Universidade Estadual de Campinas
Instituto de Computação

Caio Hoffman

Computer Security by Hardware-Intrinsic Authentication

Segurança de Computadores por meio de autenticação intrínseca de Hardware

CAMPINAS
2019

Caio Hoffman

Computer Security by Hardware-Intrinsic Authentication

Segurança de Computadores por meio de autenticação intrínseca de Hardware

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

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

Supervisor/Orientador: Prof. Dr. Guido Costa Sousa de Araújo
Co-supervisor/Coorientador: Prof. Dr. Mario Lúcio Côrtes, Prof. Dr. Diego de Freitas Aranha

Este exemplar corresponde à versão final da Tese defendida por Caio Hoffman e orientada pelo Prof. Dr. Guido Costa Sousa de Araújo.

CAMPINAS
2019

Agência(s) de fomento e nº(s) de processo(s): FAPESP, 2015/06829-2; FAPESP, 2016/25532-3; CNPq, 147614/2014-7; CNPq 200362/2016-0
ORCID: https://orcid.org/0000-0001-7213-9528

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

H675c

Hoffman, Caio, 1983-
Computer security by hardware-intrinsic authentication / Caio Hoffman. – Campinas, SP : [s.n.], 2019.

Orientador: Guido Costa Souza de Araújo.
Coorientadores: Mario Lúcio Côrtes e Diego de Freitas Aranha.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Sistemas de segurança. 3. Criptografia de dados (Computação). 4. Hardware - Medidas de segurança. 5. Funções físicas inclonáveis. I. Araújo, Guido Costa Souza de, 1962-. II. Côrtes, Mario Lúcio, 1950-. III. Aranha, Diego de Freitas,1982-. IV. Universidade Estadual de Campinas. Instituto de Computação. V. Título.

Informações para Biblioteca Digital

Título em outro idioma: Segurança de computadores por meio de autenticação intrínseca de hardware
Palavras-chave em inglês:
Computer architecture
Security systems
Data encryption (Computer science)
Hardware - Safety measures
Physical unclonable functions
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Marcos Antonio Simplicio Junior
Wang Jiang Chau
Julio César López Hernández
Lucas Francisco Wanner
Data de defesa: 16-01-2019
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Caio Hoffman

Computer Security by Hardware-Intrinsic Authentication

Segurança de Computadores por meio de autenticação intrínseca de Hardware

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo
  Instituto de Computação
  Universidade Estadual de Campinas
- Prof. Dr. Marcos Antonio Simplicio Junior
  Escola Politécnica
  Universidade de São Paulo
- Prof. Dr. Wang Jiang Chau
  Escola Politécnica
  Universidade de São Paulo
- Prof. Dr. Julio César López Hernández
  Instituto de Computação
  Universidade Estadual de Campinas
- Prof. Dr. Lucas Francisco Wanner
  Instituto de Computação
  Universidade Estadual de Campinas

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 16 de janeiro de 2019

To my family.

In order to seek truth, it is necessary once in the course of our life, to doubt, as far as possible, of all things.

(René Descartes)

Acknowledgements

First of all, I would like to thank FAPESP for the doctoral scholarship (File N°2015/06829-2) and the research internship abroad scholarship (File N°2016/25532-3). I also thank CNPQ (Files N°147614/2014-7 and N°200362/2016-0), Intel's Research grant "Energy-Efficient Security for SoC Devices – Physical Unclonable Function", and CAPES for their funds and scholarships.

I thank Professor Guido for having constantly challenged me in pursuing excellence in my work during all these years. I also thank him for having always promptly helped me on many problems, mainly, those regarding my scholarships. Finally, I would like to thank him for all opportunities that he opened to me and which allowed me to grow as a student and researcher.

I thank Professor Mario for his constant support in the development of this work. He had always been patient in my mistakes, including those I insisted in keeping doing over the time. Yet, I could always ask for help when I needed.

I thank Professor Diego for his technical support in security and research. I also thank him for having indicated me to the Intel's scholarship, which made possible that I went to Canada for the first period of internship.

I thank Professor Catherine Helen Gebotys for having me twice as visiting student in her laboratory at the University of Waterloo, Canada. I also thank her for welcoming, helping, and supporting my experiments, giving me freedom to pursue my goals.

I thank Augusto F. R. Queiroz for helping me to implement this work. He laid down the basis of the implementation of the architecture which helped me immensely and allowed me to finish this work.

I thank my friends and colleagues in Waterloo, Mustafa Faraj, Mahmoud Khalafalla, Haohao Liao, Karim Amin, for their friendship and help.

I would like thank the entire staff of the Institute of Computing, for all their help during all these years. In particular, I would like to give a special thank to Cristiane Marques Ortiz de Camargo, Denilson Ferreira Alves, and Wilson Bagni Junior.

I thank all colleagues in the LSC that always helped when I needed.

Finally, I thank God and my family for everything.

Resumo

Neste trabalho apresentamos Computer Security by Hardware-Intrinsic Authentication (CSHIA), uma arquitetura de computadores segura para sistemas embarcados que tem como objetivo prover autenticidade e integridade para código e dados. Este trabalho está divido em três fases: Projeto da Arquitetura, sua Implementação, e sua Avaliação de Segurança. Durante a fase de projeto, determinamos como integridade e autenticidade seriam garantidas através do uso de Funções Fisicamente Não Clonáveis (PUFs) e propusemos um algoritmo de extração de chaves criptográficas de memórias cache de processadores. Durante a implementação, flexibilizamos o projeto da arquitetura para fornecer diferentes possibilidades de configurações sem comprometimento da segurança. Então, avaliamos seu desempenho levando em consideração o incremento em área de chip, aumento de consumo de energia e memória adicional para diferentes configurações. Por fim, analisamos a segurança de PUFs e desenvolvemos um novo ataque de canal lateral que circunvê a propriedade de unicidade de PUFs por meio de seus elementos de construção.

Abstract

This work presents Computer Security by Hardware-Intrinsic Authentication (CSHIA), a secure computer architecture for embedded systems that aims at providing authenticity and integrity for code and data. The work encompassed three phases: Design, Implementation, and Security Evaluation. In design, we laid out the basic ideas behind CSHIA, namely, how integrity and authenticity are employed through the use of Physical Unclonable Functions (PUFs), and we proposed an algorithm to extract cryptographic keys from the intrinsic memories of processors. In implementation, we made CSHIA's design more flexible, allowing different configurations without compromising security. Then, we evaluated CSHIA's performance and overheads, such as area, energy, and memory, for multiple configurations. Finally, we evaluated security of PUFs, which led us to develop a new side-channel-based attack that enabled us to circumvent PUFs' uniqueness property through their architectural elements.

List of Figures

List of Tables

List of Symbols and Abbreviations

ALAP

As Late As Possible.

BCH Code

Bose–Chaudhuri–Hocquenghem Code.

BL

Bit Line.

CPA

Correlation Power Analysis.

CRP

Challenge-Response Pair.

CSHIA

Computer Security by Hardware-Intrinsic Authentication.

DPA

Differential Power Analysis.

ECC

Error Correcting Code or Error Correction Code.

FE

Fuzzy Extractor.

HW

Hamming Weight.

HD

Hamming Distance.

IBS

Index-Based Syndrome.

IoT

Internet of Things.

LRU

Least Recently Used.

ML

Machine Learning.

MB

Memory Block.

MT

Merkle Tree.

PRF

Pseudo-Random Function.

PUF

Physical Unclonable Function.

PTAG

PUF-based Tag.

SPUF

SRAM-PUF.

TS

Timestamp.

WL

Word Line.

$\oplus$

Bitwise XOR operator.

$||$

Concatenation operator to bit strings.

Contents

Chapter 1

Introduction

The upcoming large connectivity among different systems in the so-called Internet of Things (IoT) [21] will expand the current range of billions of Embedded System devices [62] to a scale yet unseen. Fridges, bicycles, light switches, and more [6], will have embedded capabilities of processing information and transfer it through the Internet. Although security in Embedded Systems has been a concern for a long time, the expected volume of devices intercommunicating in the IoT poses more complex challenges to security. Among these challenges, data authentication [60] and trust [37] will have to be addressed integrating local and network security. Network security has been constantly improved over decades of research and technological development, and despite requiring further improvements, it has developed technologies that are well established and allow secure data transfer, privacy, device authentication, etc. Nonetheless, current network security mostly aims at issues related to human users, while for the future, it is expected that autonomous and smart devices will be the dominating entities in the Internet. Therefore a major conundrum will be how to ensure authenticity and authentic behavior of devices to enable network security.

Enforcing authenticity and authentic behavior can be a role played by local security, yielding a layer of trust that network security can leverage on. Thus, local security needs to ensure that adversaries cannot tamper with systems' behavior and identity, even if he/she has physical access to them. In that regard, physical access not only includes physical or visual contact, but also the capability of provoking physical interference or capturing side channel information of systems at a small distance. Hence, threats against local security encompass a variety of physical attacks to Embedded Systems. For instance, a resourceful adversary can: capture electromagnetic information [35], decapsulate components and collect photonic emission [22], and even change physical behavior of components by applying laser [54], among other different non-invasive, semi-invasive, and invasive attacks [66]. Additionally, a less resourceful adversary can: tamper with systems by connecting his/her own device to buses and memories, modifying memory through controlling operating system, replacing physical memory with a tampered one, among other attacks.

To fully physically protect Embedded Systems, two major concepts need to be incorporated into their design: a robust fabrication processes and a secure architectural design. A robust fabrication process will hinder attacks of a resourceful adversary. However, that may involve technological advances that are yet to be discovered. Moreover, this protec-

tion alone does not enable data authentication and trust. Therefore, secure architectural design is the essential part of local security. Only a secure architectural design can yield data authentication and trust. In order to do that, device-intrinsic identification stamps can be used to attest data origin and code *authenticity*, and hardware-based mechanisms can guarantee *integrity* for code and data. These features implemented by an architectural design not only tackle those threats of less resourceful adversaries, but also enable verification of authenticity and authentic behavior of Embedded Systems, which is important for multiple kinds of applications, even for those that will not be integrating the Internet of Things.

The literature abounds with secure architectural designs [63, 52, 43, 58, 26] that aim at providing local security. However, many of these works have some relevant drawbacks: they overlook systems with less processing power and/or impose a large number of design modifications, from instruction sets to operating systems, requiring sometimes the redesign of the whole tool-chain [63, 52]. Some works do not handle authenticity [26], while others deal with device-intrinsic identification, but they do not take into consideration some important practical limitations of that [43, 58].

In the face of the requirements of local security, we present Computer Security by Hardware-Intrinsic Authentication (CSHIA). CSHIA is a secure architecture that leverages on Physical Unclonable Functions (PUFs) to extract unique information, like a hardware fingerprint, that is used to ensure that code and data of embedded systems will be kept as designed during system lifetime. CSHIA ensures that violation of the authenticity and integrity of code and data will be immediately detected, preventing corrupted systems from doing further actions. Moreover, CSHIA combines all that with flexible design parameters, reduced overheads, and transparent integration to processor architectures, thus avoiding modifications on peripherals and software tool-chains.

1.1 General View of This Work

The use of PUFs in security applications has been explored for more than one decade now [13]. A physical system that responds to a stimulus, called challenge, can be seen as a PUF if it has 4 properties: (a) it behaves as a *function*; (b) it is consistent over time (*robustness*); (c) its responses are *unpredictable*; (d) no other system replicates its behavior (*uniqueness*). In electronic devices, we have a particular class of PUFs: Silicon PUFs (or Electronic PUFs). In those, a binary input is the challenge, and a binary output is the response. Due to properties (a), (b), (c), and (d), binary outputs of PUFs can be used in multiple applications from hardware fingerprints to cryptographic keys [13]. CSHIA extracts from PUFs a unique binary output that is used as a cryptographic key in a Pseudo-Random Function (PRF) to generate authentication tags. An authentication tag is created for each memory block from main memory during an enrollment procedure. On runtime, CSHIA recreates these tags and verifies them against the original ones, looking for tampered memory blocks and tags. If the verification proves that both memory block and tag are authentic, execution proceeds. Otherwise, all further operations are stalled.

Therefore, CSHIA verification ensures integrity of code and data. While the tag cre-

ation using a PUF-based key gives uniqueness to every tag. The uniqueness ensures that code and data belong to a specific instance and they were originally placed there by the manufacturer/vendor during the enrollment procedure. So, CSHIA yields to Embedded Systems guarantees of authenticity and authentic behavior.

1.2 Work Relevance

Inhibiting physical attacks on Embedded Systems in the IoT age will be crucial, mainly because these devices will operate autonomously and unsupervised, allowing attackers to do on-field exploration of vulnerabilities. Consequently, code and data will be at risk of being tampered with. Thus, CSHIA's goal is to complement full physical protection when robust fabrication processes are provided, seeking to ensure authenticity and authentic behavior of an Embedded System. What is unique about CSHIA, in comparison to other works, is its endeavour towards robust use of PUFs. Although PUFs would need to have properties (a), (b), (c), and (d), physical implementations constantly fail in (a) and (b), generating a reliability problem. This work tackles this problem intensively. It not only analyzes security downsides of the problem, but also presents analyses that show how to extract stable keys from PUFs, in a complete integrated way to the architecture design.

Additionally, CSHIA architecture presents advantageous design decisions that include a dedicated bus and memory for authentication tags. This additional memory that has an isolated bus enables designers to choose many parameters such as bandwidth, clock frequency, among others, that can reduce performance overhead. The CSHIA design can deal with replay attacks, a class of attacks that violates data integrity, in different forms, allowing designers to take into account design constraints and, therefore, consider pros and cons of each replay attack solution in their system. All that can be integrated to Embedded Systems in a transparent way, avoiding modification in peripherals, operating systems, compilers, programs, etc.

Over the course of this work, while pursuing improvements in the use of PUFs in CSHIA, we developed a new threat to PUFs. Using a known technique called Template Attack [14], we present an attack that circumvents property (d) of PUFs (uniqueness). We show that in a particular PUF, called XOR Arbiter PUF, we were able to reveal unknown Challenge-Response Pairs (CRPs) of one PUF instance using information from another one. That is a relevant finding that increases awareness of security issues not only for current systems that already employ PUFs, but also for new devices that plan to join the Internet of Things. With multiple instances of the same device working in the IoT environment, using one device instance to reveal intrinsic information from other instances is a serious threat, particularly when this information is related to the system identity and authenticity, as in CSHIA's case.

All that is presented in this work in a collection of papers that were either published or have been submitted to publication. These works are described in the next section.

1.3 Publications

The papers presented in this work are pre-printed versions or are under review. Therefore, there is no copyright infringement as stated in [11]. The papers are:

1. *Computer Security by Hardware-Intrinsic Authentication.* Authors: Caio Hoffman, Mario Côrtes, Diego F. Aranha, Guido Araujo. Published at CODES+ISSS 2015: International Conference on Hardware/Software Codesign and System Synthesis. DOI: 10.1109/CODESISSS.2015.7331377 [25].

2. *Implementing a Secure Architecture for Code and Data Authenticity and Integrity in Embedded Systems.* Authors: Caio Hoffman, Augusto F. R. Queiroz, Diego F. Aranha, Mario L. Côrtes, Guido Araujo. Submitted to Microprocessors and Microsystems, Journal, Elsevier. Submission Date: 26/Aug/2018. Submission Code: MICPRO_2018_332.

3. *Applying Template Attacks on XOR Arbiter PUFs.* Authors: Caio Hoffman, Catherine H. Gebotys, Diego F. Aranha, Mario Côrtes, Guido Araujo. Submitted to DAC 2019: Design Automation Conference. Submission Date: 27/Nov/2018. Submission Code: 267-VL649.

In addition to the above papers, this work also produced a patent that was deposited in the National Institute of Intellectual Property (INPI) of Brazil:

1. Secure Architecture for Embedded Systems. Authors: Guido Costa Souza de Araújo, Mario Lúcio Côrtes, Caio Hoffman. Deposited in Brazil: 14/Jul/2015. Brazilian Code: BR1020150168314. International Deposit: 19/Jan/2017. Code: WO2017008133.

To refer to the papers we chose the following simplification:

- *Computer Security by Hardware-Intrinsic Authentication* = "*CSHIA Design*".
- *Implementing a Secure Architecture for Code and Data Authenticity and Integrity in Embedded Systems* = "*CSHIA Implementation*".
- *Applying Template Attacks on XOR Arbiter PUFs* = "*PUF Attack*".

1.4 Contributions

There are four major contributions of this work. They are, in order of importance:

1. A novel secure architecture for embedded systems that comprises:
    - A deep integration with PUFs that takes into account their limitations and characteristics, such as entropy and instability.
    - An additional memory for authentication tags that has a dedicated bus, which enables designers to choose parameters like bandwidth, clock frequency, among others.

- A flexible design that allows designers to consider pros and cons of two different countermeasure against replay attacks, enabling them to choose the best option for their system.
- A seamless integration to modern embedded systems' design flow that avoids modifications in peripherals, operating systems, compilers, programs, etc.

2. The security engine implementation and integration to the proof-of-concept prototype of CSHIA.

3. A new attack on XOR Arbiter PUFs that shows that is possible to circumvent their uniqueness property.

4. An algorithm that extracts stable cryptography keys from SRAM-PUFs.

Item 1 is presented in "*CSHIA Design*" and "*CSHIA Implementation*" papers. The CSHIA prototype was partially developed and implemented by this work. The proof-of-concept prototype, which comprises our contribution to Item 2, is presented in "*CSHIA Implementation*". "*PUF Attack*" encompasses Item 3 that was result of two periods (total 14 months) of internship as Visiting Graduate Student at the University of Waterloo, under the supervision of Professor Catherine Helen Gebotys. Finally, Item 4 is presented in "*CSHIA Design*", and additional information about it is found in Appendices A and B.

1.5 Organization

This dissertation is organized as follows: in the following subsections of this chapter, we provide necessary background information to the topics of this work. Chapter 2 presents our papers and a reading roadmap. A discussion about the strengths and weaknesses of the work, the evolution of CSHIA over time, and how the papers are interconnected can be found in Chapter 3. Chapter 4 finishes this work and sheds light on future research directions.

1.6 Physical Unclonable Functions

One of the biggest challenges in security for many years has been how to safely store secret keys. Before Physical Unclonable Functions (PUFs), these keys were kept in off-chip or on-chip non-volatile memories. Over the time, off-chip memories were shown to weaken security of systems since attackers could dump the content of these memories with simple equipment (see Figure 1.1), and thus obtain the secret keys.

On-chip memories, however, have been more difficult to attack because they lie inside the main chip of the system, which usually is the one that performs cryptographic operations. Consequently, without being able to forcedly put memory content in the output pins of the chip, attackers would not have access to what is internally stored. Then again, an attacker capable of decapsulating chips and capturing photonic emission would easily get all information stored in these memories through the emissions produced when they are powered on [49].

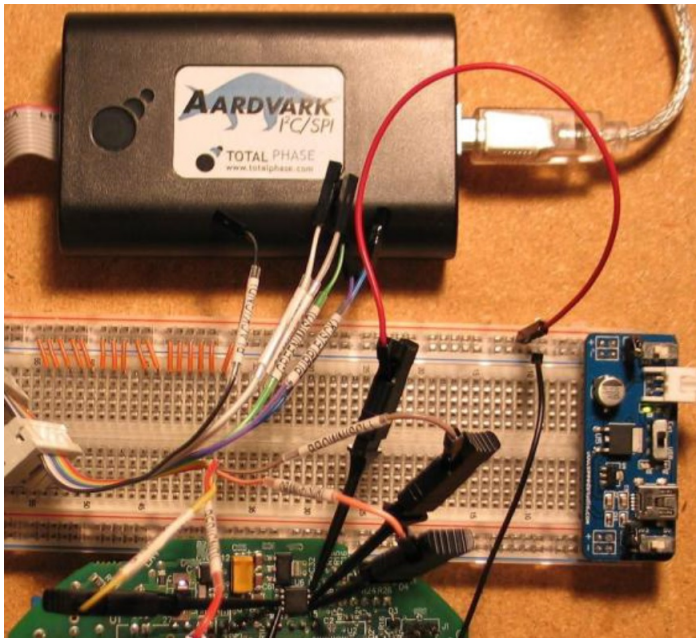


Figure 1.1: Dumping data from an EEPROM of a smart meter using Total Phase Aardvark. Photo originally displayed in [59].

Another security flaw one may see in storing secret keys in non-volatile memory regards to key generation. If someone chooses such keys, they can be biased, facilitating key deduction by attackers. On the other hand, if a vendor/manufacturer is not using a true random number generator to create keys, an attacker that discovers the underlying algorithm of key generation can break all systems that this vendor/manufacturer is distributing. Notice that it is a very common situation in software in which crackers provide their own generators that produce product keys which softwares accept as authentic.

On pursuing robust solutions for security, Silicon (or Electronic) Physical Unclonable Functions (PUFs) were proposed as highly integrative cryptographic primitives to computer system design [20]. The concept of PUFs is quite general though. Any physical system that responds to stimuli can be a candidate to be a PUF. However, systems must meet two criteria. First, a specific stimulus, which we denominate *challenge* in PUF terminology, cannot result in more than one response value. This property is basically the definition of *function*. Second, the set of all pairs stimulus-response that defines the physical system cannot be reproduced, even if we intentionally build another one with that purpose. Making these systems *unclonable*. Notice that every physical element that is a building block of physical systems is imperfect at the molecular or atomic level and, therefore, leveraging on these imperfections to generate responses to stimuli make these systems impossible to replicate.

An important aspect one might observe in PUFs is that finding a function (or its inverse) that maps challenges to responses is not trivial. Mainly because their physical construction involves a multitude of parameters that hinders the creation of predictive models of responses, which results in *unpredictability*. All these features of PUFs are desirable characteristics in cryptographic primitives for electronic systems, as they can be employed to create secret keys, uniquely identify chips, generate random numbers,

among other uses. The myriad of possible applications of PUFs led researchers not only to propose new electronic circuit designs as PUFs [51], but also to rediscover well-known circuits, like SRAMs, as interesting PUFs [32].

In the following subsections, we explore three different PUFs design that are extensively used in this work.

1.6.1 Arbiter PUF

The Arbiter PUF (APUF) is probably the most popular PUF design (see Figure 1.2). To build an APUF, one uses a flip-flop or latch and multiple pairs of multiplexers. Each pair of multiplexers is a stage. Each challenge bit is an input for a stage and it is applied to both multiplexers. The bit chooses which multiplexer output of the previous stage is the input signal of the current one. Figure 1.2 illustrates a Δ signal that is applied at the beginning of the APUF circuit and then divided into upper and lower signals[1]. At each stage, a challenge bit `0` keeps the lower and upper signal traveling through the respective lower and upper multiplexers. However, a challenge bit `1` sends the upper signal to the lower multiplexer and the lower signal to the upper multiplexer. Due to imperfections of the fabrication process, multiplexers and interconnection wires introduce delays into the signals, making them arrive at the flip-flop in different times. If the upper signal arrives first, the lower signal will lock it, resulting in $\texttt{Q} \rightarrow \texttt{1}$. Otherwise, the lower signal arrives first and locks the flip-flop, inhibiting the upper signal to be stored, resulting in $\texttt{Q} \rightarrow \texttt{0}$. Therefore, APUF leverages on intrinsic delays of its elements to procedure responses. Every challenge results in a unique pair of paths that is hard to predict through which one the signal delta arrives first. For this reason, APUFs are classified as delay-based PUFs.

Surprisingly, APUFs were found to be easy to model using Machine Learning (ML) [46]. A large set of Challenge-Response Pairs (CRPs) enables learning algorithms to figure out the intrinsic delays of each APUF stage. Consequently, that allows these algorithms to predict with very high accuracy[2] the final delay of every possible path, thereby determining responses to any challenge. Searching for PUF architectures resistant to modeling, researchers focused on developing complex constructions derived from APUF, such as the XOR Arbiter PUF, which we discuss next.

1.6.2 XOR Arbiter PUF

XOR Arbiter PUFs (XOR-APUFs) are a composition of multiple APUFs, as one can see in Figure 1.3. In the figure, a challenge is simultaneously applied to all APUFs, and then their responses are combined into one by a XOR operation. What is notorious about the XOR-APUF construction is that it only works due to the unclonability property of PUFs. As it is impossible to build two PUFs with the same set of CRPs, each APUF will generate unique and independent responses to every challenge. The combination of all individual responses using XOR results in a non-linear relation between challenge and

[1] We will use the terms upper and lower for the sake of simplicity.

[2] Accuracy is define by the sum of true positives and true negatives divided by the number of samples. For PUFs, that would be the correct prediction of ones and zeros by the total number of challenges used.

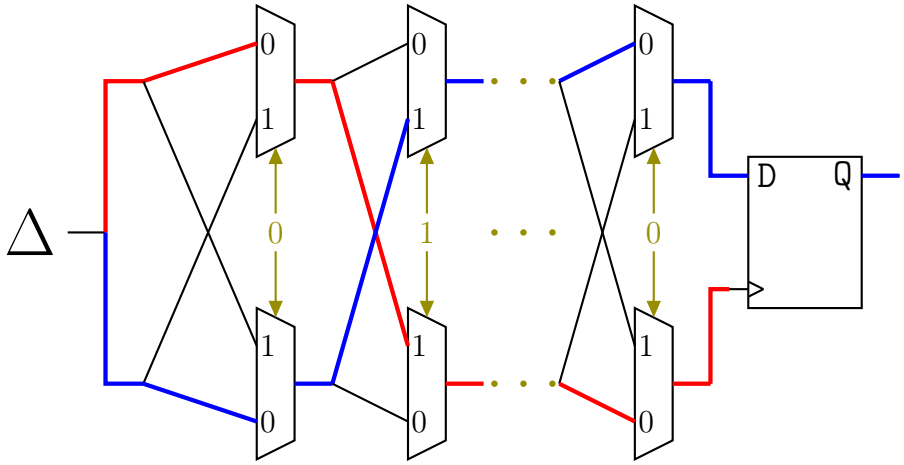


Figure 1.2: A n-bit challenge Arbiter PUF with signal race.

response that hinders modeling. As Rührmair *et al.* presents in [46], to train a logistic regression model for a XOR-APUF with five 64-bit challenge APUFs, they used 80000 CRPs and 2:08 hours of computation. However, to model a XOR-APUF with six 64-bit challenge APUFs they used 200000 CRPs and 31:01 hours of computation. That is a very significant increment of time complexity. Yet, the work of Becker in [9] convinced many researchers that XOR-APUFs were broken. Using real implementations of XOR-APUFs with large number of APUFs (up to 32), Becker achieved high accuracy on predicting responses of all PUFs evaluated. For instance, he achieved about 90 % of prediction accuracy in a XOR APUF model with 16 APUF, using over 500000 CRPs and 30 hours of training.

Then again, a very relevant point the reader should notice in Becker's attack is the enormous number of CRPs. In many applications, an attacker will not have access to such a large amount of information. For instance, a system that uses a very few challenges to generate PUF-based keys will probably not be threatened for these machine learning attacks, since responses are unlikely to be available and a small amount of CRPs hinders model accuracy. Another important point is that every model uniquely corresponds to a single instance of a PUF circuit. Other circuits will have different CRPs and demand new training.

1.6.3 SRAM-PUF

Static Random-Access Memory (SRAM) have already been validated in regard to the fundamental properties that define PUFs [30]. SRAMs are constructed of cells like the one illustrated in Figure 1.4 in which 6 different transistor, 2 PMOS (`P1` and `P2`) and 4 NMOS (`N1`, `N2`, `N3`, and `N4`), are interconnected. To understand how a SRAM cell can be a PUF, we need to understand its write process. A full description of memory read and write can be found in [61]. Suppose that the inverter formed by `P1` and `N1` has output `0`, and the inverter formed by `P2` and `N2` has output `1`. So, based on how they are connected, they are in a stable state. Assume now that the word line (`WL`) is `1`, the bit line (`BL`) is pre-charged high and left floating, and the complementary bit line $\overline{\texttt{BL}}$ is pulled down (value `0`). When `N3` and `N4` are set to `1`, `N4` draws the charge of `N2P2` through

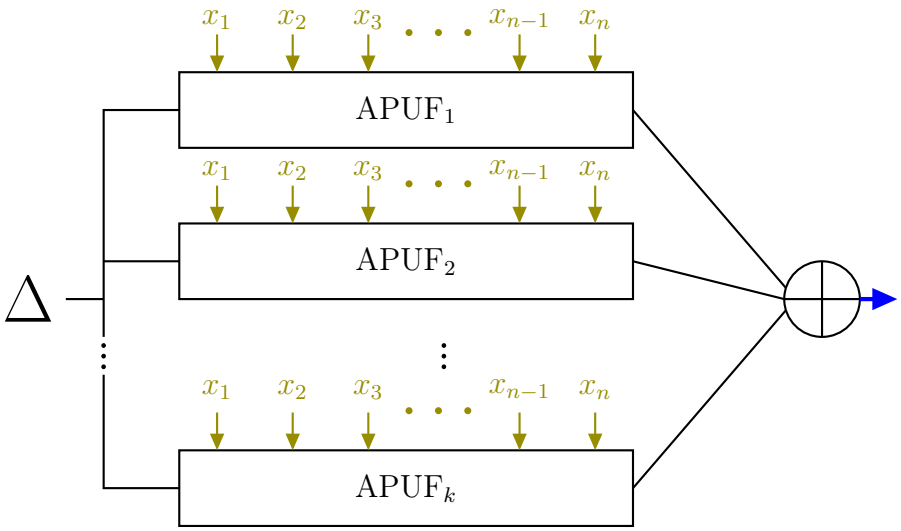


Figure 1.3: A n-bit challenge XOR Arbiter PUF, composed of k APUFs.

$\overline{\texttt{BL}}$. Simultaneously, that activates `P1` and deactivates `N1`, elevating the tension in the output of `N1P1`. Because `BL` is high, no charge moves through it. The high output of `N1P1` deactivates `P2` and activates `N2` that will keep `N2P2` output low. Now, `N3` and `N4` can be deactivated since the cell reaches a new equilibrium. Conversely, one can see that by setting $\overline{\texttt{BL}}$ high, leaving it to float, and then pulling `BL` down will revert the process.

Powering up a SRAM can lead to a high word line and both bit lines floating. In such a situation, the inverters will face a race condition to see which one resists more against being pulled down. Due to variations in the fabrication process, we can have 3 situations [41]: `P2` is the strongest transistor between `P1` and `P2` and it draws current faster, raising the voltage in the output of `N2P2`, which will deactivate `P1`; or `P1` is the strongest and that results in high output for `N1P1`, deactivating `P2`; and finally there is no significant difference between them, yielding an unstable situation, in which, on every power-up, a different transistor can win. Therefore, observing these scenarios we can consider the SRAM cell power-up as a random event because without reading the bit lines we cannot previously know which of the three situations have happened.

Hence, a SRAM-PUF (SPUF) consists of SRAM cells after a power-up. SPUF is the most known PUF of the class called memory-based PUFs. To obtain responses from SPUFs, we use addresses as challenges. Usually, responses will be memory words formed by independent and random bits. Because SRAMs are commonly smalls, even if one can address individual bits, the total number of challenges is significantly smaller than the one we are likely to have in APUFs, for instance. Next subsection discusses more about the number of challenges PUFs have.

1.6.4 Weak versus Strong PUFs

Figure 1.2 illustrates an APUF with n stages. Implementations of APUFs having from $n = 64$ to $n = 512$ stages can be found in the literature [47]. As the number of stages are the same of bits in a challenge, a 64-stage APUF will have 2^{64} possible challenges. Thus, a 512-stage APUF has 2^{512} possible challenge values, an enormous number. On the other

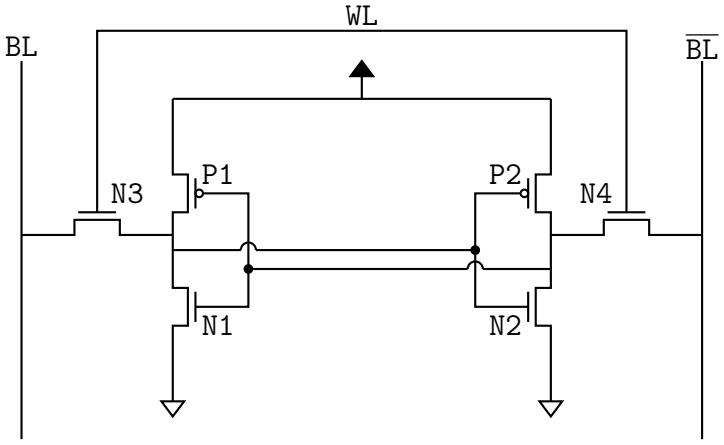


Figure 1.4: A SRAM cell. Figure based on [61].

hand, a 32-MB SRAM has 2^{28} bits, thus, even if we can obtain responses of individual cells, such a SPUF will be limited to 2^{28} challenges.

In that regard, it is common in the literature to classify APUF, and derived constructions of it (like XOR APUF), as *Strong PUFs*. SPUFs and other PUFs with small challenge space are called *Weak PUFs*. The reader should understand that these terms do not relate to security. For instance, SRAM-PUFs can generate cryptographic keys that are less biased than APUFs [30]. The "*PUF Attack*" paper gives examples of applications of weak and strong PUFs. Deep analyses regarding strong and weak PUFs can be found in [45]. Next, we discuss basic mathematical tools to assess PUFs in regard to bias, randomness, etc.

1.6.5 Assessing PUFs

A key point we did not discuss yet is how to use PUFs as cryptographic key generators, integrated circuits identifiers, etc. Figures 1.2, 1.3, and 1.4 show PUFs with one-bit output, thereby to generate n-bit long strings we must replicate a PUF n times. For SPUFs though, a memory word can be seen as a bit string. However, n can be a number too large to either be a memory word or enable cost-viable replication of PUFs. Hence, an assemblage of multiple responses of different challenges is a valid form to generate long bit strings that cryptographic applications demand. Due to *unclonability* and *unpredictability* of PUFs, we expect those strings to be unique and random. Of course, we do not have guarantees about that, unless we assess samples of these strings.

Three main metrics have been used in the literature to assess the quality (i.e. randomness and uniqueness) of bit strings generated by PUFs: Hamming Weight (HW), Hamming Distance (HD), and Entropy. The HW metric is given by Equation 1.1, where X is a collection of bits $\{x_1, \ldots, x_n\}$, $x_i \in \{0, 1\}$. Given a sample of n-bit strings, we expect that, as result of computing the HW of each bit string, we generate a Gaussian distribution with expected value of $n/2$. This value indicates that, on average, the strings do not have bias towards bits `0` or `1`. We want to avoid biased patterns since they reduce the number of bits an attacker would have to guess to figure out a cryptographic key.

$$\mathbf{HW}(X) = \sum_{i=1}^{n} x_i \tag{1.1}$$

Equation 1.2 describes the HD metric, in which X and Y are collections of n bits. So, the HD between two binary string is the sum of bitwise XORs. For a sample of fixed-length bit strings, we apply Equation 1.2 for all possible distinct pairs we can form with them. As result, we expect to obtain a Gaussian Distribution with mean $n/2$. This value shows us that, on average, half of the bits in two randomly-picked bit strings are equal. That is the minimum possible information that an attacker can obtain from true random keys. For instance, if a HD between two keys is n, an attacker knows that one string is the binary complement of another. Conversely, if a HD value is `0`, an attacker knows that both strings are equal.

$$\mathbf{HD}(X, Y) = \sum_{i=1}^{n} x_i \oplus y_i \tag{1.2}$$

It is important to notice that HW and HD complement each other information, thus one analysis does not replace the other. Furthermore, HD is usually computed between strings generated from different PUF circuits: in order to evaluate if a PUF architecture actually leverages on the variation of the fabrication process to produce unique responses. To assess randomness of bit strings created by concatenating responses, Entropy is a better metric. Given a set $\overline{\mathrm{X}}$ of m bit strings $\{X^{(1)}, X^{(2)}, \ldots, X^{(m)}\}$, where $|X^{(j)}| = n, \forall\ j \in [1, m]$, let us define $\mathbf{column}(i) = \sum x_i,\ x_i \in X^{(j)}, \forall\ j \in [1, m]$. That is, $\mathbf{column}(i)$ is the sum of all bits at the same position in all m bit strings. Thus, dividing the result of $\mathbf{column}(i)$ by m, we obtain a probability $\Pr(x_i) = \mathbf{column}(i)/m$, which informs how likely is a source to generate `0` or `1`. Using $\Pr(\cdot)$, we can define Entropy by Equation 1.3.

$$\mathbf{H}(\overline{\mathrm{X}}) = -\sum_{i=1}^{n} \Pr(x_i) \log_2 \frac{\Pr(x_i)}{1 - \Pr(x_i)} - \sum_{i=1}^{n} \log_2(1 - \Pr(x_i)) \tag{1.3}$$

Consider a large set of 128-bit strings generated by 128 APUFs. Assume that, for all i, $\Pr(x_i) = 0.6$. That is, in all sources, an output `1` has 60 % of chance of happening. Conversely, `0` has 40 % of chance $(1 - \Pr(x_i) = 0.4)$. Thus, for each possible value of i, $\Pr(x_i) \log_2 \Pr(x_i)/(1 - \Pr(x_i)) + \log_2(1 - \Pr(x_i)) = 0.6 \log_2(0.6/0.4) + \log_2 0.4 \approx -0.97$. If we sum that, for all x_i, we get approximately 124.3 bits of entropy. However, a theoretical true random source would have $\Pr(x_i) = 0.5$ and that would yield 128 bits when computing the entropy. Therefore, one could deduce from the result of our example that: almost 4 PUFs have predictable outcomes; or 4 bits carry redundant information of the other 124 bits; or still some sources are biased towards `1`. Notice that entropy is crucial information, if we aim at generating 128-bit cryptographic keys, we definitely would not want those keys having any deducible information that facilitates the attackers work.

For this reason, estimating a worst case scenario of randomness in bit strings would be a better metric for security purposes. To achieve that, we use Minimum Entropy, which Equation 1.4 defines. Using the example above, assume $\Pr(x_i) = 0.6$ again, thus $\max\{0.6, 0.4\} = 0.6$ and $-\log_2 0.6 \approx 0.74$. Hence, considering that all 128 APUFs have the same minimum entropy, the total $\mathbf{H}_{min}$ would be ≈ 94.33 bits. That is a significant reduction compared to the previously estimated entropy of 124.3 bits. In practical terms, both entropy measurements indicate that we should probably compress our 128-bit strings into smaller ones. For example, entropy says that we should only use 124 bits out of 128. On the other hand, the minimum entropy value says that we should only use 94 bits out of 128. From the security point of view, opting to compress and generate 94-bit strings would approximate our generator of a true random one [7].

In conclusion, to use PUFs in cryptographic applications, system designers should look for PUFs that allow them to assemble n-bit strings that will averagely present HW and HD values close to $n/2$, and minimum entropy close to n. These metrics are the minimum evaluation one should do in order to consider PUFs in security applications.

$$\mathbf{H}_{min}(\overline{\mathrm{X}}) = -\sum_{i=1}^{n} \log_2 \Big(\max\{\Pr(x_i), 1 - \Pr(x_i)\} \Big) \tag{1.4}$$

1.7 Robustness in PUF-based Systems

The properties presented in the last section make PUFs desirable security primitives. However, they are not perfect functions. That is, due to the effects of temperature and/or voltage variation, circuit aging, ambient noise [10], among others possible internal and external physical events, challenges can generate different responses over time, causing a *reliability* problem in PUFs. As result, PUF-based key re-extractions are unlikely to work without a redundancy scheme. Redundancy for PUFs needs to not only provide as much original information as possible, but also be secured. In other words, exposing redundant data can help attacker to learn about PUF responses.

Currently, Fuzzy Extractors (FEs) have been employed to produce robust PUF-based keys with minimum information leakage through its redundant data [8], also called *Helper Data*. FEs are secure sketches that comprises a randomized key extraction algorithm and a key recovery procedure. For the purpose of this work, we call the key extraction phase of FEs as *enrollment* and the key recovery phase as *regeneration* (or *reconstruction*). In the following subsections, we define important elements present in most of FE implementations and detail two FE constructions that we use in this work.

