Universidade Estadual de Campinas
Instituto de Computação

Augusto Fernandes Ribas Queiroz

Secure code execution using PUF authentication

Execução segura de códigos utilizando PUF para autenticação

CAMPINAS
2019

Augusto Fernandes Ribas Queiroz

Secure code execution using PUF authentication

Execução segura de códigos utilizando PUF para autenticação

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr.Guido Costa Souza de Araújo
Co-supervisor/Coorientador: Prof. Dr.Mário Lúcio Côrtes

Este exemplar corresponde à versão final da Dissertação defendida por Augusto Fernandes Ribas Queiroz e orientada pelo Prof. Dr.Guido Costa Souza de Araújo.

CAMPINAS
2019

Na versão final, esta página será substituída pela ficha catalográfica.

De acordo com o padrão da CCPG: "Quando se tratar de Teses e Dissertações financiadas por agências de fomento, os beneficiados deverão fazer referência ao apoio recebido e inserir esta informação na ficha catalográfica, além do nome da agência, o número do processo pelo qual recebeu o auxílio."
e
"caso a tese de doutorado seja feita em Cotutela, será necessário informar na ficha catalográfica o fato, a Universidade convenente, o país e o nome do orientador."

Universidade Estadual de Campinas
Instituto de Computação

Augusto Fernandes Ribas Queiroz

Secure code execution using PUF authentication

Execução segura de códigos utilizando PUF para autenticação

Banca Examinadora:

- Prof. Dr. someone
  Universidade Estadual de Campinas
- Prof. Dr.Someonel
  Universidade Federal de Santa Catarina
- Profa. Dra. someone
  Universidade Estadual de Campinas

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 28 de março de 2019

Dedicatória

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus vitae iaculis erat. Aliquam tristique consectetur ante, quis commodo lacus egestas in. Nullam semper elit nec eros pretium

Resumo

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus vitae iaculis erat. Aliquam tristique consectetur ante, quis commodo lacus egestas in. Nullam semper elit nec eros pretium dapibus. Ut eget porta metus. Mauris rhoncus vel magna non faucibus. Ut a ornare elit. Morbi sagittis quam nec risus laoreet, tincidunt volutpat ex venenatis. Sed ultrices felis quis felis scelerisque gravida a non neque. Etiam sed nisi neque. Ut lobortis pulvinar facilisis.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus vitae iaculis erat. Aliquam tristique consectetur ante, quis commodo lacus egestas in. Nullam semper elit nec eros pretium dapibus. Ut eget porta metus. Mauris rhoncus vel magna non faucibus. Ut a ornare elit. Morbi sagittis quam nec risus laoreet, tincidunt volutpat ex venenatis. Sed ultrices felis quis felis scelerisque gravida a non neque. Etiam sed nisi neque. Ut lobortis pulvinar facilisis.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Standard design techniques to secure code execution in SoCs are based on well-known cryptographic mechanisms and on (micro) architecture features to encode bus transactions [?], or isolate secure code into trusted platforms [?], among others. Although such techniques usually provide good levels of security, most of them are either inefficient, considerably impact processor (micro) architecture design, require extensive changes in the programming tool-chain [?], or are so complex [?] that may create unexpected security loopholes. Any solution up to this challenge should be able to use more than incremental approaches which try to re-use current cryptographic mechanisms to fill in security holes; the new generation of IoT SoC devices will require novel solutions which deeply integrate hardware-intrinsic security features to program execution, across the whole architecture and software stacks.

Physical Unclonable Functions (PUFs) are devices which exploit the statistical distribution of hardware-intrinsic physical parameters to design functions capable of (uniquely) mapping a set of inputs (*challenges*) to outputs (*responses*) [?]. Built upon PUF theoretical models, several constructions of essential cryptographic primitives have been proposed, mainly to support key exchange [?, ?, ?], device authentication [?], intellectual property protection [?], oblivious transfer [?, ?] and commitment schemes [?]. The myriad of cryptographic primitives which could benefit from PUFs has driven the search for efficient real-world implementation of these devices.

