, while the end of the dotted arrow shows the final reconfigured hardware configuration.

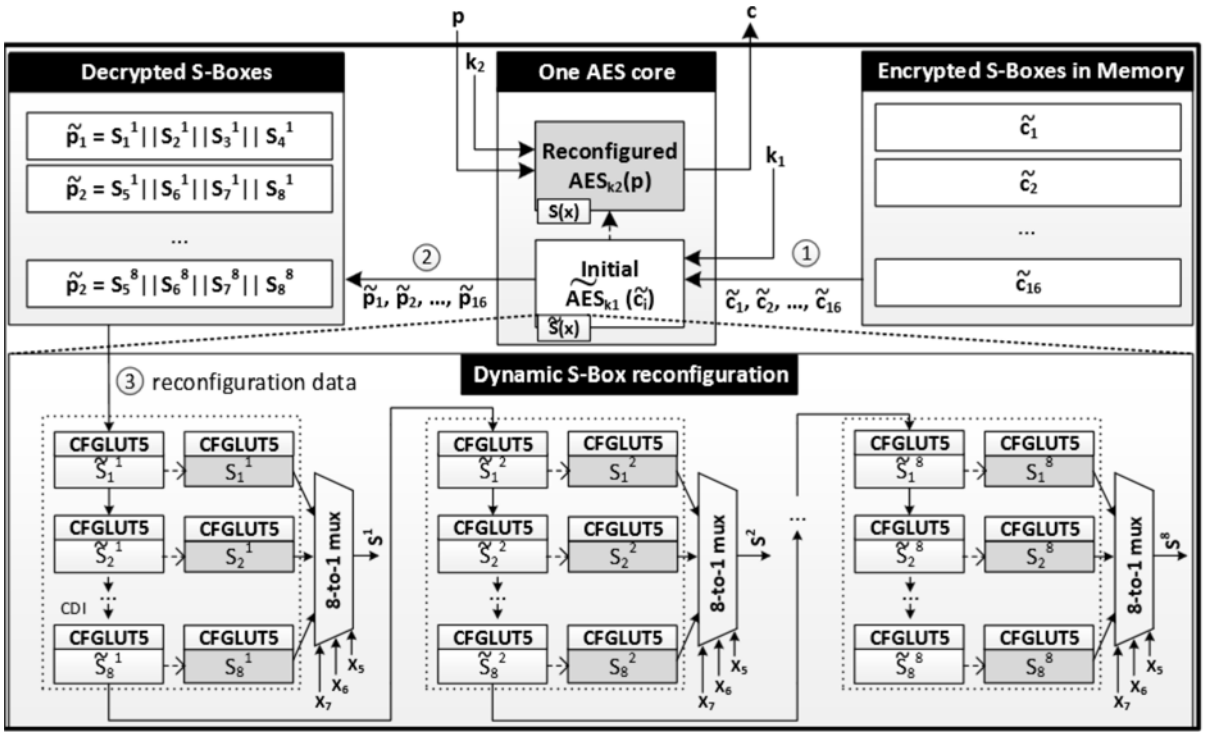


Figure 4.11: Dynamic reconfiguration of AES S-boxes using CFGLUT5 elements

Phase II – FPGA Power-Up

Directly after the FPGA power up, the 16 ciphertext blocks $\widetilde{c}_\kappa$ are *encrypted* by $\widetilde{p_\kappa} := \widetilde{AES}_{k_1}(\widetilde{c_\kappa})$ to recover the valid AES S-box values. Second, all $\widetilde{S}(x)$ instances are dynamically reconfigured to $S(x)$ (utilizing the values $\widetilde{p_\kappa}$) from the previous step. After this process, the $\widetilde{AES}$ core turns into the correct AES core. Finally, the reconfiguration key k_1 is set to the data encryption key k_2. This way, static manipulations are more challenging for an attacker, as the search patterns of a valid AES S-box are stored as encrypted data within the hardware configuration. When assuming that an attacker can locate the corresponding ciphertexts, meaningful and targeted manipulations are still limited.

We now examine the performance and utilized hardware area of our dynamically reconfigurable AES core.

Backwards Compatibility

Since not all Xilinx FPGA families support dynamic CFGLUT5 elements, we also implemented a similar approach that instantiates S-boxes using BRAM instead of dynamic LUTs. Due to similarity with the LUT-based scheme, the BRAM implementation is not further explained, although its resource usage is provided.

For performance and hardware area evaluation we used the same synthesis options (using Xilinx ISE 13.2) for each FPGA implementation and verified the functionality on a Spartan 6 FPGA board (SP601 evaluation kit with a XC6SLX16). We compared three different implementation strategies: (unprotected) static S-boxes in LUTs, (protected) S-boxes instantiated in CFGLUT5 memory, and (protected) S-boxes instantiated in BRAM. The results are depicted in Table 4.6. The amount of utilized slices, LUTs and flip-flops, etc. refer to a single instantiation of one AES. Each design contains a UART core due to the communication interface that is required for verification.

Table 4.6: Static LUT-based / dynamic DES and AES designs

Design	**#Slices**	**#FFs**	**#LUTs**			**#RAMBs**		f_{max}
			6-to-1	**5-to-1**	**Sh-Reg.**	**8**	**16**	(MHz)
AES								
Static LUT	1256	1619	3410	332	1	–	–	203
Dyn. CFGLUT5	1371	4065	1848	1376	1130	–	–	211
Dyn. BRAM	451	1444	1234	346	1	16	4	238

The performance of the AES implementations varies moderately. The larger amount of flip-flops (approximately a factor of 2.5) and shift-registers in the dynamic CFGLUT5 design, compared to the static LUT-based and BRAM designs, is due to storing the ciphertexts and the additional wires of all CFGLUT5 elements. The increased combinatorial hardware area overhead is mostly due to the smaller utilized 5-input, 1-output LUTs, where the amount of LUTs (storing precomputed S-boxes) at least doubles.

Brief Discussion of Dynamic S-Box Reconfiguration

Our proposed scheme is supposed to counterfeit an adversary who is capable of identifying and subsequently altering the original AES S-boxes in a given hardware configuration through LUT

or BRAM manipulation. This is currently one major threat in many real-world settings, because finding a given 8-input,8-output look-up table is a reasonable task. Considering the assumed capabilities of our adversary, it seems not possible to *directly and selectively* manipulate the original AES S-boxes to an arbitrary chosen Boolean function, e.g., to the identity function $f(x) = x$. Various practical hurdles need to be managed in terms of a meaningful S-box manipulation that leads to a weakened AES core. Some of them are as follows:

- An attacker cannot easily identify one or several parts of the ciphertexts (distributed as random unknown pattern in the LUTs and consequently in the bitstream). Even if he is able to locate them, any manipulation of the ciphertext will lead to the configuration of an unknown and probably non-bijective random S-box configuration being incompatible to the AES decryption function. Further, it remains unclear whether the conducted non-deterministic manipulation leads to a cryptographically weak AES version. Depending on the utilized block cipher, the adversary would be required to also indirectly manipulate the inverse S-box accordingly to ensure that the encryption and decryption are compatible.
- The initial randomly configured bijective S-boxes cannot be easily located (distributed as random unknown pattern in the LUTs and consequently in the bitstream). Even if an adversary is able to locate and replace them, it can be expected that random S-box values will be configured, again making the decryption and encryption incompatible.

To sum up, our countermeasure can prevent an adversary to inject cryptographically weak S-boxes, cf. Section 4.4.6. Note that our scheme is applicable to arbitrary S-box functions. After we proposed our countermeasure scheme, the prevention of key recovery seemed to be solved as well, but as a negative result it turned out that the AES key can still be recovered through following a different technique, which works under our specified attacker model. This new technique is described in Section 6. From this finding, we conclude that obfuscation alone does not provide the desired security level.

4.6 Conclusion

In the first part of this chapter, we demonstrated how to detect and maliciously manipulate cryptographic S-boxes within Xilinx bitstreams that encode cryptographic circuits such as DES, 3DES, and AES in order to weaken their strong cryptographic properties. We assumed a realistic practical setting, i.e., that an unknown third-party bitstream is in possession of an adversary who tries to exploit the encoded cryptographic circuit. One of the key insights is that in many practical situations an adversary does neither need to possess any high-level design information such as routing details nor does she need to reverse-engineer the entire bitstream file format encoding to be able to significantly weaken cryptographic primitives or to recover cryptographic keys. As long as a Boolean function is known to an adversary and is specific, the presented attacks (relying on LUT and BRAM manipulations) are practically feasible and pose a serious threat for several real-world applications. We demonstrated the practical feasibility for 10 out of 16 tested AES cores and for 3 out of 3 tested third-party DES cores. An attacker can decrypt all ciphertext blocks that were weakened due to the manipulated hardware configurations. The DES becomes a key-independent permutation that can be inverted by an adversary without any further information. Compared to that, the AES was manipulated in two ways: the first

bitstream alteration turns the AES into a linear function and thus all further ciphertext blocks can be decrypted with only one known plaintext/ciphertext pair. The second manipulation leads to a (partial) key leakage of AES-{128,192,256}. The presented results highlight the importance of integrity checks and that further security mechanisms must be deployed as a part of the hardware configuration, e.g., an internal difficult-to-patch self-test. This work should raise awareness that an attacker can indeed meaningfully manipulate proprietary bitstreams by purpose with moderate efforts. It is important to carefully check an intellectual property core before using it in security applications.

In the second part of this chapter, we proposed a countermeasure that can mitigate an S-box substitution attack leading to an FPGA Trojan. During further research investigations (cf. Chapter 6), we found that the scheme is still vulnerable to key recovery. Hence, obfuscation alone does not represent an appropriate solution.

To further highlight the practical relevance of this attack vector, we demonstrate the first practical third-party Xilinx bitstream manipulation used by a commercially available high-security USB flash device from Kingston, cf. Chapter 5.

Chapter 5

Real-World FPGA Trojan Insertion into a Commercial High-Security Encryption Device

To the best of our knowledge, this is the first demonstration that bitstream manipulation can severely impact the system security of a real-world device.

Contents of this Chapter

5.1 Motivation **61**
5.2 Proceeding of Inserting an FPGA Trojan **62**
5.3 Real-World Target Device **63**
5.4 Building the FPGA Trojan **68**
5.5 ARM Code Modification **70**
5.6 XTS-AES Manipulation and Plaintext Recovery **72**
5.7 Summary of Security Problems **74**
5.8 Conclusion **75**

5.1 Motivation

As part of the revelations about NSA activities [Sny14, Gre14], the notion of interdiction has become known to the public: the interception of shipments to manipulate hardware with the goal to silently install backdoors. Manipulations can occur on firmware or at the hardware level. As shown in Chapter 4, Xilinx FPGAs are particular interesting targets as they can be easily altered by replacing the corresponding bitstream. In security products, an FPGA is often one of various integrated components on a PCB similar to ASICs, which function as a trust anchor for an embedded device, e.g., to securely store cryptographic keys. Thus, usually several components are integrated on the same PCB implementing security features. Therefore, to compromise an embedded device, the weakest link should be identified. As bitstream manipulations are believed to be highly complex and time-consuming, the impact of this attack vector was underestimated in the past.

Related attacks can also be launched in "weaker" settings, for instance, by an adversary who replaces existing equipment with one that is backdoor-equipped or by exploiting reprogramming features to implant a backdoor. Other related attacks are hardware Trojans installed by Original Equipment Manufacturers (OEMs). It can be said that such attacks constitute a realistic threat, because the entire arsenal of security mechanism available to us, ranging from cryptographic

primitives over protocols to sophisticated access control and anti-malware measures, can be invalidated if the underlying hardware is manipulated in a targeted way.

Our investigation required two reverse-engineering steps related to the FPGA bitstream and to the firmware of the underlying ARM CPU. In our Trojan insertion scenario, the targeted USB flash drive is intercepted before being delivered to the victim. The physical Trojan insertion requires the manipulation of the SPI flash memory content, which contains an FPGA bitstream as well as the ARM CPU code. The manipulating of the FPGA bitstream alters the AES-256 algorithm and turns it into a linear function, which can be broken with 32 known plaintext-ciphertext pairs. After the manipulated USB flash drive has been used by the victim, the attacker is able to obtain all user data from the ciphertexts. With this proof-of-concept FPGA bitstream attack of our real-world target device, we further highlight the practical relevance of malicious bitstream manipulation attacks.

5.2 Proceeding of Inserting an FPGA Trojan

In the following, we assume that the attacker is able to intercept a device during the shipping delivery before it arrives at the actual end user. As indicated before, this is not an imaginary scenario as according to the Edward Snowden documents it is known as interdiction [SPI13].

5.2.1 Attack Scenario: Interdiction

The process of interdiction is illustrated in Fig. 5.1. Ordered products (e.g., an USB flash drive) of an end user are secretly intercepted by an intelligence service during the shipment. The target device is modified or replaced by a malicious version, e.g., one with a backdoor. The compromised device is then delivered to the end user. Intelligence agencies can subsequently exploit the firmware or hardware manipulation.

According to the Snowden revelations, hardware Trojans are placed, e.g., in monitor or keyboard cables with hidden wireless transmitters, allowing for video and key logging [SPI13]. It also can be assumed that a PC malware may be distributed with the help of a compromised firmware of an embedded device as recently demonstrated by Nohl *et al.* [NKL14]. This can have severe impacts such as secret remote access by a malicious third party or decryption of user data on physical access. It is relatively easy to alter the firmware, e.g., of an Advanced RISC Machine (ARM) processor, or other similar platforms if no read-out protection is given or no self-tests are utilized.

In contrast, altering hardware such as an ASIC is a highly complex procedure. Recently, Becker *et al.* [BRPB13] demonstrated how a malicious factory can insert a hardware Trojan by changing the dopant polarity of existing transistors in an ASIC. However, this requires a different and considerably stronger attacker scenario than the one shown in Fig. 5.1, because the modification takes place during the manufacturing process. This is a time-consuming, complex, and expensive task and therefore less practical.

On the contrary, at first glance, attacking an FPGA also seems to be similarly challenging because the bitstream file is proprietary and no tools are publicly available that convert the bitstream back to a netlist. However, in Chapter 4 we have shown that a bitstream manipulation attack can indeed be successfully conducted with realistic efforts depending on the hardware configuration.

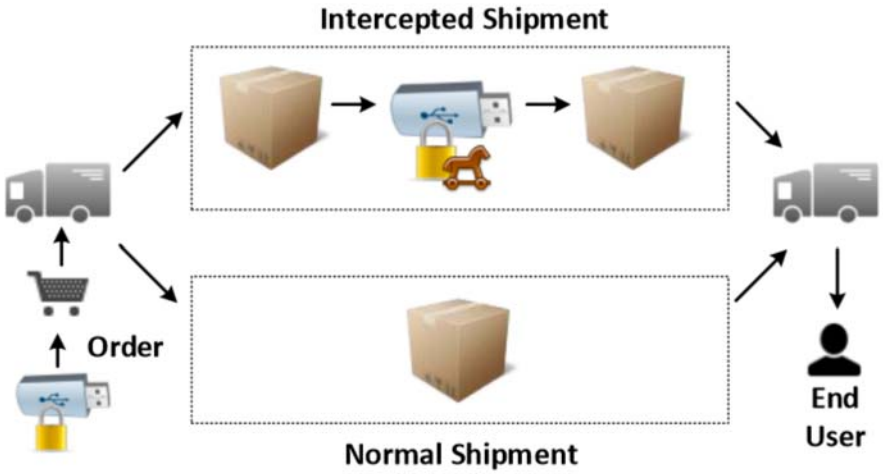


Figure 5.1: Interdiction attack conducted by intelligence services

In our case, we simulated the scenario of Fig. 5.1 by mainly manipulating the bitstream of an FPGA contained in a high-security USB flash drive that utilizes strong cryptography to protect user data. After the manipulated USB flash drive has been forwarded to and utilized for a certain amount of time by the end user, the attacker is able to obtain all user data.

5.3 Real-World Target Device

To demonstrate our FPGA Trojan insertion, we selected the Kingston DataTraveler 5000 [Kin] as the target, which is a publicly available commercial USB flash drive with strong focus on data security. This target device is overall FIPS-140-2 level 2 certified [NIS10]. It uses Suite B [NIS01b] cryptographic algorithms, in particular AES-256, SHA-384, and Elliptic Curve Cryptography (ECC). All user data on our targeted USB drive is protected by an AES-256 in XEX Tweakable Block Cipher with Ciphertext Stealing (XTS) mode. A PC software establishes a secured communication channel to the USB flash drive and enforces strong user passwords.

Due to the FIPS-140 level 2 certification, the device has to fulfill certain requirements of tamper resistance on the physical, hardware and software levels as well as on detecting physical alterations. The PCB of the Kingston DataTraveler 5000 is protected with a titanium-coated, stainless-steel casing and is surrounded by epoxy resin to prevent the undesired access to its internal hardware components.

5.3.1 Initial Steps and Authentication Process

When plugging the USB flash drive into a USB port for the first time, an unprotected partition drive is mounted making the vendor's PC software available to the user. Meanwhile, in the background, this software is copied (only once) to a temporary path from which it is always executed, cf. the upper part of Fig. 5.5.

In an initial step, the end user needs to set a password. Afterwards, the user must be authenticated to the device using the previously-set password. This means that the key materials must be somewhere securely stored, which is commonly a multiple-hashed and salted password.

On every successful user authentication (mainly performed by the ARM CPU and the PC software), the protected partition drive is mounted allowing access to the user data. Any data written to the unlocked partition is encrypted with AES by the Xilinx FPGA and the corresponding ciphertexts are written into the sectors of the micro SD card as indicated in Fig. 5.5.

When unplugging the USB flash drive and for the case that an adversary has stolen this device, he/she is not able to access the user data without the knowledge of the corresponding password. According to [Kin], after 10 wrong password attempts, the user data is irrevocably erased to prevent an attacker from conducting successful brute-force attempts.

5.3.2 Physical Attack — Revealing the FPGA Bitstream

To conduct an FPGA hardware Trojan insertion, we need to have access to the bitstream. Thus, in the first step we were able to remove the epoxy resin. Indeed, this procedure was much easier than expected. We locally heated up the epoxy resin to 200°C (by a hot-air soldering station) turning it to a soft cover and removed the desired parts by means of a sharp instrument, e.g., a tiny screwdriver (see Fig. 5.2).

Figure 5.2: Epoxy removal of Kingston DT 5000 with screwdriver

Figure 5.3: Eavesdropping the bitstream of Kingston DT 5000 with opened case

By soldering out all the components, exploring the double-sided PCB and tracing the wires, we detected that an ARM CPU configures the Xilinx FPGA through an 8-bit bus. We also identified certain points on the PCB by which we can access each bit of the aforementioned configuration bus. Therefore, we partially removed the epoxy resin of another operating identical target (USB flash drive) to access these points and then monitored this 8-bit bus during the power-up (by plugging the target into a PC USB port) and recorded the bitstream sent by the ARM CPU, cf. Fig. 5.3. Note that SRAM-based FPGAs must be configured at each power-up. By repeating

the same process on several power-ups as well as on other identical targets, we could confirm the validity of the revealed bitstream and its consistency for all targets. We should emphasize that the header of the bitstream identified the type and the part number of the underlying FPGA matched with the soldered-out component.