1.7.1 Error Correction Codes

Most of Fuzzy Extractors use Error Correction Codes (ECCs) to enable data recovery. An ECC encodes data into a string called syndrome, which contains information to recover not only the data, but also itself. By concatenating data and syndrome we form a codeword. Given k bits of data, we describe an ECC string by a tuple (n, k, t), where n is the length

of the codeword and t is the maximum number of bits that can be lost or corrupted. Any number of errors greater than t does not have any theoretical guarantee of being recoverable. Notice that a syndrome have length $n-k$ and, for a fixed-size data of length k, the greater the number of bits we want to correct, the larger n is.

ECCs work in two steps: encoding and decoding. Both steps have some similarities. For instance, we need to use the same generator polynomial to create a syndrome and to decode it. For a general understanding of ECCs, it is just required that we know the input and output of the encoding and decoding steps. An ECC encoder has a data with k bits as input. If it has less than k bits, we usually pad zeros to the data. The encoder output is the syndrome. Decoders receive data and syndrome and output the correct data[3], if the sum of errors of both are equal to or less than t. What happens to a decoder output if the number of errors is greater than t depends on the implementation. That can be source of unwanted leaking information [8]. Additionally, how decoding is implemented can leaking information, even if it is done in hardware [29]. We will not detail these attacks, but we believe that would be important to make the reader aware about them.

1.7.2 Fuzzy Extractors

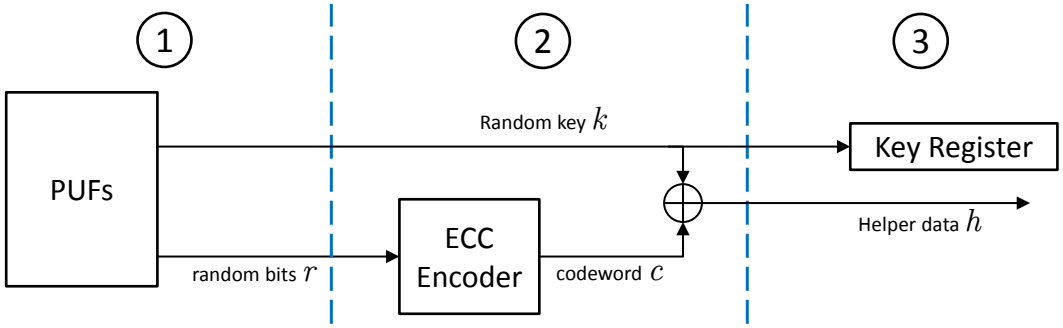
ECCs are the core part of the Fuzzy Extractors we employ in this work. To design a FE, it is crucial to use a correct ECC tuple and we explore that in the papers "*CSHIA Design*" and "*CSHIA Implementation*". In this subsection, however, we will explain key extraction and recovery procedures of both Code-Offset and Index-Based Syndrome Fuzzy Extractor.

Code-Offset Fuzzy Extractors

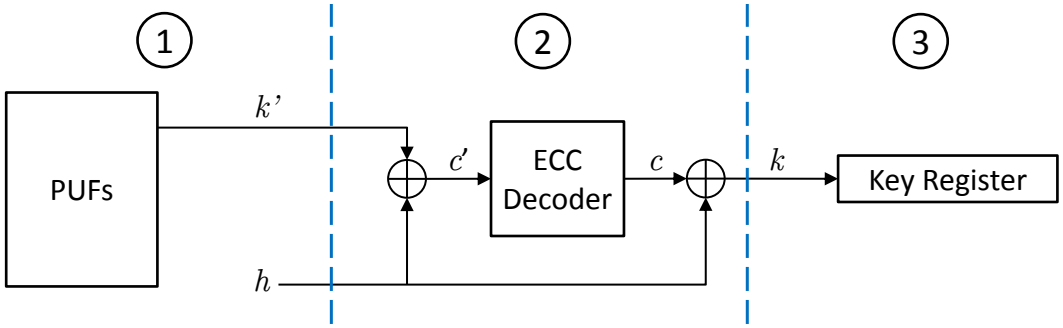
A *Code-Offset* fuzzy extractor is shown by Figure 1.5. Figure 1.5(a) and Figure 1.5(b) describe the key extraction and recovery circuits, respectively. In Figure 1.5(a), after PUFs were challenged, the system extracts a key k and a random binary word r. The ECC encoder creates a syndrome for r and their concatenation forms c. The codeword c is a random word, since r is, and thereby its syndrome is as well [18]. Due to that, a combination of c and k can be seen as a one-time pad operation since every c is only combined with a unique k, thus yielding an Information-Theoretically Secure (ITS) [40] bit string: the helper data h. Consequently, h can be externally stored because of the ITS property, which ensures that critical information will not leak. The key will be maintained on-chip while the system is operational, inhibiting attacker from collecting information about it.

Every system power-on will require a regeneration of the key. During the regeneration procedure, a slight different key k' may be extracted because of the PUF reliability problem. As Figure 1.5(b) shows, combining k' with helper data h results in a modified codeword c'. If this codeword differs at the most t bits from the original c, the ECC decoder can recover c, which combined again with h recovers the original key. This version

[3]Sometimes it is necessary to correct both data and syndrome. However, some implementations of decoders will recover only data, requiring to use the encoder again to reproduce the correct syndrome.



(a) Code-Offset Fuzzy Extractor steps during the Enrollment.



(b) Code-Offset Fuzzy Extractor steps during the Key Regeneration (or Reconstruction).

Figure 1.5: The Code-Offset Fuzzy Extractor.

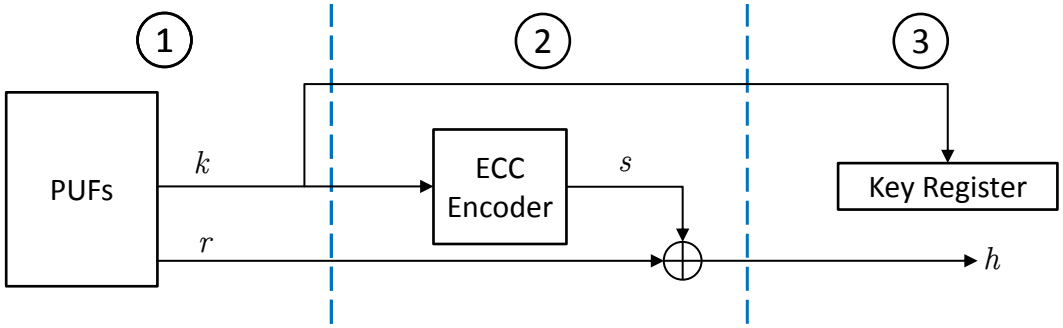
of FE that we are describing does not include a post-processing step for the key, because we assume that the employed PUFs produce random and unpredictable responses with enough minimum entropy. To post-process a key, we could apply it to a hash function.

Even though, by construction, it is safe to expose the helper data, we have downsides on doing it. First, an attacker only needs to guess the bit string r to unveil the key. Once the attacker guesses r, he/she can recreate its syndrome and thus obtain c, which in combination with the helper data gives k. Notice that r is always smaller than k in this FE, and therefore being computationally easier to figure it out. That is seen as a reduction of the entropy of key [4]. In addition, helper data manipulation by attackers has been shown to enable side channel attacks [35].

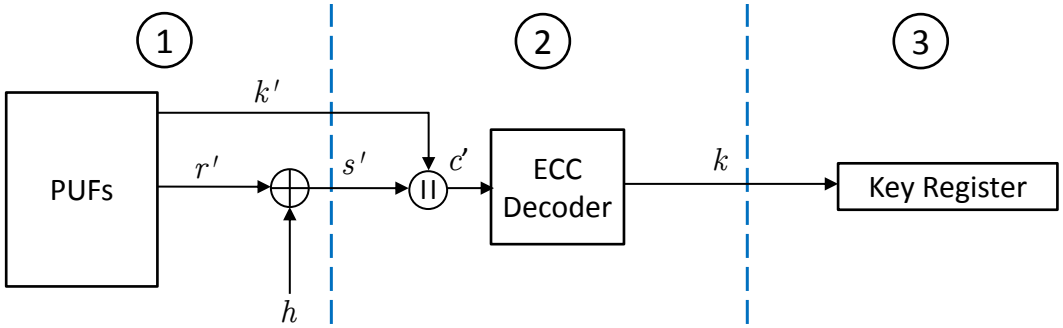
Index-based Syndrome Fuzzy Extractors

The Index-Based Syndrome (IBS) Fuzzy Extractor, proposed by Yu and Devadas in [65], has a core difference from the Code-Offset FE: a key-independent helper data. Figure 1.6 illustrates an adaptation of the IBS-FE in which the indexes only consists of inverting and non-inverting values, i.e. `1` and `0`. After extracting the key k and a bit string r from PUFs, a syndrome s is generated for the key. The syndrome is combined with r that becomes the helper data h. So, the helper data is a one-time pad encryption of the syndrome. If an attacker figures out s, he/she cannot recover the key entirely from it. In fact, only t bits of information are lost to the attacker that unveils the syndrome because t is the number of wrong bits the ECC can recover. Thus, an attacker would still need to figure out $k - t$ bits.

For key regeneration, due to unreliability of PUFs both k and r can differ from their



(a) IBS Fuzzy Extractor steps during the Enrollment.



(b) IBS Fuzzy Extractor steps during the Key Regeneration (or Reconstruction).

Figure 1.6: The Index-based Syndrome Fuzzy Extractor.

original values, so we have k' and r'. The combination of h and r' generates a syndrome s' that differs from the original s as much as r' differs from r. Concatenating k' with s' gives us c' that by differing from c in up to t bits enables the ECC decoder recover k.

The strongest feature of IBS is that we don't have entropy reduction of the key, since no shortcut to figure out the key happens. Nonetheless, researches believe that this FE is not side channel attack proof [23] and Karakoyunlu and Sunar in [29] showed that the syndrome decoding can leak side channel information. Overall, both FEs' security will depend on the length of the bit strings used. Larger lengths for k and r will hinder attacks, but they will extract a heavy toll from either PUF reliability or system cost. Using small ECC hardware enables correction of only few bits in large strings. Conversely, to correct a larger number of error in long bit strings, more complex ECC hardware is needed. Nevertheless, in "*CSHIA Implementation*", we present a FE design that tries to deal with these trade-offs.

1.8 Side Channel Attacks

A side channel attack is the one that uses leaking information of operations, such as time delay, power consumption patterns, among others, to extract important data of a system. Attackers usually seek to obtain concealed information of systems, like binary keys, passwords, etc. On PUFs, some works have been using side channel information to help machine learning algorithms to model PUFs [33, 47, 31]. On Fuzzy Extractors, Merli *et al.* have successfully applied Differential Power Analysis (DPA) and Correlation Power Analysis (CPA) as attacks on Code-offset FE [35, 36]. In this section, we discuss some

basic concepts of side channel analysis focusing on attacks on Fuzzy Extractors.

1.8.1 Understanding Power Side Channel Information

A physical implementation of a register is an assemblage of flip-flops. On the logical level, a register is a collection of `0` and `1`, where each logical value draws different charge from flip-flops' power sources. Therefore, a binary word will draw power proportionally to its number of zeros and ones or, in other words, correspondingly to its HW. Knowing that XOR APUFs are compounded by APUFs, which have flip-flops in their construction, we can assume that the set of flip-flops in a XOR APUF is like a register.

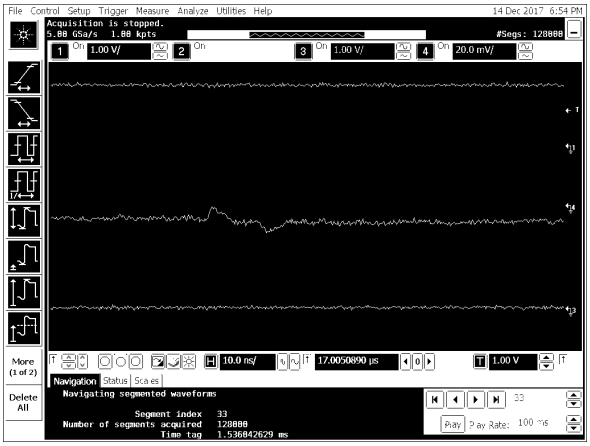
Measuring power consumption or capturing electromagnetic emission of circuits enable us to estimate information about logical values of registers. Figure 1.7 shows a collection of pictures of oscilloscope *traces* of voltage variation during responses of a XOR APUF with four 16-bit challenge APUFs. Considering that we have a 4-bit register inside this XOR APUF, we know that there is five possible HW values. Figures 1.7 (a), (b), (c), (d), and (e) show, respectively, when the flip-flops had 4 logical zeros, 3 logical zeros, 2 logical zeros, 1 logical zero, and 4 logical ones. Notice that, as the number of logical ones increases, the power consumption gets higher. This relation is a side channel information that allows attackers to realize HWs without ever accessing internal values. In particular, for the XOR APUF, this information is critical since one can infer responses only knowing the HW of the flip-flops.

Even though we have seen how one can obtain this unwanted leakage of information, capturing such data is not a easy task. Differently from our example above, real implementations have multiple PUFs simultaneously generating responses, which hinders to obtain individual information and, consequently, forces attackers to analyze mixed data with multiple sources of noise. For this reason, many works have focused on applying side channel analyses on Fuzzy Extractors since their construction enables these attacks through helper data manipulation. In the following, we explain two well-known side channel attacks using their application on Fuzzy Extractors.

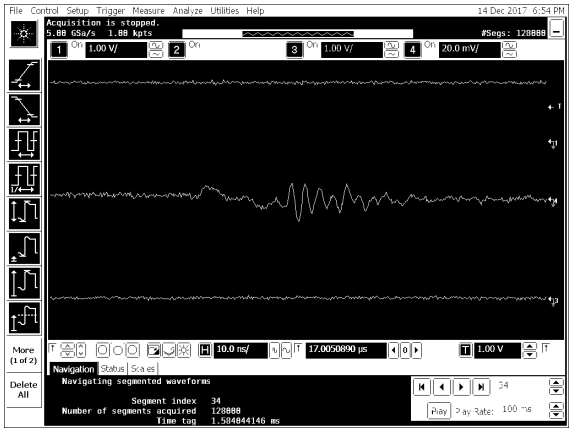
1.8.2 Differential Power Analysis

In the Differential Power Analysis (DPA) attack, an attacker wants to collect power traces from a referential state and a differential state of the system. By differentiating them (e.g. subtracting them) the attacker will discover the change that happened from one state to another. In a practical attack, attackers would get a set of thousands of power traces of a specific state of a register and then obtain another set of thousands of power traces of a new state of that register. For analysis, each set is averaged to eliminate random noisiness. From the attack perspective, the best scenario would be when a differential state only differs one bit from referential state. That allows attackers to deduce if a bit changes from one to zero or vice-versa by only looking an increment or decrement in power consumption.

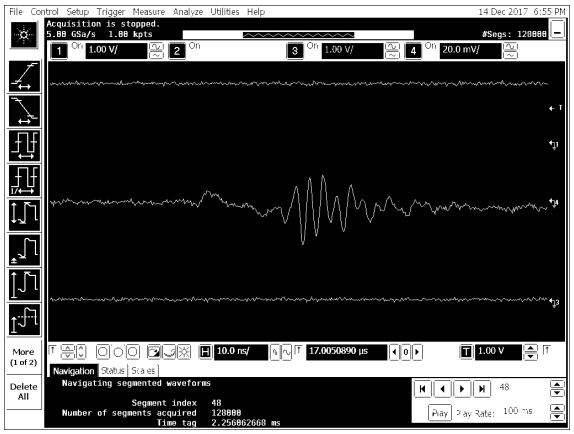
To attack a Code-offset FE using DPA, an attacker needs to be able to modify the helper data. His/her goal is to modify bits in the helper data (generally one bit per



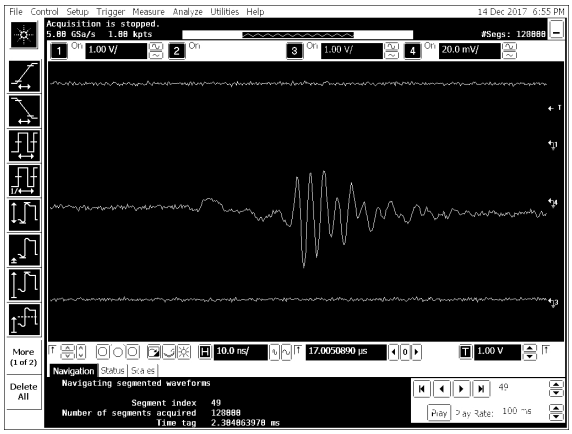
(a) A response of HW 0.



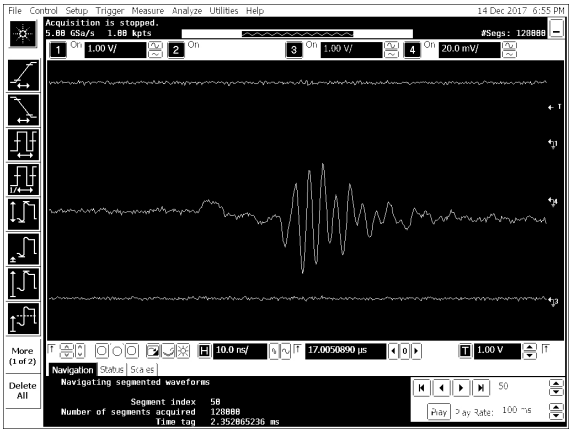
(b) A response of HW 1.



(c) A response of HW 2.



(d) A response of HW 3.



(e) A response of HW 4.

Figure 1.7: Power analysis of responses to challenges of a XOR Arbiter PUF compounded by four 16-bit challenge APUFs.

attack) and see how that changes the power consumption of the register that stores the result of $c \oplus h$ (Figure 1.5(b)). Assuming that registers power consumption increases when a bit changes from zero to one, and decreases otherwise, the subtraction of the averaged attacking traces (differential state) from the averaged referential traces (the original power consumption of $c \oplus h$) will result in a new trace with either a significant spike (positive or negative) or nothing but noise. By assumption, a spike indicates a bit-flip and its absence

indicates that no change has happened. As the attacker knows the previous value of the bit he/she changed in the helper data, he/she can deduce its value in c. One does not need to repeat the attack procedure to all bit in c, though, only to $|r|$ bits instead. As we discussed before, finding r enables to compute c and thus recover k doing $c \oplus h$.

1.8.3 Correlation Power Analysis

The principle of a Correlation Power Analysis (CPA) attack is similar to the DPA one. For the sake of understanding, we will keep using Code-Offset FE as target of attacks to explain the CPA attack. The goal of the attacker is to modify the helper data and collect traces of the key register (where the result of $c \oplus h$ is stored). However, he/she does not need a referential state. Once traces are collected, the idea is to model the relation between information provided by the traces and the changes in the helper data. In [36], the authors chose the Hamming Distance (HD) Model.

Assume that an attacker chooses to recover r (used to generate c) byte by byte using CPA. Consider that r, h and k from Figure 1.5(b) has more than 1 byte, and suppose, without loss of generality, that the attacker selects the least significant byte B_0^h of the helper data as the first to modify. For simplicity, also assume that the attacker incrementally changes B_0^h from `00000000` to `11111111`. For each value of B_0^h, the attacker has to collect thousand of power traces and then average them. After all that is done, it is possible to start the analyses.

One begins by creating hypotheses of possible values of the byte B_0^c of the codeword. The attacker assumes B_0^c is `00000000` and then computes the hamming distance between it and every value B_0^h he/she set. As traces are collections of points that consists in a pair (*time*, *power*[4]), the attacker picks the first point of each averaged trace of B_0^h and computes the correlation between the HD and the power value. Then, he/she passes to the next point, and so on, until all points have a correlation value. After that, the attacker makes a hypothesis that B_0^c is `00000001` and repeat all the calculation again. The final product of this process is 256 correlation graphs in which the one with the highest peak has the highest probability of being the correct guess of the byte in c. Repeating all that for each byte in r, c can be calculated, allowing k to be unveiled.

1.8.4 Countermeasures

A very common side channel countermeasure is to use masks, which are random binary strings to be combined with a value we want to protect from leaking information. Once again, for illustration of this countermeasure we will use Code-Offset FE. This countermeasure was proposed in [36]. As we can see in Figure 1.8, during the regeneration phase of the FE, a mask m is extracted from PUFs and encoded into a codeword c_m, which is then combined with c'. Because ECC decoders are linear, the combination of c' and c_m can be processed simultaneously. When c is combined again with h the result is k, but because the mask is kept, we actually have $k \oplus m$. Therefore, at all times, attackers will capture power consumption of registers that have $k \oplus m$. As every regeneration a new

[4]It can also be voltage, current, etc.

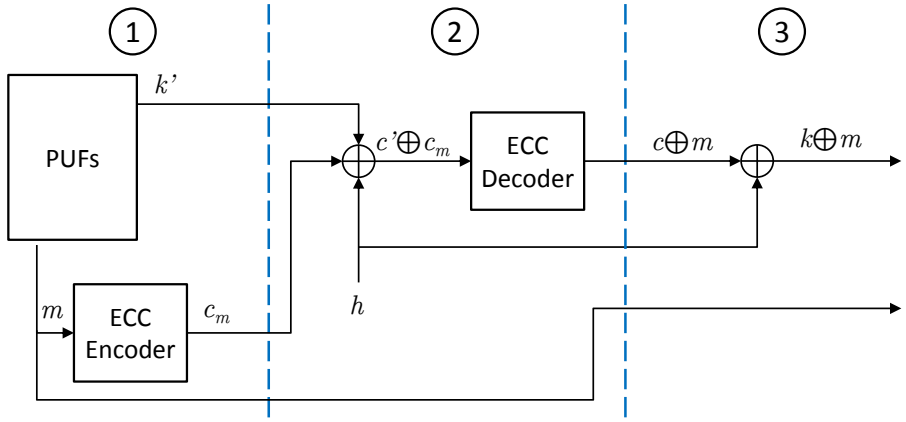


Figure 1.8: Masking countermeasure in the Code Offset Fuzzy Extractor.

mask is used, DPA and CPA will not work because the randomness of m interferes in the direct relation between data and power consumption. As result, this mask cannot be removed anytime, since it would make a new spot for attack.

This countermeasure, though, is not suitable for all systems: as a new mask has to be generated every key reconstruction, all cryptographic operations involving the key have to be linear to allow posterior discard of the mask after concluding an operation. This requirement of linearity inhibits the adoption of this countermeasure in CSHIA.

More Powerful Attacks

A class of more powerful attacks that include second order DPA and template attacks have been already shown to overcome masking [39, 64]. In the so-called high order attacks, an attacker measures side channel information of two elements or more that are related to the target of the attack, then he/she differentiates these data seeking concealed information. For instance, in a second order DPA, the attacker collects power signature from the mask register and from the one that stores the key combined with the mask. Taking the difference between these power signatures, the attacker obtains a new power signature that correspondents to the key. The differentiated data will enable the attacker to perform DPA/CPA as we discussed before.

Differently, in a template attack, the attacker profiles every possible combination between keys and masks that can be stored in the target register. Then, during the attack, he/she tries to figure out the key by matching the register's power signature with those that he/she profiled. It obviously is a very time consuming attack, yet a powerful one, nonetheless.

1.9 Key Features of Security

This section defines what the terms authenticity and integrity means in this work. Also, it defines how both can be applied together through Pseudo-Random Functions.

1.9.1 Authenticity

In [57], Varga and Guignon present one possible strong sense of being authentic. One could simply put it as something of *undisputed origin or authorship*. Although this definition comes from a philosophical text, it is possibly the best one we can apply to authenticity in Cryptography. Hence, whenever we use the term authenticity, we mean that something is irrefutably linked to a specific entity and no one can fake it or contradict it so.

As we discussed throughout this chapter, PUFs are uniquely defined by their CRPs. Further, they respond unpredictably to challenges, which makes difficult to impersonate them. These two properties enable PUFs to be used as cryptographic primitives that provide undisputed origin or authorship. To do so, we need to tie data to PUFs responses. Let us define $\mathbf{PUF}(c) = r$ as non-invertible surjective function, where r is a bit string that is an assemblage of PUF responses to the challenge c. Assume that forging $\mathbf{PUF}(\cdot)$ is computationally impossible. Let $\mathscr{P}$ be the collection of all possible $\mathbf{PUF}$ functions, that is $\mathscr{P} = \{\mathbf{PUF}_1(\cdot), \mathbf{PUF}_2(\cdot), \dots\}$. Now, let a computationally-hard-to-invert function be $f(d, k) = y$, where d, k are bit strings. Assume that x is a randomly-picked data and c is a randomly-picked challenge. Thus, $f(x, \mathbf{PUF}_i(c)) = y_i$ enables $\mathbf{PUF}_i(c)$ to authenticate x through y_i, for every $i \in \mathbb{N}$. Furthermore, there is no j such that $f(x, \mathbf{PUF}_j(c)) = y_i$, if $j \neq i$.

Therefore, every y_i is uniquely linked to $\mathbf{PUF}_i(\cdot)$, for any given challenge c and data x. Notice that it can there exist a x' such as $x \neq x'$, which and $f(x', \mathbf{PUF}_i(c)) = y_i$. Namely, two different data can be authenticated by the same $\mathbf{PUF}_i(\cdot)$. Or, different bit string can be unequivocally identified by the same origin and authorship. However, that does not break the authenticity that $\mathbf{PUF}_i(\cdot)$ gives to the sole system i from which it belongs to. Also, we assumed that $f(\cdot)$ is computationally hard to invert because $\mathbf{PUF}_i(c)$ should not be easy to unveil.

1.9.2 Integrity

Menezes *et al.* in [34] define data integrity as "the property whereby data has not been altered in an unauthorized manner since the time it was created, transmitted, or stored by an authorized source". First, notice that the term data in this section conveys a generic bit string whose semantic can be code, information, etc.

For this work, given a function $g(x) = z$, where x is a bit string randomly picked, if there exists another randomly picked bit string x', then $g(x') = z$, if and only if $x = x'$. That is, if x' has one bit or more that differs from x, then $g(x') = z'$. Thus, the input of the function is uniquely linked to its output. That is, $g(\cdot)$ attests the integrity of x through z. Notice, though, that the origin of x does not care in this definition.

The reader can see now the difference between Authenticity and Integrity. Authenticity can be seen as a person's signature, which can exist in multiple documents. Nonetheless, it is quite unique for every person and works as anti-impersonation. Integrity otherwise regards to documents one signs. In that way, we want to have proof that such a document is not tampered with. For instance, if every document has a *unique* picture of it, we can check if a document matches its picture. It does not matter whether it is signed or not, but rather whether it is the same of the picture.

1.9.3 Pseudo-Random Function

Previously, we defined two different properties using two different functions. However, in Cryptography, both properties, authenticity and integrity, are generally undissociated and they are provided by a special kind of functions: keyed hash functions or Pseudo-Random Functions (PRFs). For this work, we deploy PRFs and use them to concomitantly give authenticity and integrity to code and data.

Assume that a PRF is a function $h(x, \mathbf{PUF}_i(C)) = w_i$, where C is a set of challenges and x is some data that is in or belongs to a given system i, which has PUFs that works as a function $\mathbf{PUF}_i(\cdot)$. Notice that $\mathbf{PUF}_i(C) = \Big|\Big|_C \mathbf{PUF}_i(c)$, $\forall c \in C$. For any two data x and x' that belongs to X, the collection of data of the system, if $x \neq x'$, then $h(x, \mathbf{PUF}_i(C)) = w_i$, $h(x', \mathbf{PUF}_i(C)) = w'_i$, and $w_i \neq w'_i$. Similarly, for any given j, $j \neq i$, if $h(x, \mathbf{PUF}_i(C)) = w_i$, then $h(x, \mathbf{PUF}_j(C)) \neq w_i$. Therefore, $h(\cdot)$ gives authenticity and provides a way to verify integrity.

Obviously, everything we defined in this section stands for the theoretical point of view and real implementations may present slight different behaviors from those stated by the definitions. However, the security analyses we do throughout this work take into account these differences in order to enable concrete deployments, such as the CSHIA prototype, without compromising security.

In conclusion, this section formulated, in a non-strictly formal way, the difference between authenticity and integrity, which are rarely used or defined separately in the literature. However, in order to show the importance of using PUFs in CSHIA we believe that the separated understanding of these properties would help to read this work.

1.10 Summary

Throughout Sections 1.6, 1.7, 1.8, and 1.9, we presented the most important background contents that we believe one would need to know to understand the rest of this work. Section 1.6 presents PUFs and their properties that make them desirable cryptographic primitives. Further, we explained how the Arbiter PUF, the XOR Arbiter PUF, and the SRAM-PUF work. Section 1.7 discussed reliability issues in PUFs and how we deal with that employing Fuzzy Extractors. Then, Section 1.8 introduced side channel attacks and how some of them work. Finally, Section 1.9 discussed how this work uses the concepts of Authenticity and Integrity.

Chapter 2

Papers

In this chapter, three academic paper contributions resulting from this work are presented in their pre-printed versions; they are:

1. *Computer Security by Hardware-Intrinsic Authentication* [25].
2. *Implementing a Secure Architecture for Code and Data Authenticity and Integrity in Embedded Systems.*
3. *Applying Template Attacks on XOR Arbiter PUFs.*

Roadmap

The first paper introduces the CSHIA architecture and three important concepts: SRAM-PUFs, Fuzzy Extractors, and replay attacks. It also presents the key extraction algorithm that is used in SRAM-PUFs and provides a thorough security analysis of the CSHIA architecture. The reader is invited to priorly read Section 1.6, in particular, Subsections 1.6.3 and 1.6.5. Also, we advise the reader to read Section 1.7 up to Subsection 1.7.2, and Section 1.9 entirely.

The second paper is the core contribution of this work. It puts together all concepts developed in the previous work, and further explores the implementation of CSHIA, while doing an in-depth analysis on attacks to the architecture. The reader is invited to priorly read Sections 1.6, 1.7, and 1.9 entirely.

Finally, the third paper briefly introduces PUFs, specifically Arbiter PUFs and XOR Arbiter PUFs, and discusses side channel attacks. Then it delves into the Template Attack technique and its application to the XOR Arbiter PUF. Here, we invite the reader to priorly read Sections 1.6, 1.7, and 1.8 for better understanding of the paper. Complementary information about all papers can be found in the appendices.

Computer Security by Hardware-Intrinsic Authentication

Caio Hoffman, Mario Cortes, Diego F. Aranha, Guido Araujo
Institute of Computing
University of Campinas
{caio.hoffman, cortes, dfaranha, guido}@ic.unicamp.br

ABSTRACT

The widespread embedding of electronic devices into the daily-life objects, and their integration in the so called Internet of the Things (IoT), has raised a number of challenges for the design of Systems-on-Chip (SoCs) devices. Tiny manufacturing costs, stringent security, and ultra-low power operation constraints have considerably raised SoC design requirements. More than incremental approaches which try to re-use current cryptographic mechanisms, the new generation of IoT devices will require novel solutions which deeply integrate their hardware-intrinsic features to program execution. This paper proposes a low-cost PUF-based authentication architecture aiming to secure code execution in IoT SoCs. The solution is deeply embedded into the processor micro-architecture, so as to minimize re-design costs and performance penalties. This new architecture model not only deals with the most common threats against code and data authenticity and integrity, but also provides an approach to extract from processor's caches a stable and unpredictable key that is used in the code and data authentication process.

1. INTRODUCTION

Standard design techniques to secure code execution in SoCs are based on known cryptographic mechanisms (primitives like block ciphers and hash functions), and on (micro) architecture techniques which can be used to encode bus transactions [8], or isolate secure code into trusted platforms [11], among others. Although such techniques usually provide good levels of security, most of them are either slow, considerably impact processor (micro) architecture design, require extensive changes in the programming toolchain [28,30], or are so complex that may create unexpected security loopholes.

Any solution up to this challenge should be able to allow for a seamless integration to the current processor/programming paradigms, achieve very high-level security under low-cost and low-power. More than incremental approaches which try to re-use current cryptographic mechanisms to fill in security holes, the new generation of IoT devices will require novel solutions which deeply integrate the hardware-intrinsic 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*) [23]. Built upon PUFs' theoretical models, several constructions of essential cryptographic primitives have been proposed, mainly to support key exchange [5, 18, 32], device authentication [29], intellectual property protection [12], oblivious transfer [5, 25] and commitment schemes [5]. 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 [22], (2) characterization of the physical parameter [31], (3) modeling [24], and (4) emulation [13]. 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 [20] for some application domains (e.g. IoT devices). One of the potential applications of PUFs in IoT devices would enable program code and data integrity. Yet very few works have addressed that using PUFs [30]. 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.

This paper proposes *Computer Security by Hardware-Intrinsic Authentication* (CSHIA), a design of a new secure program execution model. The approach is a new PUF-based mechanism which aims at ensuring firmware authenticity and integrity for a given program/processor pair – while confidentiality is left for future work. Specifically, for authentication, 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. For integrity, it provides a new architecture that ensures that program instructions and data are not violated, and that the program will execute correctly during the lifetime of the device.

The contributions of this paper are:

- A new (micro) architecture model that ensures code and data authenticity and integrity.
- A strategy for high hit-rate caching of nodes in an *authentication tree*.
- A new method of extracting a stable and unpredictable key from the processor's SRAM cache.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes the proposed (micro) architecture and authentication mechanism. Section 4 discusses how CSHIA addresses typical attacks. Section 5 describes how CSHIA tackles replay attacks. Section 6 details CSHIA key extraction. Finally, Section 7 concludes the paper and Section 8 discusses future work.

2. RELATED WORK

SoC devices have a number of features which differentiate them from other traditional electronic solutions. In such devices, heterogeneous hardware IP-cores are combined with processors to perform specialized functions, non-volatile memories work as secondary storage devices, and programs are firmware code which perform a number of low-level operations to enable the cooperation of the various hardware/software modules. Ideally, for the sake of security, all firmware running on a secure SoC should have their integrity continuously verified along the device's lifetime.

Although the dedicated nature of SoCs allows the adoption of more intrusive protection, it also imposes challenging energy and performance requirements. Unfortunately, traditional security solutions based on typical cryptographic mechanisms can have an expensive 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 mechanisms and program execution, as those offered by PUFs.

Qualitative analyses of PUFs have already been done in the literature [16] motivated by several reasons like cryptographic key generation [4, 29] and true random number generation [14, 17]. Unlike those works, which aim at evaluating the quality of a standalone PUF-inspired mechanism, this paper focus on proposing and analyzing a PUF-based micro-architecture mechanism to enable secure code execution.

Most of the preliminary work to secure code execution aimed at keeping instructions and data secure from scrutiny, by using mechanisms like bus encryption. In [8] Elbaz *et al.* did 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. From many alternatives, the authors discarded public key encryption because of its high overhead. The remaining solutions store encrypted data in external memory (or even at higher level cache memories). Such schemes require on-chip secret key storage, a major shortcoming since the usage of non-volatile memories to store keys is susceptible to side-channel attacks [26].

AEGIS, the proposed secure processor by Suh *et al.* in [30], uses PUFs as a cryptography primitive to uniquely authenticate code and data in order to prevent both software and physical attacks. They present a tool-chain for developing secure software for their architecture which includes a secure operating system to manage different levels of memory protection. Although the presented tool-chain 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 tool-chains. Even though the described set of tools enables different security levels, thus minimizing performance degradation, their architecture requires 30 % more processor cycles when running a case study: Sensor Networks. Besides that, performance degradation becomes prohibitive for programs with high cache miss rates. Code and data memory overheads are considerably low, and stayed below 5 % in the case study. Nevertheless, 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.

One of the most difficult issues against active attacks on secure processor architectures is to preserve memory integrity. Memory placed outside of a secure area is exposed to any kind of manipulation an attacker could perform. Despite that, the secure area still needs to verify the integrity of the memory when communicating with it. Recently, many solutions came up in the literature proposing different approaches, costs, and overheads [7, 9, 15]. Section 4 discusses memory integrity issues in detail.

3. THE CSHIA ARCHITECTURE

CSHIA, Figure 1, is a processor architecture which aims at providing secure code execution by means of cache line based PUF authentication. For the scope of this paper, CSHIA is integrated into a typical SoC IoT device, consisting of a low-power embedded processor, I/O modules, main memory, a non-volatile memory, and (if needed) lower levels of storage.

The central idea behind CSHIA is a PUF-Tag (PTAG) Memory[1], which runs in parallel with the system main memory (Figure 1). Each entry in the PTAG Memory stores an authentication code of a cache line generated by a PUF-based device located in the Memory Controller (MCTRL).

The main architectural module affected by the design of the CSHIA architecture is the *Memory Controller* (MCTRL), which is modified to include the *PTAG Generator* (PTAG-GEN, Figure 2). Other two new architectural components are also required to complete CSHIA design, the *PTAG Memory* and the *PTAG Bus*. In a few words, when *Memory Controller* (MCTRL) reads/writes the data/instruction it uses the PTAG-GEN to compute/validate a PTAG for the cache line. Notice from Figure 1 that the PTAG bus runs in parallel to the system bus, and that no program can directly read the PTAG Memory, which is only accessible by the memory controller during read/write operations.

This section is divided in two parts. First, Section 3.1 details the main operations executed by CSHIA, from the generation of the PTAG to its validation. Second, Section 3.2 describes the mechanism used to extract the key used to generate the PTAGs and discusses firmware installation and update.

3.1 PTAG-GEN Operation

The hardware of PTAG-GEN is integrated into the processor MCTRL logic. Its operations are divided into three groups (listed below), based on the bus transactions (Memory READ, Memory WRITE and I/O).

3.1.1 PTAG Generation (memory write)

During a write operation the MCTRL writes data/instruction cache lines to memory while the PTAG-GEN computes the PTAG and stores it into the PTAG Memory. To generate the PTAG a *Pseudorandom Function* (PRF) [10] module is used 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 2). In order to ensure uniqueness, the PRF is configured using a *unique-per-device key*. As Section 3.2 describes, this key is produced by the intrinsic hardware features of a SRAM PUF. Such authentication

[1]During the system operation PTAGs reside in either volatile or non-volatile memory. However, when using a volatile memory, a non-volatile storage is required to backup the PTAGs during turn-on cycles.

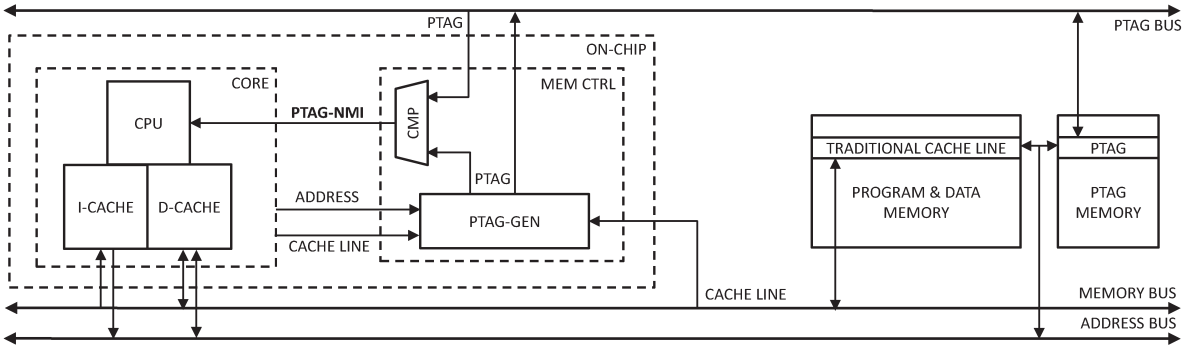


Figure 1: A system overview of the CSHIA system.

tag is specific to the core running that specific cache line, as SRAM PUF outputs are dependent on the statistical variations of the manufacturing process, and these are unique to each processor. Hence identical cache lines running on different processors will produce different PTAG values for the same inputs. Only code in the cache, for which integrity has been ensured, will be able to write to the memory. In other words, a chain of trust exists from the execution of one instruction to another, starting on the first instruction which has been validated into the system. Moreover, as only ensured code can write into memory, not only code but also program data will have its own PTAG computed and stored into the PTAG Memory. Therefore, the chain of trust created by the flow of execution of ensured instructions will also be transferred to program data. Notice that this approach allows for the existence of multithreaded programs.