Although silicon PUFs have gained a lot of attention, they are still under strong scrutiny, as they can undergo a number of attacks like: (1) reverse engineering [?], (2) characterization of the physical parameters [?], (3) modeling [?], and (4) emulation [?]. Even though there are still many concerns about the overall security of PUFs, their simplicity, low-power consumption and speed are very attractive design features for some application domains [?] (e.g. IoT devices). One of the potential applications of PUFs in IoT devices would enable integrity checking and authentication of program code and data. Yet very few works have addressed that using PUFs [?]. Thus additional research needs to be done in order not only to improve PUF security, but also to allow its integration into processor architecture and software stacks.

Recently, *Computer Security by Hardware-Intrinsic Authentication* (CSHIA) was presented in [?]. CSHIA proposes a new secure program execution model which employs a new PUF-based authentication mechanism aimed at ensuring code and data authenticity

for a given program/processor pair. Specifically, the system generates an authentication tag (called PTAG) to every instruction and data cache line at the very first moment that it runs in the processor. This authentication tag is later verified for integrity, ensuring that program instructions and data are not violated in runtime, and thus programs will execute correctly during the lifetime of the device. This work intents to extend the preliminary contributions in [?]. In particular, design a CSHIA FPGA prototype in conjunction with a security analysis and further evaluate its impact on performance and robustness .

1.1 Contributions

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus vitae iaculis erat. Aliquam tristique consectetur ante, quis commodo lacus egestas in. Nullam semper elit nec eros pretium dapibus. Ut eget porta metus. Mauris rhoncus vel magna non faucibus. Ut a ornare elit. Morbi sagittis quam nec risus laoreet, tincidunt volutpat ex venenatis. Sed ultrices felis quis felis scelerisque gravida a non neque. Etiam sed nisi neque. Ut lobortis pulvinar facilisis.

1.2 Publications

The contribuitions of this work were published in the following conferences

- Pub0 Publication 1 - description 1
- Pub1 Publication 2 - description 2

1.3 Organization of the dissertation

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus vitae iaculis erat. Aliquam tristique consectetur ante, quis commodo lacus egestas in. Nullam semper elit nec eros pretium dapibus. Ut eget porta metus. Mauris rhoncus vel magna non faucibus. Ut a ornare elit. Morbi sagittis quam nec risus laoreet, tincidunt volutpat ex venenatis. Sed ultrices felis quis felis scelerisque gravida a non neque. Etiam sed nisi neque. Ut lobortis pulvinar facilisis.

Chapter 2

Related Work

SoC devices have a number of features which distinguish them from other traditional electronic solutions. Their dedicated nature allows the adoption of more intrusive protection, while posing challenging energy and performance requirements. Unfortunately, traditional security solutions based on typical cryptographic mechanisms (e.g. bus encryption) can have a significant impact in device cost, energy efficiency and performance. One way to go around that is to consider approaches which enable a deep integration of device hardware-intrinsic features and program execution, as those offered by PUFs.

Qualitative analyses of PUFs have already been done in the literature [?] motivated by several applications such as cryptographic key generation [?, ?] and true random number generation [?, ?]. Unlike those works, which aim at evaluating the quality of a standalone PUF-inspired mechanism, this work focus on proposing and analyzing a PUF-based microarchitecture mechanism to enable secure code execution.

Most of the preliminary work on secure code execution aimed at keeping instructions and data secure from scrutiny, by using mechanisms like bus encryption. In [?], Elbaz *et al.* performed a comprehensive survey of bus encryption, where they describe many possible ways of using cryptographic algorithms in SoC architectures, so as to ensure that no malicious instruction/data would be executed by the CPU. The major shortcoming of these solutions is the usage of on-chip secret key storage in non-volatile memories which enable off-line key recovery attacks [?].

AEGIS, the secure processor proposed by Suh *et al.* in [?], employs PUFs as a cryptography primitive to uniquely authenticate code and data in order to prevent both software and physical attacks. They present a toolchain for developing secure software for their architecture which includes a secure operating system to manage different levels of memory protection. Although the presented toolchain does not require modifications in the processor architecture, it demands extensive changes in the SoC architecture, in addition to changes in the compiler and operating system. Moreover, AEGIS *does not* ensure full-time security from power-on to power-off; i.e. the system runs unprotected until the security kernel loads the system. In addition, physical attacks were neither evaluated nor simulated. Different circuits used in AEGIS, like PUFs and post-processing schemes for key extraction such as Fuzzy Extractors, have been successfully attacked with side-channel [?, ?] and semi-invasive attacks [?]. While semi-invasive attacks are hard to repeal, side-channel attacks have few known countermeasures [?] that can be easily adopted.