We also identified an Serial Peripheral Interface (SPI) flash among the components of the PCB. As we have soldered out all the components, we could easily read out the content of the SPI flash. Since such components are commonly used as standalone non-volatile memory, no read-out protection is usually integrated. At first glance it became clear that the SPI flash contains the main ARM firmware (2nd ARM image). We also found another image (1st ARM image) initializing the necessary peripherals of the microcontroller. Furthermore, we identified that the bitstream, which we have revealed by monitoring the configuration bus, has been stored in the SPI flash, cf. Fig. 5.4.

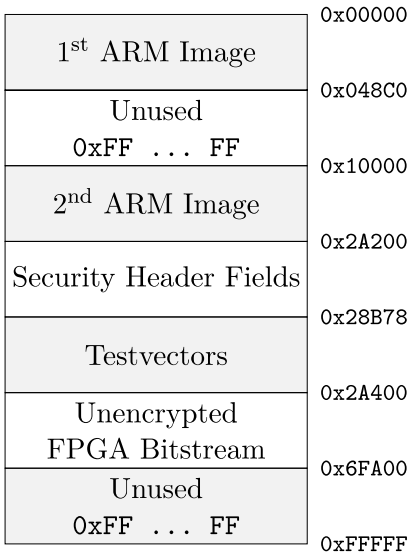


Figure 5.4: Address space layout of the SPI flash

Motivated by these findings we continued to analyze all other components of the USB flash drive and thus describe our results in the following.

5.3.3 Overview and Component Details

Based on our accomplishments described above, we could identify the following main components placed on the double-sided PCB:

- NXP LPC3131 with embedded ARM926EJ-S CPU operating at 180 MHz
- Xilinx Spartan 3E (XC3S500E) FPGA
- HSM from SPYRUS (Rosetta Micro Series II) providing ECDH, DSA, RSA, DES, 3-DES, AES, SHA-1, etc.
- 2 GB Transcend Micro SD card (larger versions also available)

- 1 MB (AT26DF081A) SPI flash

We revealed the layout of the circuit through reverse-engineering. The whole circuit is depicted in Fig. 5.5. This step was conducted by tracing the data buses of the PCB and by decompiling the PC software as well as the identified ARM firmware. Both executables were decompiled with Hex-Rays [HR12]. The resulting source code served for further reverse-engineering.

The main task of the identified ARM CPU (master device) is to handle the user authentication, while the Xilinx FPGA (slave device) is mainly responsible for the user data encryption and decryption. It should be noted that the FPGA is also partially involved in the authentication process and exhibits our main target for manipulation. We could not confirm the key storage location, but we assume that the key materials are securely stored in the HSM, cf. Fig. 5.5. As we demonstrate in this chapter, we need neither any access to the key materials nor any knowledge of the key derivation function to be later on able to decrypt sensitive user data.

As stated before, both images (ARM CPU code and FPGA bitstream) were discovered in the SPI flash that are loaded and executed during the power-up of the USB flash drive.

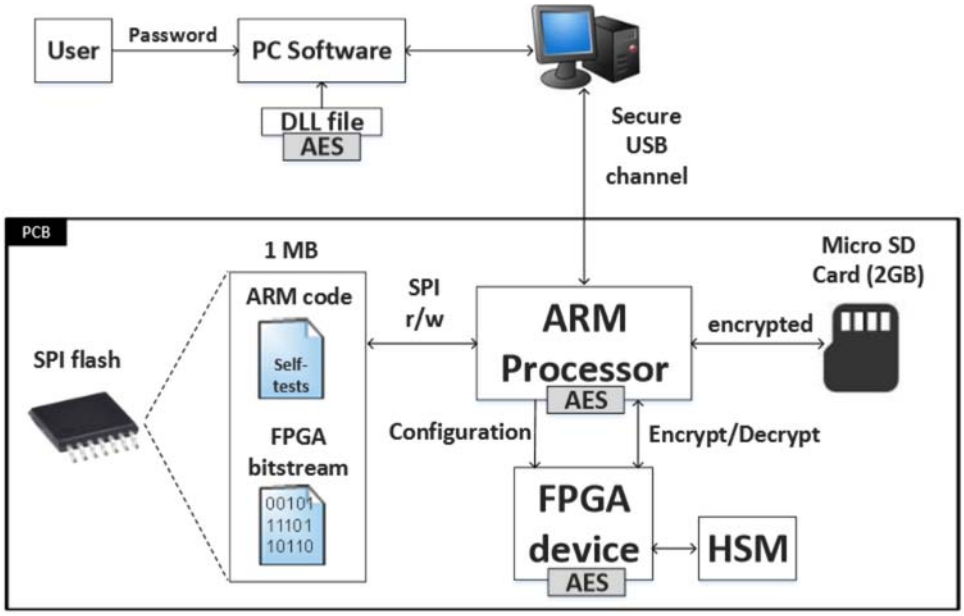


Figure 5.5: Overview of revealed circuit of our target device

5.3.4 Unlinking FPGA Trojan from the Authentication Process

During our FPGA Trojan insertion, we identified several AES cores, as shown in Fig. 5.5:

(1) *AES core in the PC Software:* used during user authentication.

(2) *AES core in the ARM code:* used during user authentication.

(3) *AES core in the FPGA:* used during user authentication (partially) as well as for encrypting user data at high speed (main purpose).

If only the functionality of the FPGA AES core is manipulated, the target device would not operate properly anymore because all three AES cores need to be consistent due to the identified authentication dependencies. To be more precise, all three AES cores are involved in the same authentication process.

As our goal is to insert a hardware Trojan by manipulating the AES core of the FPGA, we first needed to unlink the dependency (of the AES cores) between the ARM CPU and the Xilinx FPGA, cf. Fig. 5.6. Therefore, we eliminated this dependency by altering parts of the ARM firmware, but we realized that any modification is detected by an integrity check. We identified several self-tests that are conducted – by the ARM CPU – on every power-up of the USB flash drive.

Further analyses revealed the presence of test-vectors. They are used to validate the correctness of the utilized cryptography within the system. The utilized self-tests are explained in Section 5.5.1 in more detail. In Section 5.5.2, we demonstrate how to disable them and how to unlink the aforementioned dependencies.

To sum up, our intended attack is performed using the following steps:

(1) Identify and disable the self-tests,

(2) Unlink the AES dependency between the ARM and FPGA, and

(3) Patch (reprogram) the FPGA bitstream meaningfully.

Fig. 5.6 and Fig. 5.7 illustrate the impact of these steps. As can be seen, canceling the dependency allows us to alter the AES core and add an FPGA Trojan. The details of how

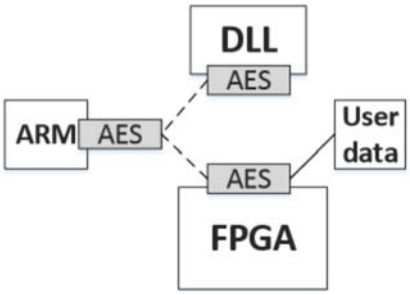


Figure 5.6: User authentication (dashed) and user data (solid) dependencies before modification

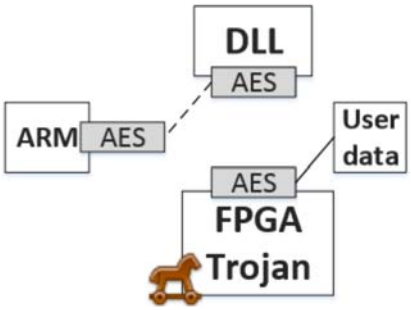


Figure 5.7: User authentication (dashed) and data (solid) dependencies after modification

we could successfully alter the FPGA bitstream to realize a hardware Trojan are presented in Section 5.4. Below, we discuss why modifying a bitstream is more suited for planting an FPGA Trojan than replacing the whole bitstream.

5.3.5 Modifying Bitstream vs. Replacing Entire Bitstream

We want to pinpoint that replacing the complete FPGA design to insert a Trojan does not necessarily mean that an attack is less complicated to be performed. Replacing the whole FPGA bitstream by a completely new design is a more challenging task. The attacker would

need to further reverse-engineer and fully understand the whole FPGA environment (ARM code, data buses, protocols, etc.) and re-implement all functions to ensure the system's compatibility. It turned out to be the easier and faster approach, since we were able to modify this third-party IP core without the need to reverse-engineer or modify any part of the routing.

Thus, we only focus on detecting and replacing the relevant parts of the utilized FPGA design. By doing so, we secretly insert a stealth FPGA Trojan that turns the AES encryption and decryption modules into certain compatible weak functions, cf. Section 5.6.

5.3.6 Manipulation – Master vs. Slave

Admittedly, on one hand the Kingston DataTraveler 5000 is not the best target device to demonstrate an FPGA hardware Trojan insertion because the embedded ARM CPU acts as the master device containing all control logic. The FPGA is merely used as an accelerator for cryptographic algorithms. In order to preserve the functionality of the USB flash drive with an active FPGA hardware Trojan the ARM CPU firmware – as previously explained – has to be customized too, i.e., the integrity check of the ARM CPU code needs to be disabled (explained in Section 5.5). At this point, the attacker can alter the firmware to not encrypt the user data at all, turning the device into a non-secure drive accessible to everyone. As another option, the attacker can secretly store the encryption key which would result in a conventional software-based embedded Trojan.

On the other hand, there are solutions which contain only an FPGA used as the master device [EGP+07]. Conventional software-based embedded Trojans are not applicable in those systems. Our attack is a proof of concept that FPGA hardware Trojans are practical threats for the FPGA-based systems where no software Trojan can be inserted. Our attack also highlights the necessity of embedded countermeasures on such systems to detect and defeat FPGA hardware Trojans.

5.4 Building the FPGA Trojan

In this section, we present the information which can be extracted from the given bitstream file followed by our conducted modification on the AES-256 core. The impact of this modification – considering the utilized XTS mode of operation – is described in Section 5.6.

5.4.1 Analysis of the Extracted Bitstream

Based on the method presented in Section 5.2, we could dump and analyze the initial memory configuration of each block RAM of the extracted bitstream. The Spartan 3E FPGA contains up to 20 block RAMs. We figured out that only 10 out of 20 block RAMs are used by the extracted FPGA design. We observed that the block RAMs are organized in a byte-wise manner fitting well to the structure of the AES state.

Our analysis revealed the presence of multiple instances of certain precomputed substitution tables. After investigating the extracted data in more detail, we obtained a structure for each table. We refer to the four identified tables whose details are depicted in Table 5.1. Each substitution table stores 256 entries that can be accessed using the input $x \in \{0, 1, ..., 255\}$.

Our analysis revealed that the following precomputed substitution tables are stored in several block RAMs:

$$\begin{aligned} \widetilde{T}(x) &= 01 \circ \mathbf{S}(x)||01 \circ \mathbf{S}^{-1}(x)||02 \circ \mathbf{S}(x)||03 \circ \mathbf{S}(x) \\ MC^{-1}(x) &= 09 \circ x||11 \circ x||13 \circ x||14 \circ x \\ S(x) &= \mathbf{S}(x) \\ S^{-1}(x) &= \mathbf{S^{-1}}(x) \end{aligned}$$

Detected tables	**Identified block RAM Data**
16 $\widetilde{T}(x)$ instances (1024 bytes each)	000: $\mathbf{S}(00)\|\|\mathbf{S}^{-1}(00)\|\|02 \circ \mathbf{S}(00)\|\|03 \circ \mathbf{S}(00)$ 001: $\mathbf{S}(01)\|\|\mathbf{S}^{-1}(01)\|\|02 \circ \mathbf{S}(01)\|\|03 \circ \mathbf{S}(01)$ ... 0FF: $\mathbf{S}(FF)\|\|\mathbf{S}^{-1}(FF)\|\|02 \circ \mathbf{S}(FF)\|\|03 \circ \mathbf{S}(FF)$
16 $MC^{-1}(x)$ instances (1024 bytes each)	000: $09 \circ 00\|\|11 \circ 00\|\|13 \circ 00\|\|14 \circ 00$ 001: $09 \circ 01\|\|11 \circ 01\|\|13 \circ 01\|\|14 \circ 01$ ... 0FF: $09 \circ FF\|\|11 \circ FF\|\|13 \circ FF\|\|14 \circ FF$
4 $S(x)$ instances (256 bytes each)	000: $\mathbf{S}(00)$ 001: $\mathbf{S}(01)$ ... 0FF: $\mathbf{S}(FF)$
4 $S^{-1}(x)$ instances (256 bytes each)	000: $\mathbf{S}^{-1}(00)$ 001: $\mathbf{S}^{-1}(01)$ ... 0FF: $\mathbf{S}^{-1}(FF)$

Table 5.1: Identified substitution tables stored in block RAM

In other words, we identified the tables which realize the inverse MixColumns transformation $MC^{-1}(\cdot)$, the SubBytes $SB(\cdot)$ and inverse SubBytes $SB^{-1}(\cdot)$. However, $\widetilde{T}(\cdot)$ is not equivalent to any T-box $(T_0, \ldots, T_3)$, cf. [KA08], but exhibits a very similar structure: one entry includes the S-box, the inverse S-box, and the S-box multiplied by two and three ($02 \circ \mathbf{S}(\cdot)$ and $03 \circ \mathbf{S}(\cdot)$). In particular $\widetilde{T}(\cdot)$ combines the SubBytes and MixColumns transformations, and thus has the same purpose as one T-box, but one remarkable difference is the storage of the inverse S-box $S^{-1}(\cdot)$. Note that all four T-boxes $T_0, \ldots, T_3$ can be easily derived from $\widetilde{T}$.

5.4.2 Modifying the Third-Party FPGA Design

Our main goal is to replace all AES S-boxes to the identity function, cf., Section 5.6. For this purpose, we have to replace all identified look-up table instances of Table 5.1. We need to replace all S-box values such that $S(x) := x$ and the inverse S-box to $S^{-1}(x) := x$. This is

essential in order to synchronize the encryption and decryption functions. Hence, it leads to the following precomputation rules for $x \in \{0, 1, ..., 255\}$:

$$\begin{aligned} \widetilde{T}(x) &= 01 \circ x || 01 \circ x || 02 \circ x || 03 \circ x \\ MC^{-1}(x) &= 09 \circ x || 11 \circ x || 13 \circ x || 14 \circ x \\ S(x) &= x \\ S^{-1}(x) &= x \end{aligned}$$

Note that the modifications must be applied on all detected instances of the look-up tables in the bitstream file, cf. Table 5.1.

In the next step, we updated the SPI flash with this new malicious bitstream and powered up the USB flash drive by plugging it into the PC. We could observe that the FPGA modification is successful as the encryption and decryption still work. This is true only when all instances of the relevant substitution tables (S-box and its inverse) are modified appropriately.

From now on we consider that the malicious AES core is running on the FPGA. Hence, in the next section, we explain in the next section how this Trojan insertion can be exploited even though a complex mode of operation (AES-256 in XTS mode) is used by our altered FPGA design.

5.5 ARM Code Modification

In this section, we briefly describe the cryptographic self-tests and ARM firmware modifications essential to enable the above presented FPGA hardware Trojan insertion.

5.5.1 Utilized Self-tests

When we reverse-engineered the ARM code using the tool IDA Pro, we were able to identify several functions that check the integrity of the ARM firmware and consistency of several cryptographic functions. Every executed self-test must return a specific integer indicating whether the test passed or not. If any self-test fails, the target device switches to an error state.

The corresponding test-vectors used by the self-tests are stored in the SPI flash. Table 5.2 provides an overview of all self-tests and the integrity checks. Besides this, we also identified several relevant security header fields that are processed by the ARM CPU, cf. Table 5.3. The ARM CPU expects to receive a specific signature (during power-up of the system) from the Xilinx FPGA to ensure that it operates correctly after the configuration process. Also, the bitstream length is coded in the header such that the ARM CPU knows the amount of configuration bytes. Lastly, a SHA-384 hash value, calculated over the main ARM firmware, is appended to ensure the program code integrity.

5.5.2 Disabling Self-tests to Modify ARM Code and FPGA Bitstream

Preliminary tests have shown that even minor code changes, which do not influence the behavior of the firmware, cause the USB flash drive to enter the error state and halt during power-up. It was concluded that there exists an implemented self-test at least checking the integrity of the code. Thus, it became a mandatory prerequisite to find and deactivate such a test. The responsible code was identified due to its specific control-flow graph and function calls.

Self-test function	**Utilized parameter of self-test**
AES-256 (CBC)	Key K = `0x2B2B...2B` (16 Bytes) IV = `0x3C3C...3C` (16 Bytes) Input x = `0x1111...11` (32 Bytes)
AES-256 (XTS)	Key K_1 = `0x2021...3F` (32 Bytes) Key K_2 = `0x4041...5F` (32 Bytes) Tweak = `0xA2566E3D7EC48F3B` Input x = `0xF0F1...FF` (16 Bytes)
SHA-{224,256,384,512}	Input x = "`abc`"
Integrity check	**Input**
SHA-384	Main ARM firmware

Table 5.2: Identified self-tests and firmware integrity check

Field Name	**Offset**	**Byte size**	**Value**
Header Signature	0x00	4	0x11223344
FPGA signature	0x04	16	"`SPYRUS_HYDRA2005`"
Bitstream length	0x14	4	0x45600
SHA-384 hash of 2^{nd} image	0x1D0	48	SHA-384(2^{nd} image)

Table 5.3: Security header fields

In addition to the firmware integrity, the correct functionality of several cryptographic algorithms is tested: the AES, ECC, and SHA in the ARM code and the AES inside the FPGA. The individual checks are performed in dedicated functions invoked by the main self-test function, and their corresponding return values are verified. Finally, the self-test succeeds only in case all individual checks are passed.

In order to disable the self-test the code was patched in a way that the function always returns zero, which is the integer representation for success. Hence, arbitrary firmware modifications and changes to the cryptographic algorithms can be applied after this patch.

5.5.3 Separating Key Derivation and FPGA AES IP-Core

As explained before, there is a software AES implementation executed by the ARM CPU and a considerably faster hardware AES instance inside the FPGA, cf. Fig. 5.6. They are both capable of ECB, CBC and XTS operation modes. The software AES is mainly used for self-tests and the hardware AES for key derivation as well as encryption and decryption of the user data stored on the USB flash drive. The key derivation requires the establishment of a secure communication channel between the PC software and the USB flash drive. The FPGA hardware Trojan weakens the AES IP-core making it incompatible to the standard AES, cf. Section 5.6. Thus, the initialization of the communication channel fails and the USB flash drive goes to an error state. To avoid such a situation the firmware has to be changed in such a way that only the

original software AES is used during the key derivation and the secure channel establishment (instead of the modified hardware AES inside the FPGA).

The ARM code internally uses a unified AES API. Four parameters are passed to its AES instance constructor routine. They hand over the key, the key length, the mode of operation and a flag indicating whether the ARM CPU or the FPGA is selected for the actual computations. The creation of all the AES instances, which are related to the key derivation as well as secure channel establishment, had to be patched. Consequently, all corresponding AES encryptions and decryptions are computed by the ARM CPU instead of the FPGA. In total, the parameters of 12 AES instance constructor calls have been patched to eliminate the AES dependency between the ARM and FPGA.