3.1.2 PTAG Verification (memory read)

During a read operation the MCTRL reads data/instruction cache lines from memory while the PTAG-GEN computes a PTAG for each cache line. As shown in Figure 2, during a read operation the cache line base address produced by the core is appended to the cache line bits read from memory and the resulting bits are fed to the PRF module. The PTAG produced in 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-Maskarable Interrupt* (NMI) is generated to the core (called PTAG-NMI), as code/data integrity might have been violated. As shown in Figure 1, in order to hide PUF latency, the data/instruction is sent to the respective cache (I$ or D$) at the same time that the PTAG-GEN computes the PTAG for that cache line and compares it to its PTAG, previously stored into the PTAG Memory.

3.1.3 I/O

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

Overall, from an architecture perspective, the main advantages of the CSHIA execution model, when compared to other techniques, like [28] are: (a) simplicity and easy of integration to current architecture/programming models; (b) separation of the system and PTAG buses, isolation of the PTAG Memory from program scrutiny; (c) improved performance, which is detailed throughout the rest of the paper.

3.2 PTAG-GEN Configuration

As discussed above, the PTAG-GEN should produce a unique PTAG for each combination of processor, cache line data and address. To achieve that, the PRF is configured by means of a PUF generated key, which is unique to each processor. This key is generated only once and regenerated at each device turn-on time. Any quality PUF available in the literature could be used to produce the key. Nevertheless, given its simplicity and good statistical metrics [16] the SRAM PUF has been selected for the purpose of this paper, and will be called from now on SPUF.

One drawback in adopting SPUF is the additional cost due to its silicon area. To compensate that, CSHIA uses the SRAM array of the L1 cache as SPUF to produce the key. When the processor is turned on the values of the L1 SRAM cells are set to zero or one according to the physical intrinsic parameters of each L1 SRAM cell. Thus, the CSHIA key is composed by a set of bits extracted from the SRAM cells of the processor L1 cache at first turn-on time.

3.2.1 Enrollment Process

Notice that, for a specific processor, the SPUF bits selected to compose the key should be identical at each processor turn-on time. As a matter of fact, the key needs to remain the same during the whole processor lifetime. An *Enrollment* process is used to ensure this after the product testing phase. It runs only once, in a controlled environment, and it is composed by three distinct stages described below: Assessment, Extractor Setup, and Firmware Installation.

• **Assessment:** During this stage, the processor's cache is turned on and off tens of times to find a set A of SPUF addresses that have stable bits to compose the PRF key k; this assessments seeks to choose bits that will have a very low bit-flip probability until the end of the hardware lifetime – Section 6 shows experiments that support the existence of such addresses in SRAMs. Although the bits selected this way will not present bit-flips during chip assessment, one cannot assure that flips will never happen as physical parameters of the SPUF cells can degrade over time. To compensate for that, a fuzzy extractor architecture [1, 19, 21] is implemented in PTAG-GEN to recover the key k, in case of some key bits are incorrect when the processor turns on.

• **Extractor Setup:** In this stage, bits are selected from the SPUF, using addresses of A, to compose two random words: r and the key k. They will be used by the fuzzy

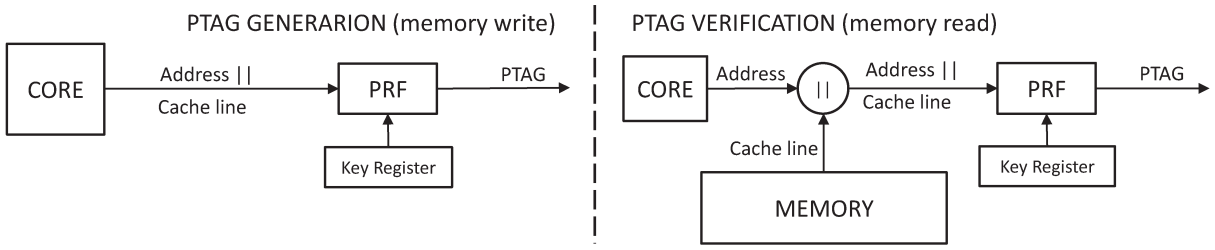


Figure 2: The PTAG-GEN during PTAG Generation (write) and PTAG Verification (read) operations.

extractor logic [1] (step 1, Figure 3 (a)). In step 2, a BCH encoder receives r and generates codeword c. Then, in step 3, the *Key Register* stores k and the fuzzy extractor computes the helper data $h = c \oplus k$. At the end of this stage, the helper data h and the set of addresses A are stored into memory using a standard store operation, which are also authenticated with PTAGs to ensure their integrity.

• **Firmware Installation:** In this stage the content of the whole memory is authenticated. First, h and A are authenticated through PTAG-GEN, generating their corresponding PTAGs, which are then stored into the PTAG Memory. The following authentications are for the firmware and the whole memory. This is a one-time procedure which can occur at the solution vendor. In this stage, the processor reads the whole content of the memory (including the firmware), produces their corresponding PTAGs and stores them back into the PTAG Memory. If necessary, firmware update can use the same procedure by running an (already authenticated) update software. In an IoT device this process will typically occur through I/O and the current (authenticated) system firmware should ensure that the updates brought into the device are also authentic. This could be done, for example, by means of an (authenticated) standard cryptographic communication channel created with the vendor server.

3.2.2 The System Reboot Process

CSHIA is operational after the Enrollment process. In this situation, a system reboot needs not only to recover the key for PTAG-GEN, but also avoid leaking any information about the key. The system reboot process also occurs in three stages: SPUF Recovery, Key Recovery, and Key Validation.

• **SPUF Recovery:** At the beginning of the system reboot, the Key Register is in an unknown state. Thus, the only way for PTAG-GEN to validate cache lines is to reproduce the original key k. At this stage, MCTRL recovers A and the helper data h from Memory and their corresponding PTAGs from the PTAG Memory (see step 1 in the Figure 3 (b)). The system now uses the addresses in A to recover k' from the SPUF.

• **Key Recovery:** Given that there is a non-zero probability that the SPUF cells flip to different values as the device is turned on, the recovered key k' might not be the same as k. To correct k' (step 2), the MCTRL computes the exclusive-or of k' and h, thus generating a codeword c' that differs from c in as many bits as k differs from k'. A BCH decoder is then used to correct c' to c. By computing $c \oplus h$ the system recovers the original value k.

• **Key Validation:** The recovered key k is stored into the Key Register (step 3, Figure 3 (b)) and is ready to be used by the PRF module. Now, before starting the regular operation, the MCTRL can validate the helper data h and A. If their PTAGs are incorrect the system will stop and generate a PTAG-NMI.

4. SECURITY ANALYSIS

Several attack scenarios can be constructed for architectures like CSHIA. Given the large hardware/software stacks and corresponding attack surfaces of modern computers, and their complex interactions, it is possible that some awkward combination of hardware/software states result in indefensible scenarios.

A security analysis across the whole (micro) architecture requires a proper selection of the parameters that compound CSHIA. For security and high performance of the PTAG generator, the chosen PRF is SipHash-2-4 [2]. The SipHash PRF uses 128-bit keys and produces 64-bit outputs. Thus the secret key k is 128 bits long and a PTAG p has 64 bits. A hardware implementation of SipHash-2-4 can produce an output for a 72-byte input (memory address concatenated with cache line) in 10 cycles, which is crucial for the CSHIA performance. As the helper data h has the same length of the key, it also has 128 bits. The random bits r, used in the generation of the codeword c, have length $|r| \geq |p| = 64$ bits. The proper value of r in CSHIA is explained in Section 6.

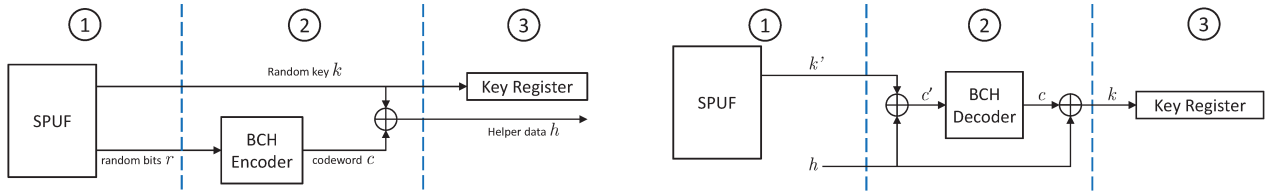
In the following, potential attacks strategies such as brute-force, modeling, and side-channels are addressed as part of a preliminary security analysis. Deep analysis to evaluate weaknesses, security holes, and tamper resistance needs simulation and implementation, which are left for future work.

4.1 Brute-force/Forgery attacks

In CSHIA, PTAGs cannot be directly read by any program and thus, unless the PTAG Memory is extracted and reverse engineered, mounting a brute-force attack would be hard. Combinations of selected ISA instructions could be used by an attacker to build forged cache lines, which produce the same PTAG as a different (valid) cache line. As said before, no access is provided to the PTAG Memory, therefore the attacker needs to directly instrument and insert data/ instructions into the buses. To do that, the attacker's code needs to have a valid PTAG to run correctly, and this is only possible if it models the PTAG-GEN function. Notice that CSHIA only allows a single brute force attack attempt to occur as a PTAG-NMI is generated, interrupting the processor at the first PTAG error detected. From the PRF security properties, the probability that an attacker correctly guesses one specific PTAG is $(1/2)^{|p|} = (1/2)^{64}$.

4.2 Enrollment/Reboot time attacks

The security of CSHIA enrollment and reboot process is



(a) Fuzzy extractor steps during the Enrollment.

(b) Fuzzy extractor steps during the System Reboot.

Figure 3: The fuzzy extractor steps for the process Enrollment and System Reboot.

also relevant. First, one should notice that two SPUF instances have different intrinsic physical parameters and thus are not equal. The probability that the key k and the random value r will repeat for any pair of SPUFs is extremely small. In addition to the literature [16, 17], Section 6 provides data that confirm the randomness of k and r over different instances of SPUF. Consequently, even if an attacker obtains the key of a CSHIA instance, valid PTAGs for other instances of the architecture could not be generated. Moreover, modifications of the values of addresses A or the helper data h stored in memory will recover wrong keys and the PTAG-GEN will not validate them. Such situations will generate a PTAG-NMI signal interrupting processor execution. An attacker could also try to manipulate the PTAGs of A and h in order to authenticate malicious set of addresses or helper data. In order to succeed, he/she needs to know the key, which is not available.

4.3 Modeling attacks

Although an attacker could collect as many cache lines and PTAGs as possible, creating a model of the PTAG mechanism should not be possible, given that the PTAG-GEN uses a PRF that was evaluated to be indistinguishable from a uniformly random function [2]. Thus, modeling does not apply. Besides that, numerical modeling attacks are not effective against SRAM-PUFs [24]. Without physical access to the device, the only way out for an attacker is guessing the PRF key, which is not an easy task when the key is randomly chosen and its size has at least 80 bits [3] – remembering that the key k is 128 bits long. An easier way to attack CSHIA would be to recover the random word r from the helper data h. Once the attacker recovers r, the same syndrome in the codeword c could be generated and then k recovered. Nonetheless, such attacks requires manipulating the helper data [21], which is not possible in CSHIA, since a PTAG protects its integrity. Thus, the only option left to the attacker is to guess r which is at least as hard as guessing a PTAG, given that $|r| \geq |p| = 64$ bits.

4.4 Side-channel/Physical attacks

Literature abounds on studies of different side-channel attacks in SRAM [13, 22] and fuzzy extractors [6, 21]. While invasive and semi-invasive attacks may completely break CSHIA security, attacks like Differential Power Analysis (DPA) [21] that needs to manipulate the system's parameter – like the addresses A and/or the helper data h – will not succeed, because all off-chip data will have their integrity verified through PTAGs. Protecting SRAM from leaking instructions or dumping data during the system reboot will ensure protection against key extraction. Besides that, some mechanisms like fully resettable SRAM [13] may difficult semi-invasive attacks. Of course, fully physical protection demands novel VLSI counter-measures to prevent invasive attacks like in [22].

4.5 Memory integrity attacks

There are basically three types of memory integrity attacks against CSHIA. Using the same terminology as in [7], those are (1) *Spoofing*, (2) *Splicing* or *Relocation*, and (3) *Replay* attacks.

• **The Spoofing attack** consists in exchanging an existing and authenticated memory block[2] (MB) to an arbitrary fake one. The CSHIA prevents such attack by verifying the PTAG, forcing the attacker to forge or guess a valid PTAG, which is only feasible with negligible probability.

• **The Splicing or Relocation attack** occurs when the attacker swaps authenticated MBs, which are located in different memory addresses. In CSHIA, the concatenation of memory addresses with cache lines forms a unique pair in the system, assuming virtual address space. Each pair should have a unique PTAG with overwhelming probability. It is very unlikely that, by swapping MBs, the new pairs would have or can be modified to have the same PTAG as the former pairs. Notice that this is different from a collision attack on public-input hash functions and requires mounting a second preimage attack instead, so the birthday paradox bound does not apply and the probability of success is equivalent to a spoofing attack.

• **The Replay attack** happens when the attacker swaps an authenticated MB for an older authenticated version. That is, differently from a relocation attack, the memory address does not change. Thus the replay attack is a temporal permutation of MBs, while the relocation attack is a spatial permutation. Notice that the replay attack affects only data cache lines (DCLs) since instruction cache lines are not updated in memory. Next section presents how CSHIA tackles this attack.

In CSHIA, the chain of trust of a program starts by tagging the boot code when the SoC device is initially loaded with its very first firmware. From this point on, all basic syscalls and I/O device drivers for DMA, UART, PCI controller, etc are tagged with PTAGs, and thus violating the integrity of these drivers will eventually result in a PTAG-NMI interrupt.

5. DEALING WITH REPLAY ATTACK

Dealing with replay attack requires keeping on-chip status of all data memory blocks. Obviously, that needs a large memory on-chip, which is impractical. To reduce the on-chip memory requirements, the standard approach is to use an *Authentication Tree*. There are different forms of building

[2]From now on memory block and cache lines are interchangeable.

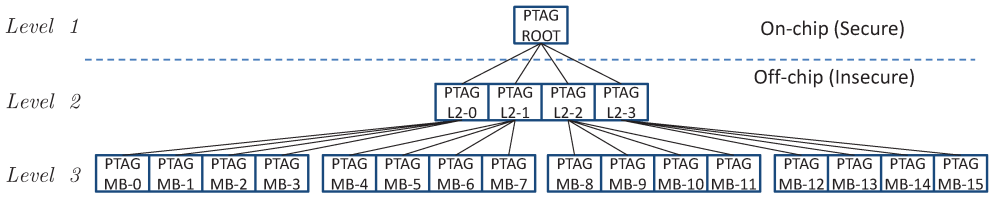


Figure 4: A pictorial example of the Merkle tree implemented in CSHIA.

authentication trees [9], [7], [15]. However, since CSHIA uses a PRF to authenticate memory blocks, a suitable approach is a Merkle Tree [9]. Section 5.1 discusses a naïve tree implementation and Section 5.2 shows how to improve it.

5.1 Merkle Tree

In a Merkle tree, with height L (or L levels) and degree d, the leaves are the tags (at *Level* L of the tree) that authenticate MBs. The next level (*Level* $L-1$) has the tags that authenticate a chunk of d tags in *Level* L. This recursive tagging ends at the root – *Level 1*, which is the only tag to be stored on-chip.

Figure 4 presents a 3-level Merkle Tree with degree $d = 4$ that ensures the integrity of a 16-block memory. The leaves are the PTAGs of each MB. The next level has 4 PTAGs, and each of them is computed by using a chunk of $d = 4$ PTAGs from the level below. Finally, the *Level 1* is the PTAG ROOT retained on-chip, stored into a MCTRL register. The computation of PTAGs from intermediate nodes follows the same PTAG generation process described before, but using chunks of PTAGs rather than cache lines. For simplicity and protection against forgery attacks, the chunk location in the tree is used in place of the memory address.

Notice in Figure 4 that when the processor requires and verifies a memory block against a replay attack, it is necessary to check $L = 3$ PTAGs (the ancestors of the MB's PTAG, all the way until the tree root), for which the MCTRL has to bring $L - 1 = 2$ from the PTAG Memory. In addition, in order to check every tag in the intermediate nodes (considered as a parent node), it is necessary to bring its remaining $d - 1 = 3$ children PTAGs from the PTAG Memory.

To illustrate the process, suppose that the processor requests MB-2. As a regular integrity check procedure, the MCTRL computes the PTAG of the MB and verifies it against the PTAG MB-2 brought from the PTAG Memory. If there is a mismatch, then there is an integrity violation. If they match, MCTRL can proceed to verify a potential replay attack. The MCTRL brings the PTAG L2-0 and has to check whether it is still valid. In order to do that, it brings all PTAG MB-2 siblings, computes the PTAG for the chunk PTAG MB-(0-3) and compares it to PTAG L2-0. If they match, the MCTRL brings all PTAG L2-2's siblings, computes the PTAG for the chunk PTAG L2-(0-3), and finally checks it against the root. In this example, there were $2 \cdot 4$ accesses to the PTAG Memory. In general, there are $(L-1) \cdot d$ memory accesses.

One should notice that (first), as soon as the MCTRL detects an invalid PTAG, a replay attack is detected and the process is aborted, issuing a PTAG-NMI command to the processor; (second) a high volume of consecutive requests produces PTAG-Bus congestion, causing performance degradation due to the need of processor stall cycles, while the MCTRL does not validate the whole chain of PTAGs. In fact, the performance penalty will be even larger when considering an eviction of a dirty block in data cache, since the MCTRL has not only to verify the PTAGs of the new MB, but also to update the PTAG of the evicted cache line, which leads to updating an entire path from leaf all the way to the root.

To illustrate an eviction situation, using Figure 4 again, suppose the processor evicted the data cache line associated to the MB-9. The MCTRL computes the PTAG MB-9 and stores it in the PTAG Memory. Now the MCTRL has to bring all PTAG MB-9 siblings in order to recompute the PTAG L2-2. After that, the MCTRL updates the PTAG L2-2 and brings all its siblings to compute PTAG ROOT, which is updated only on chip. Thus, there were 2 writes and $2 \cdot 3$ reads in the PTAG Memory, which amounts to $2 \cdot 4$ accesses to the PTAG Memory, or $(L-1) \cdot d$ in general. The entire eviction process, including the verification of the requested MB by the processor, demands $2 \cdot d \cdot (L-1)$ accesses to the PTAG Memory. For typical memory sizes, this access rate will indeed cause PTAG-Bus contention. The adoption of a cache to store PTAGs is an attractive solution for this type of problem. This is discussed next.

5.2 PTAG Cache

In [9], Gassend *et al.* propose to use the L2 processor's cache to cache the tree while allowing speculative execution until a hit happens in either the cache or the root (in the secure area). Despite being an efficient solution, since CSHIA has a dedicated bus for PTAGs, a PTAG Cache on the MCTRL fits better the CSHIA needs. In addition to that, reducing the useful space of the processor's caches is prohibitive, given the limited resources in SoCs typical of IoT devices. See the MCTRL modifications in Figure 5.

The PTAG Cache only stores PTAGs of the Merkle tree (including the PTAGs of data MB). Thus, code memory blocks are still verified directly by the MCTRL. Notice in Figure 5 that the MCTRL can directly access PTAGs either via the PTAG Bus or the PTAG Cache. The PTAG Memory stores PTAGs of code memory blocks at conventional address space. The MCTRL places PTAGs from the tree using another address space in the PTAG Memory, representing the tree location.

To compare how the PTAG Cache reduces the number of access to the PTAG Memory in relation to a CSHIA implementation without the PTAG Cache (that is, the naïve CSHIA implementation), one can measure, for each data cache miss, what is the hit rate of the PTAG Cache. To do that, assume the following experimental scenario: a 64-KB processor data cache memory with 8-way set associativity, 128 lines, and blocks of 64 bytes; and a 64-KB PTAG Cache with 16-way set associativity, 64 lines, and blocks of the same size. Notice that while a 64-byte memory block repre-

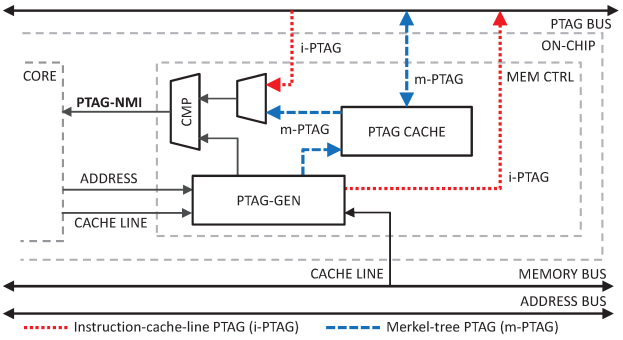


Figure 5: Modifications of the MCTRL and its behavior regarding PTAGs of instruction cache lines (i-PTAG, small dashes) and PTAGs of the Merkle tree (m-PTAG, larger dashes).

sents a cache line, 64-byte PTAG-Cache block represents a chunk of $d = 8$ PTAGs – remembering that a PTAG is 64 bits long.

The experiment was performed using a PIN TOOL to simulate the PTAG Cache and a L1 Data Cache. The PIN TOOL runs all benchmarks of the SPEC CPU2006. The experiment assumed 48 bits of virtual address space. In CSHIA, a memory with 2^{48} bytes requires 2^{42} PTAGs to authenticate all memory blocks. Consider a Merkle Tree with degree $d = 8$ and $L = 15$ levels, for a rough estimate of the entire PTAG Memory overhead, one should take into account $2^{42} \cdot 8$ bytes of memory block PTAGs plus no more than $2^{42} \cdot 8$ bytes to store all ancestors, which results in 2^{46} bytes. This is equivalent to 25 % of the entire memory. The high number of PTAGs to verify would also call into question the efficiency of the PTAG Cache. Fortunately, such high-overhead configuration is very unlikely to be found in an IoT SoC, but serves for illustrative purposes as a pessimistic reference point.

Two strategies of cache-hit policy were tested in the PTAG Cache. The strategy adopted by Gassend *et al.* in [9], which consists in stopping verifying/updating nodes of a leaf-root path at the first node found in cache, showed an overall number of misses greater – for 24 of the 30 benchmarks of the SPEC – than the strategy of caching all nodes from the leaf-root path while the integrity of a MB is verified/updated. Thus, the later strategy was adopted for the PTAG Cache.

Figure 6 shows the Miss Rate (MR) of the PTAG Cache. One can clearly identify three sets of performance levels. Many benchmarks yield very low MRs, around 1 %, and a few are in the 5 % neighbourhood. However, the benchmarks *mcf* and *sjeng* presented high MRs: 14.1 % and 15.7 %, respectively. While the high MR for *mcf* in the L1 data cache impacts directly in its PTAG-Cache MR, for *sjeng* the reasons were not clear. Further studies will be conducted on PTAG Cache's parameters in order to evaluate the best individual and overall configurations for all benchmarks.

To the best of our knowledge, CSHIA is the first architecture with a dedicated bus to deal with memory authentication and integrity verification. The results from simulating a dedicated cache to the PTAG Bus seems promising. Differently from the work in [9], in which the L2 processor's cache stores nodes of the authentication tree, instructions, and data, a dedicated cache does not affect the cache miss-rate

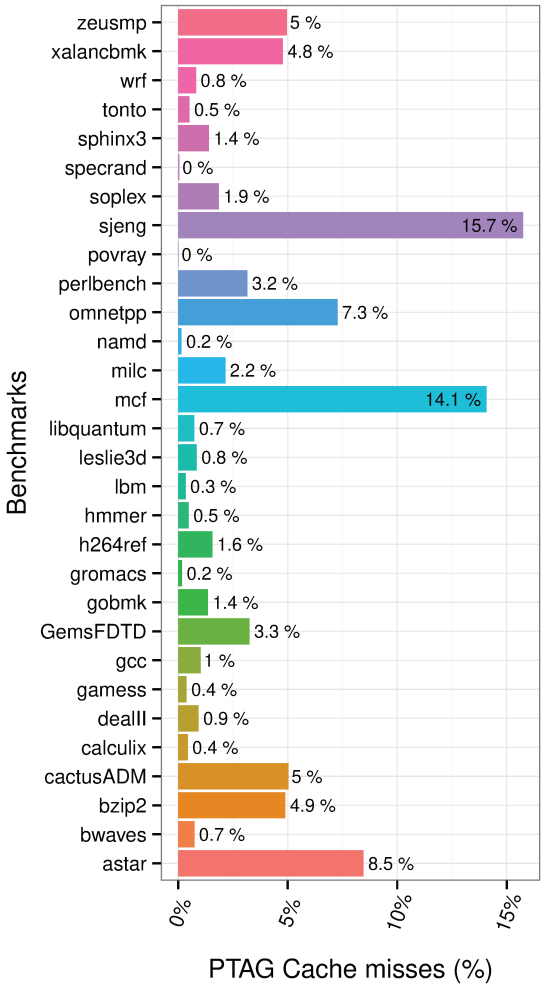


Figure 6: PTAG Cache miss rate for the SPEC 2006 benchmarks.

of programs, mainly when the L2 cache is small. Nonetheless, further studies on performance, power consumption, and area are needed for a thorough comparison with the work of Gassend *et al.* This is left for future work.

Notice also that the PTAG-Cache replay attack solution specifically employs authentication trees. However, there are other solutions to replay attack that uses caches [27], and a comparison involves figures of merit that are beyond the scope of this paper.

Finally, high PTAG-Cache MR might cause high frequency of execution stalls since, for each miss, the MCTRL has to verify an entire leaf-root path. Speculative execution would reduce this performance impact, and stalling the processor only when data leaves the secure area is a promising approach. Measuring the real impact and efficiency of such policies requires simulation and implementation. Nevertheless, preliminary results allow to conclude that the PTAG Cache may contribute to improve CSHIA performance.

6. SRAM-PUF KEY EXTRACTION

The main requirements of the key extraction process presented in Section 3 have to be validated by experimental

results. This section shows how to determine the number of assessments needed to extract a stable key for the PRF. It also shows how the statistical distribution of the possible keys behaves across different SPUFs.

6.1 SPUF Evaluation

As stated in Section 3, even choosing stable bits for the key, one cannot ensure that they will not flip and make the fuzzy extractor recover a wrong value for k. The BCH code to be used in CSHIA's fuzzy extractor (Figure 3) must satisfy two properties: (a) codeword length with the same bit-length as the key k (i.e. 128 bits); (b) syndrome length such that the random word r is at least as long as the PTAG length (i.e., $|r| \geq |p| = 64$ bits). A (127, 64, 10) instance of a BCH code has both properties and is able to detect and correct up to 10 errors. Since that code produces 127-bit codewords, a parity bit is computed over the 127-bit codeword and concatenated to it, resulting in c (Figure 3). It is worth noticing that the parity bit value is completely unknown to an attacker, since both r and the 127-bit codeword are unknown. Thus, neither a bit of c nor a bit of k will leak any information through this scheme.

At this point, one has to devise a strategy to extract stable bits and to measure the probability that these bits would not exhibit more bit-flips than the maximum allowed by the BCH-based fuzzy extractor. In a related article [17], Leest *et al.* measured the stability of SRAMs. They ran 500 power-up operations on a SRAM with the purpose of extracting random numbers and concluded that, after the first 100 power-up rounds, the amount of variability (entropy) of the measured SRAM contents becomes stable. Leest *et al.* then used this lower bound of 100 assessments to estimate the minimum entropy of other SRAMs, allowing them to discover the amount of bits they should extract in order to obtain random words with the same length. This result is useful for CSHIA implementation, because it establishes that stable random bits can be extracted from a SRAM if their locations are known.

However, even after 100 assessments, there is no guarantee that, after picking 128 bits to compose the key k, the number of bit-flip errors will not exceed the limit of the chosen BCH code (10 errors). To estimate the probability of more than 10 errors in a 128-bit key, an experiment with SRAMs available in 10 Altera DE1 development kit was conducted. Although each SRAM had 512 KB, a much larger size than a conventional L1 cache in modern processors, only the first 64 KB were analyzed.

In the first experiment, each SPUF is subjected to 200 power-up cycles (assessments), in which the first one is considered the reference assignment. Figure 7 shows the percentage amount of bits that present a different value as compared to the reference, at least once. One can notice that after around 100 assessments the curves start to flatten off, which reproduces the result by Leest *et al.*

Now, one has to determine the probability of finding more than 10 errors with respect to the reference assessment, when reading 128 bits out of these potentially stable bits. Recall that finding such a high number of errors would be a problem because the BCH code in the fuzzy extractor deals with at most 10 errors. Figure 8 shows the probability of having more than 10 bit flips when choosing 128 bits at every assessment after the 100th round. The measurements were conducted in the range 100 to 200 rounds of assessments, given that the SRAM behavior is not expected to change significantly after 200 assessments. In order to decide what is the recommended number of assessments, one can take the reference probability of one part in a million (1 p.p.m) [19] – shown as a horizontal line in the Figure 8.

One can notice in Figure 8 that, after selecting 128 bits from those that have not flipped yet after 160 assessments, it is highly unlikely to obtain unstable keys from any SPUF. However, this method requires keeping track of the addresses of 128 bit locations. Each address takes 19 bits in a 64 KB memory, resulting in an overhead of $128 \cdot 19$ bits $= 2432$ bits – equivalent to 5 cache lines. To reduce this overhead, one can look for stable memory words, rather than looking for single bits. The experiments show that picking stable words reduces not only the address storage overhead, but also the number of assessments to reach the probability of 10^{-6}. Figure 9 shows the key correction failure probability with respect to the number of assessments when picking 16-bit stable words. Unfortunately, it was not possible to find stable words with more than 16 contiguous bits for any SPUF.

6.2 Key Extraction

This section presents the key extraction algorithm that can be automated and used over the previously described data, and an analysis of the statistical properties of the resulting keys. Figure 9 clearly shows that there is strong correlation between the key correction failure probability and the number of assessments. Thus, one can run a linear regression model to predict the number of assessments that produces the expected low failure probability. After conducting such experiment, with 99% confidence at the prediction interval, 149 was found to be a conservative number of assessments to reach the 10^{-6} probability.

This analysis is conducted during the learning phase (technology maturation) over all evaluated SRAM instances, and results in a single number, the maximum number of assessments. Once this number is determined, the following algorithm can be executed for each SRAM instance, producing as result a list of reliable and stable memory addresses: (1) Reset the L1 data cache of the processor. (2) Copy data cache contents to the instruction cache. (3) Reset (power-off cycle) the data cache again. (4) If a word in the instruction cache is set as valid, compare it against the recently read data-cache word at the same address. If they differ, set the instruction-cache word as invalid, else, keep it valid. (5) Repeat the step (3) until reaching the maximum number of assessments. Now a list of reliable cache addresses is

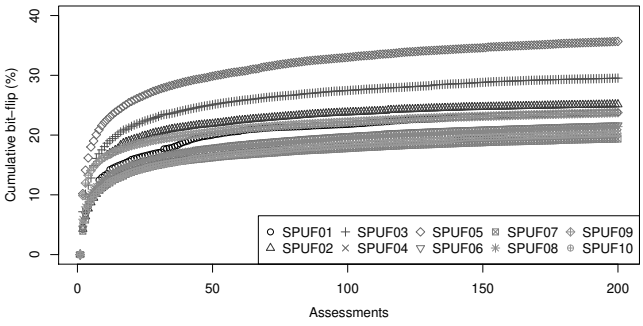


Figure 7: The fraction of SPUF bits that have flipped during 200 assessments.

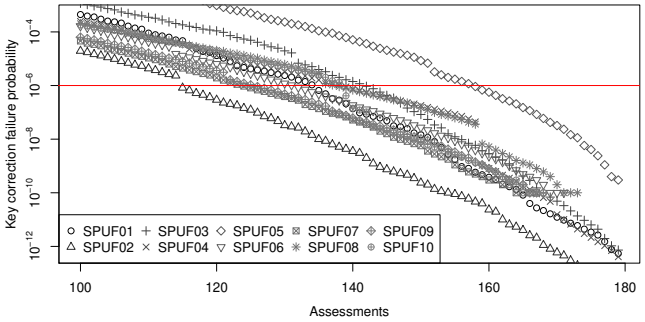


Figure 8: The probability of (127, 64, 10) BCH code failure in correcting a 128-bit key when composing it from single bits extracted from SPUFs.

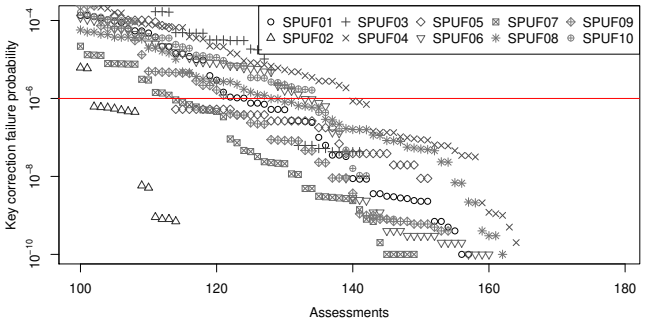


Figure 9: The probability of (127, 64, 10) BCH code failure in correcting a 128-bit key when compounding it with 16-bit words picked from SPUFs.

obtained for one specific SRAM instance.

The execution of this algorithm in each SRAM instance would take at most $O(n \cdot m)$ operations, for a memory with n words and running m assessments. However, as the algorithm execution proceeds, the number of valid lines to be verified decreases. For instance, after 100 assessments, less than 5% of n words are still valid for a 64-KB SPUF with 16-bit words. Therefore, the algorithm should run faster than $O(n \cdot m)$. After running m assessments, one can select the appropriate set of addresses of SPUF that are valid for composing the set A of addresses (see Section 3.2). In the case of 128-bit key and 16-bit words, eight addresses would be needed to compose the set of addresses A. This information is then recorded in a non-volatile memory to be used every time the system is powered up.

It is also critical to validate whether the keys extracted as described have adequate statistical properties. In particular, it is important to analyze how extracted keys differ across different SRAM instances. The established method in the literature for measuring this property is the Hamming Distance. The ideal behavior would be a normal distribution centered at 64, for a 128-bit long key. The Hamming Distance of the extracted keys presented in Figure 10 shows a random distribution centered at 61.11. This is slightly skewed towards zero, but still close to the ideal distribution. Finally, one can notice that the same analysis and result applies to the extraction of r (see Figure 3).

Differently from previous architectures that use PUFs [30], CSHIA does not need additional PUF circuitry, since the processor's SRAMs cache are used as PUFs. Also,

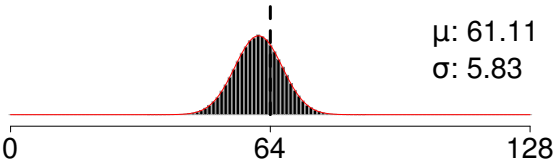


Figure 10: Hamming Distance distribution of 128-bit keys extracted from SPUFs.

CSHIA key extraction is deeply integrated to the architecture and is the first of this kind, to the best of our knowledge. Nonetheless, comparison with others SPUF key extraction processes in the literature [19] still needs to be done. Experimental results presented in this section showed that good quality keys can be extracted from SRAMs with relatively straightforward procedures.

7. CONCLUSIONS AND DISCUSSIONS

IoT devices need robust security, which goes beyond the traditional approaches based on typical cryptographic mechanisms proposed so far. Such solutions should work from system power-on to power-off, should not impact performance, and consume very low-power, while avoiding changes in the compiler and operating system tool-chains. This paper presents a particular way for achieving this goal, employing a solution deeply integrated to the processor microarchitecture which relies on intrinsic hardware information to authenticate program execution. Authentication and integrity are preserved by computing and verifying a PRF with key material extracted from the processor's cache memory. In addition, a new strategy for caching nodes from an authentication tree allows to efficiently extend security properties to external memory. The effort consists in an important step towards realizing this architecture, while several challenges remain ahead.

8. FUTURE WORK

In the near future, simulation and FPGA implementation will enable deeper analysis and evaluation of the performance and security of CSHIA. These results will inform potential changes in CSHIA and validate design decisions presented in this work. Afterwards, a real silicon prototype is intended in order to cover all levels of evaluation of the proposed architecture model.

Acknowledgment

We thank CAPES, CNPq (grant nº 147614/2014-7), and Intel Research grant "Energy-Efficient Security for SoC Devices – Physical Unclonable Functions" for supporting this work.

9. REFERENCES

[1] F. Armknecht, R. Maes, A.-R. Sadeghi, F.-X. Standaert, and C. Wachsmann. A formalization of the security features of physical functions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 397–412, Washington, DC, USA, 2011.

[2] J. Aumasson and D. J. Bernstein. Siphash: A fast short-input PRF. In *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on*

Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings, pages 489–508, 2012.

[3] E. B. Barker and A. L. Roginsky. Sp 800-131a. transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. Technical report, Gaithersburg, MD, United States, 2011.

[4] M. Bhargava and K. Mai. An efficient reliable puf-based cryptographic key generator in 65nm cmos. DATE '14, pages 1–6, March 2014.

[5] C. Brzuska, M. Fischlin, H. Schröder, and S. Katzenbeisser. Physically uncloneable functions in the universal composition framework. In *CRYPTO'11*, pages 51–70, Berlin, Heidelberg, 2011.

[6] J. Delvaux and I. Verbauwhede. Attacking puf-based pattern matching key generators via helper data manipulation. In J. Benaloh, editor, *Topics in Cryptology – CT-RSA 2014*, volume 8366 of *LNCS*, pages 106–131. 2014.

[7] R. Elbaz, D. Champagne, R. Lee, L. Torres, G. Sassatelli, and P. Guillemin. Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks. CHES'07, pages 289–302. 2007.

[8] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, C. Anguille, C. Buatois, and J. Rigaud. Hardware engines for bus encryption: a survey of existing techniques. In *Design, Automation and Test in Europe, Proceedings*, pages 40–45 Vol. 3, March 2005.

[9] B. Gassend, G. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. HPCA-9, pages 295–306, Feb 2003.

[10] O. Goldreich. *Foundations of Cryptography: Basic Tools*. New York, NY, USA, 2004.

[11] T. C. Group. Trusted platform module - main specification, 2014. accessed 02/17/2014, `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`.

[12] J. Guajardo, S. S. Kumar, G. J. Schrijen, and P. Tuyls. Physical unclonable functions, fpgas and public-key crypto for ip protection. In *FPL*, pages 189–195, 2007.

[13] C. Helfmeier, C. Boit, D. Nedospasov, and J.-P. Seifert. Cloning physically unclonable functions. In *Hardware-Oriented Security and Trust (HOST), IEEE International Symposium on*, pages 1–6, June 2013.

[14] A. V. Herrewege, V. van der Leest, A. Schaller, S. Katzenbeisser, and I. Verbauwhede. Secure prng seeding on commercial off-the-shelf microcontrollers. *IACR Cryptology ePrint Archive*, 2013:304, 2013.

[15] M. Hong, H. Guo, and S. X. Hu. A cost-effective tag design for memory data authentication in embedded systems. CASES '12, pages 17–26, 2012.

[16] S. Katzenbeisser, U. Kocabaş, V. Rožić, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann. Pufs: Myth, fact or busted? a security evaluation of physically unclonable functions (pufs) cast in silicon. CHES'12, pages 283–301, Berlin, Heidelberg, 2012.

[17] V. Leest, E. Sluis, G.-J. Schrijen, P. Tuyls, and H. Handschuh. Efficient implementation of true random number generator based on sram pufs. In D. Naccache, editor, *Cryptography and Security: From Theory to Applications*, volume 6805 of *LNCS*, pages 300–318. 2012.

[18] D. Lim, J. W. Lee, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas. Extracting secret keys from integrated circuits. *IEEE Trans. Very Large Scale Integr. Syst.*, 13(10):1200–1205, Oct. 2005.

[19] R. Maes, P. Tuyls, and I. Verbauwhede. Low-overhead implementation of a soft decision helper data algorithm for SRAM pufs. CHES'09, pages 332–347, 2009.