Chapter 3

Fundamental concepts

3.1 Physical Unclonable Functions - PUFs

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus vitae iaculis erat. Aliquam tristique consectetur ante, quis commodo lacus egestas in. Nullam semper elit nec eros pretium dapibus. Ut eget porta metus. Mauris rhoncus vel magna non faucibus. Ut a ornare elit. Morbi sagittis quam nec risus laoreet, tincidunt volutpat ex venenatis. Sed ultrices felis quis felis scelerisque gravida a non neque. Etiam sed nisi neque. Ut lobortis pulvinar facilisis.

3.1.1 Authenticity

Suppose that an attacker wants to add own code for execution in the embedded system or intends to move the data from one system instance to another. These attacks can be avoided by employing authentication mechanisms. In this solution, a key (or unique set of keys) is determined for each instance. Code data are tagged using these keys during manufacturing (an enrollment phase). On running time, this key (or set of keys) is used to regenerate tags. Only a correct key value will be able to verify what was installed during manufacture. Therefore, an instance will not accept code or data that was not tagged using its own keys.

Before the introduction of electronic Physical Unclonable Functions (PUFs) [**?**], these keys had to be inserted into systems before they were made available to customers. To do so, keys were stored on chip using non-volatile memories and the manufacturer/vendor controlled the uniqueness of the keys in each instance. The main downsides of storing key permanently include: facilitating physical attacks [**?**], and possibly increasing costs of production since it may demand integration of different technologies on the same chip.

Recently, PUFs have been employed to generated secret keys. PUFs are physical functions created to mimic random functions. Their inputs, called challenges, and outputs, called responses, are designed to have a unique relation for every PUF instance. This is achieved by leveraging on imperfections resulted from fabricating electronic devices. In regard to authenticity, the main advantage of using PUFs as key generators is that they can produce keys on running time, on-chip memories are not needed for key storage, and they are unclonable. That means that even the manufacturer itself cannot produce two PUF instances that will have the same the set of Challenge-Response Pairs () [**?**].

3.1.2 Integrity

Similarly to authentication, integrity is ensured by tagging code and data with additional information such as memory address location timestamps in general. This prevents an attacker from tampering with a system by, for instance, moving instructions from their location in memory, setting different initial values of variables, etc. The level of integrity can be done for an entire program, or memory pages, or memory blocks. That depends on the choice of designers.

Integrity can also be considered at the instruction sequence level, which we refer as Control-Flow Integrity (CFI). Hardware solutions for control-flow integrity usually require deep integration between hardware and software [**?**], that can result not only in changing

the Instruction Set Architecture () the tool-chain, but also the processor's data path, as proposed in [?, ?]. Even though the CFI protection is welcomed, due to the focused nature of embedded systems, many applications cannot afford the performance penalties and storage overhead inherently of this solution. For instance, in applications where user inputs is limited and involves fixed amounts of data, an attacker has very little room to employ a buffer overflow or similar attacks prevented by CFI. However, integrity verification regarding blocks of code and data (as mentioned above) can avoid a variety of situations that go beyond runtime attacks. For example, if an embedded system is unwatched, an attacker can upload a malicious code or modify the data in the external memory even if the system is not running. Integrity verification can prevent and indicate these violations before they reach the processor.

3.1.3 Secrecy

An embedded system can also use encryption to prevent exposure of code data stored in the external memory. Consequently, the processor can process these instructions and data only after decryption. Therefore, the major drawback on using encryption is the performance overhead that highly depends on which cryptographic primitive is employed. In addition, secrecy only prevents that an attacker obtains the information, if it is not combined with a unique key or integrity verification, the system will be vulnerable to execute code of different system instances to suffer relocation and replay attacks.

Chapter 4

CSHIA Architecture

CSHIA, illustrated in Figure 4.1, is a processor architecture which aims at providing secure code execution by means of PUF-based authentication of cache lines. The central idea behind CSHIA is a PUF-Tag (PTAG) Memory, which runs in parallel with the system main memory (Figure 4.1). Each entry in the PTAG Memory stores an authentication code of a cache line generated by a PUF-based device located on-chip.