5.5.4 Recording XTS-AES Parameters

In order to recover all user data from the USB flash drive we need several values for the attack: 32 plaintext-ciphertext pairs of the same sector, the sector number and the initial tweak value. The latter parameter is hard-coded in the firmware and was obtained by static analysis. The plaintext-ciphertext pairs are acquired at runtime during normal operation of the USB flash drive. In the ARM code, there is a highly-speed-optimized function which reads data from the embedded SD card, sends them to the FPGA for decryption and finally copies the plaintexts from the FPGA to the USB endpoint so that the computer receives the requested data. This function was intercepted at several positions in a way that the plaintext-ciphertext pairs and the initial sector number could be obtained. They are then written (only once) in one unused sector of the embedded SPI flash from where they can be read out by an attacker to launch the cryptographic attack.

As explained in Section 5.6, having this information is essential to decrypt the phony ciphertexts due to the underlying XTS mode. We practically verified the plaintext recovery of the weakly encrypted ciphertexts stored on the SD card of our target device.

5.6 XTS-AES Manipulation and Plaintext Recovery

In this section, the cryptographic block cipher mode of operation XTS is briefly presented. As already indicated in the previous sections, our target device uses a sector-based disk encryption of user data. The tweakable block cipher XTS-AES is standardized in IEEE 1619-2007 [IEE08] and used by several disk-encryption tools, e.g., VeraCrypt and dm-crypt as well as commercial devices like our targeted USB flash drive.

Each sector (usually 512 bytes of memory) is assigned consecutively to a number, called *tweak* and denoted by i in the following, starting from an arbitrary non-negative integer. Also, each data unit (128-bit in case of XTS-AES) in a sector is sequentially numbered, starting from zero and denoted by j. This pair (i, j) is used for encryption and decryption of each data unit's content.

In general, XTS-AES uses two keys (k_1, k_2). The first key k_1 is used for the plaintext encryption and the second key k_2 for the tweak encryption. The XTS-AES encryption diagram is depicted in Fig. 5.8. After the tweak encryption, the output is multiplied by α^j in the Galois field GF(2^{128}), where α is a primitive element, e.g., $\alpha = x$ and j is the data unit position in the

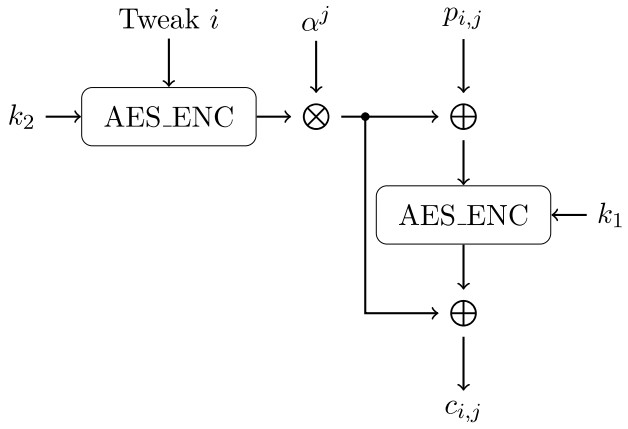


Figure 5.8: XTS-AES encryption block diagram overview

sector i. This result is then XOR-ed before and after encryption of the plaintext block we refer to as $p_{i,j}$. The encryption of one 16-byte plaintext can be described as

$$c_{i,j} = (\mathrm{AES}_{k_2}(i) \otimes \alpha^j) \oplus \mathrm{AES}_{k_1}((\mathrm{AES}_{k_2}(i) \otimes \alpha^j) \oplus p_{i,j}),$$

while the decryption is computed as follows

$$p_{i,j} = (\mathrm{AES}_{k_2}(i) \otimes \alpha^j) \oplus \mathrm{AES}^{-1}_{k_1}((\mathrm{AES}_{k_2}(i) \otimes \alpha^j) \oplus c_{i,j}).$$

5.6.1 Manipulation of AES-XTS

As explained in Section 4.4.6, a manipulated AES core ($\widetilde{\mathrm{AES}}_k(\cdot)$), for which the AES S-boxes $S(x)$ were substituted by the linear function $S^{id}(x) = x$, can be described as follows.

$$\widetilde{c} = \widetilde{\mathrm{AES}}_k(p) = \underbrace{SR(\ldots MC(SR(p)\ldots))}_{:=MS(p)} \oplus \widetilde{K} = MS(p) \oplus \widetilde{K}$$

So far, in this thesis we only discussed on how to decrypt all phony ciphertexts $\widetilde{c}$ considering a single encryption call of $\widetilde{\mathrm{AES}}_k(p)$. For our target device, we need to extent the approach for the underlying XTS mode of operation. In this case, an XTS-AES ciphertext can be described as a linear equation too:

$$\begin{aligned}
\widetilde{c_{i,j}} &= (\widetilde{\mathrm{AES}}_{k_2}(i) \otimes \alpha^j) \oplus \widetilde{\mathrm{AES}}_{k_1}((\widetilde{\mathrm{AES}}_{k_2}(i) \otimes \alpha^j) \oplus p_{i,j}) \\
&= (MS(i) \oplus \widetilde{K_2}) \otimes \alpha^j \oplus MS((MS(i) \oplus \widetilde{K_2}) \otimes \alpha^j \oplus p_{i,j}) \oplus \widetilde{K_1} \\
&= \underbrace{(MS(i) \otimes \alpha^j) \oplus MS(MS(i) \otimes \alpha^j)}_{TW_{i,j}} \oplus MS(p_{i,j}) \oplus \underbrace{(\widetilde{K_2} \otimes \alpha^j) \oplus MS(\widetilde{K_2} \otimes \alpha^j) \oplus \widetilde{K_1}}_{CK_j}
\end{aligned}$$

Since $MS(\cdot)$ is a linear function, the tweak part $TW_{i,j}$, the plaintext-related part $MS(p_{i,j})$, and the key-related part CK_j can be seen as separated XOR terms. Every plaintext $p_{i,j}$ is encrypted in this way by the FPGA hardware Trojan of our target device. An attacker only needs to gather the following information:

- 32 plaintext-ciphertext pairs $(p_{i,j}, c_{i,j}), j \in \{0, ..., 31\}$ of one sector (512-byte wide), and
- knowledge about the tweak value i corresponding to this sector.

Due to the combination of the data unit's position j and the key k_2 (through Galois field multiplication by α^j), each position j in a sector has its own constant key-related part CK_j. Further, CK_j is constant for every sector of the memory as it is independent of i. Once all 32 CK_j are known to an attacker, any plaintext $p_{i,j}$ can be easily recovered. Therefore, the attack requires only 32 plaintext-ciphertext pairs of one arbitrary sector to obtain all CK_j values, cf. Algorithm 10.

Algorithm 10 Computation of key-dependent secrets CK_j

Input: 32 Plaintext-ciphertext pairs $(p_{i,j},\ \widetilde{c_{i,j}})$ from a linearized AES-XTS, Tweak i
Output: Key-dependent secrets CK_j for $j \in \{0, 1, \ldots, 31\}$

for $j =$ 0 *to* 31 **do**
$TW_{i,j} = (MS(i) \otimes \alpha^j) \oplus MS(MS(i) \otimes \alpha^j)$
$MS_{i,j} = MS(p_{i,j})$
$CK_j = \widetilde{c_{i,j}} \oplus TW_{i,j} \oplus MS_{i,j}$
Return all CK_j

Once an attacker obtained all CK_j, any read out ciphertext that we refer to as $\widetilde{y}_{i,j}$ can be decrypted as described in Algorithm 11.

Algorithm 11 Decryption of ciphertexts that were encrypted with manipulated AES-XTS

Input: Weak ciphertext $\widetilde{y}_{i,j}$ from a modified AES-XTS with $S^{id}(\cdot)$, 32 previously obtained key-dependent secrets CK_j, Tweak i and block j
Output: Plaintext $x_{i,j}$ corresponding to weak ciphertext $\widetilde{y}_{i,j}$

1: $TW_{i,j} = (MS(i) \otimes \alpha^j) \oplus MS(MS(i) \otimes \alpha^j)$
2: $MS(p_{i,j}) = \widetilde{y}_{i,j} \oplus TW_{i,j} \oplus CK_j$
3: $p_{i,j} = MS^{-1}(MS(p_{i,j}))$
4: Return $p_{i,j}$

It is worth mentioning that the produced ciphertext still appears to be random for a victim, who visually inspects the phony ciphertexts from the micro SD card. Therefore, the victim cannot observe any unencrypted data as it would be the case if the FPGA is simply bypassed.

5.7 Summary of Security Problems

We summarize the security problems of our investigated target device and further outline which security barriers might be inserted by the vendor to improve the security of the analyzed USB flash drive.

As previously stated, during our analysis we found an HSM from SPYRUS that is directly connected to the Xilinx FPGA over a single-bit bus. According to [Mic] it provides certain

cryptographic primitives and serves as a secure storage device, e.g., for secret (symmetric) keys. We suggest including the following security measure: during the power-up of the USB flash drive, the FPGA should validate its AES implementation using the AES core provided by the HSM. It should be extremely challenging for an attacker to alter the AES core of the HSM as its internal functionality is realized by an ASIC. The HSM should decide whether the USB flash drive continues (no alteration detected) or switches to an error state (alteration detected).

To further raise the bar for an attacker, the FPGA design should include built-in self-tests for the S-box configuration as well as for the whole AES core. To be more precise, it is recommended to include several test vectors in the FPGA firmware so the FPGA can validate its consistency. Probably, the built-in self-tests do not hinder a more powerful attacker who can disable them, but the reverse-engineering efforts are significantly increased and require a more powerful adversary. Since in our attack scenario we exploited the content of the block RAMs, it is also important to assure their integrity. Their initial content can be encrypted with an appropriate mode of operation: a built-in circuitry in the FPGA design might (during the FPGA power-up) decrypt the block RAM's contents and update them with the corresponding decrypted data. By doing so, an attacker cannot replace the highly important S-boxes in a meaningful way to implant a Trojan-like functionality of the AES core. We proposed such a countermeasure in Section 4.5.

More importantly, all self-tests (including those we found) should be performed by the HSM. Therefore, the HSM should verify the integrity of the ARM code. Further, the bitstream of the FPGA must be protected (not stored in plain in the SPI flash) and its integrity must be verified e.g., by the HSM. This should prevent any modification attempt on the ARM code as well as on the bitstream, making a firmware modification attack rather challenging.

5.8 Conclusion

In this chapter, we demonstrated the first practical real-world FPGA Trojan insertion into a high-security commercial product to weaken the overall system security. We reverse-engineered a third-party FPGA bitstream to a certain extent and replaced parts of the FPGA logic in a meaningful manner on the lowest level. In particular, we significantly weakened the embedded XTS-AES-256 core and successfully canceled its strong cryptographic properties making the whole system vulnerable to cryptanalysis. Our work is a proof of concept that an FPGA can also be one of several weak points of a seemingly protected system. It is important to ensure the integrity of the FPGA bitstream even though its file format is proprietary. This is especially critical in applications where the FPGA acts as the master device. Future work must deal with counterfeiting bitstream modification attacks by developing appropriate countermeasures that have to be implemented within an FPGA design.

Chapter 6

Bitstream Fault Injections (BiFI) - Automated Fault Attacks Against SRAM-based FPGAs and AES

Targeted bitstream manipulations, requiring the detection of relevant sub-circuits, work well for the majority of AES implementations, but nevertheless there is no success guarantee for special AES implementations. In this chapter, we go one step further regarding crypto-related manipulations of third-party bitstreams and improve the success rate of key recovery. During further research investigations, we found a more generic attack strategy allowing to automatically manipulate a third-party FPGA bitstream realizing a cryptographic primitive such that the underlying secret key is revealed. Our attack strategy turned out to be surprisingly powerful as we could attack 15 out of 16 AES cores in an automated fashion. Hence, we introduce a novel class of bitstream fault injection (BiFI) attacks, which does not require any reverse-engineering to undermine cryptographic cores. As opposed to the targeted S-box manipulations in the bitstream, this kind of attack can be automatically mounted without any detailed knowledge about either the bitstream format or the design of the cryptographic primitive under attack. We further demonstrate that the bitstream encryption schemes of Xilinx FPGAs do not necessarily prevent our attack if the integrity of the encrypted bitstream is not carefully checked. As a proof of concept, we successfully attack 12 out of 13 encrypted AES cores of a Xilinx Virtex 5 FPGA.

Contents of this Chapter

6.1 Related Work **78**
6.2 Motivation and Contribution **78**
6.3 Background **79**
6.4 Attack Idea **80**
6.5 Experimental Setup and Results **83**
6.6 Analysis **91**
6.7 Discussions and Countermeasures **96**
6.8 Conclusion **98**

6.1 Related Work

Methods of how to attack cryptographic implementations and how to secure them have been studied for a long time in the scientific literature. As one of the earlier references, Boneh *et al.* [BDL97] demonstrated in 1997 that the RSA public-key scheme as well as authentication protocols are vulnerable to fault injections. The idea is to exploit transient hardware faults that occur during the computations of the cryptographic algorithm. Due to the injected faults, faulty intermediate values may propagate sensitive information to the output revealing the private key. This concept was extended by Biham and Shamir [BS97] – known as Differential Fault Analysis (DFA) – to recover the secret key from symmetric block ciphers such as DES. In 2003, Piret and Quisquater [PQ03] introduced a sophisticated fault model for AES which enables an attacker to recover the secret key with only two faulty ciphertexts.

Additionally, numerous other implementation attacks on hardware have been proposed, including power and EM side-channel attacks [MBO+05], glitch-based fault attacks [LSG+10, CT05], laser fault attacks [SA02] and photonic emission attacks [SNK+13], each requiring different expertise and equipment. For a classification of fault injection attacks, we refer to contribution [VKS11]. Notably, all proposed methods have in common that they cannot be executed automatically for different targets. They always require an experienced engineer to adjust the attack to each new target that may become a time-consuming task in a black-box scenario.

6.2 Motivation and Contribution

We introduce a new strategy to efficiently and automatically extract secrets from FPGA designs which we coin bitstream fault injection (BiFI) attack. Moreover, the goal is to reduce the required expertise as well as the engineering hours. Instead of (partially) reverse-engineering a hardware configuration as described in Chapter 4, we *manipulate an unknown bitstream without any knowledge of the configuration* resulting in faulty ciphertexts. These faulty ciphertexts can then be used to recover the secret key. The general idea that one might recover secret keys by manipulating bitstreams without reverse-engineering was first mentioned in [TML11], but no concrete attack was proposed and it remained unclear if such an attack is indeed feasible in practice. In this chapter we not only show that such attacks are indeed feasible, but also that they are much more powerful than assumed. A surprising large number of bitstream manipulations result in exploitable faulty ciphertexts. A key finding of our analysis is that it is not necessary to make targeted manipulations based on knowledge of the target design. Instead, a set of design-independent manipulation rules can be applied automatically to different regions of the target bitstream until the attack succeeds. Thus, one only needs to develop an attack tool once and can apply it to any design that implements the same cryptographic algorithm. Crucially, no FPGA reverse-engineering expertise is needed to perform the BiFI attack on different targets. We verified the feasibility of the attack with 16 different AES implementations on a Spartan 6 FPGA. Out of those, 15 designs could be successfully attacked with BiFI in an automated fashion.

It seems that bitstream encryption can prevent BiFI attacks, but this assumption turned out to be not necessarily true. Already in [TML11] it was noted that bitstream manipulations might be possible in theory on encrypted bitstreams if no integrity checks are used. However, it was also noted that the CRC feature as it is implemented in Virtex 2 through Virtex 5 FPGAs

should prevent bitstream manipulation attacks such as BiFI. Nonetheless, we could successfully attack bitstreams on a Virtex 5 with an enabled bitstream encryption scheme. We demonstrate this by successfully attacking 12 out of 13 encrypted AES cores. Hence, bitstream encryption in combination with a CRC feature is not necessarily enough to stop BiFI attacks.

Note that the majority of currently-deployed Xilinx FPGAs appear to be vulnerable to the BiFI attack. However, (automatic) bitstream manipulations attacks were widely neglected in the past due to the believed high complexity needed for reverse-engineering. In contrast, the attacks presented in this thesis show that bitstream fault injection attacks can be performed completely automatically without any user interaction and particularly without reverse-engineering of the design or the bitstream format. Hence, BiFI attacks are low cost and require very little expertise. Furthermore, in contrast to all previous works, BiFI (up to some extent) can nevertheless deal with encrypted bitstreams.

Note that during our experiments, most AES cores could be attacked with *only one* faulty ciphertext. In many cases, neither the plaintext nor the fault-free ciphertext are required. In addition to that, as a negative result, it turned out that the proposed countermeasure of Section 4.5, designed to mitigate a key recovery by means of bitstream manipulations, was vulnerable to the BiFI attack, as well. This highlights the importance of dedicated countermeasures that should be part of the hardware configuration.

6.3 Background

Since we will later introduce various LUT modification rules in the bitstream, we first provide some low-level background information on how LUTs are used in most Xilinx FPGAs. Fig. 6.1 illustrates a simplification of the most common slice configurations. Note that we ignored some hardware elements such as flip-flops and switch-boxes for the sake of simplicity.

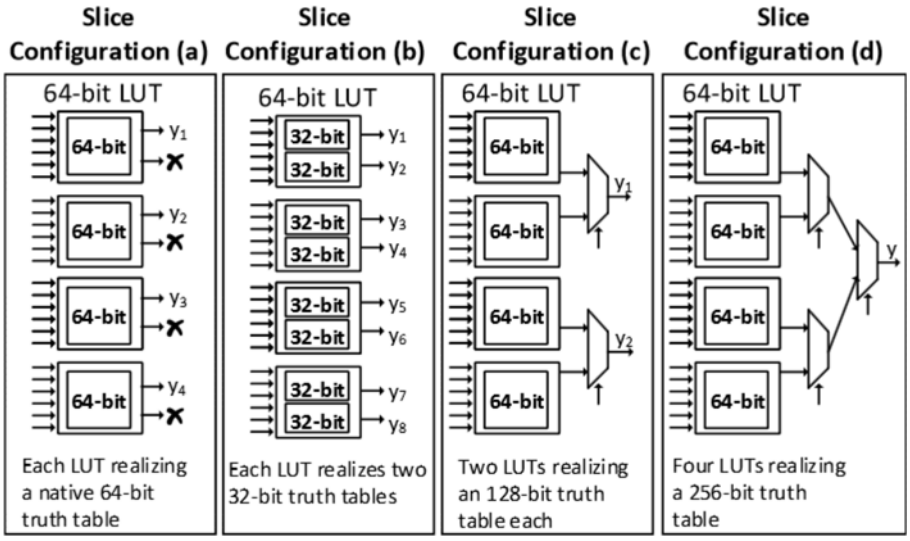


Figure 6.1: Subset of the most commonly used possible slice configurations with focus on look-up tables

Each 6-bit LUT can implement one out of 2^{64} possible $6 \mapsto 1$ Boolean functions, cf. configuration (a) in Fig. 6.1. Alternatively, each 64-bit LUT can be split into two 32-bit subtables in order to realize two different $5 \mapsto 1$ Boolean functions with shared inputs, cf. configuration (b) in Fig. 6.1. Two (resp. four) LUTs within one slice can also be combined to realize larger truth tables with 128 bits (resp. 256 bits) to realize $7 \mapsto 1$ (resp. $8 \mapsto 1$) Boolean functions, cf. configuration (c) and (d) in Fig. 6.1.