[20] M. Majzoobi, F. Koushanfar, and M. Potkonjak. Techniques for design and implementation of secure reconfigurable pufs. *ACM Trans. R. Tech. S.*, 2(1):1–33, 2009.

[21] D. Merli, D. Schuster, F. Stumpf, and G. Sigl. Side-channel analysis of pufs and fuzzy extractors. In J. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, editors, *Trust and Trustworthy Computing*, volume 6740 of *LNCS*, pages 33–47, 2011.

[22] D. Nedospasov, J.-P. Seifert, C. Helfmeier, and C. Boit. Invasive puf analysis. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 30–38, Aug 2013.

[23] R. S. Pappu, B. Recht, J. Taylor, and N. Gershenfeld. Physical one-way functions. *Science*, 297:2026–2030, 2002.

[24] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, and J. Schmidhuber. Modeling attacks on physical unclonable functions. CCS '10, pages 237–249, New York, NY, USA, 2010.

[25] U. Rührmair. Oblivious transfer based on physical unclonable functions. In A. Acquisti, S. Smith, and A.-R. Sadeghi, editors, *Trust and Trustworthy Computing*, volume 6101 of *LNCS*, pages 430–440. 2010.

[26] A. Sadeghi and D. Naccache, editors. *Towards Hardware-Intrinsic Security - Foundations and Practice*. Information Security and Cryptography. 2010.

[27] G. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. MICRO-36, pages 339–350, Dec 2003.

[28] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. ICS '03, pages 160–171, 2003.

[29] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. DAC '07, pages 9–14, 2007.

[30] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. ISCA '05, pages 25–36, 2005.

[31] S. Tajik, E. Dietz, S. Frohmann, J.-P. Seifert, D. Nedospasov, C. Helfmeier, C. Boit, and H. Dittrich. Physical characterization of arbiter pufs. CHES'14, pages 493–509. 2014.

[32] B. Škorić, P. Tuyls, and W. Ophey. Robust key extraction from physical uncloneable functions. In *Applied Cryptography and Network Security*, volume 3531 of *LNCS*, pages 407–422. 2005.

Implementing a Secure Architecture for Code and Data Authenticity and Integrity in Embedded Systems

Caio Hoffman[a,*], Augusto F. R. Queiroz[a], Diego F. Aranha[a,b], Mario L. Côrtes[a], Guido Araujo[a]

[a] *Institute of Computing, University of Campinas, Brazil*
[b] *Department of Engineering, Aarhus University, Denmark*

Abstract

Computer Security by Hardware-Intrinsic Authentication (CSHIA) was recently proposed as a secure architecture to provide code and data authenticity and integrity. The architecture of CSHIA allows embedded system designers to adjust many parameters without compromising its security. In this work, using Gaisler's Leon3 platform, we implemented an improved version of CSHIA in FPGA. We not only strengthened its security features, but also introduced new features that give designers choice between two countermeasures against replay attacks: timestamps or Merkle Tree. The performance evaluation showed that CSHIA using timestamps can have only 2.76 % average performance overhead, while CSHIA using Merkle Tree reaches a 5.77 % average performance overhead. That, in conjunction with power and area estimates, showed that CSHIA with timestamps can be a very advantageous option for constructing secure embedded systems.

Keywords: Hardware Security, Security Analyses, Physical Unclonable Functions, Replay Attacks, Merkle Tree, FPGA.

1. Introduction

The demand for code/data integrity and authenticity has steadily increased. The wide spectrum of known attacks currently poses a threat to a variety of embedded systems that need constant protection against tampering. A particular class of embedded systems which must resist many forms of tampering comprises systems equipped with a large external non-volatile memory to store software and data, such as voting machines, smart metering devices and employee attendance control systems. These systems need to provide integrity and authenticity guarantees, but usually not secrecy or confidentiality, in order to be easily audited by government authorities and independent experts.

Due to the stringent nature of available resources of embedded systems, software solutions for code and data integrity do not fit best. In addition, software authenticity would involve a third party certification authority. Therefore, hardware solutions are desirable for such systems. A myriad of hardware solutions for code and data authenticity and integrity have been proposed in the literature ([1, 2, 3, 4]), however, some of those solutions target high-end embedded systems or more powerful configurations, requiring at least a two-level cache in the processor for their performance overhead not to be prohibitive. Other approaches need modifications on the Instruction Set Architecture (ISA) or processor datapath, leading to complete redesign of code, compilers, operating systems, among others. Moreover, not all solutions provide integrity and authenticity.

Recently, an architecture aiming at code/data authenticity and integrity was proposed in [5]. The Computer Security by Hardware-Intrinsic Authentication (CSHIA) provides authenticity by authenticating all memory blocks of the external memory using a unique key extracted from Physical Unclonable Functions (PUFs) implemented in each instance. The authentication tags (called PTAGs) are computed during an enrollment procedure and later verified or updated on runtime for each memory block brought to the processor. The main advantages of CSHIA over the previous hardware solutions are that it does not require changes in the ISA or datapath, being adaptable to most of embedded system architectures while providing complete software compatibility, and also using a separate bus for the tag memory, which gives to designers freedom to match timing requirements to hide verification overhead.

Basing on Gaisler's Leon3 [6] FPGA implementation, this work presents a proof-of-concept of CSHIA. The main goal of our implementation was to improve the original version of the architecture and add more flexible design choices. Besides presenting an in depth description of the integration between the architecture and a real processor, we evaluated performance and storage overheads, computed area and power estimates, and also performed a security analysis. The CSHIA implementation enables two solutions against replay attacks: timestamps or Merkle Tree. To the best of our knowledge, it is the first time that both solutions are evaluated in the same architecture. The results showed that the CSHIA's timestamp instance is the best one, when taking into account performance degradation, area and energy overhead. It presented only 2.76 % of performance penalty on average, while the Merkle Tree instance showed an average 5.77 % reduction on performance. Finally, we discuss how the CSHIA flexible design is a very attractive solution for embedded systems when compared to state-of-the-art architectures.

*caio.hoffman@ic.unicamp.br

This work is organized as follows. Security issues in embedded system are discussed in Section 2. Section 3 describes the architecture. Section 4 provides details about the implementation of CSHIA in the Leon3's platform. Section 5 discusses experiments and results. A security analysis of the CSHIA implementation is presented in Section 6. Section 7 discusses related work and Section 8 concludes this work.

2. Security Issues in Embedded Systems

The integration between different hardware and software components, protocols, and I/O devices in embedded systems can make it difficult to protect against attacks due to the complexity that arises from such combinations. In addition, in many cases, embedded systems will run autonomously and unsupervised, facilitating physical control by an adversary. While dealing with security breaches in all components in an embedded system can be complex and inefficient, a common threat model adopted by several works in the literature [1, 2] considers that the main chip, which has the processor, can be secured and any other component, like memories, buses, among others, is vulnerable to manipulation. In this scenario, most of the threats can be grouped in three main classes of attacks, which are discussed next. Afterwards, security features for embedded systems that can be employed as countermeasures against these attacks are presented.

2.1. Threat Model

Since all components but the main processor are exposed to an attacker, external modifications reach the processor by its buses. Attempts of tampering with these components can then be reduced to tampering with the buses. Alternatively, because modern embedded systems use direct memory access, which makes processors see peripheral hardware as memory, most threats can be abstracted as attacks against the main memory.

Considering this simplification, there are three distinct attack scenarios: (i) the attacker arbitrarily inserts/modifies memory words (spoofing attack); (ii) the attacker swaps content between different memory locations (splicing or relocation attack); (iii) the attacker replaces the current content of a memory word by an older value (replay attack). These attacks against the main memory are illustrated in Figures 1 and 2. In short, by tampering with the memory contents, the attacker successfully changes the instructions/data the processor will receive when performing read operations.

2.2. Security Properties

In order to counter the attacks discussed above, a system designer can employ mechanisms implementing three security properties: authenticity, integrity, and secrecy. Although these features can be implemented through software, the stringent nature of embedded systems demands solutions that consume few clock cycles and are not power consuming. In the following, we discuss hardware implementation of those security features.

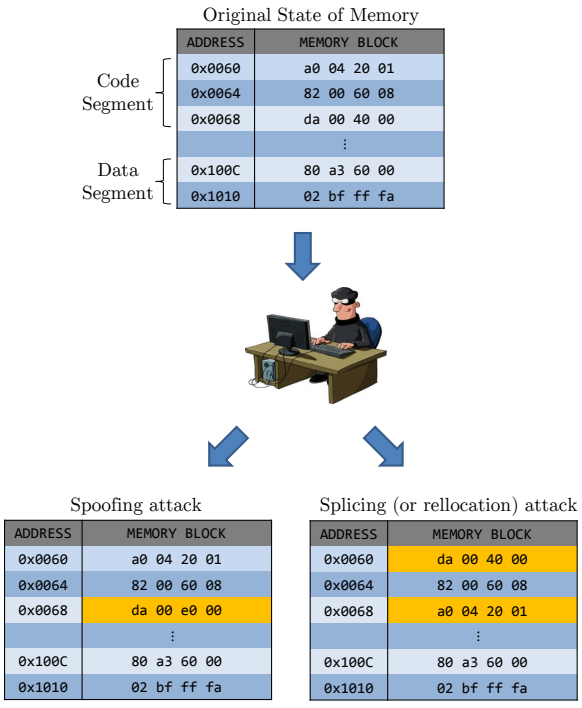


Figure 1: Spoofing and Splicing (or Relocation) attacks.

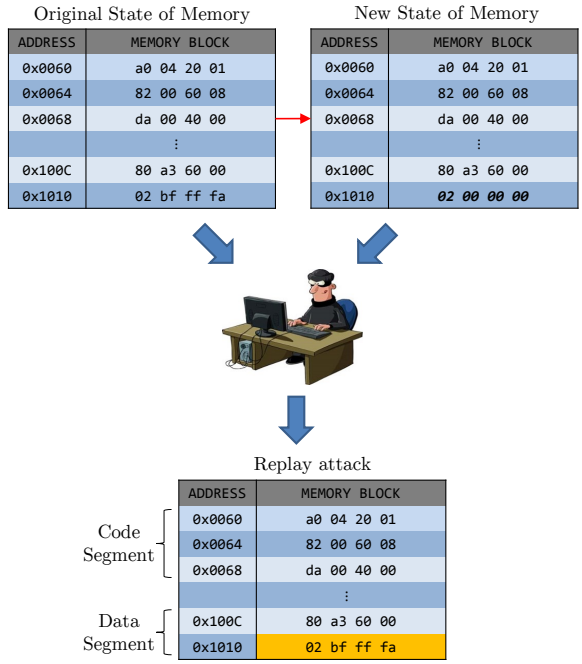


Figure 2: Replay attack.

2.2.1. Authenticity

Suppose that an attacker wants to add his/her 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 so-

lution, a key (or unique set of keys) is determined for each instance. Code and/or 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.

Previously the introduction of electronic Physical Unclonable Functions (PUFs) [7], 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 [8], 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 (CRPs) [7].

2.2.2. *Integrity*

Similarly to authentication, integrity is ensured by tagging code and data with additional information such as memory address location and/or 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 [9], that can result not only in changing the Instruction Set Architecture (ISA) and/or the tool-chain, but also the processor's data path, as proposed in [10, 11]. 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 I/O 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.

2.2.3. *Secrecy*

An embedded system can also use encryption to prevent exposure of code and/or 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 and/or to suffer relocation and replay attacks.

2.3. *Analysis of Countermeasures*

The security properties described above can be used to counter the attacks presented in the threat model. Next, countermeasures providing these security properties against attacks on embedded systems are discussed together with their impact on design.

2.3.1. *Preventing Spoofing Attacks*

Authentication tags can be computed from memory blocks using a unique on-chip secret key and robust cryptographic primitives. By verifying pairs of memory blocks and tags, a system can detect arbitrary values inserted by an attacker in memory or buses. Figure 3 shows how tags can prevent a spoofing attack, in which an attacker wants to modify instructions. Notice that unique secret keys will inhibit an attacker mimicking tag creation of any device instance, even if he/she has access to all tags and memory blocks of multiple instances.

Regarding downsides, one important matter about tags is the size of the block or part of memory that is tagged. Large blocks result in low storage overhead because the number of tags will be smaller, but high performance penalties since it takes more time to generate a tag. This situation is inverted when smaller blocks are used. Generally, the granularity of cache lines (or memory blocks) are chosen to provide the best trade-off and it is the most common choice found in the literature [12, 3, 4, 2].

2.3.2. *Preventing Splicing or Relocation Attacks*

If tags result only from processing memory blocks, a splicing or relocation attack (see it in Figure 1) can still be applied. A system can deal with this attack by creating tags that include not only content but also location in memory. Since for most of the cryptographic primitives used to generate tags the content order matters, the initial address of a block is enough. The prevention of this attack does not add any significant overhead or complexity to the system, because address length is considerable smaller than blocks and both can even be combined for fast tagging. Figure 3 shows that an attacker fails in this attack when he/she tries to swap blocks in memory. Even swapping tags will not work, due to their generation depending on the address-block pair.

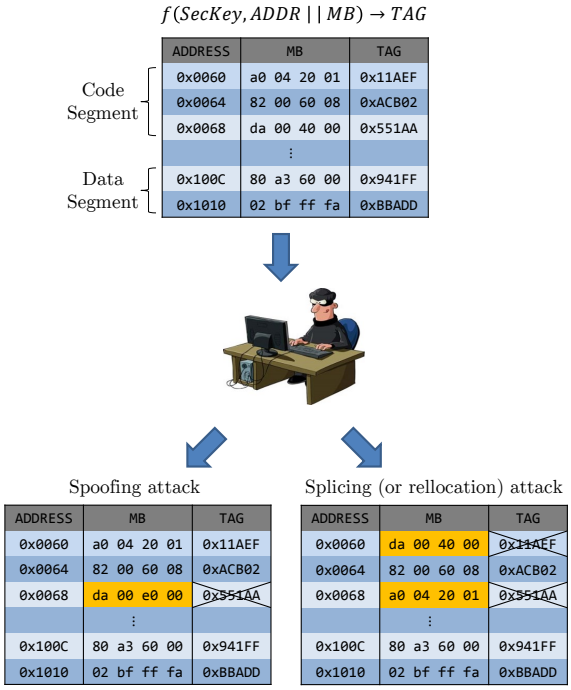


Figure 3: Preventing Spoofing and Splicing (or Relocation) attacks.

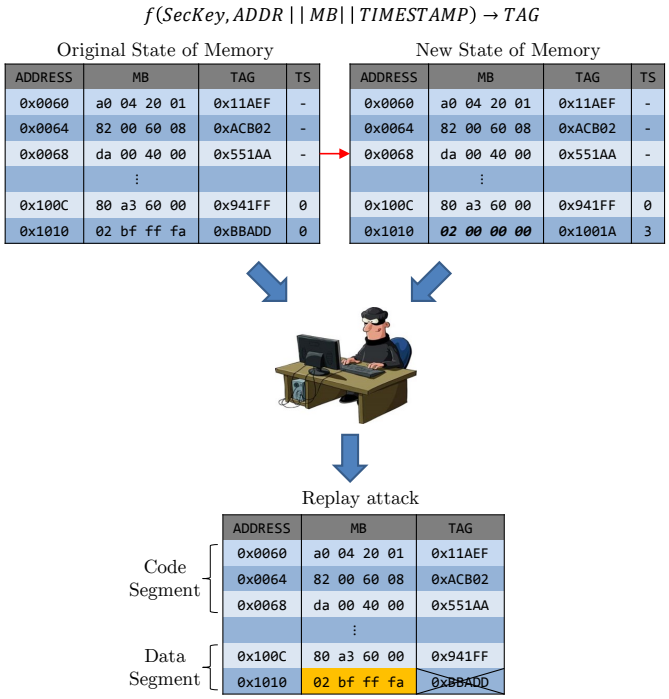


Figure 4: Replay attack prevention using Timestamps.

2.3.3. *Preventing Replay Attacks*

A final trick an attacker can do to bypass previous solutions is to replace a block and its tag by an older version. Since old values may remain consistent, the system will not detect this modification unless some information about the current state of the system is available. Figure 2 presents this situation, that usually targets data since code is not constantly modified in memory. The information of the current state can be added to the system in two forms: a tag of all tags (or root tag) or timestamps for each block of content. Figure 4 shows how timestamps can prevent replay attacks and Figure 5 shows how a root tag works. The reader should notice that in both cases neither the timestamps nor the root tag are exposed off-chip. This implies that both solutions need a non-volatile storage on chip. Hence, while timestamps need an on-chip memory, a root tag only requires memory element likely to be the size of a register.

It is worth noticing that a root tag can degrade performance significantly, to a point in which can be impractical. To minimize such impact on performance when using a root tag, a tree structure can be applied. This tree is known as Merkle Tree. The literature has proposed a vast number of approaches of how to implement this tree, and details can be found in [12, 13]. The tree is constructed by grouping tags in chunks and creating tags for them. By recursively repeating this procedure until only one tag is left, we obtain the root tag. Figure 6 illustrates this process, in which a chunk is composed of 4 tags of memory blocks, and the next level of the tree has 4 tags that originated from each descendant chunk. The root tag results from the chunk created by the tags in the second level. Notice that all intermediate tags

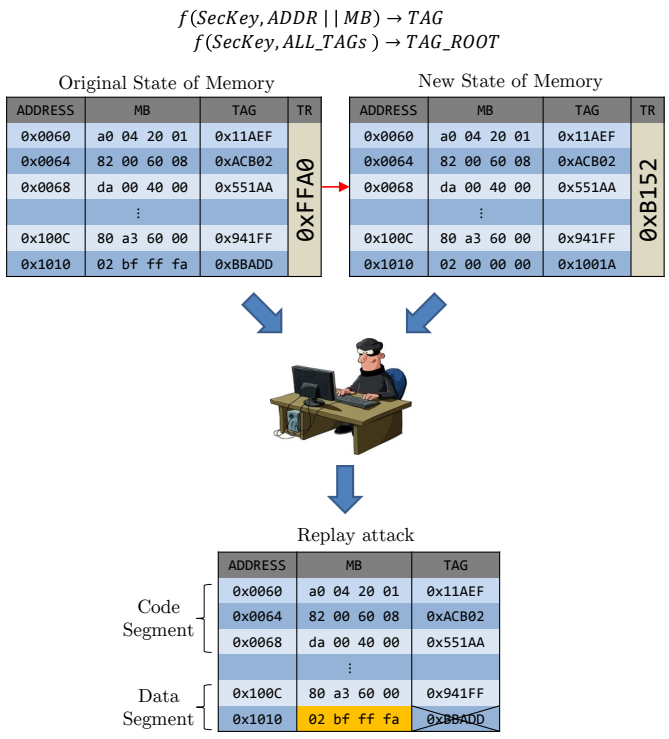


Figure 5: Replay attack prevention using Merkle Tree.

between the root tag and the leaves may be stored off-chip, with the root tag as the only one that cannot be exposed.

Even though the tree seems to be an elegant solution to reduce performance penalties of the root tag solution, usually it

is not enough. Because every read and write of memory blocks will require verification and update of the nodes in the leaf-root path, an on-chip cache for nodes of the Merkle Tree can be implemented to avoid constant complete walks from leaves all the way to the root. More details can be found in [14, 15]. Therefore, the dilemma is: timestamps are likely to provide lower performance penalties since verification is simple, but they can require larger non-volatile on-chip memories. On the other hand, Merkle Tree may only need a non-volatile on-chip register, but in order to reduce performance slowdown a cache should be added, which will adversely affect dynamic power and area. Further details in the downsides of both solutions are discussed in depth in the next sections.

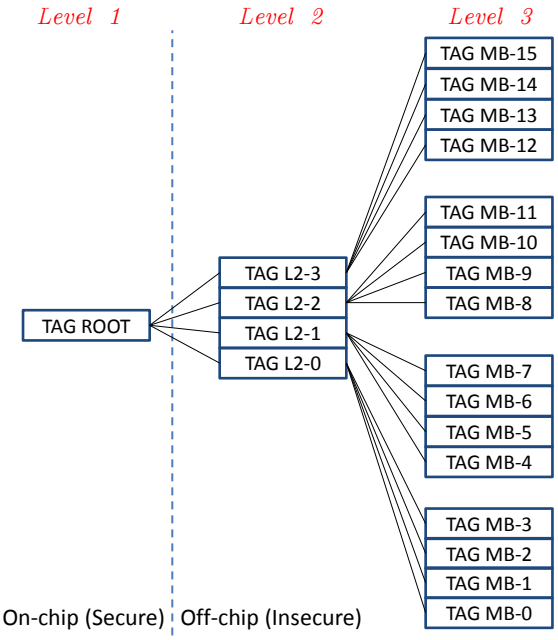


Figure 6: Merkle Tree structure to 16 memory blocks.

2.4. The Matter of Secrecy

Although secrecy mitigated some attacks by simply concealing the contents of the memory, many embedded system application do not need secrecy. For instance, applications that use open source programs and/or have data that must be audited at some point. In general, encryption primitives are slower than authentication due to their reversible nature. Different secure architectures that implement secrecy using AES report memory block encryption latency between 13 [16] to 20 cycles [3] without estimating real impact on area and power. In [17], a 16-byte block is assumed to be encrypted in 10 cycles using AES with an estimated area of 25,000 gates. However, a Pseudo-Random Function (PRF) such as SipHash [18], can be implemented in 7,900 gate-equivalent elements capable of processing 20 bytes in 5 cycles. Therefore, embedded systems can have authenticity and integrity with very low performance penalties, when secrecy is not needed. Next, we present the CSHIA architecture, which assuming the threat model and security issues and solutions discussed in this section, provides authenticity and integrity for code and data.

3. CSHIA

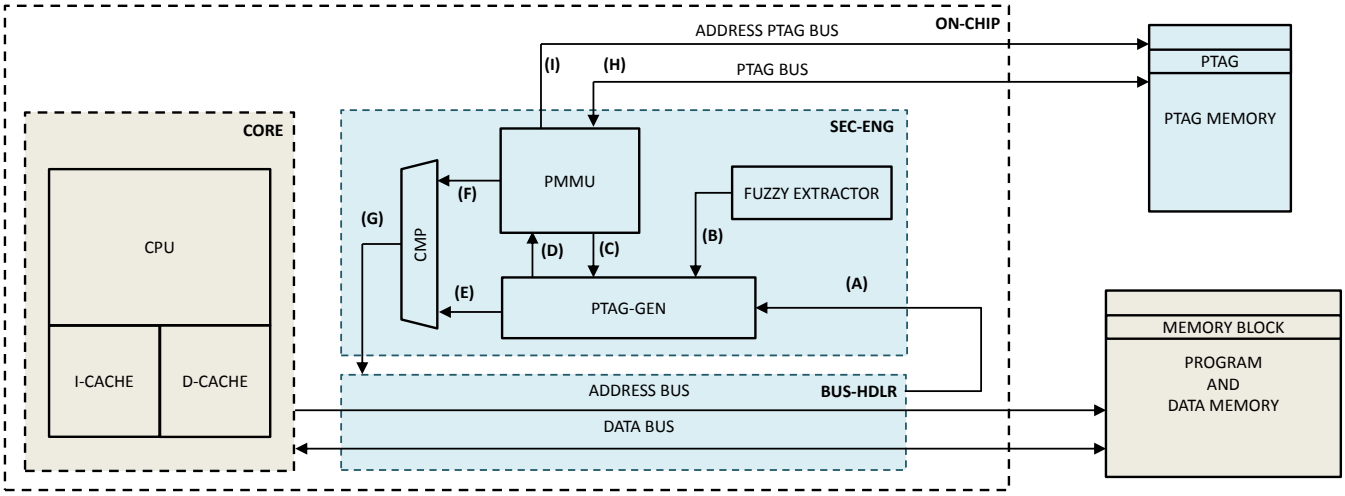
CSHIA was originally proposed in [5] as an architecture for IoT. However, we believe that CSHIA fits in a variety of embedded system applications that can benefit from its architectural design decisions. As we stated before, many embedded system applications do not need secrecy/confidentiality, but strongly require code and data authenticity and integrity. Using the original work as base, we modified some elements to provide stronger security features, as well as make CSHIA adaptable to a FPGA implementation. This section focus on presenting our CSHIA main architectural components and how they work to provide authenticity and integrity.

3.1. Components of the Architecture

As Section 2 discussed, the main resource to provide integrity are tags. Since CSHIA uses PUF-based keys to generate tags, we called them PUF-based Tags, or PTAGs for short. PTAGs are the core of CSHIA's design. They will be unique for each instance of CSHIA due to the unclonability property of PUFs. That ensures one-to-one relationship between programs and instances, providing authenticity. To handle PTAGs, three main components are added to a conventional embedded system architecture. They are: The PTAG Memory; the Bus Handler (BUS-HDLR); and the Security Engine (SEC-ENG). Figure 7 shows this design and how components communicate between themselves.

PTAG Memory is an external memory and has its own buses. This architectural decision gives freedom to designers that can choose bus width, frequency, address space, etc. Because the processor is not aware of any additional component of CSHIA, BUS-HDLR intercepts data transfers between processor and memory in order to provide them to SEC-ENG that generates tags. BUS-HDLR can also request data in behalf of the processor to main memory to form complete memory blocks that are necessary to generate PTAGs.

SEC-ENG has three major subcomponents. The main one is the PTAG Generator (PTAG-GEN), which uses input data whose length is equal to a memory block concatenated with its address to generate PTAGs. The Fuzzy Extractor is only used when the system loses its secret key. For instance, after a power cycle. Thus, when the system is powered on, the Fuzzy Extractor will extract the PUF-based key and provide it to PTAG-GEN. Finally, we have the PTAG Memory Management Unit (PMMU). The main functions of the PMMU are to store and request PTAGs from the PTAG Memory and also decode internal addresses of PTAGs to physical addresses of PTAG Memory. In addition to that, PMMU can have two distinct designs. If a designer chooses to use timestamps as solution for replay attacks, PMMU will have an internal memory to store and control timestamps of the memory blocks. However, if the solution for replay attacks is a Merkle Tree, PMMU will control verification and update of the tree, as well as it will have a cache memory, the PTAG Cache, to speed up these tasks.



(A) The BUS HANDLER (BUS-HDLR) provides memory block concatenated with physical address to the PTAG GENERATOR (PTAG-GEN).
(B) Fuzzy Extractor serves an extracted PUF-based key to PTAG-GEN.
(C) The PTAG MEMORY MANAGEMENT UNIT (PMMU) can either provide:
  (i) a chunk of PTAGs concatenated with their address location;
  (ii) or a time stamp for PTAG-GEN.
(D) PMMU receives a new PTAG when it requests generation to PTAG-GEN.
(E) PTAG-GEN provides PTAG for comparison.
(F) PMMU provides PTAG for comparison.
(G) If the comparison fails, a signal is sent to BUS-HDLR for action.
(H) PMMU sends and receives PTAGs from the PTAG Memory.
(I) PMMU decodes virtual PTAG address to physical address and sends it to PTAG Memory.

Figure 7: The CSHIA architecture.

Here is how CSHIA's components work together. BUS-HDLR checks for memory read-write operations of the processor. When it perceives a memory read it will capture memory words and/or request memory words to compose a memory block. Then it sends this memory block and its address to SEC-ENG. On its turn, SEC-ENG uses PMMU to bring the corresponding PTAG of that memory block from PTAG Memory, while PTAG-GEN computes a PTAG using the content served by BUS-HDLR. After that, the PTAG brought from PTAG Memory and the one computed are compared. If they match, SEC-ENG knows that neither the PTAG nor the memory block were tampered with. Otherwise, SEC-ENG alerts the handler that can isolate the processor or sends a non-maskable interrupt to the processor.

For write operations, the process is simpler. Once any memory block that reached the processor was verified for integrity and authenticity, BUS-HDLR can serve the cache line to SEC-ENG that uses PTAG-GEN to compute a new PTAG and PMMU sends that PTAG to PTAG Memory. We can see PMMU for now as a black box, until the end of this section, when we describe its complete functionality. The following subsections describe the two phases of CSHIA: enrollment and runtime.

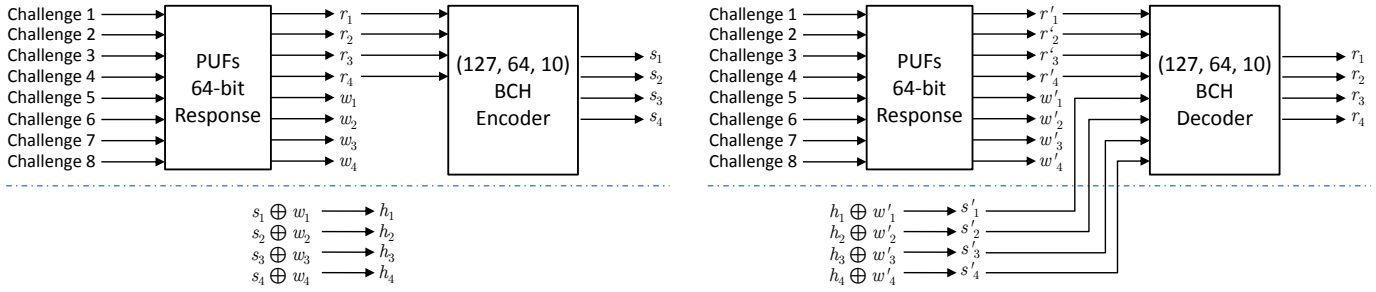
3.2. Enrollment Phase

In order to ensure authenticity and integrity, an initial procedure has to be conducted by the manufacturer/vendor. This enrollment procedure will activate the Fuzzy Extractor to extract the secret key from PUFs. Once that is done, the BUS-HDLR brings all memory blocks for tag generation. Next, we detail this procedure.

3.2.1. Key Extraction

PTAG Generator implements a Pseudo-Random Function (PRF), which is a primitive cryptographic very similar to a hash function with an important difference: the input processing is based on a secret key. In order to provide uniqueness to every CSHIA instance this key has to be unique. As aforementioned, PUFs cannot be cloned, thus they can provide this uniqueness. Nevertheless, one big conundrum of using electronic PUFs to generate keys is that they are inherently unstable. Due to their nature of leveraging on imperfection of the fabrication process, external factors such as temperature variation, voltage variation, etc., can interfere on their responses. Thus, varying responses to challenges during the lifetime of devices. In order to provide consistence in PUF responses, Fuzzy Extractor (FE) are employed. In simple terms, FEs are schemes comprised of an extraction algorithm and a recovery procedure. Becker provides a solid review and formal definitions in [19].

There are multiple ways of implementing a Fuzzy Extractor. Originally, CSHIA was proposed using a Code-offset FE, which is well-known to reduce entropy of extracted keys [20]. To strengthen the CSHIA design, we now use an adapted version of the Index-based Syndrome (IBS) FE proposed by Yu and Devadas in [21]. Figure 8 (a) illustrates the process of key extraction of CSHIA's FE. In general terms, a bit string r is extracted from PUFs. Then, the FE generates a syndrome s of r using a (n, k, t) Error Correction Code (ECC). The FE also



(a) Fuzzy Extractor during key extraction.

(b) Fuzzy Extractor during key regeneration.

Figure 8: Fuzzy Extractor actions during the enrollment and recovery procedure.

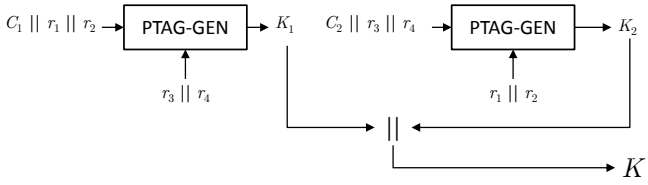


Figure 9: Key generation on CSHIA.

extracts a bit string w and combines it to the syndrome s to generate an encoded helper data h. This helper data h can be externally exposed and will not leak information about r (that can be used as secret key or derive the key).

To fully explain Figure 8 (a), the chosen parameters are detailed. First, CSHIA incorporates PUFs that produce 64-bit responses (more details in the next section). These PUFs will be responsible to generate each string r and w that are 64 bits long. To match the length of r and w, CSHIA has a $(127, 64, 10)$-BCH ECC. As Figure 8 (a) depicts, there are four bit strings r_i, which are compounded two by two and fed to the PRF (Figure 9). Such combinations were specifically designed to match the PRF chosen for CSHIA, the SipHash [18], which has an output of 64 bits and uses key of 128 bits. Therefore, the first pair of bit strings r_i is concatenated with a constant and processed by the PRF using the second pair of bit string r_i as key. That generates a hash K_1. Then, inverting their places and concatenating the second pair with a different constant, a hash K_2 is obtained. Concatenating K_1 with K_2 results in K which is the secret key of CSHIA. Notice that C_1 and C_2 in Figure 9 are replacing addresses for input of the PTAG-GEN. Further details of security will be given in the following sections, however, one can notice that assuming that each bit string r_i has at least half of their length of entropy, each part of the key will have full entropy. Hence, the key has full entropy.

3.2.2. *Full Memory Protection*

The Enrollment Phase proceeds to tag the memory range the manufacturer/vendor specified during design. Now that PTAG-GEN has an unique key, SEC-ENG orders BUS-HDLR to bring all memory blocks and deliver them to it. SEC-ENG will use PTAG-GEN to generate PTAGs, however, depending on the solution against replay attacks a designer chooses, PTAG-GEN is used differently.

Timestamps Generation. When timestamps are the solution against replay attacks, PMMU will have a timestamp memory. This timestamp memory has the depth of the number of data memory blocks the designer chose to cover. Thus, before BUS-HDLR hands in data memory blocks, PMMU will clear the entire timestamp memory to avoid uninitialized values. While SEC-ENG receives code memory blocks, generated PTAGs are just passed to PMMU that stores them in PTAG Memory. As BUS-HDLR starts to pass data memory blocks to SEC-ENG, PMMU increments the timestamp of each memory block received and passes this value to SEC-ENG, which combines with the address of the memory block. This combination is then concatenated with the memory block and then finally hashed into a PTAG. PMMU receives this PTAG and stores it in PTAG Memory.

Merkle Tree Generation. A Merkle Tree solution is more complex. The first procedure is very straightforward. SEC-ENG receives memory blocks and their addresses from BUS-HDLR and uses PTAG-GEN to generate PTAGs. PMMU receives these PTAGs and sends them to PTAG Memory. After all memory blocks had their PTAGs generated, PMMU starts to bring PTAGs of data memory blocks. As soon as a chunk of PTAGs is formed, a PTAG internal address of the chunk is calculated. PMMU provides this internal address and the chunk to PTAG-GEN that generates a PTAG. This PTAG is returned to PMMU that stores it in PTAG Memory. This process will continuously happen (as we can see in Figure 6) until PMMU identify that the last PTAG calculated has no siblings. Hence, it is the root PTAG, which must be stored inside PMMU. It is worth to clarify that PTAG internal address is an address space that facilitates computation and identification of descendants and ancestors. Each internal address is directly translated to a physical address by PMMU and this translation has as goal to minimize unused spaces in PTAG Memory. Moreover, in terms of security, this internal address mitigates a very specific attack on the tree, in which an descendant has the same PTAG as one of its ancestors. In this case, an attacker could try to perform a relocation attack likewise.

3.3. Runtime Phase

After the enrollment phase, CSHIA instances are ready for distribution. During the product lifetime, the device can be rebooted and turned off and on multiple times. While this will not affect PTAGs, which are externally stored in PTAG Memory, the secret key has to be recovered every time the system comes back from off-line periods. This recovery procedure of the Fuzzy Extractor is described next.

3.3.1. Key Regeneration

During the enrollment there were 8 challenges selected to produce four r_i and four w_i values. These challenges and helper data can be exposed off-chip and stored in PTAG Memory if the designer chooses to do so. The recovery process of the secret key can be seen in Figure 8 (b). After using the challenges and all helper data, the syndromes are recovered. Due to inconsistent nature of PUFs, the fuzzy extractor actually recovers bit-flipped versions w_i' and r_i', what leads to the BCH decoder receive r' and s'. Once bit flips in r_i values are corrected, the FE uses all r_i to regenerate the secret key as Figure 9 shows.

3.3.2. Runtime Protection using Timestamps

After recovering the key, CSHIA is ready to execute any program. Each instruction or data requested by the processor will be actually handled by BUS-HDLR. During reads, the processor requests a memory word, BUS-HDLR intercepts that request and informs SEC-ENG. While SEC-ENG then asks the PMMU to make the corresponding PTAG available, BUS-HDLR takes control of the buses and request all words that compound the memory block whose the requested word belongs. After buffering this memory block/cache line, BUS-HDLR sends it to SEC-ENG for verification. Meanwhile, PMMU made available the corresponding PTAG of that block and its timestamp (when the memory block is data and is in the covered region). After combining a timestamp (if necessary) with the address and concatenating them to the memory block, a PTAG is created. If this PTAG matches the one PMMU made available, a signal informs BUS-HDLR that it can serve the processor, otherwise the system is stalled from further action. Notice that the processor will receive instruction/data only when all three elements used for PTAG generation, timestamp, address, and memory block, match those that were used to produce the PTAG brought from PTAG Memory.

For writes, the process for serving the processor requisition can be longer. It will depend on the availability of the internal buffer of BUS-HDLR. If a write request is intercepted and it does not match any line in the buffer, this line has to be brought. But, before that, if the buffer is full, BUS-HDLR will have to discard a non-dirty line or write it back before releasing the buses to the processor. While BUS-HDLR writes back all words in a buffer line, it also sends this line to SEC-ENG. In its turn, SEC-ENG solicits to PMMU an updated timestamp for that block. After receiving it, a new PTAG is generated and SEC-ENG passes it to PMMU, which will write it into PTAG Memory. Once the dirty buffer line was all written back, BUS-HDLR starts a reading operation to bring the memory block of the data the processor wants to write. Only after that memory block is buffered, the buses will be granted to the processor.

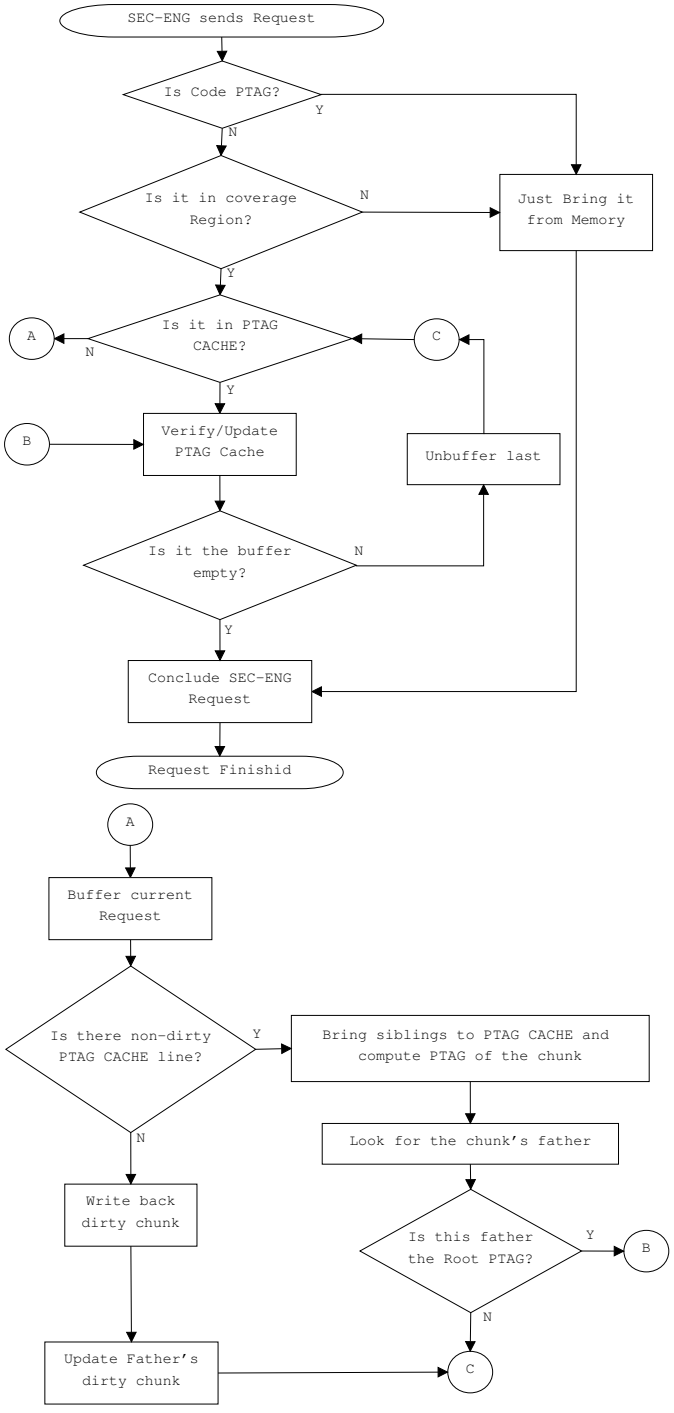


Figure 10: Flowchart of the PMMU's algorithm for verification and update of the PTAG Cache for the Merkle Tree implementation.