Figure 4.1: A system overview of the CSHIA system.

In comparison to traditional architectures, CSHIA includes two main modifications: The *Secure Engine* (SEC-ENG), which includes the *PTAG Generator* (PTAG-GEN, Figure 4.2); and the *Security-Cache* (SEC-CACHE) that controls bus traffic between the processor and the *Memory Controller* (MCTRL). Other two new architectural components are also required to complete the CSHIA design, the *PTAG Memory* and the *PTAG Bus*. In a few words, when the processor requires/sends data/instructions to the MCTRL, the SEC-CACHE sends the related cache line to the SEC-ENG for computing/validating its PTAG. Notice from Figure 4.1 that the PTAG bus runs in parallel to the system buses, and thus no program can directly read the PTAG Memory, since neither the processor nor the MCTRL are aware about the SEC-CACHE.

4.0.1 PTAG-GEN Operation

The SEC-ENG controls the PTAG-GEN based on the information delivered by the SEC-CACHE. This information is generated from bus transactions (Memory READ, Memory WRITE and I/O) between the processor and the memory controller, and which the SEC-CACHE controls. Next, each PTAG-GEN action is explained in regard to bus transactions.

Figure 4.2: The PTAG-GEN during PTAG Generation (write) and PTAG Verification (read) operations.

PTAG Generation (memory write)

During a write operation, the SEC-CACHE passes data/instruction cache lines to the SEC-ENG and the PTAG-GEN computes PTAGs and stores it into the PTAG Memory. A *Pseudorandom Function* (PRF) [**?**] module is used to generate the PTAG and takes as input the concatenation (||) of the cache line bits and the base address of the cache line provided by the core (see Figure 4.2). In order to ensure uniqueness, the PRF is configured using a *unique-per-device key*. This key is produced by the intrinsic hardware features of a PUF. Such authentication tag is specific to the core running that specific cache line, as PUF outputs are dependent on the statistical variations of the manufacturing process, and are unique to each processor [**?**]. Hence identical cache lines running on different processors will produce different PTAG values for the same inputs. Notice that only code in the cache, for which integrity has been ensured, will be able to write to memory.

PTAG Verification (memory read)

During a read operation, the SEC-CACHE passes data/instruction cache lines to the SEC-ENG and the PTAG-GEN computes PTAGs for verification. As shown in Figure 4.2, during a read operation the cache line base address produced by the core is appended to the cache line contents read from memory and the result is fed to the PRF module. The PTAG produced this way is compared to the PTAG read from memory for equality. If the previously stored PTAG and the recently computed value do not match, a *Non-Maskable Interrupt* (NMI) is generated to the core (called PTAG-NMI), as code/data integrity may have been violated. As shown in Figure 4.1, in order to hide PUF latency, the data/ instruction is sent to the respective cache (I$ or D$) at the same time that PTAG-GEN computes the PTAG for that cache line and compares it to its PTAG previously stored into the PTAG Memory.

Handling I/O

In modern computer systems, I/O operations store data directly into specific memory regions through DMA mechanism. Thus, it is not possible to trust such memory regions and CSHIA does not ensure their integrity and authenticity. Software should first perform authentication of I/O data in a higher abstraction layer and then copy it to secure areas where the CSHIA can ensure integrity and authenticity.

Chapter 5

CSHIA Prototype

Figure 5.1: The prototype architecture.

The prototype will be implemented upon a Leon 3 SPARC V8 platform from Aeroflex Gaisler [?] on an Altera DE2-115 Development Kit. To implement CSHIA's pre design (Figure 4.1), the design of two blocks are planned: the Security Engine (SEC-ENG) and the Security Cache (SEC-CACHE) , to be inserted between the processor and the memory controller as shown in Figure 5.1. The SEC-CACHE will control bus transactions between the processor and the MCTRL, and provide data to the SEC-ENG. Consequently, the SEC-ENG will control the fuzzy extractor and the PTAG-GEN. To use CSHIA's fuzzy extractor a (127, 64, 10)-BCH code instance for error correction and an internal memory that emulates a SPUFwill be used. The PTAG-GEN will use a SIPHASH-2-4 for PTAG generation and verification.