6.4 Attack Idea

The first step of the attack is to read out the bitstream of the device under attack from the non-volatile memory or by wiretapping the configuration data bus. The attack tool then repeatedly *i*) manipulates the bitstream by changing (randomly appearing) LUT contents, *ii*) configures the target device with the altered hardware configuration, and *iii*) queries the manipulated design to collect faulty ciphertexts. The faulty ciphertexts are used to recover the key by testing a set of hypotheses, e.g., that the faulty ciphertext is the plaintext XORed with the key. However, there are numerous LUTs in even small FPGAs, and testing all possible modifications on all LUT bits is not practically feasible. Therefore, we try to reduce the space for manipulations by defining particular *rules* in such a way that the faulty ciphertexts can be still used for key recovery.

6.4.1 Manipulations Rules

All the conducted manipulations target the LUTs of the FPGA, i.e., only the combinatorial logic of the design is changed. Let LUT_i be the i^{th} occupied LUT of n available LUTs on the target FPGA. As indicated in Section 2.1.1, a LUT is a 64-bit truth table T that implements a Boolean function $y = f(x)$ with $x \in \{0,1\}^6$ and $y \in \{0,1\}$. Let us denote the j^{th} binary element in this truth table as $T[j]$ with $0 \leq j \leq 63$. As stated before, we suppose that the location of the truth table in the bitstream for each LUT is known to the attacker[1], and hence he can directly modify any single bit of this truth table. With T as the original truth table and its corresponding manipulated truth table $\tilde{T}$, we define three basic operations:

(1) Clear 64-bit LUT configuration: $\tilde{T}[j] = 0, \forall j \in \{0, ..., 63\}$

(2) Set 64-bit LUT configuration: $\tilde{T}[j] = 1, \forall j \in \{0, ..., 63\}$

(3) Invert 64-bit LUT configuration: $\tilde{T}[j] = T[j] \oplus 1, \forall j \in \{0, ..., 63\}$

and accordingly we define three manipulation rules as

- $R_1[i]/R_2[i]/R_3[i]$: Clear/Set/Invert the i^{th} 64-bit LUT configuration

which cover the cases where the entire LUT forms a $6 \mapsto 1$ function, cf. configuration (a) in Fig. 6.1. Besides modifying the entire LUT, we also consider the cases where only the *upper* or *lower* half of the LUT is manipulated. As an example, we can form $\tilde{T}$ by

- $\tilde{T}[j] = 1, \forall j \in \{0, ..., 31\}$

[1] As indicated before, reverse-engineering the LUT encoding eases the attack, but is not required to be able to mount a successful attack for the majority of examined AES cores.

- $\tilde{T}[j] = T[j], \forall j \in \{32, ..., 63\}$

In other words, we only modify the upper half (first 32 bits) of the truth table. The motivation of considering these operations is the cases where a LUT realized a $5 \mapsto 2$ function, cf. configuration (b) in Fig. 6.1. Hence, we define three other rules as

- $R_4[i,h]/R_5[i,h]/R_6[i,h]$: Clear/Set/Invert the h^{th} half of the i^{th} LUT

To cover other two configurations – (c) and (d) Fig. 6.1 – where two or four LUTs are grouped to form larger truth tables, we define the next four rules as

- $R_7[i]/R_8[i]$: Clear/Set all 4 LUTs within the i^{th} slice
- $R_9[i,h]/R_{10}[i,h]$: Set/Clear $(h = 1)$ upper or $(h = 2)$ lower 2 LUTs within the i^{th} slice

Let us define the term Hamming weight of the content of all 4 LUTs within a slice as HW. Accordingly, we define two rules as

- $R_{11}[n], R_{12}[n]$: Clear/Set all 4 LUTs within slices with $HW = n$

with $n \in \{1, ..., 256\}$. In other words, by these rules we clear/set all slices that have a specific Hamming weight. The motivation for these rules is to potentially alter multiple instances of the same Boolean function simultaneously. This may result in manipulating all instances of the same S-Box in the design at once.

Based on our observations, the LUT of the control logic that examine whether a counter (e.g., AES round counter) reaches a certain value (e.g., to issue a *done* signal) have a considerably low or high HW. In other words, the content of such LUT have a high imbalance between the number of '1's and '0's. As an example, a LUT with 4 inputs $c_3c_2c_1c_0$ which checks whether $c_3c_2c_1c_0$=1010 (10_{dec} as the number of AES-128 rounds) has a HW equal to one. This case is covered by the following rules.

- $R_{13}[i,j]$: Invert bit $T[j]$ of the i^{th} LUT, if $1 \leq HW \leq 15$
- $R_{14}[i,j]$: Invert bit $T[j]$ of the i^{th} LUT, if $49 \leq HW \leq 64$

Finally, we cover the case where a LUT is replaced by a random Boolean function:

- $R_{15}[i]$: Set the i^{th} LUT to a random 64-bit value. Repeat this step 10 times.

6.4.2 Key Recovery

By applying any of the above-explained manipulation rules ($R_1 - R_{15}$), we have hit control logic and/or data processing part if a faulty ciphertext is observed. All collected faulty ciphertexts potentially exhibit a sensitive intermediate value allowing for key recovery. Hence, all faulty ciphertexts are further processed depending on which intermediate value hypothesis (we denote to $H_1 - H_{11}$) is tested to derive a set of AES key candidates. All those derived AES key candidates are then automatically tested by a C++ tool with the help of one valid plaintext-ciphertext pair (p, c).

In practice, an adversary can check whether the computation $AES^{-1}_{\text{key candidate}}(c)$ outputs the known plaintext p. Given this is true, the tested AES key candidate is equal to the correct

AES key k. A known plaintext-ciphertext pair (p, c) can for example be obtained from our AES core itself or the attacker might know part(s) of the plaintext, i.e., due to constant header information. In cases where no plaintext-ciphertext pair is available (p unknown, encryption can be invoked, ciphertext c observable), an adversary may also analyze the entropy of decrypted ciphertexts, e.g., if the decryption yields text which typically has a low entropy.

In several cases the key can be extracted without knowing the correct fault-free ciphertext, which is an advantage for an attacker, particularly for cases where he does not have full control over the AES circuit. We hence define the following key hypotheses[2]:

- $H_1[j] : \tilde{c} = rk_j$
- $H_2[j] : \tilde{c} = SB(0^{128}) \oplus rk_j$,

for $j \in \{0, \ldots, 10\}$. The hypothesis H_1 mainly deals with the cases where the state st becomes 0^{128}. Further, H_2 targets the faults which hit the control logic in such a way that the S-box input register always becomes inactive. We give a more detailed information about this concept in Section 6.6.1. If only one round key $rk_1, rk_2, \ldots, rk_{10}$ is extracted, the main key rk_0 can be easily recovered (e.g., see [KK06]). Further, we consider the following hypotheses:

- $H_3[j] : \tilde{c} = c \oplus rk_j$
- $H_4 : \tilde{c} = ka_{10}$
- $H_5 : \tilde{c} = sb_{10}$

To recover the key using H_3 is straightforward as $rk_j = \tilde{c} \oplus c$. Hypotheses H_4 and H_5 check the dependency between the faulty ciphertext and the state in the last AES round. With hypothesis H_4 the last roundkey rk_{10} can be recovered: $rk_{10} = SR\left(SB\left(\tilde{c}\right)\right) \oplus c$. A similar approach can be followed for hypothesis H_5. The next set of hypotheses are defined as:

- $H_6[j] : \tilde{c} = p \oplus rk_j$
- $H_7 : \tilde{c} = sr_1$
- $H_8 : \tilde{c} = sb_1$
- $H_9 : \tilde{c} = mc_1$
- $H_{10} : \tilde{c} = AES_{k'}(p)$
- $H_{11} : \tilde{c} = SR(SB(p)) \oplus rk_{10}$

where k' is defined as $rk'_0 = rk_0$, and $rk'_{j \in \{1,\ldots,10\}} = 0^{128}$. Using H_6 is straightforward. H_7, H_8, and H_9 can also be checked by applying the corresponding inverse functions, e.g., $rk_0 = SB^{-1}\left(SR^{-1}\left(MC^{-1}\left(\tilde{c}\right)\right)\right) \oplus p$ for H_9.

To examine H_{10}, we need to apply a specific form of the decryption function as $rk_0 = AES^{-1}_{k''}(\tilde{c}) \oplus p$ with $\forall j \in \{0, \ldots, 10\}$, $rk''_j = 0^{128}$. Hypothesis H_{11} can be seen as an AES encryption where only the steps of the last AES round are executed on the plaintext p. In this case, the last round key can be trivially computed with $rk_{10} = \tilde{c} \oplus SR(SB(p))$.

[2]Note that we stick with the definitions of AES states as introduced in Section 2.3

In each of the above hypotheses only one faulty ciphertext is used. One can also define hypotheses that used two faulty ciphertexts generated by two different bitstream manipulations for a certain plaintext. As an example, knowing $\tilde{c}_1 = ka_2$ and $\tilde{c}_2 = ka_3$ would lead to a full key recovery. In this scenario, the adversary needs to try all possible combinations between different faulty ciphertexts, and the computation complexity of the attack increases quadratically. Since it is – to some extent – in contradiction with our goal (i.e., limiting the number of rules as well as the hypotheses for key recovery), we omit the corresponding results although we have observed such successful cases in six designs in our experiments.

6.5 Experimental Setup and Results

For evaluating the effectiveness of BiFI, we use exactly the same 16 AES cores $(D_0, D_1, \ldots, D_{15})$ as introduced in Section 4.4.5. The interested reader is referred to this section as it explains which implementations are round-based, unrolled, and so forth. In order to perform the attacks, we developed a program to automatically apply all the rules R_1-R_{15} given in Section 6.4.1 one after each other. To this end, our tool first queries the original design with a particular plaintext and captures the fault-free ciphertext (p, c). Afterwards, for each rule our tool manipulates the bitstream accordingly, configures the FPGA, queries the design with the same particular plaintext p and collects the corresponding faulty ciphertext $\tilde{c}$. Another program (which we have developed as well) examines all the hypotheses H_1-H_{11} described in Section 6.4.2 by analyzing each faulty ciphertext[3].

[3]Due to larger tables leading to visualization issues, the next paragraph continues in the next page.

6.5.1 Results without Enabled Bitstream Encryption Scheme

Our setup is again based on a Spartan 6 (XC6SLX16) FPGA, where the JTAG port is used for configuration, cf. Fig. 4.2. The entire bitstream manipulation, configuration, query and collection of the faulty ciphertext takes around 3.3 seconds. We should emphasize that by manipulating the bitstream, the Cyclic-Redundancy-Check (CRC) checksum should be correct. Alternatively, the bitstream can be modified in such a way that the CRC check is simply disabled. As stated before, we conducted our attack on 16 different designs. The corresponding results are depicted in Table 6.1 indicating which manipulation rule R_1-R_{15} on which AES design D_0-D_{15} led to an exploitable faulty ciphertext.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	d.att.
Changing all 64 bits of a LUT in the bitstream																	
$R_1[i]$: Clear LUT	1	3	.	6	.	.	.	1	1	3	1	.	.	.	.	2	8
$R_2[i]$: Set LUT	2	4	3	1	.	.	.	2	2	2	.	3	.	.	.	1	9
$R_3[i]$: Invert LUT	1	9	2	2	1	.	.	.	1	2	2	2	.	.	.	2	10
Changing only 32 bits of a LUT in the bitstream																	
$R_4[i,h]$: Clear half LUT	.	4	.	7	.	.	1	4	3	3	.	.	1	1	1	2	10
$R_5[i,h]$: Set half LUT	1	3	2	3	.	.	.	3	3	2	.	3	1	1	1	1	12
$R_6[i,h]$: Invert half LUT	.	7	2	5	1	.	.	1	1	3	1	2	2	2	3	4	13
Changing two or four 64-bit LUTs in the bitstream																	
$R_7[i]$: Clear slice	.	1	.	4	.	.	.	1	.	3	1	.	.	.	.	1	5
$R_8[i]$: Set slice	1	1	1	1	.	.	.	1	.	1	.	1	.	.	.	1	3
$R_9[i,h]$: Set 2 LUTs	1	2	.	5	.	.	.	1	2	3	1	.	.	.	.	1	8
$R_{10}[i,h]$: Clear 2 LUTs	1	2	1	1	.	.	.	2	2	1	.	2	.	.	.	1	9
Clearing only LUTs with a specific slices' HW in the bitstream																	
$R_{11}[n]$: Clear slice if HW=n	.	.	.	1	.	.	.	1	.	.	.	.	.	.	.	.	2
$R_{12}[n]$: Set slice if HW=n	.	1	.	1	.	.	.	1	.	.	.	.	.	.	.	.	2
Inverting single LUT bits (64 times) for a specific HW in the bitstream																	
$R_{13}[i,j]$: Invert bits if HW ≤ 15	.	2	2	3	.	.	2	5	5	4	.	.	1	1	1	.	7
$R_{14}[i,j]$: Invert bits if HW ≥ 49	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
Configuring random Boolean functions (10 times) in the bitstream																	
$R_{15}[i]$: Set LUT randomly	7	31	16	16	1	.	.	3	7	13	1	12	5	2	5	3	14
Statistics																	
$\sum$ exploitable faulty ciphertexts	15	71	29	56	3	0	3	26	27	40	7	25	10	7	11	19	
Number of vulnerable LUTs (R_1-R_{14})	5	20	4	18	1	0	2	9	7	13	3	5	3	3	4	6	
Measurement time R_1-R_{12} (hours)	20	26	18	74	22	64	26	9	9	42	62	12	26	246	28	12	
Measurement time R_{13}-R_{14} (hours)	110	12	11	38	16	53	18	5	5	26	34	8	18	61	21	7	

Table 6.1: Overview of the experiments with regard to the different modification rules. Each entry in the table represents the number of times for which applying the manipulation rule R_i lead to an exploitable fault for design D_j. The last column "d.att." (designs attacked) shows the number of different designs D_j that could be attacked with the corresponding rule. In the experiment, several modification rules resulted in an exploitable faulty ciphertext when applied to the same LUT. The number of LUTs that lead to at least one exploitable faulty ciphertext for at least one of the manipulation rules $R_1 - R_{14}$ is shown in row "Number of vulnerable LUTs" as a reference

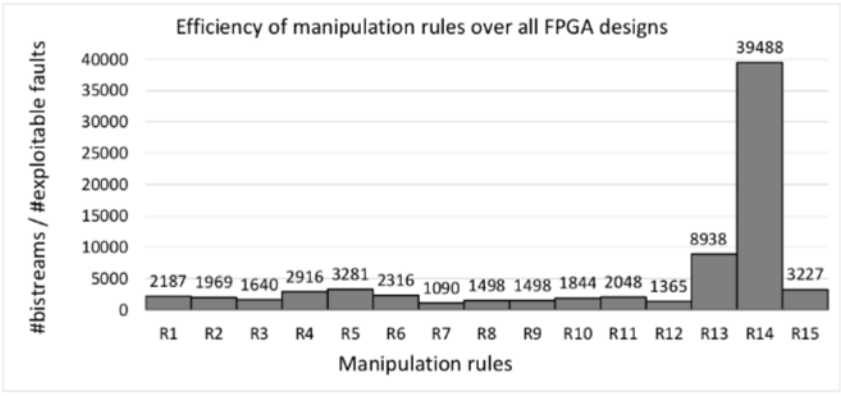
Similarly, Table 6.2 shows for each AES design which hypotheses H_1-H_{11} led to successful key recovery. For all designs, except the unrolled pipeline one (D_5), at least one hypothesis could make use of the faulty ciphertexts generated by the manipulation rules to recover the key. In Section 6.6.1, we give a detailed analysis on the exploitable faults. In short, many exploitable faults hit the control logic (i.e., the AES state machine is modified). We predict this to be the

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	d.att.
$H_1 : rk_0$	.	.	.	.	.	.	.	.	.	.	.	12	.	.	.	.	1
$H_1 : rk_{10}$	.	.	.	33	.	.	.	2	4	15	.	3	.	.	.	5	6
$H_2 : S(0^{128}) \oplus rk_0$	.	.	.	.	.	.	.	.	.	.	.	.	4	2	4	.	3
$^{\dagger}H_2 : S(0^{128}) \oplus rk_1$	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	1
$^{\dagger}H_2 : S(0^{128}) \oplus rk_2$	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	1
$H_2 : S(0^{128}) \oplus rk_{10}$	.	6	.	.	.	.	.	2	2	.	.	.	.	.	.	.	3
$H_3 : c \oplus rk_{10}$	.	.	.	.	.	.	.	2	2	.	.	.	.	.	.	.	2
$H_4 : ka_{10}$	.	.	.	.	3	.	2	.	.	.	.	.	.	.	1	1	4
$H_6 : p \oplus rk_0$	4	.	19	10	.	.	.	.	.	14	2	10	1	3	4	7	10
$^{\dagger}H_6 : p \oplus rk_2$	1	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	2
$H_6 : p \oplus rk_4$	6	.	.	1	.	.	.	.	.	1	.	.	.	.	.	.	3
$^{\dagger}H_6 : p \oplus rk_5$	.	.	.	3	.	.	.	.	.	1	.	.	.	.	.	.	2
$^{\ddagger}H_6 : p \oplus rk_6$	.	.	.	1	.	.	.	.	.	2	.	.	.	.	.	.	2
$H_6 : p \oplus rk_{10}$	4	.	.	8	.	.	.	.	.	5	.	.	.	.	.	.	3
$^{\ddagger}H_7 : sr_1$	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
$^{\dagger}H_9 : mc_1$	.	.	.	.	.	.	.	.	.	.	.	.	3	.	.	.	1
$H_{10} : AES_k(p), rk_j = 0$	.	63	10	.	.	.	1	17	17	.	4	.	2	2	2	4	10
$H_{11} : SR(SB(p)) \oplus rk_{10}$	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	1
Statistics																	
Collected responses	20333	26455	18044	75296	22442	65051	26522	8914	9080	42620	62991	12261	26226	249813	28188	12077	
Unique faulty responses	6411	7412	6022	28512	8955	3772	15887	2587	2675	15760	10137	5171	15484	45971	17246	5317	

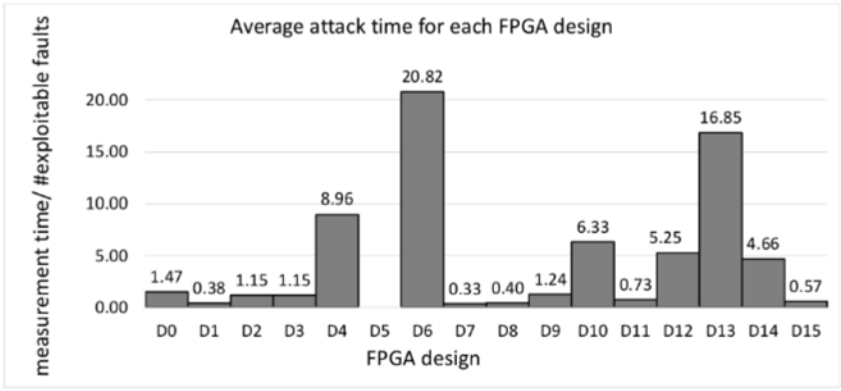
Table 6.2: Overview of the experiments with regard to the different hypotheses. Each entry in the table represents the number of times a hypotheses H_i for each design D_j could be used to recover the key from faulty ciphertexts being the result of applying the modification rules R_1-R_{15}. Some hypotheses (marked by †) succeed only for R_{15} while some other hypotheses marked with ‡ could make use of only R_1-R_{14}. The last column "d.att." shows the number of different designs that could be successfully attacked by the corresponding hypothesis. The last two rows summarize the number of collected responses (which are equivalent to the number of times a bitstream manipulation was conducted) and the number of observed unique faulty ciphertexts

reason why the design D_5 cannot be successfully attacked, since the unrolled pipeline design makes use of simple state machines.