3.3.3. Runtime Protection using Merkle Tree

In CSHIA, the basic difference between using timestamps and Merkle Tree relies on the work PMMU has to do. For a solution based on Merkle Tree, PMMU has a small PTAG cache memory that only caches PTAGs of data memory block and its

ancestors in the tree. Nonetheless, the process in which the PMMU makes PTAGs available becomes very complex. For the sake of simplicity, we provide a flowchart of the PMMU's work in Figure 10. The same process described for read and write in the runtime protection using timestamps is also valid here. That means that the work of BUS-HDLR during processor's read/write requests is the same.

To summarize Figure 10, PMMU has a buffer to solve deadlocks when replacing dirty chunks of PTAGs in the cache. Most of those deadlocks happens because right after evicting a chunk, its ancestor has to be updated and thus the current write or verification has to be stalled. For intelligibility, we omit further details of implementation, they will be found in the publication of our source code and documentation about CSHIA implementation. Finally, information about caching Merkle Tree nodes can be found in [14].

After giving an overall look in the CSHIA architecture, all basic ideas are laid down for implementation. Next, we introduce our CSHIA FPGA implementation.

4. Implementation

We chose the Leon3 platform from Cobham Gaisler [6] to implement CSHIA. Leon3 is a VHDL implementation of a SPARC V8 processor with configurable parameters, which together with some additional IP cores provide a suitable solution for embedded systems. In addition, Leon3 has a free version for academic purposes that include sophisticated debugging tools, and it is available for a variety of FPGA Development kits. Gaisler keeps an email list for support and constant updates are provided. All these features are interesting because CSHIA can be an extension of the platform available to the research community, and which also has solid design choices since Leon3 is a product available to the industry.

The implementation is based on Figure 7 in Section 3. Leon3's processor (the core) is connected through the main memory by a AMBA Bus version 2.0. In our modification, the processor's I/O master bus connects it to BUS-HDLR, which then provides a new I/O master bus for the rest of the components in the platform. Thus, BUS-HDLR is transparent to all components of the platform, even the core. One of the components that is specific of Leon3's platform is the Debug Support Unit (DSU), which allows a designer using a debug host (such as a computer) to connect to development kits running Leon3. Through the debugging connection, a program can be loaded to the FPGA memory, started, paused, among other useful functions.

We implemented CSHIA in an Altera FPGA Development Kit DE2-115. The parameters of the processor and CSHIA are in Table 1. The Altera's kit allows the processor to run at 50 MHz. The total amount of SDRAM memory dedicated to Leon3 is 128 MB. As convention all programs starts by its `.text` segment (code) at the address `0x40000000`. We set `.data` segment (data) to start at `0x40013000`, or at `0x40023000`, depending on the size of the code segment. As described in the previous section, BUS-HDLR has a buffer that stores memory words. When these words form a memory

Table 1: CSHIA FPGA implementation configuration.

Component	Parameter
Leon3 Processor	
Frequency	50 MHz
Instruction Cache	16 KB
Data Cache	16 KB
Cache Line Size	256 bits
Memory Word	32 bits
Code and Data Memory	Up to 128 MB
Code Start Address	`0x40000000`
Data Start Address 1	`0x40013000`
Data Start Address 2	`0x40023000`
BUS-HDLR Buffer	128 Bytes
Fuzzy Extractor	
ECC	(127,64,10)-BCH
PUFs	64 × 64-bit Arbiter PUFs
PTAG-GEN	
PRF	SipHash-2-4
SipHash-2-4 key	128 bits
PTAG generation	10 cycles
PTAG length	64 bits
PTAG Memory	216,064 bytes
Code and Data PTAGs	18816 words of 64 bits
Merkle Tree PTAGs	8192 words of 64 bits
Data coverage	512 KB
Total coverage	588 KB
PMMU	
Time Stamp Memory	2^{14} timestamps
Time Stamp Length	16 bits
PTAG Cache	4 KB
PMMU Buffer for Merkle Tree	2 * number of cache lines

block, it is handed to SEC-ENG. We set the size of this buffer to 4 cache lines, which gives a total of 128 bytes. The 128-bit SipHash's key is extracted from 64 Arbiter PUFs (APUFs). Although any PUF could be used, for the sake of design simplicity we chose the APUF as a proof of concept. Each APUF has a 64-bit challenge input. PTAG generation lasts 10 cycles, between SEC-ENG request and PTAG-GEN reply. Our SipHash implementation is an adaption of the VHDL version available in [22].

Continuing to look at Table 1, the PTAG Memory uses internal memory of the FPGA. This option arose due to limiting options available in the kit. Because we wanted to design a 64-bit bus memory, no better option than internal memory was available. The SRAM of the kit only allowed 16-bit words. We also could not increase the frequency of the SRAM using PLLs since its maximum frequency was limited to 125 MHz, and, to simulate a 64-bit bandwidth, we would need at least a SRAM operating at 200 MHz. The option for FPGA internal memory limited our coverage to a maximum of 512 KB of data memory, which resulted in a memory overhead of 36 % (code, data, and Merkle Tree). In addition, to reduce unused memory words in PTAG Memory, we split it into two. This allowed to create an easy decoder to separate PTAGs of memory blocks from those of chunks of PTAGs.

Due to the high utilization of internal memory, timestamp memory became limited to 2^{14} 16-bit words to cover the 512 KB of data memory. This represents 5.4% of the total 588 KB main memory coverage. The PTAG Memory utilization for this solution was up to 147 KB (code and data), or 25% overhead.

Table 1 also shows the PTAG cache's configuration. Since this cache is an internal memory as well, it was limited to 4 KB. That limitation did not prevent us of evaluating the cache in multiple configurations. We evaluated this cache in different configurations of lines, set associativity, and replacement policies. Finally, as previously discussed, PMMU requires a buffer to stall a PTAG cache write or read while an eviction is required. We calculated that this buffer needs to be at most 2 times the number of lines in PTAG Cache.

One last information about our FPGA CSHIA implementation is that it had three modes of operation. In the first mode, called *Leon3 Baseline*, the BUS-HDLR is disabled and all security-specific hardware is bypassed. A second mode, the *CSHIA-TS*, activates PMMU for timestamps only. Finally, the third mode, *CSHIA-MT*, disables timestamps and activates the cache in PMMU for supporting the Merkle Tree implementation. Being able to switch between those modes only using switch keys of the development key helped us in debugging and evaluating the performance of the architecture.

5. Experiments and Results

This section describes the experimental setup and results. First, it describes benchmarks and experiments configuration. Then, it presents experimental results on performance, area and power estimates.

5.1. Experimental Setup

The DSU connects the Leon3 platform to a computer through a debugging program called GRMON. Using GRMON, we are able to load programs, measure runtime, insert breakpoints, and set some Leon3 parameters. As benchmarks, we chose nine programs from the MiBench suite [23]: `basicmath`; `bitcount`; `susan`; `qsort`; `fft`; `fft_inv`; `sha`; `stringsearch` (or just `search` for short). These benchmarks were either executable without input files or easily modified to run without them, since we could not load input file through GRMON. Thus, for some benchmarks we incorporated input files in their data segment, and these modifications were evaluated against reference outputs. MiBench usually provides two types of inputs: small and large. We ran both inputs for most of the benchmarks, except by `basicmath`, `fft`, and `fft_inv`. The large inputs of these programs did not affect the size of the data segment and yet most of their run time was dominated by printing their outputs over GRMON.

As Section 4 discussed, the CSHIA implementation is able to cover up to 512 KB of data. This was enough for most of the benchmarks except by the large inputs of `qsort` and `sha`, as Table 2 shows. Only the `.data` and `.bss` segments of the programs were covered. We did not have enough memory to reach the beginning of the `.stack` segment and we would only be able to cover a small portion of `.heap` segment.

Each benchmark was run in eight different instances of CSHIA. (1) The first CSHIA instance is the one that BUS-HDLR is disabled, and bypasses incoming and outgoing bus transfers from the processor. We called this instance as *Leon3 Baseline*. (2) The second instance of CSHIA uses the timestamps solution against replay attacks. We defined it as *CSHIA-TS*. (3-8) The remaining instances are variations of CSHIA when a Merkle Tree is used as solution against replay attacks. As we discussed in the previous section, we evaluated two cache policies and 3 PTAG cache configurations. The 3 PTAG cache configurations are 16 lines and 8 sets, 32 lines and 4 sets, 64 lines and 2 sets. The cache replacement policies were a traditional LRU policy and As-Late-as-Possible (ALAP) [15], in which invalid and discardable chunk of PTAGs are always selected first when available. Therefore, we intuitively named this instances as (3) *CSHIA-MT*-16x8-LRU, (4) *CSHIA-MT*-32x4-LRU, (5) *CSHIA-MT*-64x2-LRU, (6) *CSHIA-MT*-16x8-ALAP, (7) *CSHIA-MT*-32x4-ALAP, (8) *CSHIA-MT*-64x2-ALAP.

Table 2: Coverage of data segment in benchmarks.

Benchmark	.data segment size (KB)	Cover (%)
`qsort_small`	54.9	100
`qsort_large`	588.6	86.99
`bitcount_small`	3.3	100
`bitcount_large`	3.3	100
`sha_small`	307.2	100
`sha_large`	3174.4	16.13
`search_small`	3.4	100
`search_large`	13.4	100
`fft_small`	2.7	100
`fft_small_inv`	2.7	100
`dijkstra_small`	31.1	100
`dijkstra_large`	31.1	100
`basicmath_small`	2.7	100
`susan_small`	23.9	100
`susan_large`	326.7	100

5.2. Performance Analysis

Table 3 shows our results. The first conclusion is that *CSHIA-TS* performs better than any instance that uses Merkle Tree. *CSHIA-TS* worst performance penalty is 8.30 % for `sha_small` and has an average performance penalty of just 2.76 %. Because CSHIA could not entirely cover `sha_large`, its performance penalty ended up being smaller than its counterpart. The `bitcount` and `fft` benchmarks had inconsistent results in some cases, when comparing all instances together. Delving into reasons for that, we found out that they are dependent of random number generation and this was affected by the intervention of CSHIA in the AMBA bus. Therefore, for those benchmarks, the performance difference between CSHIA instances should not be considered significant. Another observation regards to `qsort_small` and `qsort_large`. They presented similar behavior of the `sha` benchmarks, despite CSHIA almost entirely covers the data segment of `qsort_large`.

The results for *CSHIA-MT* show that increasing the number of sets and reducing the number of cache lines seems to provoke a performance penalty. The average performance of *CSHIA-MT*-16x8-ALAP and *CSHIA-MT*-16x8-LRU are worse than *CSHIA-MT*-64x2-ALAP and *CSHIA-MT*-64x2-LRU, respectively, with 7.05% and 5.99% against 5.99% and 5.77%. Therefore, cache memory for *CSHIA-MT* is likely to improve

Table 3: Performance overhead in % of the evaluated CSHIA instances in comparison of running times in *Leon3 Baseline*.

Benchmarks	*CSHIA-TS*(%)	*CSHIA-MT* instances					
		16x8-LRU(%)	32x4-LRU(%)	64x2-LRU(%)	16x8-ALAP(%)	32x4-ALAP(%)	64x2-ALAP(%)
qsort_small	3.77	9.90	9.90	9.90	11.31	10.38	9.91
qsort_large	0.05	0.05	0.05	0.05	0.05	0.05	0.05
bitcount_small	0.00	0.00	0.00	0.00	0.00	0.00	0.00
bitcount_large	2.43	2.42	2.43	0.00	0.00	0.00	2.43
sha_small	8.31	16.61	16.58	16.55	24.85	16.55	16.55
sha_large	1.78	4.51	4.63	4.75	5.34	4.99	4.63
search_small	0.10	0.10	0.10	0.00	0.10	0.10	0.00
search_large	0.00	0.01	0.00	0.01	0.00	0.00	0.01
fft_small	0.00	1.07	2.14	1.07	1.07	1.07	1.07
fft_small_inv	0.92	0.00	0.00	0.00	0.00	0.00	0.00
dijkstra_small	6.40	15.37	15.37	14.09	16.65	15.36	15.38
dijkstra_large	7.35	16.90	16.91	16.90	18.37	17.64	16.90
basicmath_small	1.73	1.73	1.73	1.73	1.73	1.74	1.73
susan_small	1.37	2.74	2.74	2.73	4.10	2.73	2.74
susan_large	7.23	18.40	19.15	18.72	22.23	19.78	18.41
Average	**2.76**	**5.99**	**6.12**	**5.77**	**7.05**	**6.03**	**5.99**

performance if a designer chooses to add more cache lines than sets. For cache replacement policies, we can observe in Table 3 that LRU has better overall performance than ALAP. These numbers do not invalidate the results of Su *et al.* in [15], but we would recommend pure LRU as replacement policy.

Because verification of PTAGs of code memory blocks is equal in *CSHIA-TS* and *CSHIA-MT*, the only way to improve performance of CSHIA is reducing the number of accesses to PTAG Memory for data memory blocks. Thus, increasing the PTAG cache size may lead *CSHIA-MT* to obtain better performance than *CSHIA-TS*. Obviously, these choices need to take into account other variables such as area and power, which we discuss next.

5.3. Area and Power Estimates

Since we did not have access to standard tools from industry to synthesize VHDL, we used the area and power proportionality relation [24] to compute our estimations. For that, we used well-known open tools like CACTI 5.3 [25] for cache memories estimative of power and area, and Ahmed *et al.*'s work [26] that presents area and power for a synthesized Leon3 processor on 65 nm LPLVT (Low Power Low Voltage Threshold) process using ST Microelectronics libraries.

Ahmed *et al.* presented their Leon3 design separating area, static and dynamic power for the core and its cache memory. We ignore their cache memory values since they differ from our implementation. Moreover, our main goal is to estimate area of logic elements. Thus, we will assume a proportional relation between their core area, 0.191 mm^2, and the number of FPGA logic elements of the *Leon3 Baseline* implementation, which is 23,629 in the Altera's DE2-115 development kit.

By this proportional relation between area and logic elements, our estimate for the *CSHIA-TS* and *CSHIA-MT*, without additional memories, is 0.246 mm^2 and 0.264 mm^2, respectively. As we said, area and power can be proportional, and thus we can use similar reasoning to estimate power. From Ahmed *et al.*'s work, static and dynamic power (at 100 MHz) are 85.3 μW and 5.75 mW, respectively. Those numbers result in static power of 109.48 μW for *CSHIA-TS* and 117.41 μW for *CSHIA-MT*. In terms of dynamic power, we obtained 7.41 mW for *CSHIA-TS* and 7.94 mW for *CSHIA-MT*.

Table 4: Area and power for CSHIA implementation without considering instruction and data cache memories of the processor.

Instance	Area (mm^2)	Static Power (mW)	Dynamic Power (mW)
Leon3 Baseline			
Core	0.191	85.3×10^{-3}	5.75
CSHIA-TS			
Core	0.246	109.48×10^{-3}	7.41
Memory	0.141	72.00	7.15
Total	0.387	72.11	14.56
CSHIA-MT-64x2			
Core	0.264	117.41×10^{-3}	7.94
Cache	0.274	6.90	100.98
Total	0.538	7.02	108.92
CSHIA-MT-32x4			
Core	0.264	117.41×10^{-3}	7.94
Cache	0.544	6.45	165.85
Total	0.808	6.57	173.79
CSHIA-MT-16x8			
Core	0.264	117.41×10^{-3}	7.94
Cache	–	–	–
Total	–	–	–

We used CACTI to estimate how the timestamp memory and PTAG Cache affects the design. From Table 1, the total timestamp memory size was 2 bytes $\times$ 2^{14} (or 32 KB). Even though CACTI does not offer an option for non-volatile estimative, a DRAM like estimation provides an insight of area and power. For the PTAG Cache, we estimated 4-KB PTAG Cache with 64 lines and 2 sets, 32 lines and 4 sets, and 16 lines and 8 sets. Unfortunately, CACTI was not able to perform the computation for this last configuration. All estimations are summarized in Table 4.

Even if our estimates are not very accurate, they allow to analyze which solution would provide the best trade-off among area, power, and performance penalties. Thus, based on our numbers, the *CSHIA-TS* would be the best solution. Of course, that would only apply to this specific memory size we evaluated. As we will discuss in the next section, 16-bit times-

tamps will not provide the same security as our *CSHIA-MT* instances with PTAGs of 64 bits. In addition, if the coverage of the data segment needs to be increased, the timestamp memory can reach prohibitive configurations for power and area. In such a situation, *CSHIA-MT* would be capable of offering this higher coverage without impacting in on-chip power and area. Nonetheless, higher penalties in performance would happen. Among all *CSHIA-MT* instances, we would definitely indicate *CSHIA-MT*-64x2-LRU as the best choice due its performance and smaller area and power overheads.

6. Security Analysis

Security features of a system are limited, sometimes by constrained resources, other times when they are surpassed by new technologies. Therefore, the deeper an analysis goes the better the deployment a system can have. In this regard, CSHIA was designed to employ security features to embedded systems when: (a) hiding memory content is not a requirement; (b) control-flow attacks will not be a major threat; (c) redesign of processors, IP components, and tool-chains to insert security features cannot be done or is prohibitive. Any embedded system which fits these three scenarios would probably benefit the most of using CSHIA's design. As previous sections showed, CSHIA is very customizable and can provide good performance with acceptable area and power overheads.

For our CSHIA FPGA implementation, we assumed the threat model discussed in Section 2. Nevertheless, we now add some additional considerations. For instance, once tampered memory blocks or PTAGs are detected, the system halts; what is inside of the chip will not be stored back to memories, as well as the processor will not receive any instruction or data. CSHIA does not assure any security claim if a resourceful adversary has physical access to the processor chip, because in this scenario an adversary can extract side-channel information through a variety of non-invasive, semi-invasive, and invasive attacks, which demand distinct countermeasures.

In the following subsections, we analyze the security strengths and weaknesses of the CSHIA design under different attacking scenarios.

6.1. Brute-force/Forgery Attacks

Scenarios where an attacker tries to bypass CSHIA integrity verification by inserting manipulated PTAGs have a small probability of being successful. PTAGs cannot be directly read by any program and thus, unless the PTAG Memory is externally inspected and manipulated, or reverse-engineered, an attacker cannot modify its contents. A tampered PTAG Memory poses the same threat as tampered PTAGs inserted into the PTAG Buses. Tampered PTAGs will only match tampered memory blocks if the attacker has the same PUF-based key of the instance of CSHIA he/she is trying to attack. This can only happen if he/she guesses the key which has a probability of 1 in 2^{128} of happening, or if the attacker guesses correctly PTAGs for the tampered memory blocks. That has a probability of 1 in 2^{64}. Although, this value does not match the standard security level for personal computers, servers, and high performance systems (see NIST report in [27]), they can significantly decrease threats to embedded systems, requiring more resourceful attackers and reducing chances of successful attacks.

6.2. Modeling attacks

The seminal work of Rührmair *et al.* [28] showed that APUF and its variations are easy to model using machine learning. One would argue that the FPGA implementation of CSHIA is weakened due to the usage of APUFs to generate the PRF's secret key. However, the key point in which CSHIA relies is that there is no PUF response available to an attacker. Although the challenges that are used to extract the key and its syndrome can be made externally available, they do not provide enough information to create a model of an APUF. As showed by Rührmair *et al.*'s work, thousand of challenge-response pairs are needed to create an accurate model. In addition, modeling PTAGs generation is not possible because SipHash was evaluated to be indistinguishable from a uniformly random function [18]. Therefore, such model would have success equivalent to brute-force attacks.

6.3. Fuzzy Extractor Attacks

Assuming that an attacker would not be granted access to the system during its enrollment phase, CSHIA's FE is very secure. First, exposing the challenges used to extract the key and its syndrome encoders (bit string w from Figure 8 (a)) does not give any information to an attacker. Second, our 64-bit challenge APUFs have enough entropy (about 80 % of minimum entropy[1]) to make the pairs $r_1||r_2$ and $r_3||r_4$ produce k_1 and k_2 with entropy of about 64 bits and, therefore, resulting in 128 bits of entropy for the secret key K. Third, the attack described by Becker in [19], in which an attacker changes the helper data in order to force the BCH decoder to fail and set bits in r, is unlikely; since the success of it depends on ECC design. In our BCH implementation, setting up more than 10 bit flips in the syndrome showed no modification of r. Therefore, this attack would not be successful. The FPGA CSHIA implementation uses the VHDL BCH generator available in [29]. A possible weakness of our FE would happen if an attacker figures out all syndromes s_i of the bit strings r_i (from which we derive the key). Once the attacker knows that, he/she can generate all possible 2^{64} bit strings and compute their syndromes. Then he/she searches for those that matches the s_i values. Notice, though, that to unveil each s_i the attacker must either know PUFs responses or apply a side-channel attack like the one Merli *et al.* proposed in [30]. Then again, strengthening the FE security would not be hard, since we could increase the length of r using a larger BCH encoder/decoder.

6.4. Memory Integrity Attacks

We discussed in Section 2 the three main attacks against main memory: *spoofing*, *splicing* or *relocation*, and *Replay* attacks.

[1] This result came from the APUF implemented in this work but evaluated in a Xilinx Spartan 3.

The capability of CSHIA preventing spoofing and splicing attacks depend on the length of PTAGs. An attack is only successful if an attacker can guess a PTAG, which has probability 1 in 2^{64} of happening.

For replay attacks, we showed two solutions. The first solution was the *CSHIA-TS* instance, which had 16-bit timestamps for each data memory block. That means that after 2^{16} memory writes of a specific memory block its timestamp would be reset, enabling a replay attack. Recall that our integrity covered 512 KB of data, which demanded a 32-KB timestamp memory. If we set larger timestamps, as the same length of PTAGs, we would need 128 KB of internal memory. Yet, notice that 512 KB did not cover all the data memory of some programs. For instance, the `sha_large` had more than 3 MB of data memory, to cover this with 64-bit timestamps the non-volatile timestamp memory size goes over 512 KB.

In case of instances of CSHIA that uses Merkle Tree, the 64-bit PTAGs results in 2^{32} attack complexity for replay attacks. This reduction is due to Birthday Paradox [31]. It is known that for any collision-resistant hash function, an attacker can find a collision (with a coin toss probability) between two values if he/she collects at least square root of the total possible outcomes of the function. Although a PRF should not have this problem if the key is changed periodically, because CSHIA's secret key is fixed for its lifetime, the PRF is reduced to an unknown hash family. Hence, to find a collision in CSHIA FPGA implementation with 50% probability, an adversary would need to collect 2^{32} memory blocks of 32 bytes written back to *the same memory address*. This means that, in the average case scenario, the attacker might find a collision after collecting 32 GB of data of one specific memory address, besides collecting all PTAGs in the tree that relate to this address as well. And it is crucial to notice that, in this scenario, the attacker does not choose values, the 50% probability regards to two random values found in the 32 GB data collected. This attack is unequivocally feasible, but requires significant resources.

Finally, of course, using 128-bit PTAGs we can increase CSHIA robustness against replay attacks, but to keep the same size of PTAG Memory in relation to the main memory, memory blocks and chunk of PTAGs would have to be 512 bits. Then again, 128-bit PTAGs would need another PRF since SipHash does not meet this specification. Additionally, larger memory blocks would increase the number of cycles to generate PTAGs, therefore, compromising performance even further for CSHIA Merkle Tree instances.

7. Related Work

A fine list of works in the literature has influenced this work. Their weaknesses and strengths, targeted systems, and construction helped us to make design choices to implement a proof of concept of CSHIA.

In 2003, Yang, Zhang, and Gao [32] proposed an improved version of XOM, an architecture for digital copyright protection. The architecture provides authenticity through a pairwise private/public key. Every instance of a XOM architecture has a unique private key. Secrecy is provided by encrypting software using specific symmetric keys chosen by the software vendor. These keys are encrypted using XOM's public key and therefore only the instance that has the correspondent private key will be able to execute the software. XOM provides integrity protection by hashing memory blocks, but it only prevents spoofing and splicing. Replay attacks are left uncovered. The differential of XOM is to isolate programs in compartments, which have their own tags and keys. Due to this isolation, new instructions had to be added to the processor in order to relax constraints of architecture. For example, to enable sharing data between isolated programs. From the point of view of targeted market, XOM is suitable for very high end embedded systems or above since they simulated their architecture using a processor capable of out-of-order execution and XOM's performance highly depends on the existence of second-level cache (L2) and its size. Finally, area overhead is not estimated, implementation is through simulation, and the averaged performance slowdown is 16.76% on the tested benchmarks, however, they were able to reduce this slowdown to 1.28% implementing an additional cache memory.

The first architecture that proposed to use PUFs for key generation was AEGIS, the work presented by Suh *et al.* in [1]. AEGIS is a tamper-resistant and tamper-evident architecture. Meaning that it hinders tampering threats, but the system still indicates if an attacker successfully overpasses the security features. AEGIS is a complete solution in which not only new instructions are provided, but also system calls, security modes, and different divisions of memory into new regions. It can securely run even under an untrusted operating system. AEGIS provides a Merkle Tree to prevent replay attacks and uses a small cache to store nodes and reduce performance penalties. All this security comes with downsides such as an almost 100% area increase in comparison to the processor baseline, and the modification of the entire toolchain: compilers, operating systems, and even programs. Due to the complexity of the architecture, targeted systems are preferable high-end embedded systems or above. Although the authors are not very clear about overall performance penalties of the architecture, when they used their full protection mode and an architecture configuration consisting in 32 KB instruction/data cache and 16 KB Merkle Tree cache, the worst benchmark performance overhead was 3.3%. However, when the architecture configuration is 4 KB instruction/data cache and 2 KB Merkle Tree cache, the same benchmark has a performance overhead of 73.1%.

Rogers, Milenković, and Milenković presented in 2007 a secure architecture [33]. Different from the previous works described above, they truly focused on embedded system since they assume that processors would not have second-level cache memory (L2), but would present a separated L1 into data and instruction memories. Their architecture provides integrity and secrecy for memory blocks of instructions, and data integrity is not discussed. The architecture uses virtual address to compound encrypted blocks, in order to thwart splicing attacks, they came up with a interesting solution of encrypting an unique PUF-generated (or thermal-noise generated) key together with the program this key authenticates. Thus, when two programs

present a collision between virtual addresses, an attacker will not be able to switch programs because their key will be different. Although their architecture was only simulated, they estimated a power consumption overhead over the baseline system. Setting up a simulation of an ARM processor with small instruction L1 cache of 1 KB resulted in a power consumption overhead as high as twice the baseline's value. In terms of performance penalties for the tested benchmarks, the results also were very detrimental for a small instruction L1, achieving an overhead of 2 times greater than baseline's performance, and becoming negligible for a 8 KB instruction cache in the best scenario. At last, for a standard memory block of 256 bits, their storage overhead reached 50% of the main memory, which is high.

In 2009, Vaslin *et al.* proposed a security approach for off-chip memory in embedded microprocessors [2]. Vaslin *et al.* used the One-Time-Pad (OTP) scheme to provide integrity and secrecy. Their architecture encrypts a timestamp, the memory address, and a padding value using AES. Then, this encrypted content is combined with the cache line. Because they used memory address and timestamp, relocation and replay attacks are thwarted. However, to inhibit spoofing attacks, memory blocks need tags and Vaslin *et al.* proposed using CRC32. One critical point is that their architecture not only needs an internal timestamp memory but also a CRC32 memory. That led to an internal memory of at least 18.8% of the size of main memory. Nonetheless, Vaslin *et al.*'s architecture was able to achieve a worst case performance impact of 10% in the tested benchmarks. However, the area overhead in the FPGA tested almost tripled.

FEDTIC, the 2010 work of Hong and Guo [3], is an architecture for integrity verification and secrecy for embedded systems. Their main contribution is a unique engine that uses one AES hardware instance for encryption, decryption, and tagging of cache lines. Stamps are used to prevent replay attacks. However, instead of a one-time-pad scheme, Hong and Guo used AES in output feedback mode which allowed them to use shifted encrypted blocks to compose a tag. In terms of achievements, for a 512 bit cache lines, their external memory overhead was less than 7% and for the tested benchmarks a maximum internal memory for timestamps needed was 5 KB. We should notice that this internal memory is not a cache and thus will increase with the program size. The average performance penalty was 7.6% and the maximum 30.72%. Their evaluation used a combination of simulation and FPGA implementation.

Bobade and Mankar presented in [4] a secure architecture for embedded system. Their architecture provides integrity and secrecy through an Elliptic Curve Cryptographic engine. The main difference regarding the others architectures presented here is that they use the timestamps as private keys. Thus, cache lines are encapsulated with their address and time stamp (for integrity verification purpose), and then encrypted with the public key to be stored in external memory. As the timestamps are stored in an internal memory, the decryption can be done with reprocessing the pair private/public key and the integrity is ensured by the correct decryption of the triad encapsulated: data, address, and time stamp. Although Bobade and Mankar synthesized their architecture for a FPGA, they only simulated the architecture and did not use any benchmark. Nonetheless, they computed the overhead of slices and LUTs over their baseline processor, which was over 76%. Memory overhead was 25%. In addition, they estimated power increment over baseline. Despite the dynamic power more than doubled in all processor's frequency simulated, the static was kept stable.

Recently, Sepulveda, Wilgerodt, and Pehl in [34] has proposed a Multi-Processors System-on-Chip that provides memory integrity and authenticity through PUFs. The proposed architecture innovates by targeting multi-processors. They also used SipHash to provide integrity tags to memory blocks to protect against all three major threats we have discussed before. One key difference on their replay attack solution is that they use session tokens instead of timestamps. While that is an innovative way, it may not be sufficient to protect against replay attacks, since tokens are updated during idle periods and booting time. Thus, in a long period of execution, in which a specific memory block can be written back multiple times to memory, an attacker might mount a replay attack. One interesting point is that Sepulveda *et al.*argue that CSHIA needs

Table 5: Summary of Related Works in comparison with CSHIA.

Work	Target Architecture	Most Positive Feature	Downside
XOM	High-End embedded systems and above	Program isolation	Does not provide protection against replay attacks.
AEGIS	High-End embedded systems and above	A complete solution	Integration with standard products can be difficult due to modification imposed to the whole toolchain.
[33]	Embedded Systems	Program Isolation	High memory overhead.
[2]	Embedded Systems	Uses AES in OTP mode combined with CRC32 to provide integrity with low on-chip memory overhead.	High area overhead in a FPGA implementation.
FEDTIC	Embedded Systems	Uses one AES component to encryption, decryption and authentication	Can impose large on-chip non-volatile memory.
[4]	Embedded Systems	Security is based on public-key cryptography.	No performance evaluation.
[34]	MPSoC	First PUF based secure architecture for multiple cores.	Does not estimate area and power increment in regard to the baseline system.
CSHIA	Embedded Systems	Design Flexibility.	Does not provide concrete estimative of area and power.

deep modifications in SoC and CPU. However, we believed that this work demonstrates that only minor modification are needed and they are all transparent to the core and does not affect how it works. It is also important to notice that the authors used a similar Code-offset Fuzzy Extractor CSHIA had originally employed, which, as we discussed in the previous section, is less secure than the one we currently proposed, since it reduces entropy of the key. Finally, they estimated area and power of the components of their architecture, and did performance evaluation which, by computing an average degradation, was 5.6% on the tested benchmarks.

Table 5 presents a summary of most positive feature and downside of CSHIA and related works. A fair comparison of performance among the works is quite hard to be performed, due to a variety of benchmarks, baseline cores, choice of platforms, etc. However, a qualitative analysis over design choices can still be done. For instance, PUFs have been constantly claimed to be a better solution for key generation than storing on-chip key. In that regard, our implementation is more advantageous than those that did not use them. Moreover, we carefully analyzed major threats presented in the literature in order to propose a secure employment of a PUF-based key. Because embedded system applications can have a very specific nature, our concern since the beginning was to propose a flexible architecture, which is characterized by its additional bus for the PTAG Memory and the choice between timestamps or Merkle Tree as replay attack solution. Thus, although we were not able to precisely estimate power and area, we believe that we presented a solid solution for the security of embedded systems.

8. Conclusions

The main goal of this work was to present a proof-of-concept implementation of CSHIA, evaluating performance, area and power. Throughout the text, we described an in-depth implementation and evaluation of the architecture. We achieved a flexible solution for embedded systems that allows designers to: (1) specify a bus and memory configuration that provides integrity; (2) select any PUF design that meets security constraints; (3) choose between two countermeasures against replay attack. All that being completely transparent for other components in the chip. In terms of performance, CSHIA timestamp instance showed an average 2.76% performance overhead, while the fastest CSHIA instance with Merkle Tree showed an average 5.77% performance penalty. Taking into account our estimative for area and power consumption, and the comparison against related work, the *CSHIA-TS* instance seems to be a great option for secure embedded systems. Finally, we provided an in depth security analyses, as well as a discussion of relevant works in literature. Soon CSHIA's FPGA implementation will be made available for the research community.

Acknowledgment

The authors thank the São Paulo Research Foundation (FAPESP process nº 2015/06829-2) and Intel (grant "Energy-Efficient Security for SoC Devices – Physical Unclonable Functions") for supporting this work.

References

[1] G. E. Suh, C. W. O'Donnell, I. Sachdev, S. Devadas, Design and implementation of the aegis single-chip secure processor using physical random functions, in: Proceedings of the 32Nd Annual International Symposium on Computer Architecture, ISCA '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 25–36. doi:10.1109/ISCA.2005.22.
URL http://dx.doi.org/10.1109/ISCA.2005.22

[2] R. Vaslin, G. Gogniat, J.-P. Diguet, E. Wanderley, R. Tessier, W. Burleson, A security approach for off-chip memory in embedded microprocessor systems, Microprocessors and Microsystems 33 (1) (2009) 37 – 45, selected Papers from ReCoSoC 2007 (Reconfigurable Communication-centric Systems-on-Chip). doi:https://doi.org/10.1016/j.micpro.2008.08.008.
URL http://www.sciencedirect.com/science/article/pii/S0141933108000823

[3] M. Hong, H. Guo, Fedtic: A security design for embedded systems with insecure external memory, in: T.-h. Kim, Y.-h. Lee, B.-H. Kang, D. Ślkezak (Eds.), Future Generation Information Technology, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 365–375.

[4] S. D. Bobade, V. R. Mankar, Developing configurable security algorithms for embedded system storage, International Journal of Computer Science and Telecommunications 6 (7).

[5] C. Hoffman, M. Cortes, D. Aranha, G. Araujo, Computer security by hardware-intrinsic authentication, in: Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2015 International Conference on, 2015, pp. 143–152. doi:10.1109/CODESISSS.2015.7331377.

[6] Leon3 processor - gaisler, http://www.gaisler.com/index.php/products/processors/leon3, web Site. Accessed: 08-Feb-2017.

[7] B. Gassend, D. Clarke, M. van Dijk, S. Devadas, Silicon physical random functions, in: Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS '02, ACM, New York, NY, USA, 2002, pp. 148–160. doi:10.1145/586110.586132.
URL http://doi.acm.org/10.1145/586110.586132

[8] A.-R. Sadeghi, D. Naccache, Towards Hardware-Intrinsic Security: Foundations and Practice, 1st Edition, Springer-Verlag New York, Inc., New York, NY, USA, 2010.

[9] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, Y. Jin, Hafix: Hardware-assisted flow integrity extension, in: Proceedings of the 52Nd Annual Design Automation Conference, DAC '15, ACM, New York, NY, USA, 2015, pp. 74:1–74:6. doi:10.1145/2744769.2744847.
URL http://doi.acm.org/10.1145/2744769.2744847

[10] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, J. Zambreno, Codesseal: Compiler/fpga approach to secure applications, in: P. Kantor, G. Muresan, F. Roberts, D. D. Zeng, F.-Y. Wang, H. Chen, R. C. Merkle (Eds.), Intelligence and Security Informatics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 530–535.

[11] A. K. Kanuparthi, M. Zahran, R. Karri, Architecture support for dynamic integrity checking, IEEE Transactions on Information Forensics and Security 7 (1) (2012) 321–332. doi:10.1109/TIFS.2011.2166960.

[12] R. Elbaz, D. Champagne, R. Lee, L. Torres, G. Sassatelli, P. Guillemin, Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks, in: P. Paillier, I. Verbauwhede (Eds.), Cryptographic Hardware and Embedded Systems - CHES 2007, Vol. 4727 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 289–302. doi:10.1007/978-3-540-74735-2_20.
URL http://dx.doi.org/10.1007/978-3-540-74735-2_20

[13] M. Hong, H. Guo, S. X. Hu, A cost-effective tag design for memory data authentication in embedded systems, in: Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '12, ACM, New York, NY, USA, 2012, pp. 17–26. doi:10.1145/2380403.2380414.
URL http://doi.acm.org/10.1145/2380403.2380414

[14] B. Gassend, G. Suh, D. Clarke, M. van Dijk, S. Devadas, Caches and hash trees for efficient memory integrity verification, in: High-Performance

Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on, 2003, pp. 295–306. doi:10.1109/HPCA.2003.1183547.

[15] L. Su, S. Courcambeck, P. Guillemin, C. Schwarz, R. Pacalet, Secbus: Operating system controlled hierarchical page-based memory bus protection, in: Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09., 2009, pp. 570–573. doi:10.1109/DATE.2009.5090729.

[16] Z. Liu, Q. Zhu, D. Li, X. Zou, Off-chip memory encryption and integrity protection based on aes-gcm in embedded systems, IEEE Design Test 30 (5) (2013) 54–62. doi:10.1109/MDAT.2013.2255912.

[17] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, Z. Wang, Architecture for protecting critical secrets in microprocessors, SIGARCH Comput. Archit. News 33 (2) (2005) 2–13. doi:10.1145/1080695.1069971.
URL http://doi.acm.org/10.1145/1080695.1069971

[18] J. Aumasson, D. J. Bernstein, Siphash: A fast short-input PRF, in: Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings, 2012, pp. 489–508. doi:10.1007/978-3-642-34931-7_28.
URL http://dx.doi.org/10.1007/978-3-642-34931-7_28

[19] G. T. Becker, Robust fuzzy extractors and helper data manipulation attacks revisited: Theory vs practice, IEEE Transactions on Dependable and Secure Computing PP (99) (2017) 1–1. doi:10.1109/TDSC.2017.2762675.

[20] F. Armknecht, R. Maes, A.-R. Sadeghi, F.-X. Standaert, C. Wachsmann, A formalization of the security features of physical functions, in: Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 397–412. doi:10.1109/SP.2011.10.
URL http://dx.doi.org/10.1109/SP.2011.10

[21] M.-D. M. Yu, S. Devadas, Secure and robust error correction for physical unclonable functions, IEEE Des. Test 27 (1) (2010) 48–65. doi:10.1109/MDT.2010.25.
URL http://dx.doi.org/10.1109/MDT.2010.25

[22] Siphash-2-4 vhdl implementation, https://github.com/pemb/siphash, web Site. Accessed: 2-Oct-2018.

[23] Mibench version 1.0 – welcome to the homepage for mibench: a free, commercially representative embedded benchmark suite., http://vhosts.eecs.umich.edu/mibench/, web Site. Accessed: 08-May-2018.

[24] M. Nemani, F. N. Najm, High-level area and power estimation for vlsi circuits, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18 (6) (1999) 697–713. doi:10.1109/43.766722.