5.0.1 Prototype Configuration

Since the processor may ask for an arbitrary number of words from memory, and the architecture needs to check for the integrity of a full memory block in every cache miss, a buffer in SEC-CACHEwill be used to hold isolated memory words required by the processor. When the processor demands a memory word, the buffer controller will request all the other words from the main memory to fit one PTAGblock. That will allow CSHIA to authenticate entire memory blocks and,with the proper memory size, can also speedup sequential requests of the processor. The buffer will be configurable to fit a variable PTAG block.

Chapter 6

CSHIA Evaluation

For fault injection and performance evaluation, we will use Gaisler's debugging interface GRMON which is a general debug monitor for the LEON processor, and for SOC designs based on the GRLIB IP library. GRMON includes the following functions:

- Read/write access to all system registers and memory
- Built-in dis-assembler and trace buffer management
- Downloading and execution of LEON applications
- Breakpoint and watchpoint management
- Remote connection to GNU debugger (GDB)
- Support for USB, JTAG, RS232, PCI, Ethernet and SpaceWire debug links

6.0.1 GRMON Debug Monitor

The GRMON debug monitor is intended to debug system-on-chip (SOC) designs based on the LEON processor. The monitor connects to a dedicated debug interface on the target hardware, through which it can perform read and write cycles on the on-chip bus (AHB). The debug interface can be of various types: the LEON2 processor supports debugging over a serial UART and 32-bit PCI, while LEON3 also supports JTAG, ethernet and spacewire debug interfaces. On the target system, all debug interfaces are realized as AHB masters with the debug protocol implemented in hardware. There is thus no software support necessary to debug a LEON system, and a target system does in fact not even need to have a processor present.

GRMON can operate in two modes: command-line mode and GDB mode. In command-line mode, GRMON commands are entered manually through a terminal window. In GDB mode, GRMON acts as a GDB gateway and translates the GDB extended-remote protocol to debug commands on the target system. GRMON is implemented using three functional layers: command layer, debug driver layer, and debug interface layer. The command layer consist of a general command parser which implements commands that are independent of the used debug interface or target system. These commands include program downloading and flash programming. The debug driver layer implements custom commands which are related to the configuration of the target system. GRMON scans the target system at start up, and detects which IP cores are present and how they are con- figured. For each supported IP core, a debug driver is enabled which implements additional debug commands for the specific core. Such commands can consist of memory detection routines for memory controllers, or program debug commands for the LEON processors. The debug interface layer implements the debug link protocol for each supported debug interface. The protocol depends on which interface is used, but provides a uniform read/write interface to the upper layers. Which interface to use for a debug session is specified through command-line options during the start of GRMON.