It can be seen from Table 6.1 that many manipulation rules lead to exploitable faulty ciphertexts. It is also worth mentioning that each manipulation rule was successful for at least one design. To compare the efficiency of the manipulation rules, we computed a ratio between the number of performed bitstream manipulations and the number of exploitable faults, cf. Fig 6.2a. Note that a lower ratio means that the underlying manipulation rule is more efficient, since the average number of manipulations required for an attack becomes smaller. As stated before, each manipulation rule led to at least one exploitable faulty ciphertext. However, some of them are more efficient than other ones. The most efficient one is R_7 (i.e., clear an entire slice), and R_{13} and R_{14} are among the worst manipulation rules. On the other hand, we should emphasize that in R_{13} and R_{14} each bit of the target LUT is independently manipulated. Hence, the number of manipulated bitstreams in these two rules is considerably higher compared to the other rules. We would like to stress that on average every 3227 random manipulations (R_{15}) led to an exploitable faulty ciphertext (cf. Fig 6.2a) indicating that it is also a solid strategy. Nevertheless, manipulations rules R_1-R_{12} are a bit more efficient than random manipulations with an average 1971 manipulations required to observe an exploitable faulty ciphertext. As stated before, the entire manipulation, configuration, and query takes around 3.3 seconds. Hence, in average $1971 \times 3.3 = 1.8$ hours of bitstream manipulations are needed per exploitable faulty ciphertext for rules R_1-R_{12}. However, this time varies significantly depending on the targeted



(a)



(b)

Figure 6.2: a) The ratio the number of performed bitstream manipulations over the number of exploitable faults. b) The average attack time (in hours) until an exploitable faulty ciphertext is obtained for each of the targeted design (using modification rules R_1-R_{12})

design. Figure 6.2b shows the average time of bitstream manipulations (over manipulation rules R_1-R_{12}) needed for an exploitable fault for each of the 16 AES designs.

6.5.2 Experimental Setup with Enabled Bitstream Encryption Scheme

To prevent reverse-engineering and IP-theft, Xilinx FPGAs are equipped with bitstream encryption. We also investigated to what extent the above-presented attack can be efficient when the underlying bitstream is encrypted. To this end, we take a closer look at this feature integrated in several Xilinx FPGAs. When this protection mechanism is enabled in the vendor's software, the user can chose a 256-bit AES key k as well as a 128-bit initial vector IV. Excluding the header, the main body of the bitstream is encrypted using $AES256_k(\cdot)$ in Cipher Block Chaining (CBC) mode. Thus, the corresponding bitstream data of size m is divided into n 16-byte plaintext blocks with $p_{i\in\{1,\ldots,\frac{m}{16}\}}$, and sequentially encrypted as

$$c_i = AES256_k(p_i \oplus c_{i-1}), \text{ for } i > 0 \text{ and } c_0 = IV. \tag{6.1}$$

Analogously, the decryption is performed by a dedicated hardware module on the FPGA as

$$p_i = AES256_k^{-1}(c_i) \oplus c_{i-1}, \text{ for } i > 0 \text{ and } c_0 = IV. \tag{6.2}$$

The key needs to be programmed once into the target FPGA either in its volatile (BBRAM) or non-volatile memory (eFUSE). At every power-up, if the FPGA receives an encrypted bitstream, it runs the corresponding CBC decryption and configures its internal elements accordingly. In relatively old Xilinx FPGAs, i.e., Virtex 4, Virtex 5, and Spartan 6 families, the integrity of the encrypted bitstream is examined by a 22-bit CRC. In Virtex 4 and Virtex 5 FPGAs, the CRC checksum is not included in the encrypted part, and the corresponding command to enable the CRC is involved in the (*unencrypted*) header of the bitstream. Hence, the attacker can easily disable such an integrity check by patching the encrypted bitstream. However, in case of Spartan 6 the encrypted part of the bitstream contains the CRC as well. Therefore, any bitstream manipulation most likely leads to CRC failure (see Section 6.5.4 for more information). Further, in more recent Xilinx products, e.g., Virtex 6 and the entire 7-series, the integrity (as well as authenticity) is additionally examined by an Hash-based Message Authentication Code (HMAC), which also disables any bitstream manipulation.

	D_0	D_2	D_3	D_4	D_5	D_6	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	d.att.
$R_3[i]$	12	7	11	1	.	2	6	1	8	6	4	5	6	12
$R_6[i,h]$	25	11	14	2	.	2	8	2	11	8	7	13	13	12
$\sum$ exploitable faulty ciphertexts	**37**	**18**	**25**	**3**	**.**	**4**	**14**	**3**	**19**	**14**	**11**	**18**	**19**	**12**

Table 6.3: Overview of the BiFI attack on encrypted bitstreams. Two modification rules R_3 and R_6 were tested and each table entry represents the number of exploitable faulty ciphertexts

Therefore, in order to investigate the efficiency of our BiFI attack when the bitstream is encrypted, we conducted our experiments on a Virtex 5 FPGA. Obviously it is desirable to control the effect of manipulation of the bitstream, e.g., to avoid the propagation of the changes. If l bits of ciphertext block c_{i+1} – in CBC mode – are toggled, its effect on the plaintext block p_{i+1} is not predictable. However, it also directly changes the corresponding l bits of the next plaintext block p_{i+2}, and interestingly such a manipulation does not propagate through the entire bitstream. This concept is illustrated in Figure 6.3.

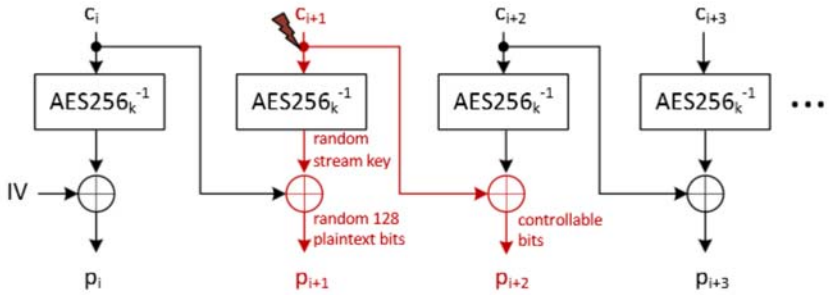


Figure 6.3: The impact of faulting one ciphertext block in case of CBC decryption

	D_0	D_2	D_3	D_4	D_5	D_6	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	d.att.
$H_1 : rk_0$	.	.	.	.	.	.	.	.	7	.	.	.	5	2
$H_1 : rk_{10}$	.	.	16	.	.	.	7	.	.	.	.	.	.	2
$H_2 : S(0^{128}) \oplus rk_0$	.	.	.	.	.	.	.	.	.	3	4	13	.	3
$H_2 : S(0^{128}) \oplus rk_1$	2	.	.	.	.	.	.	.	.	.	.	.	.	1
$H_3 : c \oplus rk_{10}$	.	.	.	.	.	.	.	1	.	.	.	.	.	1
$H_4 : ka_{10}$	.	.	.	3	.	.	.	.	1	.	.	.	.	2
$H_6 : p \oplus rk_0$	17	12	.	.	.	.	.	.	11	7	6	5	13	7
$H_6 : p \oplus rk_2$	9	.	1	.	.	.	1	.	.	.	.	.	.	3
$H_6 : p \oplus rk_3$	.	.	2	.	.	.	.	.	.	.	.	.	.	1
$H_6 : p \oplus rk_5$	4	.	5	.	.	.	.	.	.	.	.	.	.	2
$H_6 : p \oplus rk_6$	.	.	1	.	.	.	.	.	.	.	.	.	.	1
$H_6 : p \oplus rk_8$	2	.	.	.	.	.	.	.	.	.	.	.	.	1
$H_6 : p \oplus rk_{10}$	3	.	.	.	.	.	6	.	.	.	.	.	.	2
$H_{10} : AES_k(p), rk_j = 0^{128}$	.	6	.	.	.	4	.	1	.	2	1	.	1	6
$H_{11} : SR(SB(p)) \oplus rk1$	.	.	.	.	.	.	.	.	.	2	.	.	.	1
Statistics														
Collected responses	86400	86400	86400	86400	86400	86400	86400	86400	86400	86400	86400	86400	86400	
Unique faulty responses	4655	7659	16959	12118	9432	12124	8089	19638	4677	11706	12469	21945	5269	

Table 6.4: Overview of the exploitable faulty ciphertexts of the different hypotheses for 13 different designs with enabled bitstream encryption

6.5.3 Setup and Results with Enabled Bitstream Encryption

On our Virtex 5 (XC5VLX50) setup – with bitstream encryption enabled – we examined 13 AES designs (D_0, D_2-D_6, D_9-D_{15}) out of the previously expressed 16 designs[4]. If we ignore the unpredictable changes on plaintext p_{i+1}, toggles on the bits of ciphertext c_{i+1} lead to the same toggles on plaintext p_{i+2} (see Fig. 6.3). Therefore, we can only apply the rules R_3 and R_6 which toggle the entire LUT or a half of a LUT. Further, the unpredictable manipulation of plaintext p_{i+1} may also hit a utilized LUT. In short, manipulation of ciphertext c_{i+1} based on R_3 and R_6 indirectly applies the rule R_{15} to other LUTs as well. More importantly, the unpredictable changes on plaintext p_{i+1} can lead to misconfiguration of switch-boxes, and hence short circuits[5]. In such scenarios, the FPGA cannot be configured, and needs to be restarted.

As an attacker, we know which parts of the bitstream (either unencrypted or encrypted) belong to LUTs' configuration. However, in case of the encrypted bitstream we cannot explore which LUTs are utilized. Therefore, the rules R_3 and R_6 need to be applied to all available LUTs (28,800 in our Virtex 5 (XC5VLX50) FPGA). Hence, the attack takes longer compared to targeting an unencrypted bitstream. Further, since the Virtex 5 FPGA equipped in our setup is larger (hence, uses a larger bitstream) than the Spartan 6, each configuration takes around 6.6 seconds, i.e., two times slower than the previously-shown experiments, which in sum turns into 6.8 days to apply both rules (R_3 and R_6) on all available LUTs. Table 6.3 shows the corresponding result of the attacks on the targeted 13 AES designs.

Similar to the unencrypted case, only the unrolled pipeline design D_5 cannot be successfully attacked. Notably, an average of 11.5 hours is needed for a successful attack over all designs. For further details, we refer to Table 6.4 which shows all successful hypotheses leading to the exposure of the key.

[4]Due to their e.g., hard-coded macros not compatible with Virtex 5, the designs D_1, D_7, and D_8 could not be synthesized on this FPGA.

[5]Based on our observations, the currently-available FPGAs in the market are protected against such short circuits, preventing them being destroyed.

We also conducted another attack in which we considered the encrypted part of the bitstream as a complete black-box, i.e., without directly targeting the LUTs. In order to minimize the effect on plaintext block p_{i+2}, we only toggled the most significant bit of one ciphertext block. In other words, we tried to apply only R_{15} on plaintext block p_{i+1}. We conducted this bitstream manipulation to each encrypted block once and could successfully attack 11 out 13 designs. Attacking one AES core takes approximately 8 days, and led again to various exploitable faulty ciphertexts[6]. To conclude, knowing the exact locations of the LUT contents in the (encrypted) bitstream is not necessarily essential. In the next section, we briefly explain how an attacker can figure out whether the bitstream encryption scheme of a particular Xilinx FPGA is vulnerable to the BiFI attack.

6.5.4 Testing the Bitstream Encryption Vulnerability of Xilinx FPGAs

In an initial step, we used the Xilinx tool to generate an encrypted bitstream for a Virtex 5 FPGA with enabled CRC-check (*bs_enc_crc_on*). For a quick test, we randomly modified one encrypted block in the bitstream and tried to configure the corresponding Virtex 5 FPGA. As expected, the FPGA refused to configure the manipulated bitstream. In the next step, we generated another encrypted bitstream for the same FPGA design and using the same key k and the same IV, but with disabled CRC-check (*bs_enc_crc_off*).

Disabling the CRC-check means that two 22-bit CRC values toggle to a fixed default sequence (defined by Xilinx) and that one bit toggles in the bitstream header which is responsible for disabling/enabling the CRC check.

The comparison of both encrypted bitstreams *bs_enc_crc_on* versus *bs_enc_crc_off* revealed that only the unencrypted parts of the file are different, i.e., the encrypted blocks are identical, cf. Fig. 6.4. Therefore, we concluded that the encrypted parts of the bitstream does not contain the checksum. Otherwise, due to the default CRC sequence at least one encrypted block would be different.

To evaluate our findings on Virtex 5, we *i*) first observed all bit toggles due to *bs_enc_crc_on* versus *bs_enc_crc_off*, then *ii*) applied the same bit toggles on the target encrypted bitstream (with enabled CRC-check), *iii*) applied various manipulation rules (Section 6.5.3), and finally *iv*) configured the manipulated bitstream into the FPGA device. It turned out that the Virtex 5 FPGA accepted the manipulated bitstream, and hence, there is no appropriate integrity check leading to feasible BiFI attacks.

We repeated the same experiment on a Spartan 6 (SLX75) FPGA and noticed that one (out of two) CRC sequences is part of the last encrypted block C_N, cf. Fig. 6.5. Therefore, for these kinds of bitstreams the integrity is ensured providing protection against BiFI attacks unless side-channel attacks are used to recover the underlying bitstream encryption key.

One further observation we made is that the Spartan 6 FPGAs in general denies any encrypted bitstream with disabled CRC-check. This is not the case for Virtex 5.

6.5.5 Discussion on Altera Bitstream Encryption Scheme

The underlying mode of encryption and the employed integrity check determine whether a BiFI attack can be mounted on an encrypted bitstream. While we used Xilinx FPGAs in our practical

[6]It is noteworthy that in this experiment several times the FPGA "crashed" i.e. could not be programmed until it was restarted manually. This never happened for the manipulation rules that only targeted LUTs.

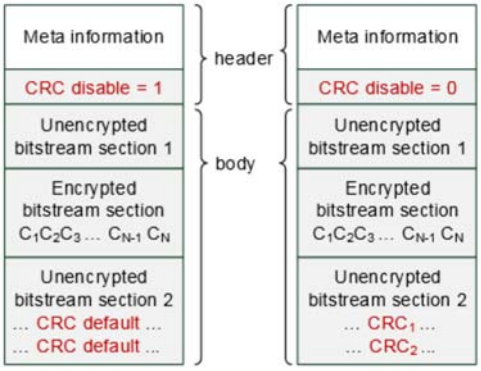


Figure 6.4: Virtex 5 (VLX50) bitstreams, left: encryption enabled and CRC off (*bs_enc_crc_off*), right: encryption enabled and CRC on (*bs_enc_crc_on*)

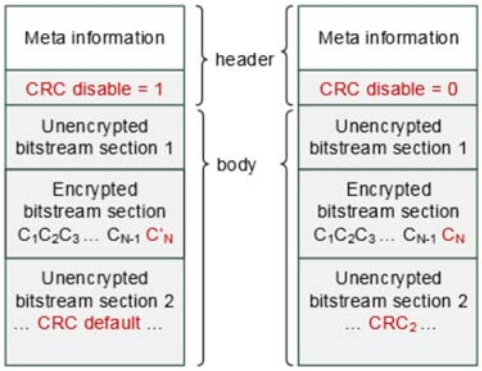


Figure 6.5: Spartan 6 (SLX75) bitstreams, left: encryption enabled and CRC off (*bs_enc_crc_off*), right: encryption enabled and CRC on (*bs_enc_crc_on*)

experiments, below we discuss about the feasibility of our attack on Altera's bitstreams with enabled encryption. In the recent families of Altera FPGAs (similar to that of Xilinx FPGAs), an HMAC authentication process is integrated. Hence, such devices are not susceptible to our attacks (unless the bitstream encryption and authentication is circumvented e.g., by a side-channel analysis attack [MOPS13]).

However, the older Stratix II and Stratix III families use AES in counter mode and a simple CRC for integrity check. The underlying scheme in Stratix II and Stratix III are similar except *i*) AES-128 replaced by AES-256 in the later one, and *ii*) arithmetic counter (of the counter mode) replaced by a sophisticated pseudo-random-number generator (for more information we refer to Section 3.3).

Both devices generate a stream key which is XORed with the plaintext blocks to form the encrypted bitstream. The decryption process (performed on the FPGA) follows the same concept as depicted in Figure 6.6. In this case, if an adversary manipulates the bitstream by toggling l bits of the ciphertext block c_{i+1}, the corresponding l bits of the same plaintext block p_{i+1} toggle, and the changes propagate neither to other bits of the same block nor subsequent blocks. Therefore, compared to the encryption feature of Xilinx FPGAs, the attacker has more control over the manipulations, hence higher efficiency of BiFI attacks. More importantly, since the CRC is

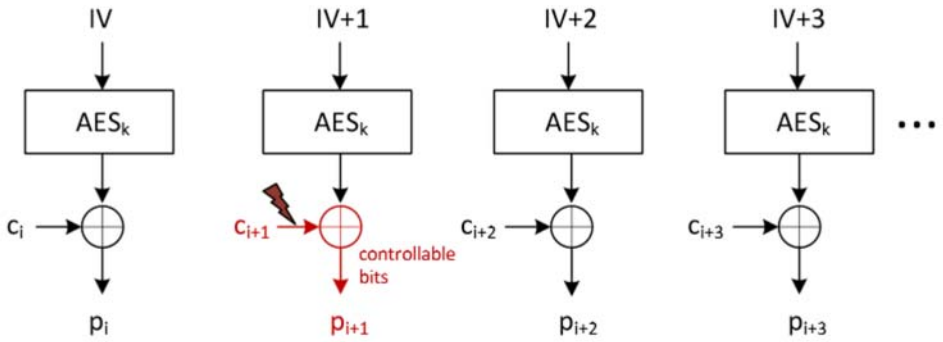


Figure 6.6: The decryption in counter mode as it is used for bitstream encryption in Stratix II FPGAs. Toggling a single ciphertext bit results in a predictable toggle of a plaintext bit

linear, it can be trivially predicted how the CRC checksum should change by any toggle made on a ciphertext block (similarly on a plaintext block). More precisely, the attacker can toggle any arbitrary bit(s) of the encrypted bitstream and correspondingly modify the CRC checksum. Therefore, the counter mode makes BiFI attacks considerably easier if a CRC integrity check is employed. Although we have not yet practically examined it, we are confident that our attack can be easily and successfully applied on Altera Stratix II and Stratix III FPGAs.