[25] Cacti – an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model, http://www.hpl.hp.com/research/cacti/, web Site. Accessed: 08-May-2018.

[26] S. Z. Ahmed, J. Eydoux, L. Rouge, J. B. Cuelle, G. Sassatelli, L. Torres, Exploration of power reduction and performance enhancement in leon3 processor with esl reprogrammable efpga in processor pipeline and as a co-processor, in: 2009 Design, Automation Test in Europe Conference Exhibition, 2009, pp. 184–189. doi:10.1109/DATE.2009.5090655.

[27] E. B. Barker, A. L. Roginsky, Sp 800-131a. transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths, Tech. rep., Gaithersburg, MD, United States (2011).

[28] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, J. Schmidhuber, Modeling attacks on physical unclonable functions, in: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10, ACM, New York, NY, USA, 2010, pp. 237–249. doi:10.1145/1866307.1866335.
URL http://doi.acm.org/10.1145/1866307.1866335

[29] E. Jamro, The design of a vhdl based synthesis tool for bch codecs, The university of HuddersfielWeb Site. Accessed: 06-May-2018.

[30] D. Merli, D. Schuster, F. Stumpf, G. Sigl, Side-channel analysis of pufs and fuzzy extractors, in: J. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, Y. Beres (Eds.), Trust and Trustworthy Computing, Vol. 6740 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 33–47. doi:10.1007/978-3-642-21599-5_3.
URL http://dx.doi.org/10.1007/978-3-642-21599-5_3

[31] A. J. Menezes, P. C. Van Oorschot, S. A. Vanstone, Handbook of applied cryptography, CRC press, 1996.

[32] J. Yang, Y. Zhang, L. Gao, Fast secure processor for inhibiting software piracy and tampering, in: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, IEEE Computer Society, Washington, DC, USA, 2003, pp. 351–.
URL http://dl.acm.org/citation.cfm?id=956417.956576

[33] A. Rogers, M. Milenkovic, A. Milenkovic, A low overhead hardware technique for software integrity and confidentiality, in: 2007 25th International Conference on Computer Design, 2007, pp. 113–120. doi:10.1109/ICCD.2007.4601889.

[34] J. Sepulveda, F. Willgerodt, M. Pehl, Sepufsoc: Using pufs for memory integrity and authentication in multi-processors system-on-chip, in: Proceedings of the 2018 on Great Lakes Symposium on VLSI, GLSVLSI '18, ACM, New York, NY, USA, 2018, pp. 39–44. doi:10.1145/3194554.3194562.
URL http://doi.acm.org/10.1145/3194554.3194562

Applying Template Attacks on XOR Arbiter PUFs

ABSTRACT

One of the fundamental properties of PUFs is *uniqueness*, which results from the intrinsic characteristics of each PUF instance. However, PUF architectures employ elements whose physical characteristics and behavior may be very similar among different instances, leaking unwanted information. We explore that with Template Attacks in XOR-APUFs, in which challenge-respose pairs (CRPs) are profiled in one FPGA instance of the PUF to predict responses of a different FPGA instance, obtaining up to 80% of accuracy. Our attack only needs few CRPs for profiling (at most 170), but it can be applied to different instances without training, reducing the attack time in comparison to approaches based on machine learning.

KEYWORDS

Template Attacks, PUFs

ACM Reference Format:

. 2019. Applying Template Attacks on XOR Arbiter PUFs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Physical Unclonable Functions (PUFs) have been under scrutiny for more than a decade now [5], some of their fundamental security properties, such as unpredictability and reliability, have been defeated in a variety of attacks. Machine learning attacks have undermined unpredictability of the main PUF implementations [4, 13, 18]. Other approaches have either amplified PUF reliability issues [11], or taken advantage from loopholes in the construction of secure sketches, that add robustness to PUFs, to deploy attacks [3, 7]. Yet one fundamental property of PUFs have withstood most attacks so far: uniqueness. It comes from specially designed patterns in PUFs. Through imperfections originated from the fabrication process, these patterns give a particular behavior for each PUF instance. It has been assumed that not even the manufacturer itself would be able to perfectly create two identical PUF instances.

Although replicating imperfections seems to be unlikely, Helfmeier *et al.* showed that a SRAM-PUF could be forcedly cloned using Decapsulation, Photonic Emission, and Focused Ion Beams [8]. In this attack, the authors cloned responses (outputs) from a SRAM PUF instance using Decapsulation and Photonic Emission and then forced a different instance to present the same responses using Focused Ion Beams. In another example, Becker [2] captured challenges (inputs) and responses of a XOR Arbiter PUF and modeled it using a machine learning algorithm. This algorithm was then implemented into an evaluation board with a microcontroller to mimic the PUF. Despite these examples, to the best of our knowledge, no attack has yet explored similarities of physical characteristics among different PUF instances. Even though all elements that compose a PUF architecture are unique in each instance, some of those elements may present similar physical behavior, which enables attacks to reveal information of one PUF instance using another.

Template attacks have been designed to use information learned from one device instance (for instance, electromagnetic emissions) to obtain concealed information from another instance. In this work, we apply a Template Attack to XOR Arbiter PUFs (XOR APUFs), and we show that, with at most 170 challenge-response pairs (CRPs) profiled from a FPGA, we were able to reproduce responses with 80% accuracy of another FPGA implementing the same PUF, despite almost 50% inter-chip CRP variation. XOR APUFs have been widely explored in the literature and also are deployed in real applications [2], thus, using them to apply this attack enables us to compare our results to solid ones from the literature. Moreover, if this attack can be further applied to different PUFs, it can become an additional threat that designers will have to take into account in the future since attackers could circumvent the uniqueness property of PUFs through architectural elements, and use information leaked by those elements to predict CRPs from different PUF instances.

This work is organized as follows. Section 2 represents a short background in PUFs and side-channel attacks against them. Section 3 briefly presents the Template Attack. Section 4 describes the attack on XOR APUFs and Section 5 discusses results. Afterwards, Section 6 discusses and concludes this work.

2 BACKGROUND

Although Physical Unclonable Functions (PUFs) are well-established in the literature, we present a brief and not strictly formal introduction to them. In particular, this section focuses on the general idea and functionality of the Arbiter PUF and its most common derivation, the XOR Arbiter PUF. Then, we discuss recent side-channel attacks against PUFs.

2.1 PUFs

A PUF is a physical system that behaves similarly to a bijective function, in which an input is a physical stimulus called *challenge* and an output is called *response*. Every physical instance of a PUF has a unique set of Challenge-Response Pairs (CRPs). This property comes from the intrinsic nature of all physical elements that are imperfect at some level (molecular level, atomic level, etc.). For instance, in electronic devices, the fabrication process is inexorably uneven and thus no device instance is equal to another. Hence, electronic PUFs exploit this inequality to uniquely generate pairs of binary inputs and outputs.

PUFs have been classified into two categories: Weak and Strong PUFs [1]. These terms do not necessarily describe a security weakness or strength, but rather they classify whether a PUF construction has a large number of CRPs. Classical examples of Weak

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

PUFs are Ring Oscillators [20] and SRAM PUFs[12]. Generally, they are good key generators. The most representative Strong PUFs are the Arbiter PUF (APUF) and its variations, such as the XOR APUF [2], Feedback-Forward PUF (FFPUF) [11], Lightweight PUF (LPUF) [14]. These are preferable to be used in applications such as security protocols, key establishment and device authentication [11].

The APUF is a good baseline for study, because its design is very intuitive and easy to explain. Figure 1 presents a 4-bit challenge APUF. The APUF is a delay-based PUF, thus its response is generated after evaluating the delay suffered by a signal Δ. Δ is divided into two paths at the beginning, each challenge bit affects both paths in every stage (usually implemented by a multiplexer component). When Δ arrives at the flip-flop at the end, imperfections in the paths will have made one signal slightly faster than the other. If it is the superior path, the inferior locks a bit 1 in the flip-flop. Otherwise, the inferior path locks a 0. Thus, the flip-flop acts as an arbiter that decides which path is the fastest.

Rührmair *et al.* in [18] revealed that APUF is very susceptible to machine learning (ML) attacks. However, a more complex version of it, the XOR APUF, was shown to be more resilient to such attacks under certain configurations. Figure 2 illustrates a XOR APUF that can be compounded by a variable number n of 4-bit challenge APUFs. For XOR APUFs, a challenge is simultaneously applied to all APUFs and the individual parallel responses are combined to produce the final response. In Rührmair *et al.*'s work, a XOR APUF with 5 64-bit challenge APUFs took about 2 hours to be modeled with satisfactory prediction accuracy. However, this time increased 15 times when the XOR APUF contained 6 APUFs. Thus, increasing the number of APUFs in XOR APUFs, despite causing instability, seems to hinder machine learning attacks.

Motivated by overcoming limitations of machine learning attacks, hybrid attacks have recently been proposed. They take advantage of side-channel attacks to unveil intrinsic or concealed information that are fed into machine learning algorithms, aiming at modeling PUF behavior. Next, we take a look at some important works that explored those attacks.

2.2 Side-Channel Attacks on PUFs

In a simple definition, a side-channel attack uses information leaked through operations, such as timing delay, power consumption patterns, among others, to extract secret data from a system. An attacker usually seeks to obtain cryptographic keys, passwords, personal information, etc. Against PUFs, side-channel attacks are usually aimed at revealing responses that are not made available.

In [13], Mahmoud *et al.* presented a machine learning attack on XOR APUFs and Lightweight PUFs that uses simulated side-channel

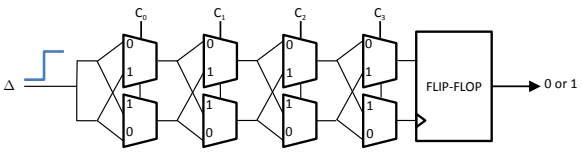


Figure 1: A 4-bit challenge Arbiter PUF.

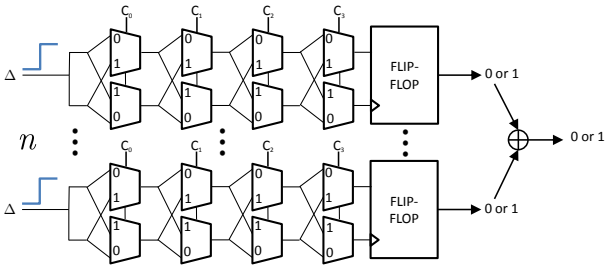


Figure 2: A 4-bit challenge n-XOR Arbiter PUF.

information. Their PUF simulation, using SPICE, provided power consumption of grouped APUFs that compound a single response of XOR APUFs or LPUFs. At the end of APUFs, group of flip-flops (or latches) can be seen as a register and, ideally, its Hamming Weight (HW) is directly correlated to its power consumption. Hence, the simulations provided power consumption that was translated into HWs used by the machine learning algorithms to model the PUFs.

Rührmair *et al.* work in [19] expanded Mahmoud *et al.*'s, by adding a timing side-channel attack. This new attack uses PUFs implemented on FPGAs, and deploys a circuit called timing signature extraction to get the timing side channel information. As the previous attack, the timing information was fed into machine learning algorithms to model PUFs.

Using also power side-channel information from simulations, Becker and Kumar attacked Controlled APUFs in [19]. Controlled PUFs are more secure PUFs due to the fact that neither challenges nor responses are available. In short, a challenge is not directly applied to PUFs, but to a challenge generator instead. Only then, derived challenges from the generator are used in the PUFs, whose responses are post-processed through a one-way function. Therefore, from the machine learning attack perspective, this PUF is much harder to attack since a ML algorithm would unlikely achieve a correct model, if a one-way function indistinguishable from a pseudo-random one is used. Thusly their side-channel attack could provide intrinsic PUF information to ML algorithms, overcoming the implementation obstacle.

Finally, in [11], Kumar and Burleson applied a hybrid side-channel attack on FFPUFs. They used a silicon implementation of FFPUFs and applied fault injection attacks through voltage and temperature variation to determine delay difference between APUF paths. This time difference information was then fed into ML algorithms.

As the reader can notice, most of the works aimed at modeling one single instance and then predict its unknown CRPs. However, how larger a threat would it be if one single profiled PUF instance can reveal CRPs of any other PUF instance? We start to explore this possibility in the next section that describes the Template Attack and introduces how it can be applied to PUFs.

3 TEMPLATE ATTACK

The seminal work on Template Attack published by Chari *et al.* in [6] is based on detection of signal with Gaussian noise in information theory. The key point of this attack is the intrinsic noise produced by every operation in an electronic device. Once a profile (template) of the noise is created for a device instance, the same

operation from another instance can be revealed by matching its noise with the profile.

In short, one can describe noise as a multivariate Gaussian distribution. Even though every operation on a specific time may have a unique noise, large samples of noises in different times, in a specific device instance, will result in a certain noise population distribution with a mean and variance. Thus, repeating the same operation in a different instance is likely to produce an intrinsic noise represented by that population. This feature, if applicable to PUFs, may be a threat to the uniqueness property since one can circumvent it through electronic noise of architectural elements, which could be found in a single statistical population of noise, despite originating from different PUF instances. Thus, investigating this attack is an essential evaluation for the security of PUFs.

3.1 Template Construction

An attacker builds the template from traces obtained by an oscilloscope. A trace (or frame) is a set of samples (or points), where the ordinate is generally power, or voltage, or current, and the abscissa is discrete time. An operation O will be a collection of traces F. Assuming the ordinate is voltage, a trace F will be a pairwise $\{T, V\}$, where $V = (v_1, v_2, \ldots, v_n)$ and $T = (t_1, t_2, \ldots, t_n)$ are arrays. Notice that the notion of operation is defined by the attacker. For instance, the whole process of encryption in AES can be considered an operation, as well as the substitution of a specific byte during the SubBytes step within AES. Overall, the more an attacker knows about the system the more specific the attack will be.

For each targeted operation, an attacker will need to collect a large number of traces, usually more than thousands per operation. Commonly, the only desirable change in an operation is a small variation in the information that is processed. For example, one would like to solely detect a byte variation in a specific register during a specific AES round. The consistence in the window of observation allows us to simplify $F = V$. For a given operation O_i from the set of observed operations $\mathbf{O} = \{O_1, O_2, \ldots, O_k\}$, and given its set of traces (or frames) $\{F_1^i, F_2^i, \ldots, F_m^i\}$, a template $\mathbf{T}^i$ is a matrix of covariance[1] Σ (Eq. 1) and an average array M (Eq. 2)

$$\Sigma^i[x, y] = \text{cov}(F_u^i[x], F_v^i[y]), \forall u, v \in [1, m], \tag{1}$$

$$M^i[x] = \frac{1}{m} \sum_{j=1}^{m} F_j^i[x], \tag{2}$$

where x and y are in the selection S of points between 0 and n. This selection has an important role in the whole attack, but we leave this discussion for later in this section. Notice that size of Σ is $|S|^2$ and the size of M is $|S|$. So, once the attacker computed the template of each operation, he/she can collect traces of the target. Remembering that the idea is to attack another instance of the electronic device used to assemble the template. The attacker's goal then is to find out which operation happens in the attacked device (recalling the example above, it would be to find out which byte is stored in a specific register during a specific AES round).

[1] In [6], they computed the covariance matrix using noise vectors of the traces, which can be computed by Eq. 3. However, as one can demonstrate, the result will the same if we compute the matrix using the traces.

3.2 Attacking

Given a collection $\mathbf{F}$ of attack traces $\{F_1^A, F_2^A, \ldots, F_l^A\}$, the attacker computes the noise vector N (Eq. 3), and using the probability density function of a multivariate Gaussian distribution (Eq. 4), he/she can compute the probability of guessing which operation was occurring in the target instance, given a trace F_u^A applied to Equation 5.

$$N_u[x] = F_u^A[x] - M[x], u \in [1, l], x \in S \tag{3}$$

$$p^i(N_u) = \frac{1}{\sqrt{(2\pi)^{|S|} \det(\Sigma^i)}} \exp\left(-\frac{1}{2} N_u^T \Sigma^{i^{-1}} N_u \right) \tag{4}$$

$$p^i(O^i | F_u^A) = \frac{p^i(F_u^A | O^i) \cdot p(O^i)}{\sum_{j=1}^{k} (p(F_u^A | O^j) \cdot p(O^j))} \tag{5}$$

The probability $p(O^i)$ in this work is equal to $1/|\mathbf{O}|$ because all operations have the same probability of occurring (like in [16]). The probability of $p^i(F_u^A | O^i)$ is, in fact, the PDF (Eq. 4) itself. Thus, a simplified equation for the probability of $p^i(O^i | F_u^A)$ is Eq. 6. For all traces $\mathbf{F}$ captured by the attacker, he/she computes their average, $\overline{\mathbf{F}}$, and uses the Eq. 7 to compute the probability of guessing the correct operation in the target device. Notice that $\mathbf{N}_u$ is the noise vector for $\overline{\mathbf{F}}$.

$$p^i(O^i | F_u^A) = \frac{p^i(N_u)}{\sum_{j=1}^{k} p^j(N_u)} \tag{6}$$

$$p^i(O^i | \overline{\mathbf{F}}) = \frac{p^i(\mathbf{N}_u)}{\sum_{j=1}^{k} p^j(\mathbf{N}_u)} \tag{7}$$

3.3 Point Selection

As mentioned previously, an operation can have a variable time depending on the focus of the attack. A large trace would make the computation too much time consuming or even infeasible. In order to reduce the trace to a few points of interest, one can use some point selection techniques. It is important to notice that this selection can increase the accuracy of the attack since it is possible to remove points that are not related or poorly related to the operation. Common techniques are the sum of differences [9] and correlation power analysis (CPA), which is the same principle of the CPA attack [15] but applied to point selection.

Therefore, the set S described above is formed by choosing the greatest values that those techniques provide. The number of points can be arbitrary and one can also select the greatest values within a minimum distance between them. Setting a minimum distance can avoid that all chosen points belong to a specific peak of the traces, which sometimes reduces the information available.

4 TEMPLATE ATTACK ON XOR APUFS

As we discussed before, flip-flops of a group of APUFs can be seen as a register. Consider that each possible HW of this register is an operation. Without loss of generality, let us assume that an attacker has as target a XOR APUF with 8 APUFs. For simplicity, let us use the following notation to refer to XOR APUFs containing n APUFs:

Xn-APUF. Now consider that the attacker has full control of a sample instance of a device that contains one X8-APUF. The attacker's goal is to figure out a specific operation in another instance of that device, the target instance. Thus, the attack goes as follows: first the attacker characterizes all possible HW from the example instance, and then captures few traces of the target instance for a specific challenge (or challenges) that he/she wants to unveil its HW.

During profiling, the attacker can choose multiple challenges for the same HW. That can help classification of traces because in real scenarios a specific HW can show variable power signatures. This multiplicity is not a requirement tough. We assume that an attacker will collect k operations, being k a multiple of 9 (all possible HWs for 8 flip-flops). For each operation, m traces from the sample instance will be captured and then the attacker will profile the sample instance using Equations 1 and 2.

In the following, the attacker has access to the target instance, and he/she is able to collect traces of the challenges he/she wants. For each operation, the attacker collects l traces. Then, after using a point selection technique, the attacker can use Equations 3 and 4 to compute the probabilities to be applied in Eq. 6, or Eq. 7 if l is greater than 1. HWs are guessed by identifying the HW related to the challenge with highest probability. The attacker can directly convert HW of XOR APUFs into responses, when desired.

5 RESULTS

Using the experimental design discussed in the previous section, we now present our experiments. First, this section details equipment, setup, experimental flow, etc. Then, we discuss our results comparing them to those in the literature.

5.1 Experimental Setup

5.1.1 Equipment and Tools. For the experiments, we implemented a 128-bit challenge X8-APUF and X16-APUF in two FPGAs: SASEBO-GII (Xilinx Virtex 5) and Mojo V3 (Xilinx Spartan 6, Figure 3). Because we only had one SASEBO-GII, we changed place and routing to distinguish between the profiling instance and the attacking instance. We collected voltage traces using an Agilent Infiniium DSO90604A oscilloscope connected to SASEBO-GII by an SMA probe and to Mojo V3 by an EM probe. All implementations had a trigger signal, which is illustrated by Figure 4. The center region in Figure 4 shows the moment a challenge is changed, which leads to a response right after that.

With both FPGAs we had a problem to obtain a clear indication of response in the oscilloscope. We noticed that, if the signal of interest was not assigned in the top level entity of the VHDL implementation, we were not able to capture a distinguishable power signature, even after collecting millions of traces for a single operation. It might have been our probe that was not sensitive enough for the experiment. Nonetheless, after multiple tests, we notice that the Xilinx compiler assigns all top level signals to pins in the I/O bank, even those that were not explicitly assigned. And, despite not sending or receiving data, those signals will draw higher current from FPGA, which will generate enough voltage variation or electromagnetic emission to be captured by the oscilloscope. *That also enabled us to directly capture XOR responses.*

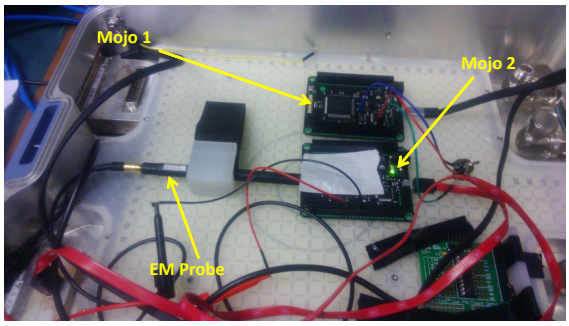


Figure 3: Profiling Mojo (1) and attacking Mojo (2).

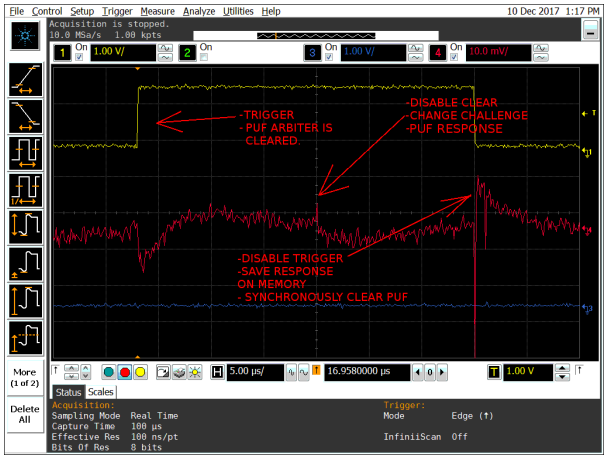


Figure 4: An example of captured traces.

5.1.2 Setup. Figure 5 shows the general flow of our experiments for each PUF attacked. It is worth to point out that we used Linear Feedback Shift Registers (LFSR) to generate the challenges that were applied to the PUFs. Furthermore, for SASEBO-GII, we used Xilinx's Chipscope not only to download CRPs, but also to set attacking challenges into the PUF designs. For Mojo V3, we implemented a serial protocol for the same function.

Once traces are collected from the sample device, it is necessary to select points of interest. We tested multiple point selection strategies. Because our traces were set to 500 timing samples, we chose 5, 10, 15, 20, and 40 points of interest that had minimum distance between them of 0, 5, 10, and 20 points. We applied both Sum Of Differences and CPA as point selection. We hoped that one of these configurations would be outstanding. However, despite the volume of configurations, they turned out to be inconclusive. That is, no specific configuration had consistently outperformed others in the experiments. Each attack had a different successful configuration. We believe that factors such as different boards, different PUF implementations, different days for the experiments may have contributed to that.

We chose to average the attacking traces (Eq. 7), instead of multiplying the probability of each trace [16], because the averaged trace consistently yielded the highest probabilities. Moreover, choosing the challenge with the highest probability predominantly had better

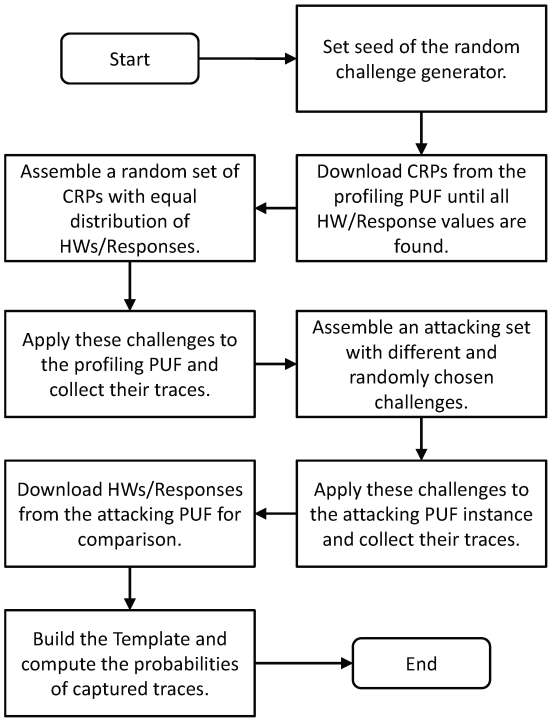


Figure 5: Overall experimental flow for each PUF.

results than combining the probabilities of the same HWs or responses as well. Overall, it is important to calibrate the experiments to find out which configurations can give the best results.

5.1.3 Experiments. We designed three experiments for each 128-bit XOR APUF divided into two attacks. First, we attacked the same PUF instance used to built the template. This attack would allow us to compare our numbers against those in the literature produced by ML. Then, we attacked a different PUF instance. Recalling that for SASEBO-GII, we changed routing and placement to simulate a different PUF instance in the same board. The experiments were:

- **(A)** *Profile*: 10 random CRPs for each HW in SASEBO-GII.
  Attack: 10 random and different CRPs per HW for X8-APUF and 5 for X16-APUF.
- **(B)** *Profile*: 10 random CRPs for each HW from Mojo (1).
  Attack: 10 random and different CRPs for each HW for X8-APUF and 5 for X16-APUF[2].
- **(C)** *Profile*: 25 random CRPs for each response from Mojo (1).
  Attack: 25 random and different CRPs for each response.

Therefore, we profiled all X8-APUF's HWs with 90 challenges and all X16-APUF's HWs with 170 challenges. For responses, our total number of profiling challenges was 50, 25 for each response. In regard to traces, we collected 50,000 traces per challenge for HW profiling, and 20,000 traces per challenge for response profiling. All attacking challenges had up to 2,500 traces collected.

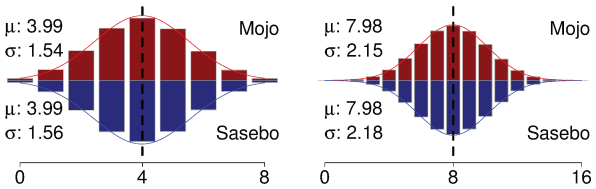
5.1.4 PUF Evaluation. Finally, in order to properly analyze CRP prediction capability, it is necessary to determine whether PUF instances are biased. We performed two evaluations of our PUFs instances. First, we evaluated the Hamming Distance (HD) between random challenges applied to each profiling instance (intra-chip CRP distance). Then, we computed the HD between PUF instances of Mojo (1) and Mojo (2) (inter-chip CRP distance).

Figure 6 (a) presents the HD the for X8-APUF instances of both FPGAs, Sasebo and Mojo. Figure 6 (b) shows the HD for the X16-APUF instances. We used over 5000 and 30000 CRPs to generate Figure 6 (a) and Figure 6 (b), respectively. Notice that we computed HDs using internal APUF responses and there were responses of each possible HW, but they were not equally distributed in our analyzed set. Additionally, we needed to generate more CRPs to find all 17 HWs. Overall, these analyses show that our PUF instances were well balanced.

Figure 7 (a) shows the HD between Mojo (1) and (2) for the X8-APUF instances, while Figure 7 (b) shows it for the X16-APUF instances. We would like to draw your attention to the fact that these distributions were not computed by comparing each response of Mojo (1) against all others of Mojo (2), but rather they are the sole difference of the same challenge applied to Mojo (1) and (2). Hence, we can see that, on average, these challenges had the highest distance possible, and response prediction is not accidental.

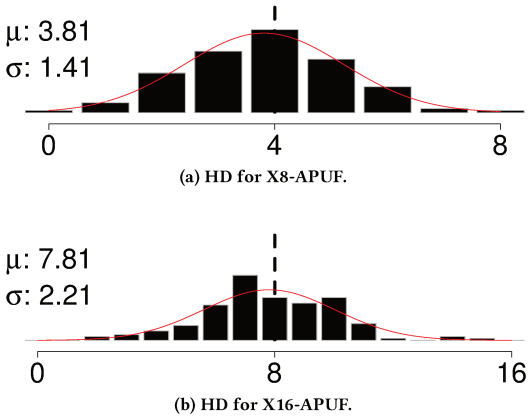
5.2 Results from Attacks

The prediction accuracy of our Template Attack is shown in Table 1. One can seen that we had better results applying the attack in the same instance we profiled (Table 1 (a)). Noticeably, HW prediction accuracy was quite low for the attack instances (Table 1 (b)).



(a) HD for X8-APUF. (b) HD for X16-APUF.

Figure 6: Intra-chip CRP distance for Sasebo and Mojo.



(a) HD for X8-APUF.

(b) HD for X16-APUF.

Figure 7: Inter-chip CRP distance between Mojo (1) and (2).

[2]We change routing and placement here as well. Even using a different FPGA, CRPs were not significantly different from Mojo (1)

Nonetheless, having trouble in classifying HW number is not new in the literature [10, 17].

When converting HWs into responses, results were improved in all attacks and achieved similar numbers. These improvements come from HWs that produce the same response in the XOR operation, despite they did not match when we compared them. The strongest results came from the experiments aiming at the responses. The prediction accuracy of 82% and 78% when attacking the same instance that was profiled is not far from some results obtained by ML algorithms and presented in recent works. For instance, Becker and Kumar in [4] had worst results using more than 20,000 challenges in their simulation with noise. Also, Becker in [2] present 80% accuracy in a training set for 128-bit X16-APUF using 500,000 CRPs. In addition to that, our prediction accuracy of 80% and 74% show that attacking different instances by using the profile of another can be feasible. Finally, another crucial point is attack efficiency. Once the template is created, to determine a response (or HW) of a single challenge, an attacker will spend seconds to minutes. Of course, the profiling phase probably takes longer than the training phase of ML algorithms, however ML attacks have to train each PUF instance to be attacked, while our template attack would not need that.

6 DISCUSSIONS AND CONCLUSIONS

PUFs are expected to provide uniqueness and unpredictability for each instance, which would result in unique secret keys, electronic fingerprints, etc. However, elements in PUFs' architectures can physically behave in similar ways among different instances, which leads to unwanted side channel information leakage. This work exploited this fact by applying Template Attacks on XOR APUFs. Our attack used real side channel information learned from PUFs implemented in a FPGA instance and predicted responses from another FPGA instance, implementing the same PUF, with up to 80 % of accuracy. This attack used a small amount of CRPs (at most 170) for profiling and its application can be quickly done in any other instance without any training. While machine learning attacks would require to train each specific instance to be attacked.

Although our work only examined XOR APUFs, this attack might be applicable to different PUFs. In addition, using XOR APUFs enabled us to compare our results to solid ones reported in the literature. Despite several previous works had performed side channel attacks on PUFs, none of those attacks have used information learned from one PUF instance to predict responses of a different PUF instance, to the best of our knowledge. Besides, further improvements in attack technique can lead to stronger results.

Our experiments used I/O bank signal amplification, however, it is likely that a very sensitive EM probe placed over the area of the PUF responses of a decapped die may reveal similar attack viability. Countermeasures against the proposed attack are clearly needed, with particular attention to the leakage of flip-flops of XOR APUFs. We believe our results are important to broaden the possibilities of attacking PUFs, contributing to improve their security evaluation.

Table 1: Accuracy results of experiments (A), (B), and (C).

(a) Attacking the same instance of profiling.

	128-bit X8-APUF			128-bit X16-APUF		
Prediction Type	(A)	(B)	(C)	(A)	(B)	(C)
HW	57 %	50%	-	28%	40%	-
HW into Responses	73%	70%	-	66%	69%	-
Responses	-	-	78%	-	-	82%

(b) Attacking a different instance.

	128-bit X8-APUF			128-bit X16-APUF		
Prediction Type	(A)	(B)	(C)	(A)	(B)	(C)
HW	29 %	23%	-	16%	19%	-
HW into Responses	63%	68%	-	67%	61%	-
Responses	-	-	80%	-	-	74%

REFERENCES

[1] Frederik Armknecht, Roel Maes, Ahmad-Reza Sadeghi, Francois-Xavier Standaert, and Christian Wachsmann. 2011. A Formalization of the Security Features of Physical Functions. In *SP '11*. 16. https://doi.org/10.1109/SP.2011.10

[2] Georg T. Becker. 2015. The Gap Between Promise and Reality: On the Insecurity of XOR Arbiter PUFs. In *CHES 2015*. https://doi.org/10.1007/978-3-662-48324-4_27

[3] G. T. Becker. 2017. Robust Fuzzy Extractors and Helper Data Manipulation Attacks Revisited: Theory vs Practice. *IEEE Transactions on Dependable and Secure Computing* (2017). https://doi.org/10.1109/TDSC.2017.2762675

[4] Georg T. Becker and Raghavan Kumar. 2014. Active and Passive Side-Channel Attacks on Delay Based PUF Designs. *IACR Cryptology ePrint Archive* (2014).

[5] C. H. Chang, Y. Zheng, and L. Zhang. 2017. A Retrospective and a Look Forward: Fifteen Years of Physical Unclonable Function Advancement. *IEEE Circuits and Systems Magazine* (2017). https://doi.org/10.1109/MCAS.2017.2713305

[6] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. 2002. Template attacks. In *CHES*. Springer, 13–28.

[7] Jeroen Delvaux and Ingrid Verbauwhede. 2014. Attacking PUF-Based Pattern Matching Key Generators via Helper Data Manipulation. In *Topics in Cryptology - CT-RSA 2014*. Vol. 8366. 106–131. https://doi.org/10.1007/978-3-319-04852-9_6

[8] C. Helfmeier, C. Boit, D. Nedospasov, and J.-P. Seifert. 2013. Cloning Physically Unclonable Functions. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*. 1–6. https://doi.org/10.1109/HST.2013.6581556

[9] NewAE Technology Inc. 2017. Template Attacks. https://wiki.newae.com/Template_Attacks [Online; accessed 26-Feb-2019].

[10] D. Karakoyunlu and B. Sunar. 2010. Differential template attacks on PUF enabled cryptographic devices. In *2010 IEEE International Workshop on Information Forensics and Security*. 1–6. https://doi.org/10.1109/WIFS.2010.5711445

[11] Raghavan Kumar and Wayne Burleson. 2015. *Side-Channel Assisted Modeling Attacks on Feed-Forward Arbiter PUFs Using Silicon Data*. Springer International Publishing, Cham, 53–67. https://doi.org/10.1007/978-3-319-24837-0_4

[12] Vincent Leest, Erik Sluis, Geert-Jan Schrijen, Pim Tuyls, and Helena Handschuh. 2012. Efficient Implementation of True Random Number Generator Based on SRAM PUFs. In *Cryptography and Security: From Theory to Applications*. Vol. 6805. 300–318. https://doi.org/10.1007/978-3-642-28368-0_20

[13] Ahmed Mahmoud, Ulrich Rührmair, Mehrdad Majzoobi, and Farinaz Koushanfar. 2013. Combined Modeling and Side Channel Attacks on Strong PUFs. *IACR Cryptology ePrint Archive* 2013 (2013), 632.

[14] M. Majzoobi, F Koushanfar, and M. Potkonjak. 2008. Lightweight secure PUFs. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*. 670–673. https://doi.org/10.1109/ICCAD.2008.4681648

[15] Dominik Merli, Frederic Stumpf, and Georg Sigl. 2013. Protecting PUF Error Correction by Codeword Masking. *IACR Cryptology ePrint Archive* 2013 (2013), 334. http://eprint.iacr.org/2013/334

[16] Elisabeth Oswald and Stefan Mangard. 2006. *Template Attacks on Masking—Resistance Is Futile*. Springer Berlin Heidelberg, Berlin, Heidelberg, 243–256. https://doi.org/10.1007/11967668_16

[17] Stjepan Picek, Annelie Heuser, Alan Jovic, and Axel Legay. [n. d.]. *Climbing Down the Hierarchy: Hierarchical Classification for Machine Learning Side-Channel Attacks*. https://doi.org/10.1007/978-3-319-57339-7_4

[18] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. 2010. Modeling Attacks on Physical Unclonable Functions. In *CCS '10*. ACM, 237–249. https://doi.org/10.1145/1866307.1866335

[19] Ulrich Rührmair, Xiaolin Xu, Jan Sölter, Ahmed Mahmoud, Mehrdad Majzoobi, Farinaz Koushanfar, and Wayne Burleson. 2013. Efficient Power and Timing Side Channels for Physical Unclonable Functions. In *CHES 2014*. 476–492. https://doi.org/10.1007/978-3-662-44709-3_26

[20] G. Edward Suh and Srinivas Devadas. 2007. Physical unclonable functions for device authentication and secret key generation. In *DAC '07*. 9–14. https://doi.org/10.1145/1278480.1278484

Chapter 3

Discussion

Four themes are highlighted in this work: (1) The proposed secure architecture. (2) The improvement and implementation of the architecture. (3) The Proposed Key Extraction Algorithm. (4) The new attack on PUFs. In this chapter, we discuss how the papers are interconnected, what changes needed to be put in place to go from designing CSHIA to implementing a real prototype of it. Moreover, we also discuss how studying side channel attacks allowed us to strengthen the architecture security and develop a new attack.

3.1 Architecture Design

The original proposal for CSHIA was to develop an architecture deeply integrated with PUFs, in such a way that its security strengthens and weaknesses would be related to or relied upon those present in PUFs. In the "*CSHIA Design*" paper, which proposed the CSHIA architecture model, we achieved this goal by presenting how the integration of the key extraction process would impact the architecture security. The CSHIA security parameters presented in the paper are intrinsically related to the properties of the proposed PUF-based key mechanism. The usage in the Fuzzy Extractor of a BCH code instance that matches a PUF-reliability model is an example of that. In the following, we dive into an analysis of the overall design of the CSHIA architecture discussing its advantages and downsides.

3.1.1 Design Analysis

The main feature of CSHIA is to yield authenticity and integrity for code and data. For authenticity, we designed CSHIA to take advantage of cryptographic keys that are extracted from SRAM-PUFs (SPUFs). SPUFs have been shown to provide binary outputs that not only have consistent high entropy values, but also are unique to every SPUF instance. Although Katzenbeisser *et al.* [30] have provided evidences that support such claims on SPUFs, we confirmed these properties on the CSHIA architecture by running a set of experiments that led to the results presented in Figure 10 of the "*CSHIA Design*" paper. The figure shows a distribution of Hamming distance between random keys extracted from different SRAM-PUFs after taking into account the analyses done in Section

6 of "*CSHIA Design*". Therefore, authenticity in CSHIA comes from each key that is random, unique, and has high entropy. By using such keys in the process of digesting memory blocks into tags, the architecture creates a unique set of pairs memory-block/PTAG for each CSHIA instance, which, without knowing the secret key, is hard to recreate.

On integrity, CSHIA presented an on-the-fly mechanism to generate and verify tags. Although performance evaluation was not done in "*CSHIA Design*", the concept of a dedicated bus and memory for integrity tags is an advantageous feature since it allows the designer to explore different and special design specifications (such as bandwidth, latency, etc.) with the goal of reducing the performance overhead. But the key point of CSHIA's integrity mechanism is its security evaluation presented in Section 4 of "*CSHIA Design*". That section discusses worst case scenarios and security limitations, which are the most important matter in secure computing systems since the security strength needs to match the asset value [55].

Despite all data we provided in "*CSHIA Design*", some points did not offer an in-depth analysis, resulting in downsides that we discuss next.

3.1.2 Critical Analysis

The "*CSHIA Design*" paper had some downsides, and CSHIA, as a model, presented some features that were difficult to implement. Overall, the main drawbacks are:

(A) It would be very hard to extract keys from SRAM cache memories of the processor.

(B) Lack of evidence that SRAM bits would not change due to wear-out.

(C) Lack of deep analyses against replay attacks.

(D) Lack of performance, area and power estimates.