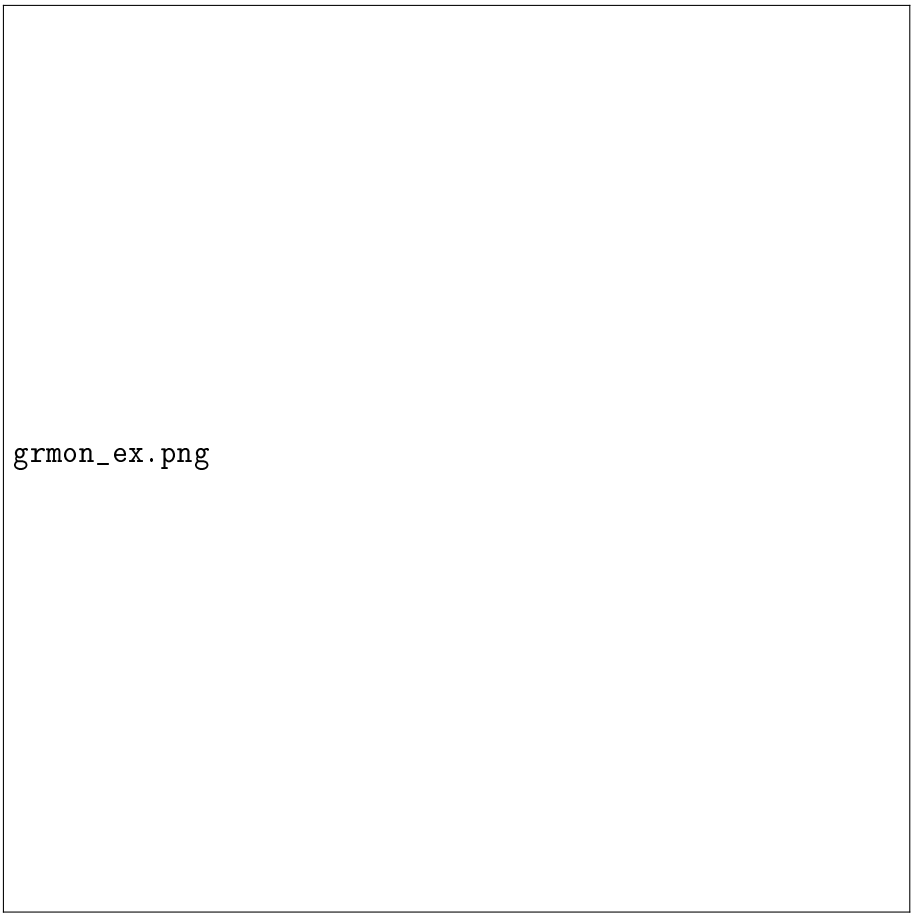

Figure 6.1: GRMON Interface

6.0.2 Performance Analysis

For correctness, the CSHIA prototype will be evaluated using 8 benchmarks from the MiBench suite: `basicmath`; `bitcount`; `crc32`; `dijkstra`; `fft`; `qsort`; `sha`; `stringsearch`. Benchmark outputs will be compared to the outputs of reference execution of MiBench and also to those produced by the Leon baseline implementation. In addition, program performance will be compared among three architectures: (1) Unmodified Leon 3 (baseline); (2) CSHIA (unsecured), whose authentication and integrity verification will be disabled by making the bus traffic bypass the SEC-CACHE; and (3) CSHIA (secure), the architecture proposed herein. All architectures will use a 50-MHz clock, 16-KB L1 data and instruction caches with 256-bit cache lines.

As Leon 3 (baseline), and consequently both CSHIA (unsecured) and CSHIA (secure), do not support system calls, benchmarks that read files will be modified to obtain their data from hard-coded integer or string vectors. Large MiBench inputs will be used for all benchmarks.

6.0.3 Fault Injection Attack

Many attack scenarios proposed in [**?**] can be used to evaluate the CSHIAprototype, in most of them the attack is performed by inserting a modified memory block into the bus

or main memory. Thus, verifying how those insertions can happen and how they will affect the behavior of CSHIA is crucial. This section presents an execution of a fault injection attack planned to be executed in the real prototype.

[language=c, tabsize=1, numbers=right, numbersep=-5pt, firstline=126, lastline=135, basicstyle=]sha.c

Figure 6.2: C code for the `sha` initialization function.

[language=[x86masm]Assembler, tabsize=1, numbers=right, numbersep=-5pt, firstline=2055, lastline=2056, basicstyle=]sha.objdump

Figure 6.3: SPARC assembly code where the constant `0x67452301` is assigned to a global register.

The attack scenario is the following. An attacker wants to tamper with the message integrity scheme of a node from a sensor network. The Secure Hash Algorithm (SHA) is used in this network to create a *Message Authentication Code* (MAC) for incoming and outgoing messages. The goal of the attacker is to make a particular node reject all incoming messages with valid MACs and generate invalid MACs for outgoing messages that all other nodes will reject. This attack may be hard to detect since all nodes will be active and promptly responding, but the network is malfunctioning. For fault injection, we will use Gaisler's debugging interface, GRMON, to directly insert memory words into the AMBA bus. Faults will be inserted as bus memory write operations that can be executed during the runtime of a program, after a breakpoint. An attacker could easily achieve that by connecting a physical adapter to the external pins of the processor. As implementations of SHA are open, we assume the attacker knows all the source code and decides to tamper with the initialization process, in which constant values are assigned, as shown in Figure 6.2. The attacker then learns that line 3 from Figure 6.2 is compiled to the two instructions in Figure 6.3. Given that, he/she replaces the memory word `05 19 d1 48`, a `sethi` instruction, by `05 11 11 11`, which represents the substitution of constant `0x67452301` by `0x44444701`. After applying the attack, all digest values will differ from those generated by an authentic implementation of SHA. This initial scenario will be used to evaluate the CSHIAprototype.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

7.2 Future Work