6.6 Analysis

So far we have only expressed the manipulation rules as well as the hypotheses which we used to conduct successful attacks. In the following, we give more details on a few cases, where observed faulty ciphertexts led to key recovery. By doing so, we disclose exploitable hardware structures that might be helpful for the security engineer. He can for example design hardware layouts that avoid the presented easy-to-exploit hardware structures.

6.6.1 Evaluation of Observed Faults

For a couple of exploitable faulty ciphertexts, we investigated at the netlist level, what exactly caused this faulty behavior. To do so, we used the FPGA Editor (provided by the Xilinx ISE toolchain) to analyze the LUTs whose modification in the bitstream led to a key exposure. Due to the large number of faults, we only cover a small subset of exploitable faults that are representative for a class of similar possible faults. Hence, the provided analysis is not a comprehensive study and only aims at providing the reader with an intuition of what can happen during the attack. It is noteworthy that the presented figures are a simplified high-level representation of the usually more complex hardware configuration.

Hitting the Control Logic

A successful key recovery on the round-based design D_0 was due to manipulating a LUT whose two output bits are used to control a 3-to-1 multiplexer controlled by two bits (m_1, m_2). cf. Figure 6.7. A part of this design performs the following operations

- CLK cycle 1: $state = p \oplus rk_0$,
- CLK cycles 2-10: $state = mc_j \oplus rk_j, j \in \{1, ..., 9\}$,
- CLK cycle 11: $c = sr_{10} \oplus rk_{10}$.

Depending on the clock cycle (i.e., the round counter) the targeted LUT (which controls the multiplexer) switches between the plaintext p $(0, 0)$, the state after MixColumns mc_j $(0, 1)$, and the final state after the final ShiftRows sr_{10} $(1, 0)$, cf. the upper part of Figure 6.7. The 128-bit multiplexer output is XORed to the corresponding 128-bit round key.

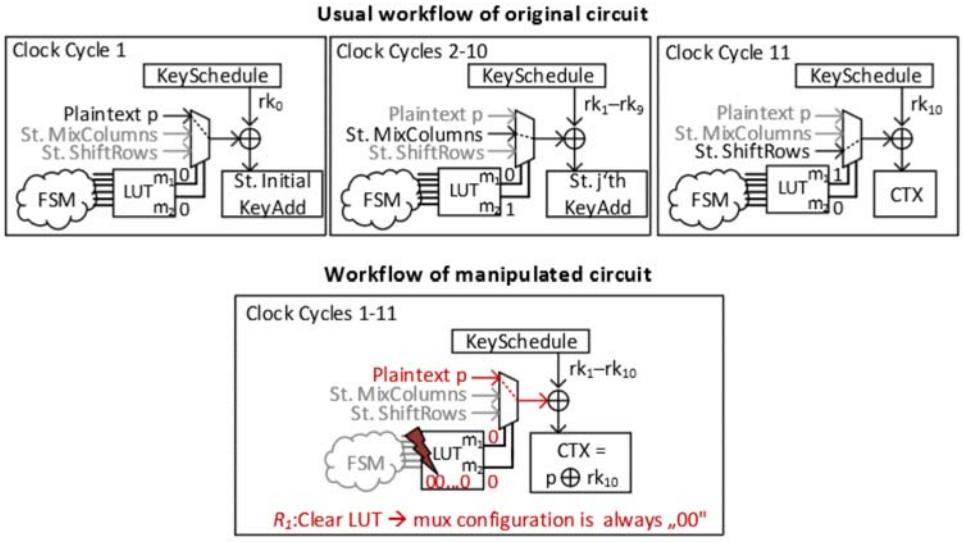


Figure 6.7: Manipulation rule R_1 (*Clear LUT*), round-based design D_0, consequence: plaintext p (instead of sr_{10}) is XORed to the last AES round key rk_{10}

By applying the rule R_1 (*Clear LUT*), the LUT outputs are permanently fixed to $(0, 0)$, and hence, the multiplexer always outputs p regardless of the clock cycle, cf. the lower part of Figure 6.7. More precisely, by such a bitstream manipulation, the following operations are performed

(1) CLK cycle 1: $state = p \oplus rk_0$,

(2) CLK cycles 2-10: $state = p \oplus rk_j, j \in \{1, .., 9\}$,

(3) CLK cycle 11: $\tilde{c} = p \oplus rk_{10}$.

The circuit outputs $\tilde{c} = p \oplus rk_{10}$ instead of $c = sr_{10} \oplus rk_{10}$, which is the motivation to test hypothesis H_6 for key recovery.

Update Mechanism of Flip-flops - Never Update 128-bit Key Register

We noticed a manipulated LUT whose output controls the update of a couple of flip-flops. As an example, a LUT might control the *CE* signal (*Clock Enable*) of a 128-bit state or key register. The flip-flops' content is updated on e.g., the rising edge of the clock, only if the CE signal is '1'. The manipulation rule R_1 (*Clear LUT*) turns such a LUT into constant '0', hence always disabling the update. We have observed many cases where the round key registers are never updated. Such a LUT modification can hence turn flip-flops into read-only elements, while they only output their initial value[7].

An example is depicted in Figure 6.8. It shows that the key schedule output is not stored by the flip-flops and they output always '0'. Therefore, such a manipulation rule can affect all round keys such that $rk_{j\in\{1,...,10\}} = 0^{128}$.

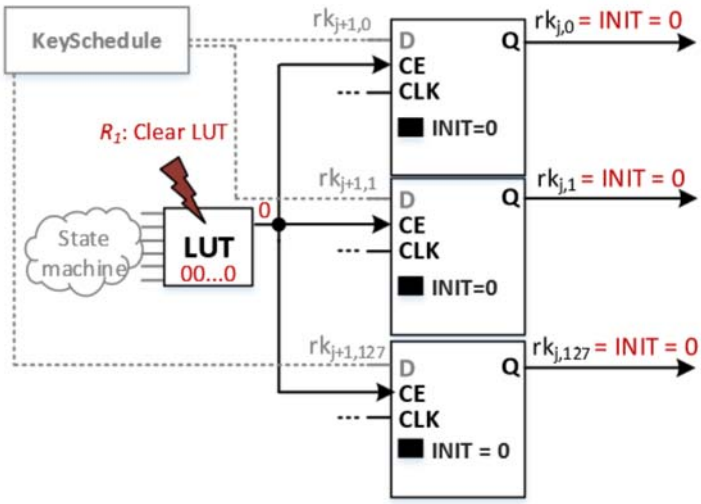


Figure 6.8: Manipulation rule R_1 (*Clear LUT*), round-based design D_{15}, group of flip-flops forming a 128-bit round key register ($rk_{j,0}$ - $rk_{j,127}$) used for XOR with the current AES state. Due to the manipulation, none of the round key flip-flops are updated. Instead, they always remain '0'

Based on our observations – depending on the design architecture – the initial KeyAdd operation is still conducted by the correct round key rk_0. More precisely, the manipulated AES core performs the following operations:

(1) $state = p \oplus rk_0$,

(2) $state = MC(SB(SR(state))) \oplus 0^{128}, j \in \{1, ..., 9\}$,

(3) $\tilde{c} = SB(SR(state)) \oplus 0^{128}$.

Therefore, the hypothesis H_{10} (defined in Section 6.4.2) can examine whether a manipulation hit the corresponding LUT, and consequently recover the key.

[7] In Xilinx FPGAs the initial value of every flip-flop can be defined. Without any definition, the default value (usually '0') is taken by the synthesizer.

Update Mechanism of Flip-flops - Never Update S-box Input Register

We have observed a similar scenario for a 32-bit AES state. As an example, we focus on the AES design D_1 with 32-bit datapath, where updating the complete 128-bit AES states requires at least four clock cycles. Similar to the prior case, the *CE* pin of the registers are controlled by a LUT. If such a LUT is manipulated by R_1 (*Clear LUT*), the register will have always its initial value, cf. Figure 6.9.

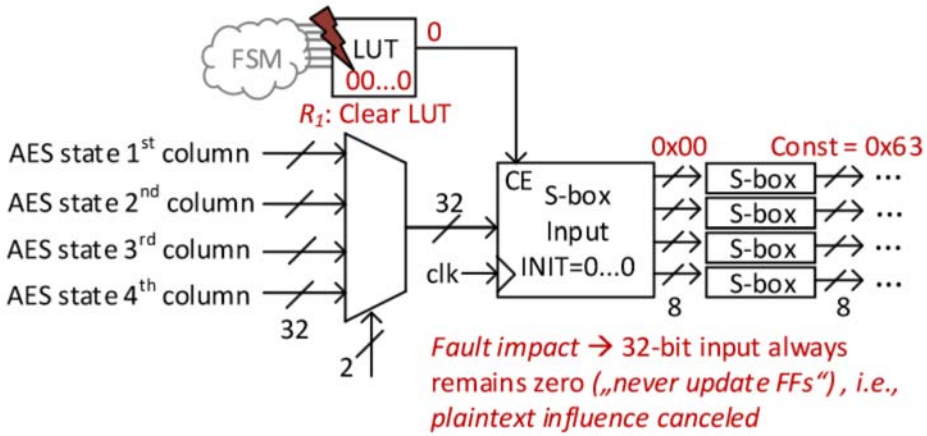


Figure 6.9: Manipulation rule R_1 (*Clear LUT*), word-based design D_1, due to the bitstream manipulation the S-box inputs remain zero, this results into the leakage of the last round key rk_{10}

It is noteworthy that in this design, the four aforementioned AES S-boxes are used only for the SubBytes operation, i.e., the key schedule circuitry employs separate S-box instances. Even though all the main AES operations (ShiftRows, AddRoundkey, MixColumns, etc.) operate correctly, the round output is not stored into the S-box input registers. Therefore, the manipulated design outputs $\tilde{c} = SB(SR(0^{128})) \oplus rk_{10} = SB(0^{128}) \oplus rk_{10}$, which trivially leads to recovering the last round key rk_{10}, i.e., hypothesis H_2.

These results indeed indicate that preventing registers from updating their state can lead to various exploitable ciphertexts. However, as expressed above, the feasibility of the key recovery depends on the initial value of the registers. Hence, if the registers (in HDL representation) are initialized by arbitrary values, the aforementioned attacks would need to guess the initial value instead of 0^{128}, which obviously complicates the attacks.

Update Mechanism of Flip-flops - Invert Updating

We also observed that the manipulation rule R_3 (*Invert LUT*) resulted in inverting a signal that controls when the output register (128-bit flip-flops so-called *out_reg*) should be updated by trivially being connected to its *CE* pin. In the underlying AES core, the *out_reg* is not updated during the encryption except when the encryption is terminated thereby storing the ciphertext. To this end, the corresponding LUT output (so-called *update_out_reg_enable*) is '1' at only one clock cycle, cf. upper part of Figure 6.10.

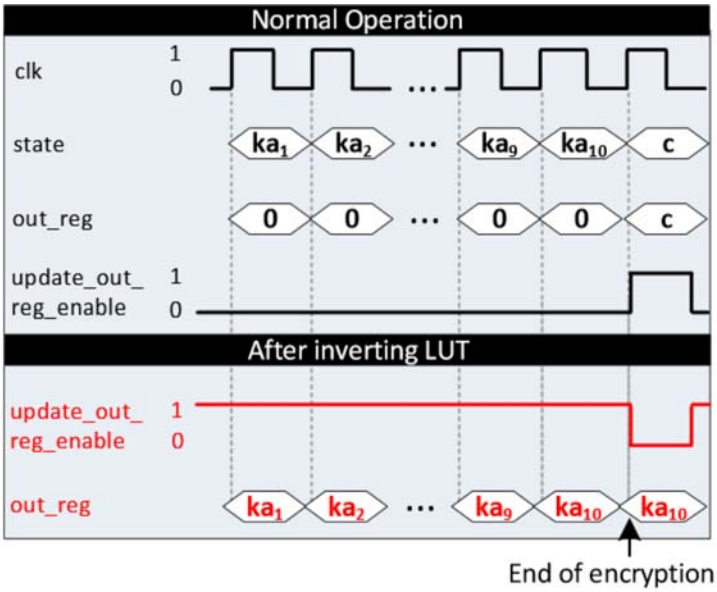


Figure 6.10: Manipulation rule R_3 (*Invert LUT*), round-based design D_4, Due to the LUT inversion of the *update_out_reg_enable* control signal, the relevant output register *out_reg* is updated at the wrong clock cycles, i.e., the modified AES core fails to copy the correct ciphertext c and writes the leaking state ka_{10}

By the aforementioned manipulation, the LUT output *update_out_reg_enable* is inverted, and the output register *out_reg* stores the cipher state after KeyAdd operation at all cipher rounds except the last one. Consequently, the final state after the last KeyAdd operation (ka_{10}) is given as faulty output instead of the correct ciphertext c. By examining hypothesis H_4 it can be tested whether such a LUT is hit which directly leads to key recovery.

Hitting the State Machine or Round Counter

One of the most common observed exploitable faults was due to manipulation of a LUT that (partially) processes the state machine or counter signals. As a consequence, in many cases the manipulated AES core finished its operations earlier than the fault-free case. Such a manipulation leads to the exposure of various intermediate values, e.g., the j^{th} Keyadd operation (ka_j). A representative example is illustrated in Figure 6.11. We have observed that the manipulated LUT realizes the following function:

$$final\ round\ =\ \overline{rnd_0}\ \cdot\ rnd_1\ \cdot\ \overline{rnd_2}\ \cdot\ rnd_3 \tag{6.3}$$

Such a LUT controls the AES core to stop the operations when the round counter reaches $(rnd_3, rnd_2, rnd_1, rnd_0) = (1, 0, 1, 0) = 10_{dec}$, obviously corresponding to 10 cipher rounds of AES-128. Inverting certain bits of this LUT's content, e.g., by rule R_{13}, can lead to a decrease or an increase of the rounds that the AES core processes.

Similarly, if manipulation rule R_2 (*Set LUT*) is applied, the targeted LUT (which, e.g., controls the *DONE* signal) forces the AES core to terminate the operation right after the start,

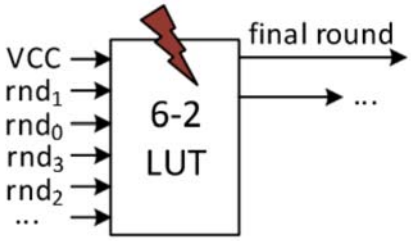


Figure 6.11: Manipulation rule R_{13} (*Invert bits if* $HW \leq 15$), round-based design D_6, consequence: modification of AES round counter threshold

cf. Figure 6.12. The state machine control flow is therefore affected, and as a consequence an intermediate state (e.g., $p \oplus rk_0$) is given instead of the ciphertext.

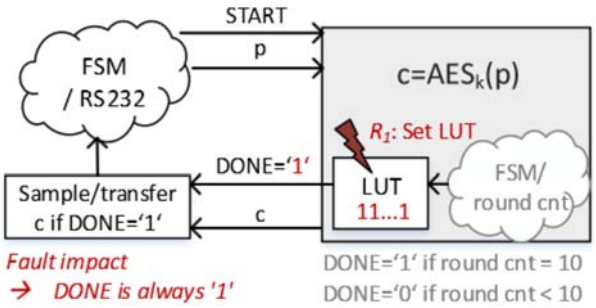


Figure 6.12: Manipulation rule R_2 (*Set LUT*), round-based design D_9, consequence: the AES core permanently signalizes *DONE*='1'

Hitting Pass-Through LUT of Countermeasure-Protected AES Core

To our surprise, even our countermeasure-protected AES core of Section 4.5 could be successfully attacked. Even though it was designed to protect against bitstream manipulations, in this case the implementation could not prevent the attack. The reason is that the synthesizer placed and routed one LUT to act as a pass-through element, i.e., it decides which decrypted values (through a 1-bit data bus) are configured into the actual S-boxes (i.e., cascaded and multiplexed CFG5 LUTs) of the AES core, cf. the flagged LUT in Fig. 6.13. In this particular case, the faulted LUT was reprogrammed to always output a logical zero, subsequently leading to zeroed S-boxes. As explained in Section 4.4.6, such a modified AES core will output the last round key rk_{10}. For the full work-flow of this countermeasure, the interested reader is referred to Section 4.5.

6.7 Discussions and Countermeasures

One way to counter BiFI attacks might be to include built-in self tests (BIST) and conventional fault attack countermeasures such as redundancy and error-detection circuitry. A BIST might

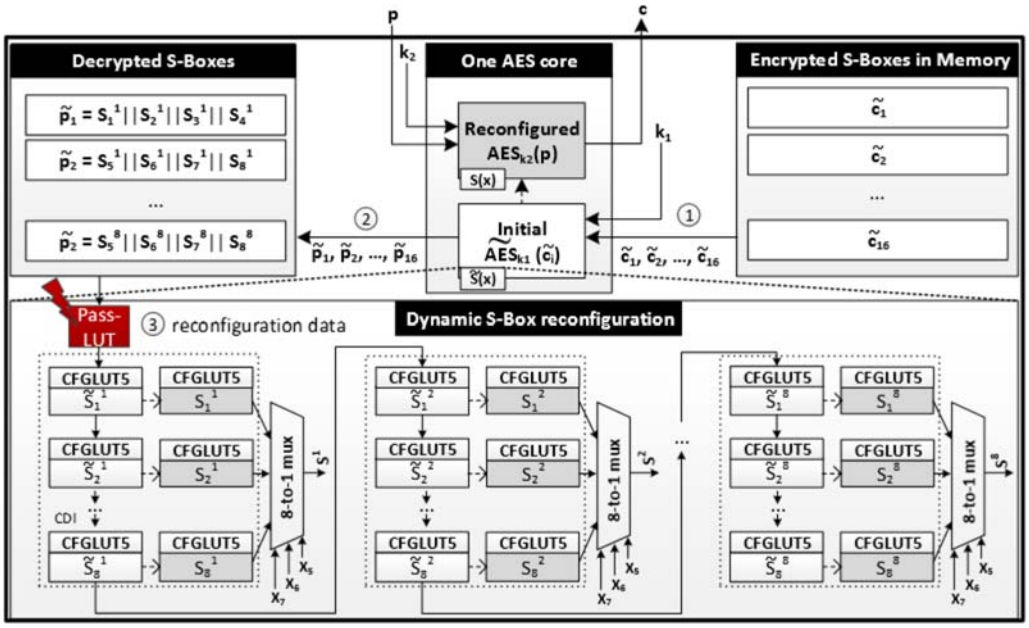


Figure 6.13: Manipulation rule R_1 (*Clear LUT*), round-based and countermeasure-protected design D_3, consequence: the 1-bit reconfiguration data bus is constantly set to "0" leading to a reconfiguration of zeroed truth tables of all dynamic CFG5 LUTs, which implement all AES S-box instances

prevent any manipulation attempt as conducted in this chapter since it will detect that the encryption core is not functioning correctly. However, in such a case an attacker could try to perform a two-step BiFI attack: in the first step, the attacker tries to trigger a BIST failure by identifying/manipulating a LUT belonging to the AES circuitry. The attacker then tries to disable the BIST by continuously modifying the bitstream until a faulty ciphertext is returned instead of the BIST failure. Although we did not verify this practically, the occurrence of a decision-making LUT, indicating a failure or success of the BIST, is likely. Manipulating such a LUT to always yield a "success" is then trivial and can be automatized by means of bruteforce.