(E) Lack of evidence that side channel attacks would not work (as claimed in Section 4 of "*CSHIA Design*").

The Item **(A)** is a strong and reasonable concern. Mainly, due to the need of powering up and down cache memories independently. We recall that such a procedure would be done only once in order for our proposed extraction algorithm[1] to determine stable memory words. In response to that, we clarify to the reader that non-volatile memories such as NAND Flash [42], Memristor [44], among others need to use circuitry that is capable of resetting memory cells. Obviously, this may complicate design and increase cost, but it enables the usage of SRAM as PUFs which have been already shown as those that generate the most random cryptographic keys with the highest entropy [30]. Although our proposed extraction algorithm may not work right away in current processors, we believe that it can be implemented in future systems.

For Item **(B)**, we must clarify that our experiments were done in multiple weather situations that include different temperatures, humidity, and days, which, by the way, were some times consecutive and others largely separated. Therefore, we believe that

[1] See Appendix B for more details.

climate may not impact in the reproduction of experiments and analyses. Furthermore, we are providing in this work the Appendices A and B that contain additional analyses and data. These Appendices show that the algorithm provides reliability and they also discuss and propose additional actions towards higher reliability in key extraction. However, in a large scale production of CSHIA there would be many instances that would stop working due to unrecoverable changes in SPUFs bits. As we stated in "*CSHIA Design*", the Fuzzy Extractor would be capable to correct up to 10 errors, beyond that the instances would unequivocally produce different PTAGs from those that were generated during the enrollment. In such cases, the manufacturer/vendor could offer a secure and authentic process to re-enroll CSHIA, programming a different key for extraction and regenerating PTAGs for the current state of the memory.

Item **(C)** reflects the limited analysis of replay attacks in "*CSHIA Design*". Despite we dedicated an entire section in "*CSHIA Design*" for replay attacks, we did not include comparisons to support our cache policy results and did not discuss the impact of the birthday paradox bound to replay attacks in our architecture. These limitations were solved in the "*CSHIA Implementation*". In this paper, we compared not only different caching policies for Merkle Tree, but also different countermeasures against replay attacks. That is a strong contribution, since it allows to determine clear trade-offs between those countermeasures. Regarding to why we chose the specific caching policies of "*CSHIA Implementation*", it is worth to point out that we did an in-depth analytical evaluation of different policies that is presented in Appendix C. The analyses in the appendix went deeper than the one we did in the "*CSHIA Design*" paper and included additional cache policies. After the evaluations, we realized that the cache policy in "*CSHIA Design*" would probably slowdown performance and would only fit well as a cache policy for a cache memory outside the security area of the system (the main chip). Thus, after obtaining the results presented in the appendix, we chose to evaluate in "*CSHIA Implementation*" only caching policies with good estimated performance.

Regarding area and power estimates (Item **(D)**), we were not concerned in presenting them in "*CSHIA Design*". However, we targeted these estimates in "*CSHIA Implementation*". Then again, estimating area and power showed to be harder than we expected. VHDL tools do not provide any relation between logic elements of FPGA and ASIC metrics. Thus, the most realistic way to estimate area and power is running tools that synthesize VHDL into ASIC library cells, which we were not able to do. Due to that, we chose to estimate area and power by directly correlating FPGA's logic elements and area and power of an already synthesized Leon3 implementation, which is shown in "*CSHIA Implementation*". We also could not provide performance estimates in "*CSHIA Design*" since at that time we did not have a full-fledge implementation of the architecture. In "*CSHIA Implementation*", however, we measured performance of our prototype in several benchmarks. Yet, as we discussed in "*CSHIA Implementation*", comparison among different works are complicated. Many works use Instructions per Second as performance measurement, which we could not obtain from the freeware version of Gaisler's GRMON. Thus, we used running time as a measurement. Moreover, in those works the selection of benchmarks and platforms varies. Some works simulated their architectures [26], others ran in different FPGA soft microprocessors [52, 48]. Therefore, under such a scenario, we

are confident to state that "*CSHIA Implementation*" presents enough evidence to support CSHIA's viability and capacity as a secure architecture for Embedded Systems.

Finally, Item **(E)** deals with a misunderstanding that we incurred in "*CSHIA Design*". When analyzing CSHIA's security (Section 4 in "*CSHIA Design*"), we stated that by generating a PTAG for the helper data and by verifying it after recovering the key we would prevent side channel attacks such as those presented in Merli *et al.*'s work [35]. Since then we had the opportunity of studying and applying real side channel attacks and we understood that it would be definitely possible to perform a side channel attack on CSHIA, as performed by Merli *et al.*, because it is possible to stop verifying the helper data's PTAG by resetting the chip after key recovery. In the face of that, we do not claim any physical security against non-invasive, semi-invasive, and fully-invasive attacks in "*CSHIA Implementation*". Additionally, further studies on side channel attack on Fuzzy Extractor led us to develop a new attack on PUFs, which is presented in "*PUF Attack*".

Overall, "*CSHIA Implementation*" addresses most of the drawbacks about CSHIA's design. However, some concepts presented in "*CSHIA Design*" had to be evolved. Next section discusses CSHIA's evolution.

3.2 Architecture Evolution

Some design choices done in "*CSHIA Design*" showed to be infeasible on an FPGA implementation of the architecture. For instance, due to the impossibility of extracting keys from SPUFs in current FPGAs, we needed to implement different PUFs to extract the CSHIA PRF key. Another example of change from design to implementation was the placement of the security components of CSHIA. Next, we provide more details about the changes that were required to create a path from design to an FPGA implementation.

3.2.1 Key Extraction and Fuzzy Extractor

One of the major changes from CSHIA's design was to use on-chip APUFs, instead of extracting keys from on-chip SPUFs as planned during design. There are two main reasons for that. First, it was not possible to implement our proposed algorithm for key extraction because we could not manipulate internal SRAMs in the FPGA chip as we needed. Second, even though it would be possible to use SRAMs from FPGA development kits to extract keys, they are placed externally to the FPGA chip and, since we expected to apply side channel attacks in a complete and running CSHIA implementation, the use of external memories increases power consumption of the FPGA chip and facilitates side channel attacks. Therefore, external SRAMs would basically allow us to determine memory word values as they are transfered, undermining the purpose of using the SRAM to extract secret key.

In conjunction to the key extraction issue, our knowledge about Fuzzy Extractor matured and the choices we made in "*CSHIA Design*" were found to be weak in terms of security. The main concern was the reduction in the key entropy, an intrinsic result of the chosen Fuzzy Extractor, and which, as stated in "*CSHIA Design*", was limited to 64 bits. In "*CSHIA Implementation*", we modified our Code-Offset Fuzzy Extractor [35] to an

adapted IBS Fuzzy Extractor [65] that does not reduce the entropy of the key. Although researchers consider side channel attacks to be feasible in the IBS FE [23], to the best of our knowledge, until now no concrete attack has been shown in the literature. As a matter of fact, we noticed that in such a Fuzzy Extractor a more effective side channel attack would be one that targets PUFs themselves rather than the structure of the Fuzzy Extractor. This is also a reason that led us to investigate side channel attacks on PUFs.

Another important issue about the original CSHIA's Fuzzy Extractor, that motivated us to redesign it, was that the known published countermeasure against side channel attacks [36] did not fit to CSHIA's mechanism of tag generation. The proposed countermeasure consists in adding random masks to the key during the error correction phase. At every recovery a new mask would be used. Therefore, new and different tags would be produced after every key recovery. Since PRFs are not linear, a mask cannot be removed if SipHash (CSHIA's PRF) digests it combined with a memory block. In spite of trying to come up with a different countermeasure to our original Fuzzy Extractor, we found out that a better solution was to redesign it.

3.2.2 Security Components

During CSHIA's design phase, it seemed reasonable to implement its components in the memory controller since it would be the one to control memory access. However, because implementation must take into account the construction of the platform, a component intercepting the bus between processor and main memory was found more convenient to adapt to the Leon3's platform. This change resulted from the work of Augusto F. R. Queiroz, who co-implemented CSHIA. Augusto, who is co-author of "*CSHIA Implementation*", implemented the BUS-HDLR and an initial SEC-ENG. These components were capable of capturing memory blocks but they did not have any security component.

Therefore, this work herein rebuilt SEC-ENG from scratch to accommodate all security features we presented in "*CSHIA Implementation*", including Merkle Tree management, timestamps management, PUFs, Fuzzy Extractor enrollment and regeneration. This work herein also fixed timing issues in BUS-HDLR resulted from the new SEC-ENG. Besides, the proposal and evaluation of different countermeasures against replay attacks and different cache configurations are contributions of this work herein.

We would like to point out that all benchmark analyses and modifications are part of this work herein. As stated in "*CSHIA Implementation*", these modifications in benchmarks were made to enable execution over Leon3's platform that does not have support to file system. Finally, we also point out that this work herein performed memory attacks on CSHIA to evaluate its security claims. Theses attacks consisted in tampering with loaded programs in the development kit's memory and PTAGs in PTAG Memory that was instantiated as a FPGA internal memory. In both cases, we used development tools from Altera and Gaisler to deploy these attacks. Appendix D provides an example of replay attack and how to perform it in each configuration of our CSHIA prototype.

3.2.3 Offering Different Replay Attack Countermeasures

Another goal of this work that came along with the implementation (not present in "*CSHIA Design*"), was to evaluate different Replay Attack Countermeasures. Most of research in this area focus on one type of countermeasure. Despite different constructions, these countermeasures will ultimately look like a tree of integrity [17, 27] or a value representing current state [53]. Thus, the analyses and solutions discussed in "*CSHIA Implementation*" can be taken further by modifying the current kind of Merkle Tree and timestamps implementation. This can help researchers to incorporate future Merkle Tree-based and timestamp-based approaches to CSHIA, thus enabling the usage of the architecture for multiple evaluations.

3.2.4 Targeting Embedded Systems

"*CSHIA Design*" presented CSHIA as an architecture for IoT devices. They are a particular class of Embedded Systems, which the main feature is the connection to the Internet, in addition to some specificities [38]. In "*CSHIA Implementation*", we chose to abandon the focus in IoTs because we noticed that our architecture could be employed in multiple embedded systems in general, but it would not fit in all IoT devices due to the variety of applications. In particular, we perceived CSHIA as a good architecture to preserve integrity of a data memory in applications that need to be audited. However, one fundamental point for any secure Embedded System, regardless whether they are targeting an IoT environment, is its cost in terms of area, power, and performance. In "*CSHIA Implementation*", we showed that CSHIA does not significantly slowdown performance, which is a very important metric for Embedded Systems since they are expected to use less resourceful processing systems. Despite that, we know that such results are limited to the suite of benchmarks we evaluated and cannot be generalized to any embedded application.

Furthermore, another important analyses we provided in "*CSHIA Implementation*", that is fundamental to Embedded Systems, were area and power overheads. But we did not limit our analyses in estimating increments of power and area only, we delved into the trade-offs between strengthening security and increasing power and area overheads. We discussed how a more secure timestamp solution against replay attacks will likely impose higher power and area overheads in an Embedded System due to a larger internal memory for timestamps. And, while that can keep good performance, it can compromise the applicability of the system due to higher energy and production costs. On the other hand, we argued that Merkle Tree does not yield as good performance as timestamps, but it can keep constant cost for different Embedded Systems applications. Even though our estimates for area and power overheads are not very precise, this analysis help designers to consider pros and cons of each situation. Additionally, the implementation of CSHIA will offer a possibility of concrete evaluation of those different scenarios, which may facilitate Embedded Systems design.

Overall, CSHIA can only be used in IoT devices, if it can be applicable to Embedded Systems in general. What "*CSHIA Implementation*" tried to do was to show that CSHIA can offer good performance with clear trades-offs between security and overheads.

3.2.5 Downsides and Limitations

Some downsides of CSHIA were neither deeply addressed nor addressed at all in both "*CSHIA Design*" and "*CSHIA Implementation*". I/O data tampering may be an issue in the architecture. In "*CSHIA Design*", we assumed that software countermeasure would be employed. In "*CSHIA Implementation*", we target embedded systems such that I/O tampering would not be a concern. In reality, I/O tampering can be critical since most systems will behave according to the input data.

Another issue is when Embedded Systems employ virtual memory. It is complicated to integrate CSHIA components in architectures that will have both virtual and physical memory address spaces. For instance, in some ARM processors [3], first level instruction caches work with virtual addresses, however, first level data caches work with physical addresses. In addition to that, working with virtual address space needs special care about repeated addresses to memory blocks that happen when the operating system set the same virtual address to different processes [43]. Moreover, a large virtual address space will require PTAGs to be stored in secondary memory, which will require help of the operating system to manipulate them. Making CSHIA not transparent anymore for the rest of the Embedded System components. Finally, Merkle Tree for virtual address space requires additional changes in operating systems in order to avoid unnecessary allocation of memory [12].

Other feature missing in CSHIA that one may see as a limitation is the fact it does not provide secrecy. Despite arguing in "*CSHIA Implementation*" that many Embedded Systems would not need encryption, adoption of secrecy might do CSHIA a more suitable choice for IoT devices since data/intellectual property theft can be a concern. However, as discussed in "*CSHIA Implementation*", cryptographic primitives for encryption would increase area and power and probably slowdown performance of CSHIA as well, thus reducing its strong points. Finally, one should notice that, if needed, secrecy can be employed through software in CSHIA. Also, secrecy does not solve the problem of privacy for the Internet of Things, which is a complicated matter per se [37]. Therefore, even without secrecy, CSHIA can yield desirable security features to IoT devices.

From the secure computer architecture point of view, we see these downsides and limitations as future challenges to work on an improved version of the architecture.

3.3 Dealing with Physical Attacks

One of the primary goals of CSHIA was to provide robustness against physical attacks. From our security model perspective, CSHIA do provide physical security if the main chip is not under physical access of an adversary. However, such a claim is hardly realistic for Embedded Systems that will probably run autonomously and unsupervised. Given all CSHIA components, the Fuzzy Extractor and the PUFs are those that are the most sensitive with respect to security. We explored countermeasures to attacks to Fuzzy Extractors, but we did not devise any new strategy to defend them against the current attacks. And, while exploring side channel attacks on PUFs, fostering countermeasures, we came up with a new attack that can pose a major threat to one of the fundamental

strengths in PUFs: their uniqueness.

The results presented in "*PUF Attack*" show an attack that has not been explored in the literature. Aiming at flip-flops, which are structural elements of the XOR Arbiter PUF's architecture, we were able to characterize their power consumption and then use that to predict HW and responses of other physical instances of this PUF. The importance of revealing such an attack viability is that it increases vulnerability awareness of system designers and security researchers. In an IoT environment, where devices will run autonomously and unsupervised, attackers will have physical access and be able to manipulate these devices. In that regard, they can capture power consumption and use the attack we proposed to unveil PUFs responses that are probably going to be used for key generation. Thus, weakening applications of cryptography in IoT devices.

But then again, our experimental setup in "*PUF Attack*" is far from a realistic scenario since we used isolated PUFs and we were able to set and control the trigger time of the attack. An isolated PUF on a chip is doubtedly to exist since it would generate only one bit of response. Finding the best time to trigger the capture of side channel information is a major trouble in real scenarios. However, showing that the concept can be applicable is the fundamental strength of the work. As stated in "*PUF Attack*", a sensitive and precise probe placed over a decapsulated chip may be capable of capturing power traces in a more realistic scenario (with multiple PUFs on chip). Additionally, further improvements on the technique we presented can show whether our attack is a good alternative to machine learning attacks. As we described in "*PUF Attack*", we did not achieve the top results of machine learning algorithms, but we used a significant smaller number of CRPs. This can make template attacks more viable since we may have scenarios where obtaining a large number of CRPs is not feasible.

Despite all that, the possibility that different PUF instances can show similar features should motivate research to seek new countermeasures against side channel attacks, besides looking for the viability of this attack in other PUFs. For small PUFs, the use of Erasable PUFs [28] can be a solution. Those PUFs limit the number of times a challenge can be applied, thus limiting the number of traces one can collect for each challenge. However, for larger PUFs, as used in "*PUF Attack*", such a strategy may not work, since there is a large number of challenges that can give the same operation result (Hamming Weight or Response). Therefore, in those cases, we still need new solutions.

3.4 Summary

In this chapter, we discussed the 3 papers that comprise this dissertation. "*CSHIA Design*" presented a secure computer architecture that provides code and data authenticity. The architecture design leverages on Physical Unclonable Functions to provide its security features. We discussed positives and negatives aspects of the design. Then, we presented "*CSHIA Implementation*", which aimed at implementing the architecture and improving negative aspects related to design. Finally, we talked about how we tried to improve the architecture's robustness against side channel attacks, and how that led us to develop a new attack on one kind of PUF. This endeavour was presented in "*PUF Attack*".

Chapter 4

Conclusion

In this work we presented CSHIA, a secure computer architecture for embedded systems that aims at providing authenticity and integrity for code and data. The work encompassed three phases: Design, Implementation, and Security Evaluation. In design, we laid out the basic ideas behind CSHIA, how integrity and authenticity are employed through the use of Physical Unclonable Functions (PUFs), and we proposed a key extraction algorithm that would take advantage of processors' intrinsic memories (e.g. caches). When we did its implementation, we built CSHIA as a flexible FPGA design with improved security features. Using the FPGA implementation we then evaluated its performance and overheads, such as area, energy, and memory size. Finally, as a side effect of the security evaluation of CSHIA, we delved into studying side channel analyses, which led us to develop a new side-channel-based attack on a particular type of PUF. That might push research towards developing new countermeasures.

Based on "*PUF Attack*", "*CSHIA Design*", and "*CSHIA Implementation*", and the discussion of Chapter 3, we believe that this work is a strong contribution to the research community. As result of this work, we have a secure architecture implemented in a solid platform, the Leon3 processor. Throughout this work, we emphasized the flexible design of CSHIA, which can now be explored in multiples ways. For instance, different clock frequencies, bandwidth, latencies can be evaluated for the PTAG memory. Also, different countermeasures against replay attacks can be evaluated to minimize overheads. Although some design proposals in "*CSHIA Design*" were not possible to be done in "*CSHIA Implementation*", we believe that CSHIA should be seen as work in progress since it can benefit from further improvements. Maybe, in the near future, it would be possible to apply our proposed algorithm in SRAMs to extract keys for the CSHIA implementation. Nonetheless, we presented an in-depth analysis of SRAM-PUF's reliability which may be applicable to other memory-based PUFs.

In addition to CSHIA, we presented an endeavour towards secure use of PUFs, which resulted in a new attack on XOR Arbiter PUFs. Despite the favouring conditions of our attack, and that it does not achieve the same level of prediction accuracy of the best works in the literature, we believe that it can be seen as a new threat to PUFs. If further works improve it, there is no doubt that this attack can become one of the most serious threat to devices that employ PUFs. Furthermore, it also leads to further questions on which PUFs would be safe against this attack and which countermeasure are needed.

Future Directions

As discussed in Section 3.2.5, to broaden the adoption of CSHIA, countermeasures against I/O related attacks, like buffer overflow, must be addressed. Some recent works have tackled these issues with integrated hardware-software solutions [15]. That kind of integration can be advantageous to CSHIA since it already has an integrity verification of memory blocks, which could be adapted to include control-flow integrity.

Moreover, managing virtual address and operating system awareness must be taken in consideration to amplify the spectrum of Embedded System applications that can employ CSHIA since modern micro-architecture design and operating systems use virtual address spaces. There are many challenges associated to this issue and they will certainly open up new research paths.

At the last, CSHIA will not be robust enough without technological countermeasures against physical attacks that circumvent architectural protection. As presented in this work, PUFs may be not enough to ensure code and data authenticity against resourceful adversaries. In regard to the upcoming challenges in security, the matter of preserving authenticity and authentic behavior is crucial for proper work of future embedded systems, mainly, in an IoT era. Therefore, countermeasures and vulnerability monitoring will need constant attention for the years to come.

Bibliography

[1] Cacti. http://www.hpl.hp.com/research/cacti/. Web Site. Accessed: 03-Feb-2016.

[2] D. Appello, A. Fudoli, V. Tancorre, F. Corno, M. Rebaudengo, and M.S. Reorda. A bist-based solution for the diagnosis of embedded memories adopting image processing techniques. In *Memory Technology, Design and Testing, 2002. (MTDT 2002). Proceedings of the 2002 IEEE International Workshop on*, pages 12–16, 2002.

[3] ARM. Cortex-a9 - technical reference manual. Technical report, 2008-2010.

[4] Frederik Armknecht, Roel Maes, Ahmad-Reza Sadeghi, Francois-Xavier Standaert, and Christian Wachsmann. A formalization of the security features of physical functions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 397–412, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Frederik Armknecht, Roel Maes, Ahmad-Reza Sadeghi, Berk Sunar, and Pim Tuyls. Memory leakage-resilient encryption based on physically unclonable functions. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 685–702. Springer Berlin Heidelberg, 2009.

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[7] Boaz Barak, Ronen Shaltiel, and Eran Tromer. True random number generators secure in a changing environment. In Colin D. Walter and Koç, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 166–180, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[8] G. T. Becker. Robust fuzzy extractors and helper data manipulation attacks revisited: Theory vs practice. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–1, 2017.

[9] Georg T. Becker. The gap between promise and reality: On the insecurity of xor arbiter pufs. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 535–555. Springer Berlin Heidelberg, 2015.

[10] Mudit Bhargava and Ken Mai. An efficient reliable puf-based cryptographic key generator in 65nm cmos. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, March 2014.

[11] IEEE Author Center. Avoid infringement upon ieee copyright. https://ieeeauthorcenter.ieee.org/choose-a-publishing-agreement/avoid-infringement-upon-ieee-copyright/. Web Site. Accessed: 29-Sep-2018.

[12] David Champagne, Reouven Elbaz, and RubyB. Lee. The reduced address space (ras) for application memory authentication. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security*, volume 5222 of *Lecture Notes in Computer Science*, pages 47–63. Springer Berlin Heidelberg, 2008.

[13] C. H. Chang, Y. Zheng, and L. Zhang. A retrospective and a look forward: Fifteen years of physical unclonable function advancement. *IEEE Circuits and Systems Magazine*, 17(3):32–62, 2017.

[14] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *CHES*, pages 13–28. Springer, 2002.

[15] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. Hafix: Hardware-assisted flow integrity extension. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 74:1–74:6, New York, NY, USA, 2015. ACM.

[16] Jay L Devore. *Probability and Statistics for Engineering and the Sciences*. Cengage learning, 2011.

[17] Reouven Elbaz, David Champagne, RubyB. Lee, Lionel Torres, Gilles Sassatelli, and Pierre Guillemin. Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 289–302. Springer Berlin Heidelberg, 2007.

[18] Jean-Bernard Fischer and Jacques Stern. An efficient pseudo-random generator provably as secure as syndrome decoding. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 245–255, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[19] B. Gassend, G.E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 295–306, Feb 2003.

[20] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 148–160, New York, NY, USA, 2002. ACM.

[21] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[22] C. Helfmeier, C. Boit, D. Nedospasov, and J.-P. Seifert. Cloning physically unclonable functions. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, pages 1–6, June 2013.

[23] M. Hiller, D. Merli, F. Stumpf, and G. Sigl. Complementary ibs: Application specific error correction for pufs. In *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 1–6, June 2012.

[24] Caio Hoffman. CSHIA VHDL Source Code. http://caiohoffman.org/cshia. Informations: caiohoffman <at> gmail <dot> com.

[25] **© 2015 IEEE. Reprinted, with permission, from** C. Hoffman, M. Cortes, D.F. Aranha, and G. Araujo. Computer security by hardware-intrinsic authentication. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2015 International Conference on*, pages 143–152, Oct 2015.

[26] Mei Hong and Hui Guo. Fedtic: A security design for embedded systems with insecure external memory. In Tai-hoon Kim, Young-hoon Lee, Byeong-Ho Kang, and Dominik Ślkezak, editors, *Future Generation Information Technology*, pages 365–375, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[27] Mei Hong, Hui Guo, and Sharon X. Hu. A cost-effective tag design for memory data authentication in embedded systems. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 17–26, New York, NY, USA, 2012. ACM.

[28] Chenglu Jin, Xiaolin Xu, Wayne Burleson, Ulrich Rührmair, and Marten van Dijk. Playpuf: Programmable logically erasable pufs for forward and backward secure key management. Technical Report 2015/1052, Cryptology ePrint Archive, 2015.

[29] D. Karakoyunlu and B. Sunar. Differential template attacks on puf enabled cryptographic devices. In *2010 IEEE International Workshop on Information Forensics and Security*, pages 1–6, Dec 2010.

[30] Stefan Katzenbeisser, Ünal Kocabaş, Vladimir Rožić, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. Pufs: Myth, fact or busted? a security evaluation of physically unclonable functions (pufs) cast in silicon. In *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'12, pages 283–301, Berlin, Heidelberg, 2012. Springer-Verlag.

[31] Raghavan Kumar and Wayne Burleson. *Side-Channel Assisted Modeling Attacks on Feed-Forward Arbiter PUFs Using Silicon Data*, pages 53–67. Springer International Publishing, Cham, 2015.

[32] Vincent Leest, Erik Sluis, Geert-Jan Schrijen, Pim Tuyls, and Helena Handschuh. Efficient implementation of true random number generator based on sram pufs. In David Naccache, editor, *Cryptography and Security: From Theory to Applications*, volume 6805 of *Lecture Notes in Computer Science*, pages 300–318. Springer Berlin Heidelberg, 2012.

[33] Ahmed Mahmoud, Ulrich Rührmair, Mehrdad Majzoobi, and Farinaz Koushanfar. Combined modeling and side channel attacks on strong pufs. *IACR Cryptology ePrint Archive*, 2013:632, 2013.

[34] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[35] Dominik Merli, Dieter Schuster, Frederic Stumpf, and Georg Sigl. Side-channel analysis of pufs and fuzzy extractors. In JonathanM. McCune, Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, Angela Sasse, and Yolanta Beres, editors, *Trust and Trustworthy Computing*, volume 6740 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin Heidelberg, 2011.

[36] Dominik Merli, Frederic Stumpf, and Georg Sigl. Protecting PUF error correction by codeword masking. *IACR Cryptology ePrint Archive*, 2013:334, 2013.

[37] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad hoc networks*, 10(7):1497–1516, 2012.

[38] A. Mosenia and N. K. Jha. A comprehensive study of security of internet-of-things. *IEEE Transactions on Emerging Topics in Computing*, 5(4):586–602, 2017.

[39] Elisabeth Oswald and Stefan Mangard. *Template Attacks on Masking—Resistance Is Futile*, pages 243–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[40] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 2009.

[41] Andrei Pavlov and Manoj Sachdev. *CMOS SRAM circuit design and parametric test in nano-scaled technologies: process-aware SRAM design and test*, volume 40. Springer Science & Business Media, 2008.

[42] Volnei Pedroni. *Digital Electronics and Design with VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[43] A. Rogers, M. Milenkovic, and A. Milenkovic. A low overhead hardware technique for software integrity and confidentiality. In *2007 25th International Conference on Computer Design*, pages 113–120, Oct 2007.

[44] Garrett S. Rose, Nathan McDonald, Lok-Kwong Yan, and Bryant Wysocki. A write-time based memristive puf for hardware security applications. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 830–833, Piscataway, NJ, USA, 2013. IEEE Press.

[45] Ulrich R"uhrmair, Heike Busch, and Stefan Katzenbeisser. *Strong PUFs: Models, Constructions, and Security Proofs*, pages 79–96. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[46] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 237–249, New York, NY, USA, 2010. ACM.

[47] Ulrich Rührmair, Xiaolin Xu, Jan Sölter, Ahmed Mahmoud, Mehrdad Majzoobi, Farinaz Koushanfar, and Wayne Burleson. Efficient power and timing side channels for physical unclonable functions. In *CHES 2014*, pages 476–492.

[48] Johanna Sepulveda, Felix Willgerodt, and Michael Pehl. Sepufsoc: Using pufs for memory integrity and authentication in multi-processors system-on-chip. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, GLSVLSI '18, pages 39–44, New York, NY, USA, 2018. ACM.

[49] S. Skorobogatov. Using optical emission analysis for estimating contribution to power analysis. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, volume 00, pages 111–119, 09 2009.

[50] Lifeng Su, S. Courcambeck, P. Guillemin, C. Schwarz, and R. Pacalet. Secbus: Operating system controlled hierarchical page-based memory bus protection. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 570–573, April 2009.

[51] G. Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 9–14, New York, NY, USA, 2007. ACM.

[52] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 25–36, Washington, DC, USA, 2005. IEEE Computer Society.

[53] G.E. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 339–350, Dec 2003.

[54] 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.

[55] ARM Security Technology. Building a secure system using trustzone technology. Technical report, 2005-2009.

[56] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 51–62, Washington, DC, USA, 2008. IEEE Computer Society.

[57] Somogy Varga and Charles Guignon. Authenticity. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2017 edition, 2017.

[58] Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet, Eduardo Wanderley, Russell Tessier, and Wayne Burleson. A security approach for off-chip memory in embedded microprocessor systems. *Microprocessors and Microsystems*, 33(1):37 – 45, 2009. Selected Papers from ReCoSoC 2007 (Reconfigurable Communication-centric Systems-on-Chip).

[59] Don C. Weber. Optiguard: A smart meter assessment toolkit. Technical report, InGuardians, Inc., July 2012.

[60] Rolf H Weber. Internet of things–new security and privacy challenges. *Computer law & security review*, 26(1):23–30, 2010.

[61] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 3rd edition, 2005.

[62] Marilyn Wolf. *High-performance embedded computing: applications in cyber-physical systems and mobile computing*. Newnes, 2014.

[63] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 351–, Washington, DC, USA, 2003. IEEE Computer Society.

[64] Huilin Yin and Ruiying Zhao. *Template-Based and Second-Order Differential Power Analysis Attacks on Masking*, pages 8–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[65] Meng-Day (Mandel) Yu and Srinivas Devadas. Secure and robust error correction for physical unclonable functions. *IEEE Des. Test*, 27(1):48–65, 2010.

[66] YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. zyb@is.iscas.ac.cn 13083 received 27 Oct 2005.

Appendix A

Modeling SRAM Failure Probability

In this appendix we show how we modeled the Failure Probability presented in Figures 8 and 9 of the paper "*CSHIA Design*".

A.1 Assumptions

Herein consider that an assessment is an observation of logical values of SRAM cells after a power-on. Given a sequence of consecutive power-on/off cycles, let us define $n_f(t)$ as the cumulative number of SRAM cells that, at the time t, have presented at least once a different value from those they had in their initial assessment. In terms of modeling, we can assume that each assessment is an event that happens in a discrete time T_i. Given a function $F(t)$, which describes a discrete state of the cumulative percentage of flipped bits in any give time, hence:

$$F(t) = \frac{n_f(t)}{N} \tag{A.1}$$

Where N is the total number of bits in the memory. Figure A.1 shows the behavior of $F(t)$ for 10 SRAMs. Given a random variable X, the probability of finding flipped bits in a time t is

$$\Pr(X = n_f(t)) = F(t) \tag{A.2}$$

Based on the behavior of the cumulative distribution of Figure A.1, we establish our main *assumption*, that is

$$\lim_{t \to \infty} \Pr(X = n_f(t)) \approx \Pr(X = n_f(T_{200})) \tag{A.3}$$

A.2 Probability of having a Bit-Flip

Given our assumption, we would like to know the probability of randomly selecting a bit of a SRAM at the moment t, where $t \leq T_{200}$, such that it will not flip at $t \to \infty$. Namely,

$$1 - \Pr(X = n_f(t) | X = n_f(T_{200})) \tag{A.4}$$

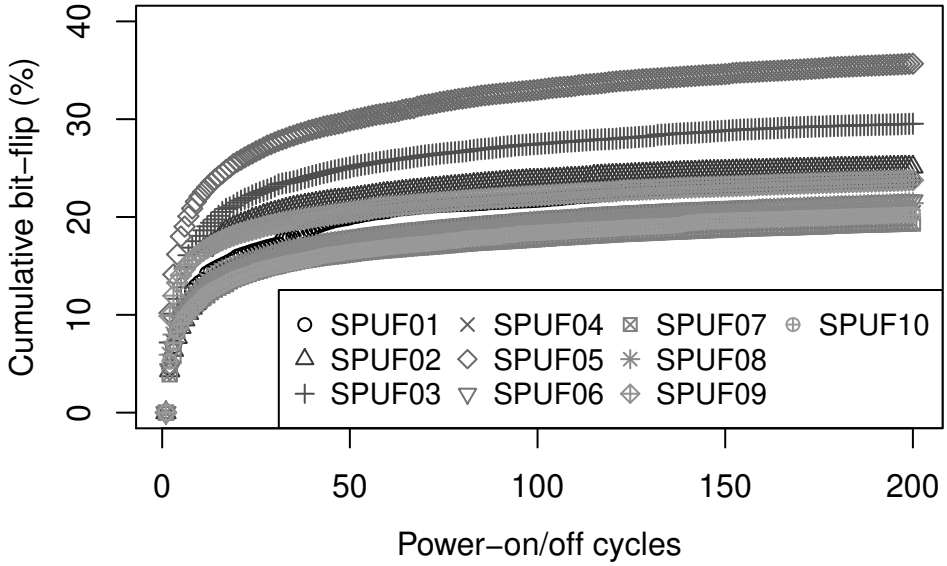


Figure A.1: The fraction of SPUF bits that have flipped during 200 power-on/off cycles (assessments).

Now let us describe Equation A.4, given Bayes Rules [16], we have:

$$\Pr(X = n_f(t)|X = n_f(T_{200})) = \frac{\Pr(X = n_f(T_{200})|X = n_f(t)) \cdot \Pr(X = n_f(t))}{\Pr(X = n_f(T_{200}))} \tag{A.5}$$

We know that $\Pr(X = n_f(T_{200})|X = n_f(t))$ is:

$$\frac{\Pr(X = n_f(T_{200}) \cap X = n_f(t))}{\Pr(X = n_f(t))} \tag{A.6}$$

For any given time $t \leq T_{200}$, $n_f(T_{200}) \cap n_f(t) = n_f(t)$. Thus, $\Pr(X = n_f(T_{200}) \cap X = n_f(t)) = \Pr(X = n_f(t))$. From all that, we conclude:

$$\begin{aligned} 1 - \Pr(X = n_f(t)|X = n_f(T_{200})) &= 1 - \frac{\Pr(X = n_f(t))}{\Pr(X = n_f(T_{200}))} \\ &= 1 - \frac{F(t)}{F(T_{200})} \end{aligned} \tag{A.7}$$

Therefore, Equation A.7 is the probability of selecting a bit at t that will not have flipped in $t \to \infty$.

A.3 Modeling Key Extraction

Let us assume that a key K is compounded of n randomly selected bits from a SRAM. Let us also assume that we have a tolerance of δ bit-flips that can happen at any given time t. Given an extraction algorithm $\texttt{Extr}(\cdot)$ to obtain K. *We want to determine a model for the probability of this algorithm extracting K with no more than δ bit-flips at all times.*

We have two possible models to go with. First, this algorithm randomly extracts bits to compound K and they can be chosen more than once for any possible K. In a second model, each bit is extracted once to compound K and cannot be used again, even for different values of K. The first model demands that not only the source (SRAM) is a true random number generator, but also that the bit selector is truly random, which, in practical terms, is hard to implement. On the other hand, the second model is more stringent in the number of possible ways to compound K, but it only depends on the randomness provided by the SRAM.

Statistically, the first model includes replacement, which allows multiple choices of a same element and can be modeled by cumulative binomial distribution. The second model is without replacement and can be modeled by the cumulative hypergeometric distribution. Despite not finding a proof of which model strengthens security more, given the reasoning above, we chose to model $\texttt{Extr}(K)$ without using replacement.

A.4 Key Failure Probability

Assume that K is a set of m words with length l from a SRAM, such as $m \cdot l = n$ bits. Let us assume that the algorithm $\texttt{Extr}(K)$ knows at any given time t which words did not have any bit-flip for all $T_i \leq t$. Based on the assumption of Equation A.3, if the algorithm picks K at the moment t, we can determine the probability that such a selection will not have more than δ bit-flips in the future, if we know how many words show bit-flips in $t \to \infty$.

In $t \to \infty$, we will find memory words in certain configurations, depending on its length. For instance, if $l = 1$, meaning 1-bit word, in $t \to \infty$ we can find words that have never had a bit-flip and those that did. If $l = 2$, in $t \to \infty$ we can find words that have never had a bit-flip, those that had 1 bit-flip, and those that had 2 bit-flips. And, so on. Therefore, to minimize the selection of words that in concatenation will have had more than δ bit-flips in $t \to \infty$, we need to compute the probabilities of selecting words that has no bit-flip at T_i but may show 1, 2, or more bit-flips in $t \to \infty$. Given that, we can establish a threshold probability that enables us to figure out a time in which we can extract a K that is unlikely to present more than δ bit-flip in $t \to \infty$.

Let be $q_j(t)$ the cumulative amount of words with j bit-flips at the time t. Given a time T_i and being $T_{200} = t \to \infty$. Let us define the function:

$$Q_j(T_i) = \begin{cases} q_0(T_{200}) & , j = 0 \\ q_j(T_{200}) - q_j(T_i) & , j \neq 0 \end{cases} \tag{A.8}$$

The number of all possible ways of choosing m words without bit-flips in T_i is

$$\binom{q_0(T_i)}{m}$$

At the time T_{200}, $q_0(T_i) - q_0(T_{200})$ will be the number of words that did not have any

bit-flip at T_i but now has at least one bit-flip. In fact, $q_0(T_i) = q_0(T_{200}) + q_1(T_{200}) - q_1(T_i) + \cdots + q_l(T_{200}) - q_l(T_i) = Q_0(T_i) + Q_1(T_i) + \cdots + Q_l(T_i)$.

Now, we want to determine the probability of picking words at the time T_i that will have no bit-flip in T_{200}. That is given by:

$$\frac{\binom{Q_0(T_i)}{m}}{\binom{q_0(T_i)}{m}} \tag{A.9}$$

In other words, Equation A.9 is the probability of choosing m words at the time T_i that will have no bit-flip in T_{200}, given all possible ways of choosing m words with no bit-flip at T_i. Because, we have a tolerance of δ bit-flips, we can choose any combination of words that has no bit-flip at T_i, but they will have at the most δ bit-flips in T_{200}. This will become:

$$\begin{aligned} &\frac{\binom{Q_0(T_i)}{m} + \binom{Q_0(T_i)}{m-1} \cdot \binom{Q_1(T_i)}{1} + \binom{Q_0(T_i)}{m-1} \cdot \binom{Q_2(T_i)}{1}}{\binom{q_0(T_i)}{m}} + \cdots \\ + \quad &\frac{\binom{Q_0(T_i)}{m-1} \cdot \binom{Q_\delta(T_i)}{1} + \binom{Q_0(T_i)}{m-2} \cdot \binom{Q_1(T_i)}{1} \cdot \binom{Q_2(T_i)}{1}}{\binom{q_0(T_i)}{m}} + \cdots \end{aligned} \tag{A.10}$$

Equation A.10 takes into account all possible ways to assemble K with words whose the sum of bit-flips is less than or equal to δ in T_{200}. Notice that this probability is very stringent because it assumes that the selected words will have simultaneous up to δ bit-flips. While in reality, the bit-flips might not happen simultaneously. Thus, this assumption worsens our scenario. Statistically, the result of such a restriction comes from the fact that we are using the cumulative distribution of bit-flips (from Figure A.1) rather than the density distribution of bit-flips.

Now, assume that m can be written as the sum in Equation A.11, where m_i is the number of *selected words* with i bit-flips.

$$m = m_0 + m_1 + m_2 + \cdots \tag{A.11}$$

Given the following equation:

$$\delta \geq 0m_0 + 1m_1 + 2m_2 + \cdots + \delta m_\delta \tag{A.12}$$

We want to determine the set of all possible solutions S of the system formed by Equations A.11 and A.12. For the sake of exemplification, let us exam these equations. Assume $\delta = 10$ and suppose that we picked $m = 8$ words in T_i. Now, in T_{200}, 7 words present 1 bit-flip and 1 presents 2 bit-flips, i.e. $m_1 = 7$ and $m_2 = 1$. This follows $m = m_1 + m_2 = 8$. From Equation A.12, $10 \geq 0 \cdot 0 + 1 \cdot 7 + 2 \cdot 1 + \cdots + 10 \cdot 0 = 9$.