Having figured out how to disable the BIST, in a second step one can proceed to apply the BiFI attack. Similar attack strategies might be applicable for other countermeasures. Which fault attack countermeasures are the most promising defenses against BiFI is still an open and interesting research question.

6.7.1 Impact on Other Fault Attack Types

This chapter focused on generating exploitable permanent faults using bitstream manipulation. However, we would like to highlight that the presented approach to recover an AES key from permanent faults is not restricted to these attacks. Basically, several other fault techniques can be used to create similar exploitable faulty ciphertexts. For example, it was shown that laser fault attacks can also be used to change the configuration of the FPGA, e.g., in [CML+11] or [TLG+15]. Typically, the goal of most laser fault attacks on FPGA designs is to cause transient faults in one round of the AES encryption to recover the key using a differential

fault analysis. Permanent faults are usually not intended and are seen (e.g., in [CML+11]) mainly as an obstacle from an attacker's perspective. However, the results in this chapter show that permanent faults can actually be a very powerful attack vector. The key insight is that even random configuration errors (rule R_{15}) have a high chance to result in exploitable faulty ciphertexts. Hence, the same attack idea can also be performed with random (or targeted) laser fault injection. A clock glitch or power glitch during the configuration of the bitstream might also be used to cause such configuration faults. Therefore, investigating how the BiFI attack can be extended to other fault techniques or cryptographic algorithms is an interesting future research direction.

6.8 Conclusion

This chapter introduces a new attack vector against cryptographic implementations on SRAM-based FPGAs. In this attack – so-called bitstream fault injection (BiFI) – the faults are injected systematically by configuring the target FPGA with malicious bitstreams. As a key insight of the BiFI attack, it can be automated so that no reverse-engineering of the target design is needed. Our attack, which is based on injecting permanent faults, is feasible in many practical realistic scenarios, where the attacker can manipulate the FPGA configuration and observe the faulty outputs of the target design. The adversary indeed manipulates the bitstream to alter and exploit the configuration maliciously.

Our experimental results with 16 AES implementations on a Spartan 6 FPGA showed that 15 out of these 16 designs could be successfully attacked in a few hours on average. The larger the design is (i.e., the more LUTs are utilized), the longer the attack takes. We furthermore showed that such a key recovery is even possible for *some* FPGAs when the bitstream encryption is enabled. The time required for the attack (in case of the encrypted bitstream) depends on the size of the underlying FPGA (more precisely on the number of available LUTs). It can range from hours (for low and mid-range FPGAs) up to weeks (for high-range FPGAs, e.g., with a million LUTs).

In short, the BiFI attack is non-invasive and requires neither a sophisticated setup nor to develop a complex reverse-engineering framework. Indeed, it can be conducted by engineers without particular expertise. Furthermore, BiFI demonstrated the necessity of finding appropriate countermeasures that should be researched and included as part of the hardware configuration, even in cases with an enabled bitstream encryption scheme.

Part III

Conclusion

Chapter 7

Conclusion

In this thesis, we have demonstrated the first practical attacks on third-party hardware configurations, which implement cryptographic algorithms, by directly manipulating the relevant functions through bitstream modification. In the following, we summarize our findings, draw a conclusion, and outline the directions for future research.

Contents of this Chapter

7.1 Impact of Bitstream Encryption Vulnerabilities 101

7.2 Impact of Bitstream Manipulations 102

7.3 Future Directions 103

7.1 Impact of Bitstream Encryption Vulnerabilities

The majority of today's bitstream encryption schemes of Altera and Xilinx FPGAs are vulnerable to side-channel attacks or no bitstream encryption is offered at all. IP theft and cloning have been considered as the major drawbacks of insecure bitstream encryption schemes. While previous research has successfully demonstrated that the Xilinx bitstream file formats can be almost entirely reverse-engineered, it still remained unclear whether the manipulation of an unknown third-party bitstream can lead to a security breach. Additionally, it was unknown whether such an attack could be conducted in a reasonable time span. In this thesis, we proved that concrete attacks against cryptographic primitives are indeed possible and surprisingly effective. Hence, the loss of integrity and confidentiality of a third-party bitstream implementing security-critical functionality is more problematic than commonly assumed. Considering the fact that billions of SRAM-based FPGAs have been sold and that the lifespan of an FPGA is approximately 10 to 15 years, we expect that the demonstrated attacks will be relevant for the next 5 to 10 years. Additionally, once further research proves security vulnerabilities of the currently deployed side-channel protected bitstream encryption schemes of newer FPGA generations such as Xilinx's Ultrascale, the presented bitstream manipulation attacks will become practically relevant for these newer devices as well.

7.2 Impact of Bitstream Manipulations

One major contribution of this thesis is the practical demonstration of bitstream manipulations potentially leading to key recovery or Trojan insertion, especially in those cases where a hardware configuration (implementing block ciphers such as DES and AES) makes use of precomputed S-box tables. Once all attacker tools are prepared and the targeted bitstream of an embedded device can be replaced by a malicious one, the actual manipulation can be conducted quickly utilizing our proposed S-box detection algorithms.

The most remarkable result is that an attacker neither requires any routing information of a third-party hardware configuration nor does he have to reverse-engineer the entire bitstream file format or the entire built circuitry, making it a realistic attacker scenario. It was demonstrated that in most cases for a variety of AES implementations it is sufficient to only obtain the LUT and BRAM encoding for targeted manipulations. Surprisingly, a considerably weaker attack model than expected is required to conduct successful practical attacks even though the underlying hardware configuration is potentially complex and the file formats are proprietary. Meaningful bitstream manipulations were believed to be too complex and time-consuming, but they *are* indeed practical, and constitute a serious threat for many embedded devices. Therefore, they should be considered by a defender as a potential security issue. The practical relevance of this attack vector was highlighted by successfully attacking the bitstream of a Xilinx FPGA which was embedded in a real-world high-security USB flash drive from Kingston intended to securely encrypt user data. Another remarkable result is the fact that it is not necessary to develop a complex framework for reverse-engineering hardware circuits, which requires a considerable amount of development time and assumes a more powerful attacker. To summarize, the work of this thesis raises awareness for security-related issues arising due to the specific nature of SRAM-based FPGAs.

Another strength of bitstream manipulation attacks is their non-invasiveness. They may succeed in situations where side-channel or laser fault injection attacks fail to undermine the security of an embedded device. Those kind of implementation attacks often require expensive equipment or expertise, and thus, our proposed attack vector represents an effective alternative.

In this thesis, we proposed a concrete countermeasure to impede targeted bitstream manipulations that otherwise would easily lead to key recovery or Trojan insertion. By introducing the BiFI attack, we found an even more powerful strategy for recovering cryptographic keys from AES cores that are executed by Xilinx FPGAs. As a negative result, we observed that our proposed countermeasure is still vulnerable to key recovery. Therefore, we draw the conclusion that it is challenging to design appropriate countermeasures at the hardware configuration layer. BiFI does not need to algorithmically detect any hardware primitives. In addition to that, the attack is design-independent, as we could also attack AES cores using non-precomputed S-boxes. We hence conclude that an attacker requires significantly less knowledge for conducting successful attacks.

Major security concerns result from the synthesizer which is currently not designed for security-related applications. Our findings indicate that many exploitable hardware structures are unintentionally inserted by the synthesizer. It is rather difficult to protect a cryptographic design against all kinds of bitstream manipulations. BiFI also turned out to be surprisingly powerful in attacking a Xilinx Virtex 5 FPGA with an enabled bitstream encryption scheme. We expect that many more devices are vulnerable to such attacks.

The main take away message is that SRAM-based FPGAs constitute a risk for security-related tasks unless secure countermeasures are implemented by the FPGA vendor to provide the corresponding level of integrity and authenticity.

7.3 Future Directions

There are various potential research directions which should be considered for future work. Since many unprotected FPGAs are deployed making use of potentially easily exploitable bitstreams, one interesting research field is to examine which hardware structures describing a cryptographic circuit are best suited to sufficiently prevent targeted and untargeted malicious bitstream manipulations.

Finding difficult-to-manipulate hardware structures could therefore be a promising research field. This should involve classifying exploitable hardware structures that synthesizers unintentionally add to a cryptographic hardware configuration. Knowing which hardware structures are problematic, a specifically designed synthesizer could try to avoid them or extend them in such a way that it sufficiently counters malicious hardware configuration manipulations beforehand. Since FPGAs allow easy replacement of a hardware configuration, many real-world embedded devices deploying an FPGA would benefit from updating its flawed bitstream by a more secure variant. This should be considered as raise-the-bar countermeasure.

Evaluating the security of bitstream encryption and authentication of newer FPGA generations is another desirable research area. Newer FPGA families deploy side-channel resistant bitstream encryption and authentication schemes, and are currently not vulnerable to the presented bitstream manipulation attacks. Therefore, those devices are currently considered to be secure. By demonstrating a security flaw leading to a bitstream encryption or authentication key leakage, our presented bitstream manipulation attack would become practically relevant for these devices. This is one reason why we believe that designing a special-purpose synthesizer for cryptographic functions is an important research field. Besides this, showing weaknesses in newer bitstream encryption and authentication schemes raises security awareness, i.e., it gives an idea to what extent FPGAs are suited to securely execute cryptographic functions.

Evaluating alternative hardware configuration manipulation strategies exhibits also a noteworthy research area. In 2015, Tajik *et al.* [TLG+15] demonstrated that a given hardware configuration of an SRAM-based CPLD can be partially altered and controlled *after* the initial configuration of the device. The authors carried out a laser fault injection attack resulting in a permanent change of one LUT, similar to our presented BiFI attack. As we have demonstrated, this can lead to a key leakage. Even though the laser fault-injection attack was only demonstrated for CPLDs, it is likely that it can be applied to FPGAs as well. This is because both technologies work similarly and are based on SRAM cells. Hence, evaluating whether this kind of attack can be conducted on newer FPGA generations with enabled bitstream encryption and authentication could again raise security awareness and further highlight the importance of finding difficult-to-manipulate hardware structures.

Part IV

Appendix

Bibliography

[A. 15] A. C. Aldaya, A. J. C. Sarmiento, S. Sánchez-Solano. AES T-Box tampering attack. *Journal of Cryptographic Engineering*, pages 1–18, 2015. 8

[Aka07] Akashi Satoh. *Cryptographic Hardware Project: IP Cores*, 2007. http://www.aoki.ecei.tohoku.ac.jp/crypto/web/cores.html. 47

[Alt08] Altera. Stratix III FPGA Development Kit. http://www.altera.com/products/devkits/altera/kit-siii-host.html, 2008. 22

[Alt15] Altera. ALTERA Annual Report for Form 10-K for 2014. https://www.sec.gov/Archives/edgar/data/768251/000076825115000008/altera10k12312014.htm, 2015. 5, 6

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology - EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997. 78

[BKL+07] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007. 39

[BRPB13] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy Dopant-Level Hardware Trojans. In *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, pages 197–214, 2013. 62

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Advances in Cryptology - CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997. 78

[BSH12] F. Benz, A. Seffrin, and S.A. Huss. BIL: A tool-chain for bitstream reverse-engineering. In *Field Programmable Logic and Applications (FPL)*, pages 735–738. IEEE, Aug 2012. 7

[CML+11] Gaetan Canivet, Paolo Maistri, Régis Leveugle, Jessy Clédière, Florent Valette, and Marc Renaudin. Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA. *J. Cryptology*, 24(2):247–268, 2011. 97, 98

[Cor12] Altera Corporation. Stratix III FPGA: Lowest Power, Highest Performance 65-nm FPGA, 2012. http://www.altera.com/devices/fpga/stratix-fpgas/stratix-iii/st3-index.jsp. 22

[CSPN13] R.S. Chakraborty, I. Saha, A. Palchaudhuri, and G.K. Naik. Hardware Trojan Insertion by Direct Modification of FPGA Configuration Bitstream. *Design Test, IEEE*, 30(2):45–54, April 2013. 8

[CT05] Hamid Choukri and Michael Tunstall. Round Reduction Using Faults. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 13–14, 2005. 78

[Dri08] Saar Drimer. Volatile FPGA design security – a survey (v0.96), April 2008. 8

[DWZZ13] Zheng Ding, Qiang Wu, Yizhong Zhang, and Linjie Zhu. Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation. *Microprocessors and Microsystems - Embedded Hardware Design*, 37(3):299–312, 2013. 7

[EGP+07] Thomas Eisenbarth, Tim Güneysu, Christof Paar, Ahmad-Reza Sadeghi, Dries Schellekens, and Marko Wolf. Reconfigurable trusted computing in hardware. In *Workshop on Scalable Trusted Computing, STC 2007*, pages 15–20. ACM, 2007. 68

[ET13] LANGER EMV-Technik. Near-field probes. https://www.langer-emv.de/en/category/near-field-probes/19, 2013. 25

[Fek14] Fekete Balazs. *AES encryption all keylength :: Overview*, 2014. http://opencores.org/project,aes_all_keylength. 47

[GKA+10] K. Gaj, J. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B.Y. Brewster. ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 414–421, Aug 2010. 12

[GLS11] Steve Guccione, Delon Levi, and Prasanna Sundararajan. JBits: Java based interface for reconfigurable computing. In *CCS 2011*. ACM, 2011. 6

[Gre14] Glenn Greenwald. *No Place to Hide: Edward Snowden, the NSA and the Surveillance State*. Metropolitan Books, 2014. 61

[Hem14] Hemanth. *AES128 :: Overview*, 2014. http://opencores.org/project,aes_crypto_core. 47

[HLK02] Edson L. Horta, John W. Lockwood, and Sérgio T. Kofuji. *Using PARBIT to Implement Partial Run-Time Reconfigurable Systems*, pages 182–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. 6

[HR12] Hex-Rays. IDA - Interactive DisAssembler, 2012. http://www.hex-rays.com. 66

[HSH+09] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, May 2009. 35

[IEE08] IEEE Std 1619-2007. IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices, 2008. 72

[Jer11] Jerzy Gbur. *AES core modules :: Overview*, 2011. http://opencores.org/project,aes_128_192_256. 47

[KA08] Bhupathi Kakarlapudi and Nitin Alabur. FPGA Implementations of S-box vs. T-box iterative architectures of AES, 2008. 69

[Kin] Kingston Technology. Protect sensitive data with FIPS 140-2 Level 2 validation and 100 per cent privacy. 63, 64

[KK06] Tim Kerins and Klaus Kursawe. A Cautionary Note on Weak Implementations of Block Ciphers. In *Workshop on Information and System Security*, page 12, Antwerp, BE, 2006. 40, 50, 82

[LPL+10] Christopher Lavin, Marc Padilla, Philip Lundrigan, Brent E. Nelson, and Brad L. Hutchings. Rapid prototyping tools for FPGA designs: RapidSmith. In *Proceedings of the International Conference on Field-Programmable Technology, FPT 2010, 8-10 December 2010, Tsinghua University, Beijing, China*, pages 353–356, 2010. 6

[LSG+10] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault Sensitivity Analysis. In *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2010. 78

[MBKP11] A. Moradi, A. Barenghi, T. Kasper, and C. Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs. In *CCS 2011*, pages 111–124. ACM, 2011. 5

[MBO+05] E. De Mulder, P. Buysschaert, S. B. Ors, P. Delmotte, B. Preneel, G. Vandenbosch, and I. Verbauwhede. Electromagnetic Analysis Attack on an FPGA Implementation of an Elliptic Curve Cryptosystem. In *EUROCON 2005 - The International Conference on "Computer as a Tool"*, volume 2, pages 1879–1882, Nov 2005. 78

[MC13] Mini-Circuits. Amplifier Data Sheet. http://www.minicircuits.com/pdfs/ZFL-1000LN+.pdf, 2013. 25

[Mic] Rosetta Micro. ENSURING TRUST IN CYBERSPACE. 74

[Mic10] Michael Calvin McCoy. *A collection of my practical VHDL and Verilog modules*, 2010. https://github.com/abhinav3008/inmcm-hdl/tree/master/AES/Basic_AES_128_Cipher. 47

[MKP12] A. Moradi, M. Kasper, and C. Paar. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures - An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism. In *CT-RSA 2012*, volume 7178 of *LNCS*, pages 1–18. Springer, February 2012. 5

[MOPS13] Amir Moradi, David Oswald, Christof Paar, and Pawel Swierczynski. Side-channel Attacks on the Bitstream Encryption Mechanism of Altera Stratix II: Facilitating Black-box Analysis Using Software Reverse-engineering. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 91–100. ACM, February 2013. 5, 8, 18, 21, 23, 24, 26, 90

[Mot13] Moti Litochevski and Luo Dongjun. *high throughput and low area aes core :: Overview*, 2013. http://opencores.org/project,aes_highthroughput_lowarea. 47

[MS16] A. Moradi and T. Schneider. Improved Side-Channel Analysis Attacks on Xilinx Bitstream Encryption of 5, 6, and 7 Series. In *Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2016. 5

[Ngu16] Jean-Francois Nguyen. Analysing the Bitstream of Altera's MAX-V CPLDs, July 2016. 7

[NIS99] NIST. FIPS-46-3: Data Encryption Standard (DES), 1999. http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf. 40, 42

[NIS01a] NIST. FIPS 197 Advanced Encryption Standard (AES), 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. 17

[NIS01b] NIST. Suite B Cryptography, 2001. 63

[NIS10] NIST. DataTraveler 5000 FIPS 140-2 Level 2 certification, 2010. 63

[NKL14] Karsten Nohl, Sascha Krißler, and Jakob Lell. BadUSB - On accessories that turn evil. BlackHat, 2014. 62

[Not08] Jean-Baptiste Note. debit, January 2008. 7

[NR08] Jean-Baptiste Note and Éric Rannaud. From the bitstream to the netlist. In *Proceedings of the ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays, FPGA 2008, Monterey, California, USA, February 24-26, 2008*, page 264, 2008. 6

[NSA99] NSA. *Round 2 Analysis*, 1999. http://csrc.nist.gov/archive/aes/round2/r2anlsys.htm#NSA. 47

[PHK17] K. Dang Pham, E. Horta, and D. Koch. BITMAN: A Tool and API for FPGA Bitstream Manipulations. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 894–897, March 2017. 7

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In *Cryptographic Hardware and Embedded Systems – CHES 2003*, volume 2779 of *Lecture Notes in Computer Scienes*, pages 77–88. Springer, 2003. 78

[RS02] A. K. Raghavan and P. Sutton. JPG - a partial bitstream generation tool to support partial reconfiguration in virtex FPGAs. In *Proceedings 16th International Parallel and Distributed Processing Symposium*, pages 6 pp–, April 2002. 6

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Sciene*, pages 2–12. Springer, 2002. 78

[SBMP17] Pawel Swierczynski, Georg T. Becker, Amir Moradi, and Christof Paar. Bitstream Fault Injections (BiFI) - Automated Fault Attacks against SRAM-based FPGAs. *IEEE Transactions on Computers*, PP(99):1–1, January 2017. 9

[SFK+16] Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, Amir Moradi, and Christof Paar. Interdiction in Practice - Hardware Trojan Against a High-Security USB Flash Drive. *Journal of Cryptographic Engineering*, pages 1–13, June 2016. 9

[SFKP15] Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, and Christof Paar. FPGA Trojans Through Detecting and Weakening of Cryptographic Primitives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1236–1249, August 2015. 9

[SFP+15] Pawel Swierczynski, Marc Fyrbiak, Christof Paar, Christoph Huriaux, and Russell Tessier. Protecting against Cryptographic Trojans in FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 151–154. IEEE, May 2015. 9

[SMOP14] Pawel Swierczynski, Amir Moradi, David Oswald, and Christof Paar. Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):34:1–34:23, January 2014. 5, 8

[SNK+13] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. Simple Photonic Emission Analysis of AES. *J. Cryptographic Engineering*, 3(1):3–15, 2013. 78

[Sny14] Bill Snyder. Snowden: The NSA planted backdoors in Cisco products, 05 2014. 61

[SPI13] SPIEGEL Staff. Inside TAO: Documents Reveal Top NSA Hacking Unit, December 29 2013. http://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969.html. 62

[SWS+11] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French. Torc: towards an open-source tool flow. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays,*

FPGA 2011, Monterey, California, USA, February 27, March 1, 2011, pages 41–44, 2011. 7

[Tar13] Tariq Ahmad. *fast AES-128 Encryption only cores :: Overview*, 2013. http://opencores.org/project,aes-encryption. 47

[TLG+15] S. Tajik, H. Lohrke, F. Ganji, J. P. Seifert, and C. Boit. Laser fault attack on physically unclonable functions. In *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 85–96, Sept 2015. 97, 103

[TML11] Steven Trimberger, Jason Moore, and Weiguang Lu. Authenticated encryption for FPGA bitstreams. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 83–86. ACM, 2011. 78

[VKS11] I. Verbauwhede, D. Karaklajic, and J. M. Schmidt. The Fault Attack Jungle - A Classification Model to Guide You. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 3–8. IEEE, 2011. 78

[Wik] Wikipedia. Pearson correlation coefficient. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient. 23

[WL] Clifford Wolf and Mathias Lasser. Project icestorm. http://www.clifford.at/icestorm/. 7

[Xil] Xilinx. Spartan-6 Family FPGAs. http://team358.org/files/programming/ControlSystem2009-/control_system/FPGA-Spartan3_Spartan6-comparison.pdf. 14

[Xil15] Xilinx. Xilinx Annual Report for Form 10-K for 2014. https://www.sec.gov/Archives/edgar/data/743988/000074398815000022/xlnx0328201510k.htm, 2015. 6

[ZAT06] D. Ziener, S. Assmus, and J. Teich. Identifying FPGA IP-Cores Based on Lookup Table Content Analysis. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, Aug 2006. 6

List of Abbreviations

3DES Triple-DES

AES Advanced Encryption Standard

API Application Programming Interface

ARM Advanced RISC Machine

ASIC Application Specific Integrated Circuit

BiFI Bitstream Fault Injection

BRAM Block Random Access Memory

CLB Configurable Logik Block

CPA Correlation Power Analysis

CPLD Complex Programmable Logic Device

DES Data Encryption Standard

DNF Disjunctive Normal Form

DSO Digital Storage Oscilloscope

DSP Digital Signal Processing

ECC Elliptic Curve Cryptography

EEPROM Electrically Erasable Programmable Read-Only Memory

EM Electro-Magnetic

FF Flip Flop

FPGA Field Programmable Gate Array

HD Hamming Distance

HDL Hardware Description Language

HMAC Hash-based Message Authentication Code

HSM Hardware Security Module

IC Integrated Circuit

IOB Input Output Block

IP Intellectual Property

IV Initialization Vector

JTAG Joint Test Action Group

LFSR Linear Feedback Shift Register

LSB Least Significant Bit

LUT Look-Up Table

NSA National Security Agency

OEM Original Equipment Manufacturer

PCB Printed Circuit Board

PC Personal Computer

PUF Physically Unclonable Function

RAM Random Access Memory

SCA Side-Channel Analysis

SHA Secure Hash Algorithm

SPI Serial Peripheral Interface

SRAM Static Random Access Memory

USB Universal Serial Bus

VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

XTS XEX Tweakable Block Cipher with Ciphertext Stealing

List of Figures

2.1 Building blocks of Xilinx FPGAs . . . 11
2.2 Exemplary CLB content . . . 12
2.3 Overview of one slice . . . 12
2.4 Simplified Spartan 6 FPGA architecture . . . 12
2.5 Example for configuring a truth table of one LUT using two different input permutations $(x_5, x_4, x_3, x_2, x_1, x_0)$ and $(x_0, x_4, x_3, x_2, x_1, x_5)$ . . . 13
2.6 Simplified design flow of Xilinx FPGAs, which translates a high-level hardware layout to a low-level hardware configuration . . . 15
2.7 Overview of system model. A proprietary bitstream file implements an unknown circuit (e.g., AES), which configures an FPGA once it is powered-up. After this phase, a control circuit provides an interface to the encryption application. In practical applications, the bitstream and FPGA are integrated on the same PCB 16
2.8 On the left, a 6-to-1 LUT with 6 input bits and 1 output bit is depicted. The LUT is a truth table T with 64 entries that are stored in the bitstream . . . 16
2.9 Overview of the AES encryption algorithm . . . 17
2.10 Key schedule of AES-128 . . . 17

3.1 Device under attack - Official development board containing a Stratix III FPGA 22
3.2 Observed execution order of the encryption module of Stratix III FPGAs . . . 24
3.3 Overview of the function f responsible for updating AES inputs . . . 24
3.4 Stratix III FPGA development kit, a) the original FPGA, b) and c) removing the metal cap of the FPGA, d) the decapsulated FPGA with an EM probe at the optimal position . . . 25
3.5 Correlation coefficient for the HD of the row-wise consecutive *ShiftRows* bytes using 365,000 traces measured during one power-up of the Stratix III . . . 26

4.1 A fully mapped and routed hardware configuration showing a more specific part of the FPGA grid with occupied hardware resources implementing an AES design. If the corresponding intermediate file format is given in a practical attack scenario, the hardware configuration could be viewed with the help of Xilinx's FPGA editor. It shows the switch-matrix, routing, and distribution of utilized slices. Note that such a hardware circuit is encoded by the proprietary bitstream file. Therefore, an attacker does not possess this representation . . . 30
4.2 SP601 evaluation kit featuring a Xilinx Spartan 6 FPGA serving as target device for our proof-of-concept bitstream manipulation attack . . . 31
4.3 Overview of the DES encryption algorithm . . . 41
4.4 DES round function f . . . 41
4.5 Modified DES with canceled f-function . . . 41

4.6 Simplified overview of a slice of a Spartan 6 FPGA realizing an 8-input,1-output Boolean function (256 bits of memory) with four 6-input,1-output LUTs (64 bits of memory each). Our example implements one S-box output column of AES. Eight of those instances are needed to implement one AES S-box 43
4.7 FPGA grid view of a Xilinx Spartan 6 XC6SLX16, where each cell represents one slice. It shows the results of the proposed 8-input, 8-output S-box detection algorithm for AES hardware configurations. Any cell containing a number shows that the i^{th} (with $i \in \{0,1,\ldots, 7\}$) AES S-box output column is implemented by the corresponding slice, i.e., it is successfully detected. White cells represent unused slices, whereas gray cells denote the usage of arbitrary combinatorial logic within one slice, i.e., at least one out of 4 LUTs is configured 45
4.8 Key schedule of AES-192 52
4.9 Initial LUT with Boolean function f_{init} 55
4.10 Reconfigured LUT with Boolean function f 55
4.11 Dynamic reconfiguration of AES S-boxes using CFGLUT5 elements 57

5.1 Interdiction attack conducted by intelligence services 63
5.2 Epoxy removal of Kingston DT 5000 with screwdriver 64
5.3 Eavesdropping the bitstream of Kingston DT 5000 with opened case 64
5.4 Address space layout of the SPI flash 65
5.5 Overview of revealed circuit of our target device 66
5.6 User authentication (dashed) and user data (solid) dependencies before modification 67
5.7 User authentication (dashed) and data (solid) dependencies after modification . . 67
5.8 XTS-AES encryption block diagram overview 73

6.1 Subset of the most commonly used possible slice configurations with focus on look-up tables 79
6.2 a) The ratio the number of performed bitstream manipulations over the number of exploitable faults. b) The average attack time (in hours) until an exploitable faulty ciphertext is obtained for each of the targeted design (using modification rules R_1-R_{12}) 86
6.3 The impact of faulting one ciphertext block in case of CBC decryption 87
6.4 Virtex 5 (VLX50) bitstreams, left: encryption enabled and CRC off (*bs_enc_crc_off*), right: encryption enabled and CRC on (*bs_enc_crc_on*) 90
6.5 Spartan 6 (SLX75) bitstreams, left: encryption enabled and CRC off (*bs_enc_crc_off*), right: encryption enabled and CRC on (*bs_enc_crc_on*) 90
6.6 The decryption in counter mode as it is used for bitstream encryption in Stratix II FPGAs. Toggling a single ciphertext bit results in a predictable toggle of a plaintext bit 91
6.7 Manipulation rule R_1 (*Clear LUT*), round-based design D_0, consequence: plaintext p (instead of sr_{10}) is XORed to the last AES round key rk_{10} 92
6.8 Manipulation rule R_1 (*Clear LUT*), round-based design D_{15}, group of flip-flops forming a 128-bit round key register ($rk_{j,0}$ - $rk_{j,127}$) used for XOR with the current AES state. Due to the manipulation, none of the round key flip-flops are updated. Instead, they always remain '0' 93

6.9 Manipulation rule R_1 (*Clear LUT*), word-based design D_1, due to the bitstream manipulation the S-box inputs remain zero, this results into the leakage of the last round key rk_{10} 94
6.10 Manipulation rule R_3 (*Invert LUT*), round-based design D_4, Due to the LUT inversion of the *update_out_reg_enable* control signal, the relevant output register *out_reg* is updated at the wrong clock cycles, i.e., the modified AES core fails to copy the correct ciphertext c and writes the leaking state ka_{10} 95
6.11 Manipulation rule R_{13} (*Invert bits if* $HW \leq 15$), round-based design D_6, consequence: modification of AES round counter threshold 96
6.12 Manipulation rule R_2 (*Set LUT*), round-based design D_9, consequence: the AES core permanently signalizes *DONE*='1' 96
6.13 Manipulation rule R_1 (*Clear LUT*), round-based and countermeasure-protected design D_3, consequence: the 1-bit reconfiguration data bus is constantly set to "0" leading to a reconfiguration of zeroed truth tables of all dynamic CFG5 LUTs, which implement all AES S-box instances 97

List of Tables

1.1 List of Xilinx FPGA families and Altera FPGA devices, which are vulnerable to side-channel attacks. No side-channel attacks for the UltraSCALE and UltraSCALE$^+$ family have been reported so far. Note that the Xilinx 7 series includes the Kintex, Artix, and Virtex families 5

2.1 General shape of a 6-input, 1-output look-up table, e.g., used by Xilinx Spartan 3 FPGAs 13

2.2 General shape of a 4-input, 1-output look-up table, e.g., used by Xilinx Spartan 6 FPGAs 13

3.1 Required differences between Stratix II and Stratix III FPGAs when performing reverse-engineering and a side-channel attack 27

4.1 Generating 65 bitstreams for one LUT 34

4.2 General shape of a 6-input,4-output S-box 37

4.3 Overview of evaluated DES implementations 38

4.4 Overview of evaluated AES implementations 48

4.5 General shape of an 8-input,8-output AES S-box. 56

4.6 Static LUT-based / dynamic DES and AES designs 58

5.1 Identified substitution tables stored in block RAM 69

5.2 Identified self-tests and firmware integrity check 71

5.3 Security header fields 71

6.1 Overview of the experiments with regard to the different modification rules. Each entry in the table represents the number of times for which applying the manipulation rule R_i lead to an exploitable fault for design D_j. The last column "d.att." (designs attacked) shows the number of different designs D_j that could be attacked with the corresponding rule. In the experiment, several modification rules resulted in an exploitable faulty ciphertext when applied to the same LUT. The number of LUTs that lead to at least one exploitable faulty ciphertext for at least one of the manipulation rules $R_1 - R_{14}$ is shown in row "Number of vulnerable LUTs" as a reference 84

6.2 Overview of the experiments with regard to the different hypotheses. Each entry in the table represents the number of times a hypotheses H_i for each design D_j could be used to recover the key from faulty ciphertexts being the result of applying the modification rules R_1-R_{15}. Some hypotheses (marked by †) succeed only for R_{15} while some other hypotheses marked with ‡ could make use of only R_1-R_{14}. The last column "d.att." shows the number of different designs that could be successfully attacked by the corresponding hypothesis. The last two rows summarize the number of collected responses (which are equivalent to the number of times a bitstream manipulation was conducted) and the number of observed unique faulty ciphertexts 85
6.3 Overview of the BiFI attack on encrypted bitstreams. Two modification rules R_3 and R_6 were tested and each table entry represents the number of exploitable faulty ciphertexts 87
6.4 Overview of the exploitable faulty ciphertexts of the different hypotheses for 13 different designs with enabled bitstream encryption 88

List of Algorithms

1 Pseudo-code for AES input update function f 25

2 LUT encoding extraction for any k-input,1-output Xilinx FPGA with $k \in \{4, 6\}$ 34
3 BRAM encoding extraction for Xilinx FPGAs 36
4 Computation of $\pi_p(\cdot)$ of the p^{th} permutation with $p \in \{0, 1, \ldots, 719\}$) 39
5 Detection of all 8 DES S-boxes being distributed over 32 6-input,1-output LUTs 40
6 Detection of AES S-boxes being distributed over 32 6-input,1-output LUTs . . . 46
7 Decryption of ciphertexts that were encrypted with $S^{id}(\cdot)$ 49
8 Reconstruction of the full main key of AES-128 51
9 Partial key reconstruction of AES-192/256 52

10 Computation of key-dependent secrets CK_j 74
11 Decryption of ciphertexts that were encrypted with manipulated AES-XTS . . . 74

About the Author

Author information as of July 2018.

Personal Data

Name Pawel Swierczynski

Address
Chair for Embedded Security, ID 2/625
Universitätsstr. 150
44801 Bochum, Germany

E-Mail pawel.swierczynski@rub.de

Date of birth October 29, 1986

Place of birth Tychy, Poland

Education

01/2013 - 09/2017	**Doctoral candidate**, *Ruhr-Universität Bochum*, Electrical and Information Engineering.
10/2010 - 01/2013	**M.Sc.**, *Ruhr-Universität Bochum*, IT Security/Information Engineering.
10/2007 - 10/2010	**B.Sc.**, *Ruhr-Universität Bochum*, IT Security/Information Engineering.

Professional Experience

01/2013 - 09/2017	**Research Assistant**, *Ruhr-Universität Bochum.* Chair for Embedded Security (EMSEC).
04/2010 - 05/2010	**Intern**, *KPMG*, Essen.

Publications and Academic Activities

Peer-Reviewed Journal Papers

- Marc Fyrbiak, Sebastian Wallat, Pawel Swierczynski, Max Hoffmann, Sebastian Hoppach, Matthias Wilhelm, Tobias Weidlich, Russell Tessier and Christof Paar. HAL - The Missing Piece of the Puzzle for Hardware Reverse Engineering, Trojan Detection and Insertion. *IEEE Transactions on Dependable and Secure Computing, 2018, to appear.*
- Pawel Swierczynski, Georg T. Becker, Amir Moradi, and Christof Paar. Bitstream Fault Injections (BiFI) - Automated Fault Attacks against SRAM-based FPGAs. *IEEE Transactions on Computers*, PP(99):1–1, January 2017.
- Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, Amir Moradi, and Christof Paar. Interdiction in Practice - Hardware Trojan Against a High-Security USB Flash Drive. *Journal of Cryptographic Engineering*, pages 1–13, June 2016.
- Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, and Christof Paar. FPGA Trojans Through Detecting and Weakening of Cryptographic Primitives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1236–1249, August 2015.
- Pawel Swierczynski, Amir Moradi, David Oswald, and Christof Paar. Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):34:1–34:23, January 2014.

Peer-Reviewed Conference Proceeding

- Pawel Swierczynski, Marc Fyrbiak, Christof Paar, Christoph Huriaux, and Russell Tessier. Protecting against Cryptographic Trojans in FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 151–154. IEEE, May 2015.
- Amir Moradi, David Oswald, Christof Paar, and Pawel Swierczynski. Side-channel Attacks on the Bitstream Encryption Mechanism of Altera Stratix II: Facilitating Black-box Analysis Using Software Reverse-engineering. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 91–100. ACM, February 2013.

Technical Reports

- Marc Fyrbiak, Sebastian Wallat, Pawel Swierczynski, Max Hoffmann, Sebastian Hoppach, Matthias Wilhelm, Tobias Weidlich, Russell Tessier and Christof Paar. HAL - The Missing

Piece of the Puzzle for Hardware Reverse Engineering, Trojan Detection and Insertion. *IACR Cryptology ePrint Archive*, 2017:783, 2017.

- Pawel Swierczynski, Georg T. Becker, Amir Moradi, and Christof Paar. Bitstream Fault Injections (BiFI) - Automated Fault Attacks against SRAM-based FPGAs. *IACR Cryptology ePrint Archive*, 2016:641, 2016.
- Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, Amir Moradi, and Christof Paar. Interdiction in Practice - Hardware Trojan Against a High-Security USB Flash Drive. *IACR Cryptology ePrint Archive*, 2015:768, 2015.
- Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, and Christof Paar. FPGA Trojans Through Detecting and Weakening of Cryptographic Primitives. *IACR Cryptology ePrint Archive*, 2014:649, 2014.

Invited Talks

- Side-Channel Attacks on the Bitstream Encryption Mechanism of Altera Stratix II. *11th CryptArchi Workshop: Cryptographic Architectures Embedded in Reconfigurable Devices, June 25rd 2013, Fréjus, France.*
- SHA-3 - Portierung auf einer ATmega163 Smartcard. *23th SmartCard Workshop: Februray 2nd 2013, Darmstadt, Germany.*

Research Visits

- University of Massachusetts, Amherst, 09/12/2015 – 10/12/2015.
- University of Massachusetts, Amherst, 03/09/2015 – 04/10/2015.
- University of Massachusetts, Amherst, 06/05/2014 – 07/05/2014.
- University of Massachusetts, Amherst, 10/30/2013 – 11/14/2013.

Awards and Stipends

- CAST-Förderpreis IT-Sicherheit 2013, third price in category master/diploma thesis.

Participation in Selected Conferences, Workshops and Summer Schools

- FCCM 2015, *Vancouver, Canada.*
- UbiCrypt Summerschool 2013, *Bochum, Germany.*
- CRYPTO 2013, *Santa Barbara, USA.*
- CHES 2013, *Santa Barbara, USA.*
- Smartcard Workshop 2013, *Darmstadt, Germany.*

- CryptArchi Workshop 2013, *Fréjus, France.*
- FPGA 2013, *Monterey, USA.*