Hence, those words can generate a stable key, since they do not surpass our limit of 10 cumulative bit-flips. On the other hand, having picked 8 words in T_i that now in T_{200} are in the following configuration $m_0 = 2, m_1 = 2, m_2 = 2, m_3 = 2$ does not give us a solution for Equation A.12. Thus these words cannot form a stable key.

Given all reasoning above, a general Key Failure Probability is:

$$\Pr(T_i) = \frac{\sum_{s \in S} \left[\prod_{m_j \in s} \binom{Q_j(T_i)}{m_j} \right]}{\binom{q_0(T_i)}{m}} \tag{A.13}$$

In Equation A.13, subset s is a solution for Equations A.11 and A.12 and S is the set of all possible solutions of these equations. Thusly, to generate Figures 8 and 9 in "*CSHIA Design*" we used Equation A.13 with $\delta = 10$. For Figure 8, we used $l = 1$ and thus K was $m = 128$. That is, $m = n = 128$ bits. We assumed that an assessment $i = T_i$. Therefore, each point in Figure 8 is a pairwise $(i, \Pr(i))$ for 10 SRAMs. After computing those points we used the linear regression as described in the paper. In Figure 9, $l = 16$, and $m = 8$ words. That gives us $n = 128$ bits.

Overall, we hope this appendix helps the understanding of Section 6 of "*CSHIA Design*".

Appendix B

Extractor Algorithm

In this appendix we give more details of the key extraction procedure presented in "*CSHIA Design*". We present the extraction algorithm and we evaluate its results. Moreover, we discuss further improvements to be applied to the algorithm.

B.1 Algorithm Parameters

During production tests and evaluation of memories, bitmaps of the memory cells are created using techniques like *Built-in Self-Test* (BIST) [2] to enable fault detection. If these bitmaps are captured over time, after multiple power on/off cycles, it is possible to use them to perform the analysis made in Section 6 of "*CSHIA Design*" whose model we presented and discussed in Appendix A. Given a set of SPUFs prototypes and assuming them as samples of a particular fabrication technology. The linear regression with a confidence interval done in Section 6 of "*CSHIA Design*" allows us to predict the number of power on/off cycles that any SPUF of such a technology would need to enable the selection of memory words that will have a small probability of having more than δ bit-flips at any given time in the future.

For the purpose of exemplification, let us assume a set of SPUFs prototypes {SPUF04, SPUF06, SPUF07, SPUF08, SPUF10} from Figure A.1. After collecting bitmaps of those SPUFs, we modeled their failure probability in extracting 128-bit keys with tolerance of 10 bit-flips. A linear regression of this sample with 99 % confidence in the prediction interval is shown by Figure B.1. Selecting 128-bit keys after 150 power on/off cycles gives a probability lower than 10^{-6} that such keys will present, in the future, more than 10 simultaneous bit-flips.

Although a sample of only 5 individuals of the population may lead to a big margin of error, even with 99 % confidence, the way the analysis is made would not change for larger samples. In that regard, determining how many power on/off is needed for a large population will depend on two key factors: the size of the sample of prototypes and the number of bitmaps available. Having a large sample and thousands of bitmaps available of each individual of the sample will make the analysis very robust. Regardless that, the number of power on/off cycles will continue to be the main result and the input of the algorithm that determines which words we should use to form a stable key.

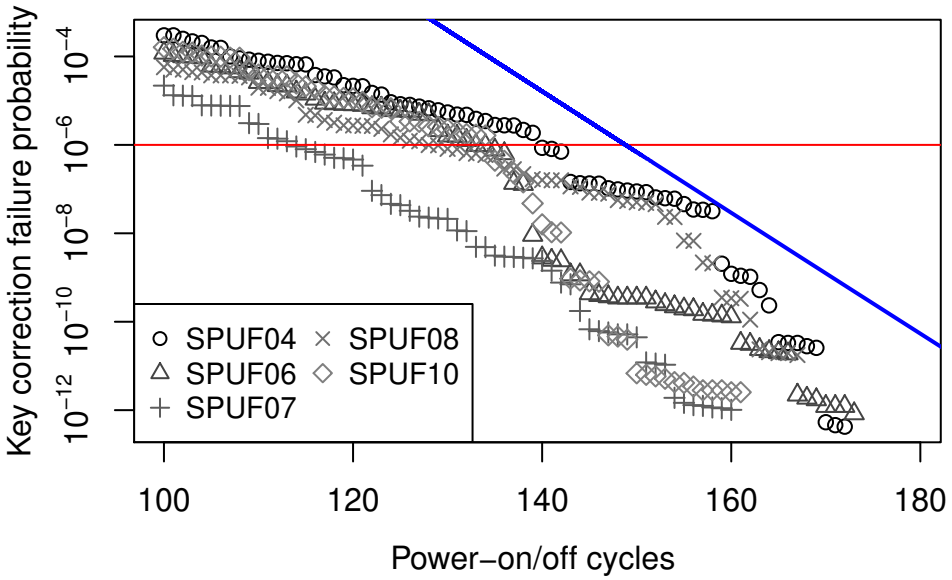


Figure B.1: Probability of more than 10 simultaneous bit-flips occurring when extracting 128-bit keys from SRAMs versus the number of cycles of power off and on.

B.2 SRAM Address Selection

In order to clarify what was proposed in Section 6 of "*CSHIA Design*", we introduce a descriptive algorithm that determines SRAM words that will have low probability of having bit-flips over their lifetime usage. Given two SPUFs available, for instance, first level instruction and data cache memories, and also given the number of power on/off, the algorithm 14 returns a set of address that indicates stable words.

Data: $m \leftarrow$ number of power ups.
Data: $X \leftarrow$ SPUF target.
Data: $Y \leftarrow$ SPUF reference.
Result: Addresses of stable words in the X.

```
1  PowerUp(X);
2  CopyAllWordsFromTo(X, Y);
3  SetValidWord(AllWords(Y));
4  for i := 1 to m do
5  |  PowerUp(X);
6  |  for j := 1 to Length(AllWords(X)) do
7  |  |  if ValidWord(Y[j]) then
8  |  |  |  if X[j] ≠ Y[j] then
9  |  |  |  |  SetInvalidWord(Y[j]);
10 |  |  |  end
11 |  |  end
12 |  end
13 end
14 return ValidAddresses(Y);
```

Algorithm 1: Method to determine the addresses of stable words in SPUFs.

The algorithm works making one SPUF as reference and the other as the target. The target SPUF is the one we want to determine stable memory words. Function PowerUp(·)

turns the target SPUF off and on. Function CopyAllWordsFromTo(·) copies the words of the target to the reference SPUF. After making the words in the reference equal to the target, the algorithm sets all memory words of the reference as stable ones. For each PowerUp(·) of the target, its words are compared to the reference. If they do not match, they are set invalid in the reference by the function SetInvalidWord(·). The worst case scenario of this algorithm is $m \cdot n$, where m is the number of power-on/off cycles and n is the number of words in the SPUFs. At the end of at least m loops, the reference SPUF has a set of words that had never been different from their initial state, which is the first power-on of the target SPUF. The algorithm then returns these addresses through the function ValidAddresses(·). Therefore, the returned addresses are those that, in the target SPUF, are the most unlike to have bit flips at any future power-on/off cycle.

B.3 Key Reliability Analysis

To simulate a scenario of usage of the algorithm, suppose now that we have Algorithm 14 implemented in ASIC instances of CSHIA. Assume that the algorithm runs with the following parameter, the target SPUF is the Data Cache Memory, the reference is the Instruction Cache Memory, and the number of power-on/off is 150, which comes from simulating SPUF04, SPUF06, SPUF07, SPUF08, and SPUF10 as sample instances during production and whose analyses is presented by Figure B.1. Given that, we simulated SPUF01, SPUF02, SPUF03, SPUF05, and SPUF09, from Figure A.1, as first level data cache memories of final ASIC CSHIA instances. Therefore, the enrollment procedure of each instance executes Algorithm 14 to determine the set of addresses, from those caches, in which stable memory words to compose cryptographic keys can be found.

For the sake of evaluation, we used the returned addresses from the simulation of the enrollment of SPUF01, SPUF02, SPUF03, SPUF05, and SPUF09, to randomly form 1 million different keys of 128 bits for each SPUF. For every key, we simulated a shutdown and turned-on of each CSHIA instance and we evaluated whether each key have presented more than 10 bit-flips. Namely, whether these keys overpassed our tolerance. We did this procedure for 200 times. From SPUF01, SPUF02, SPUF03, SPUF05, and SPUF09, only SPUF09 presented keys that had more than 10 simultaneous bit-flips. From one million keys, exactly 3046 presented more than 10 bit-flips in some moment of 200 simulated key regenerations. That is, only 0.3 % of SPUF09's keys overpassed our established tolerance.

B.4 Improving Key Reliability

Despite only 0,3 % of keys might fail, it is possible to lower this number for SPUF09 by using Temporal Majority Voting (TMV) [5]. TMV would consist in extracting the key and saving it in a temporary register. Powering down and up the SPUF one more time and extracting the key again and saving it in another register, and so on. Doing that an odd number of times we can count how many time the same bit value, for each bit in the key, appeared. The value that scores the most wins.

To evaluate how TMV can reduce the instability in keys, we chose majoring the votes

by 3. In such a scenario, we evaluate 600 power-on/off cycles of SPUF09, which would represent 200 key regenerations of an ASIC instance of CSHIA with TMV. The number of SPUF09's keys that, in this scenario, presented more than 10 bit-flips went down to only 11, a reduction of almost three orders of magnitude.

B.5 Algorithm Discussion

To implement Algorithm 14 in a real ASIC instance of CSHIA, the processor needs to pass by some deep modifications, mainly because it would need to be able to independently power off and on each cache memory. This seems to be hard to materialize, but it is possible. It is worth to remember that memories such as Phase-Change Memories, Flash, Memristors, need a reset circuit to modify memory cells into different states. Thus, such a circuit capability will incur in higher cost of production of ASICs, but we believe that that is a good feature to have in processors since it enables extraction of keys from SRAMs.

Appendix C

Evaluation of Cache Policies for Merkle Tree

One of the main ways to deal with replay attacks is using Merkle Tree, as presented in Section 5 of "*CSHIA Design*". For the purpose of reducing performance penalties, cache memory for nodes in the tree are a common solution. In this chapter, we exam four cache policies to reduce memory access to PTAG Memory.

C.1 Caching Policies

Regarding Merkle Tree, caching policies are strategies to reduce accesses to the memory of tags. Recall that every data memory block verification or update needs to walk the whole tree from leaf to the root. If the Merkle Tree is externally stored, that means a significant number of memory accesses, which is proportional to the height of the tree. Due to the tree structure, caching policies for Merkle Tree may differ from those of data/instruction cache memories, mainly because tree addresses may hinder sequential access and increase misses and evictions.

In the following, we present 4 caching policies for Merkle Tree and we discusses each one:

• **Caching.Path** always caches nodes from a leaf-root path, while the tag of a data memory block is verified/updated. The basic principle of this policy is an untrusted cache memory, placed outside of the secure zone of a system and which can be under control of an adversary. Therefore, if a node of the tree is found in cache, it is not necessary to access the external memory of tags, but verification must go up to the root tag, which inexorably needs to belong to the secure zone of the system.

• **CHTree**. This policy was adopted by Gassend *et al.* in [19]. It consists in stopping verifying/updating nodes of a leaf-root path at the first node found in cache. This policy leverages on caches located inside the secure zone of systems and, therefore, cannot be under the control of an adversary.

• **Read.Hit**. This policy mixes CHTree and Caching.Path. For verifying a leaf-root path, this policy adopts CHTree, namely, it stops verifying nodes of the tree at the first one found in cache. However, a problem in CHTree is possible scenarios of inconsistence.

If a tree node is found in cache and updated, its ancestors can be left outdated. So, when those updated nodes need to be evicted their ancestors need to be updated to keep consistence. If the ancestors are not in the cache, they need to be brought. This can create another eviction and lead to a long sequence of evictions and updates. If all nodes in the leaf-root path are always updated, evictions are not necessary.

• **SecBus** [50] modifies CHTree, which we can refer to *as soon as possible* (ASAP) policy replacement, and establishes an *as late as possible* (ALAP) policy. ALAP delays eviction of dirty nodes by vacating clean nodes in the cache instead. When all nodes in cache are dirty, the cache controller should use an eviction policy in order to avoid deadlock. For the sake of this work, evictions are placed by the *least recently used* (LRU) policy.

C.2 Simulating Caching Policies

For evaluation of each cache policy, we simulated them in a PIN Tools cache simulator. The simulated system had a 64-KB First level Data Cache (FDC) memory with 8-way set associativity, 128 lines, and blocks of 64 bytes. For the Merkle Tree cache, we established a 64-KB memory cache with 16-way set associativity, 64 lines, and blocks of 512 bits. Notice that while a 64-byte memory block represents a cache line, 64-byte tree cache block represents a chunk of $d = 8$ tags, considering tags with the same length of a PTAG, namely, 64 bits. The address space of the simulation is 48 bits, which results in a Merkle Tree with $L = 15$ levels. We ran all benchmarks of the SPEC CPU2006.

Figure C.1 shows the miss rate of the Merkle Tree cache for each policy. Caching.Path shows the lowest rates. However, a deeper analysis in the results showed that the total number of memory accesses of Caching.Path was too large due to the repeating process of always verifying/updating all nodes in a leaf-root path, which led to a small percentage of misses, but the amount of misses itself is significant.

Another measurement we took from our simulation was the ratio between Merkle Tree cache misses and FDC misses. This gives us the impact of data memory block miss in the number of accesses to the external memory of tags. The results of this metric for the four caching policies are presented in Figure C.2. Notice that in the figure Caching.Path does not seem to be as much exceptional as it was in Figure C.1. For almost all benchmarks the performance of all policies were sightly similar. Two benchmarks are distinctive: *cactusADM* and *sjeng*. In *sjeng*, CHTree and SecBus perform worst than the others due to the high number of evictions these benchmarks present. In the case of *cactusADM*, the number of reads overwhelms the number of writes and, because Caching.Path must always walk all the leaf-root path, that results in a larger number of misses for this policy.

Next, we investigate the occurrence of evictions (external memory writes) in the Merkle Tree cache compared to FDC misses. This elucidates the impact of a data memory block miss and the expected number of memory writes in the external memory of tags. Figure C.3 shows that Caching.Path and Read.Hit have lower numbers of evictions than CHTree and SecBus in almost all benchmarks.

Overall, from Figure C.2 and Figure C.3, one may conclude that the expected number

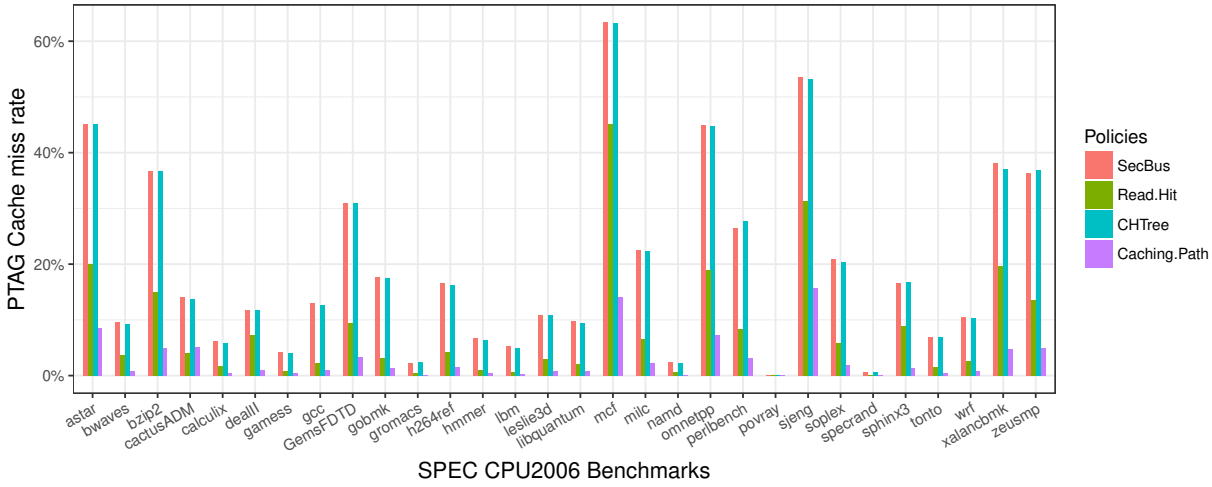


Figure C.1: PTAG Cache miss rate for the SPEC CPU2006 benchmarks.

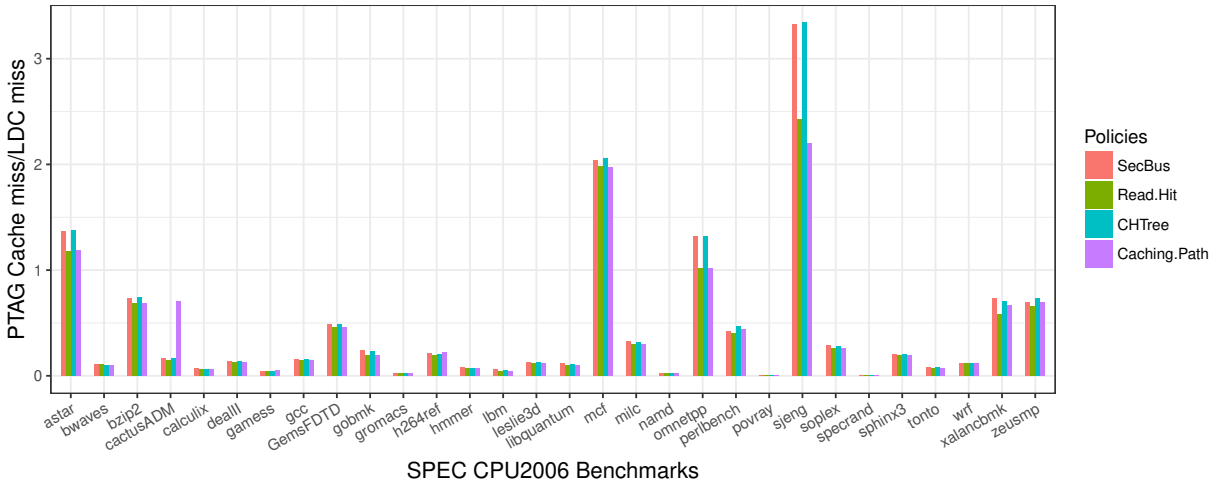


Figure C.2: PTAG Cache miss over the L1 data cache miss for the SPEC CPU2006 benchmarks.

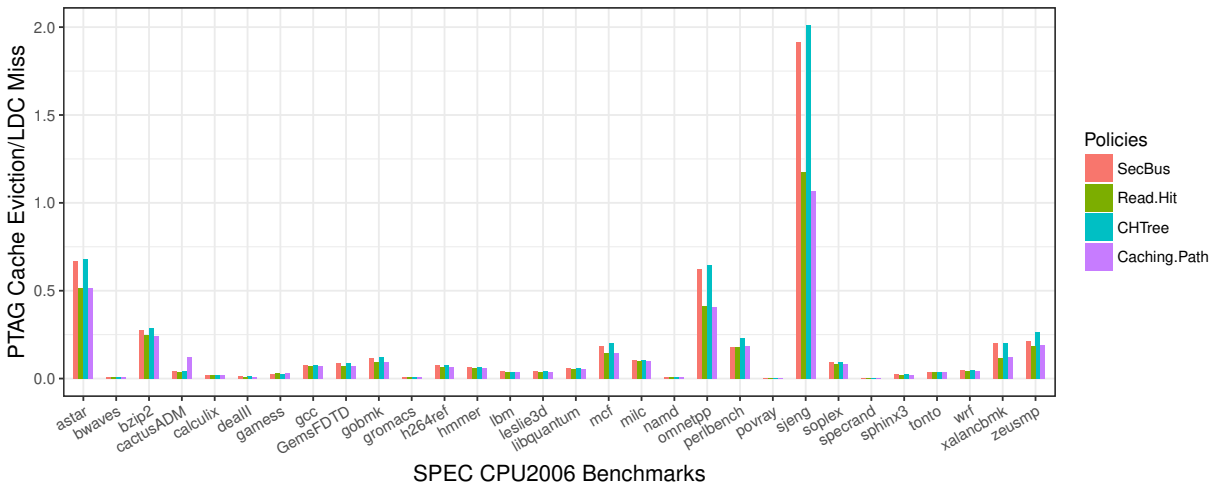


Figure C.3: PTAG Cache eviction over the L1 data cache miss for the SPEC CPU2006 benchmarks.

of misses and evictions for all policies are pretty similar in the Merkle Tree cache. Being *sjeng* the benchmark that presents the worst results, showing at least two misses in the Merkle Tree cache for every data cache miss and at least one Merkle Tree cache eviction for every data cache miss.

C.3 Estimating Performance

We used our number of hits, misses, evictions, and accesses from our cache simulations to estimate performance of CSHIA, in order to verify the impact of caching policies. To do that, we modeled our system like Figure C.4 describes. Observe in the figure that we are assuming a simultaneous process of verification/update of PTAGs with data memory block read/write. From Figure C.4 we obtained an analytical formulation, Equation C.1, that gives the ratio between main memory access cycles and PTAG verification/update cycles. This ratio if greater than 1 indicates that processors would be stalled after receiving data from main memory due to PTAG verification/update.

The statistical parameters of the equation are in Table C.1. Notice that each access A to PTAG Cache would generate a number of hits H and misses M. Each miss M or eviction E will generate $M+E$ accesses to PTAG Memory. Finally, each PTAG Cache miss M or write W needs a PTAG computation. To estimate the cycles in the Equation C.1, we used CACTI [1] to generate a 64-KB SRAM cache, a 256-MB DRAM main memory, and a 64-MB DRAM PTAG Memory[1] based on the technological parameters from [56]. We normalized all variables by the sequential access time of the cache (parameter c), which is lowest time value. The PTAG computation cycles s comes from the SipHash digest, which we set as 10 cycles.

Figure C.5 shows that for almost all benchmarks SecBus and CHTree have the best performance; the only exception is the benchmark *sjeng*. Moreover, SecBus and CHTree

$$C_{\text{ratio}} = \Big[A \cdot k + (H + M) \cdot c + (M + E) \cdot m + (M + W) \cdot s\Big] \cdot (A \cdot n)^{-1} \tag{C.1}$$

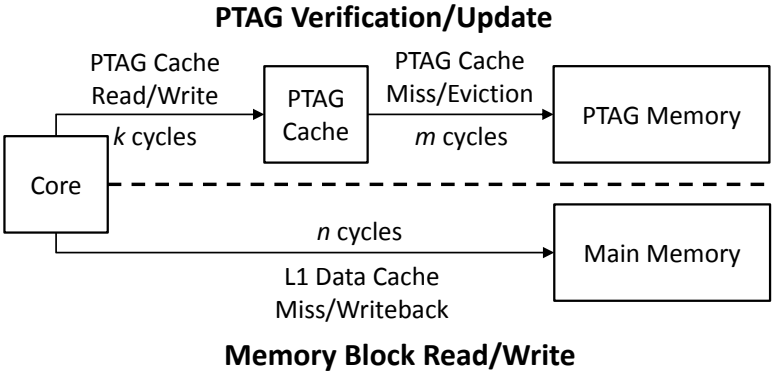


Figure C.4: A L1 data cache miss or write-back triggering a PTAG cache verification or update.

[1]It is the 25 % main memory overhead.

Table C.1: Variables of Equation C.1.

Variable	Description
A	Number of accesses to main memory or PTAG Cache due to FDC miss or eviction.
H	PTAG Cache Hits.
M	PTAG Cache Misses.
E	PTAG Cache Evictions.
W	PTAG Cache Updates.

Table C.2: Timing parameters for Equation C.1.

Parameter	Description	Cycles
c	PTAG Cache sequential access	1
k	PTAG Cache access	8
m	PTAG Memory access	44
n	Main Memory access	54
s	PTAG computation cycles	10

spent more cycles with PTAG verification/update than main memory access in only seven benchmarks. Thus, for most of the benchmarks in SPEC, we estimate that CSHIA would run transparently, as if no integrity verification was provided. Despite L1 Instruction Cache misses were not taken into account in this evaluation, they would not imply core stalls, since PTAG Cache is not used for instruction cache lines and main memory access spends as many cycles as it takes to compute a PTAG and access the PTAG Memory (from Table C.2, $m + s = n$).

However, the result in Figure C.5 goes in a different direction of what Figures C.3, C.2, and C.1 showed. That is because in those analyses only ratios were taken into account. For our analytical model, the number of memory operations (accesses, reads, and writes) is more important and critical than any ratio. Thus, SecBus and CHTree

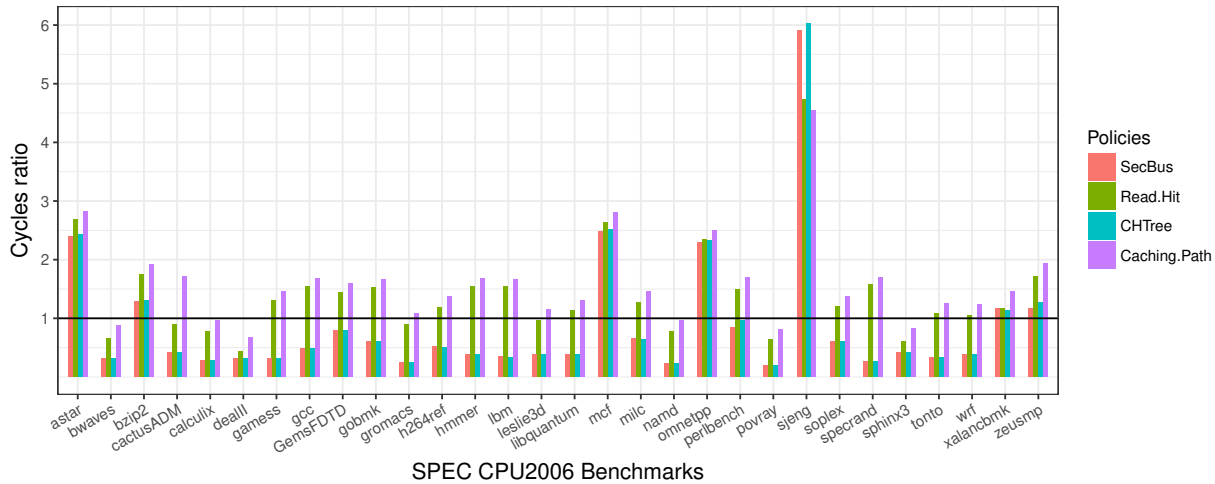


Figure C.5: PTAG verification cycles over the main memory access cycles for the SPEC CPU2006 benchmarks.

have an estimated performance better than Read.Hit and Caching.Path, despite the last two presented better ratios regarding misses and evictions. Overall, the analyses provided in this chapter allowed us to focus on the implementation of SecBus and CHTree in our CSHIA prototype.

Appendix D

Performing Attacks on CSHIA Prototype

This chapter shows the application of three replay attacks on the prototype of CSHIA. All attacks have the same purpose: reuse legit initial memory word values to prevent program output. The first attack is on CSHIA in bypass mode (no security). The second attack is on CSHIA running timestamp protection (*CSHIA-TS* instance). And, finally, the third attack is on CSHIA running Merkle Tree protection (*CSHIA-MT* instance). All these experiments were run in Altera's Development Kit DE2-115. The software used was GRMON v3.0.14 64-bit eval version and Altera's Quartus 13.1. DE2-115 was connect to a desktop computer running Fedora 23 by an Ethernet port. In addition, the target program is an adapted version of MiBench's `sha` in which we incorporated the small input file to the `.data` segment of the compiled program. A VHDL source code of CSHIA's prototype for DE2-115 will be made available in [24] together with the adapted benchmarks we used in this work.

D.1 Attacking CSHIA in Bypass Mode

Once we program CSHIA in the FPGA, one needs to set the prototype to bypass mode. That corresponds to put switch keys `SW0` and `SW1` to high, as Figure D.1 depicts. After that we run GRMON to connect to the development kit and then load `sha`'s binary in the kit's DRAM. Figure D.2 shows the command line executed in Fedora 23's Bash and Figure D.3 shows GRMON's terminal output after loading the binary.

Figure D.4 presents the range of memory addresses that will be the target of our attack. In Figure D.5, we establish a breakpoint at the address `0x40092160`, which allows us to observe bus transactions targeting that address, and we run `sha`. Then, we examine the content of the memory after the first break point. To attack, we write zero in two memory words located at `0x4009216C` and `0x4009217C`. As Figure D.6 shows, once we checked that that memory block content was changed, we delete the break point and finally continue to execute `sha`. Notice that the program finishes without printing its output. The correct and expected output for `sha`'s small input is presented in Figure D.7.

The attack we demonstrated happens in the `.data` segment of the program that starts

at `0x40045B00`, as one can see in Figure D.3. It can be classified as a replay attack since an attacker chooses old values that memory addresses `0x4009216C` and `0x4009217C` possessed at the beginning of the execution. Our CSHIA prototype currently covers the address range `0x40000000`–`0x40092FFF`. Therefore, this attack should not succeed if either of the countermeasures CSHIA provides were activated. In the following sections, we examine how an attacker proceeds to apply a replay attack against each countermeasure and we show that the prototype prevents he/she to succeed.

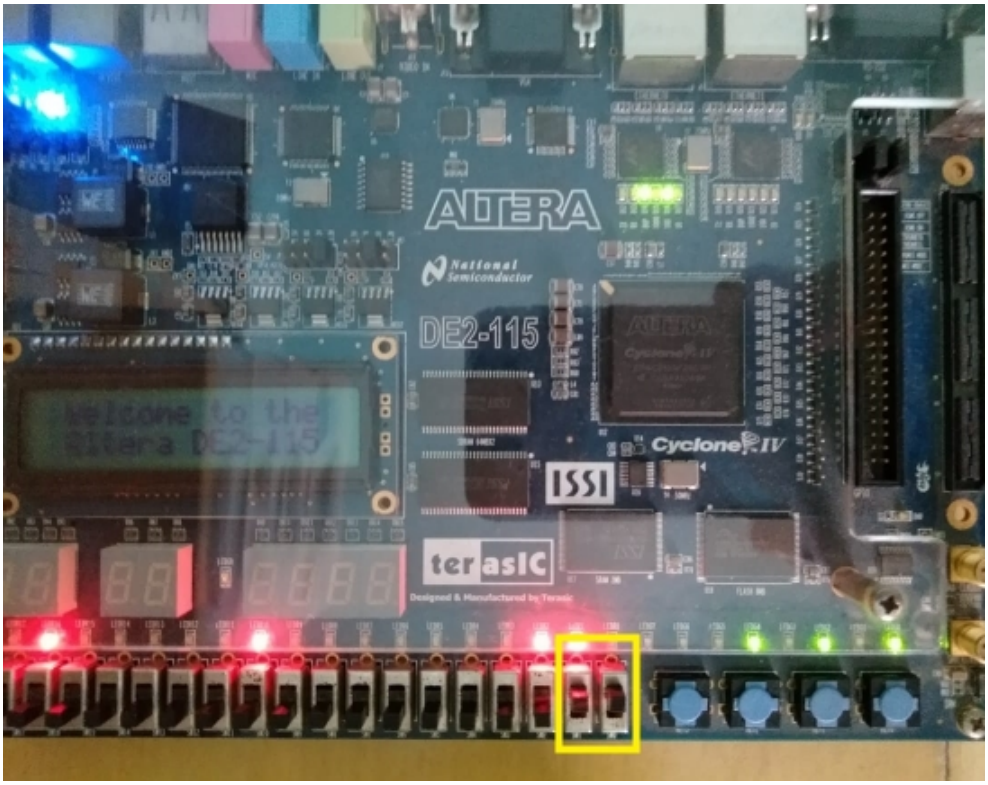


Figure D.1: Configuring CSHIA to bypass mode in Altera's FPGA DE2-115.

```
1 > linux/bin64/grmon -u -v -eth -dsudelay 250 -log sha.small.bypass.250
    delay.attack.txt"
```

Figure D.2: Command line to execute GRMON.

```
1 grmon3> load sha_small
2 40000000 .text                    0B                [>                ]   0%
3 40000000 .text                 57.0kB /  57.0kB     [===============>] 100%
4 40045B00 .data                    0B                [>                ]   0%
5 40045B00 .data                307.2kB / 307.2kB     [===============>] 100%
6 Total size: 364.28kB (25.95Mbit/s)
7 Entry point 0x40000000
8 MiBench/security/sha/sha_small loaded
9 1073741824
```

Figure D.3: Loading `sha` into the FPGA memory.

```
grmon3> mem 0x40092160
0x40092160  00000000  00000000  00000000  00000000    ................
0x40092170  00000000  00000000  00000000  00000000    ................
0x40092180  00000000  00000000  00000000  00000000    ................
0x40092190  00000000  00000000  00000000  00000000    ................
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure D.4: Initial memory content at `0x40092160`.

```
grmon3> bp bus 0x40092160
Bus watchpoint 1 at 0x40092160
1
grmon3> run
Unknown watchpoint hit
0x400028d4: c0262010  clr   [%i0 + 0x10]  <std+64>
SIGTRAP
grmon3> mem 0x40092160
0x40092160  00000000  00000000  00000000  000a0001    ................
0x40092170  00000000  00000000  00000000  40092160    ............@.!‘
0x40092180  400048fc  400048a4  4000484c  00000000    @.H.@.H.@.HL....
0x40092190  00000000  00000000  00000000  00000000    ................
0 0 0 655361 0 0 0 1074340192 1073760508 1073760420 1073760332 0 0 0 0 0
```

Figure D.5: Executing `sha` with a breakpoint at `0x40092160`. After reaching the breakpoint, we can visualize the memory block content at `0x40092160`.

```
grmon3> wmem 0x4009216C 0x0
grmon3> wmem 0x4009217C 0x0
grmon3> mem 0x40092160
0x40092160  00000000  00000000  00000000  00000000    ................
0x40092170  00000000  00000000  00000000  00000000    ................
0x40092180  400048fc  400048a4  4000484c  00000000    @.H.@.H.@.HL....
0x40092190  00000000  00000000  00000000  00000000    ................
0 0 0 0 0 0 0 0 1073760508 1073760420 1073760332 0 0 0 0 0
grmon3> bp delete
grmon3> cont

Program exited normally.
SIGTERM
```

Figure D.6: Forcing null words into the CSHIA memory, when it is in bypass mode. Then, after deleting the breakpoint, `sha` is continued and finishes running.

```
320c22e9 7b1ed440 77d2e55a bbe2481a 2b24a55b

Program exited normally.
SIGTERM
```

Figure D.7: The expected output from normally running `sha` with its small input file.

D.2 Attacking *CSHIA-TS*

The CSHIA Timestamp instance (*CSHIA-TS*) has internal timestamp and PTAG memories, as presented in "*CSHIA Implementation*" in Chapter 2. To instanciate *CSHIA-TS*, we need to program the FPGA through Quartus. After running the same commands in Figure D.2 and D.3, we need to press and release `KEY2` and then press and release `KEY3` in this specific order. Figure D.8 shows `KEY2` and `KEY3` highlighted by orange and lightblue frames, respectively. Notice that in Figure D.8 both `SW0` and `SW1` are down, which is highlighted by a yellow frame. `KEY2` is the CSHIA's Fuzzy Extractor enrollment. `KEY3` is a combination of the Fuzzy Extractor regeneration and PTAG Memory enrollment. Currently, our prototype does not have a reset to kick off the regenation only.

Once the steps aforementioned are concluded, we can examine the current status of both timestamp and PTAG memories using *In-System Memory Content Editor* of Quartus (see Figures D.9 and D.10). The timestamp (Figure D.9) and PTAG (Figure D.10) highlighted by yellow frames correspond to the memory block that starts at `0x40092160`. The next steps are those presented in Figure D.5, which are to establish the breakpoint and run the program until it stops at the breakpoint[1]. When that happens, the timestamp and PTAG memories have new values that are highlighted in Figures D.11 and D.12, which respectively show the timestamp and PTAG related to the memory block at `0x40092160`.

In our threat model, an attacker does not have access to the timestamp memory, which

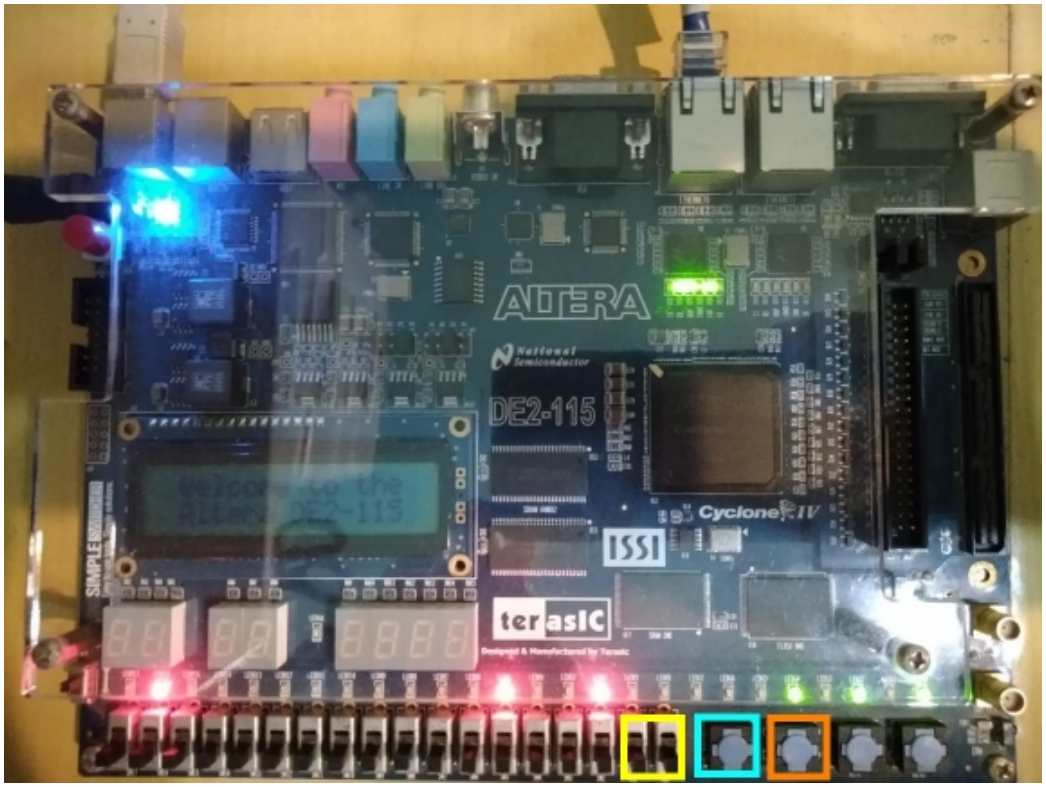


Figure D.8: Configuring CSHIA to timestamp mode in Altera's FPGA DE2-115.

[1]Sometimes the execution will stop multiple times before reaching the memory values exhibited in Figure D.5. In those cases, we continue execution until it reaches that status.

would be an on-chip memory, but he/she does have access to the memory of PTAGs (an off-chip memory). In this situation, the attacker proceeds writing old values in main memory, as Figure D.6 shows, and he/she also writes back the inicial value of the PTAG which corresponds to the memory block he/she has just changed (Figure D.13).

Finally, sure that theses modifications would allow he/she to suceed, the attacker proceeds with the execution of the program. However, differently from what happened in Section D.1, the system hangs up because tampered data will not reach the processor. As Figure D.14 presents, the execution of `sha` stalls in a call instruction of the `_vprintf_r` function. This corraborates that the purpose of the attack is output suppression. Hence, *CSHIA-TS* inhibits unauthentic behavior to happen. Notice that, if we manually modify the timestamp memory and reset the highlighted value in Figure D.11, the program finishes as shown in Figure D.6.

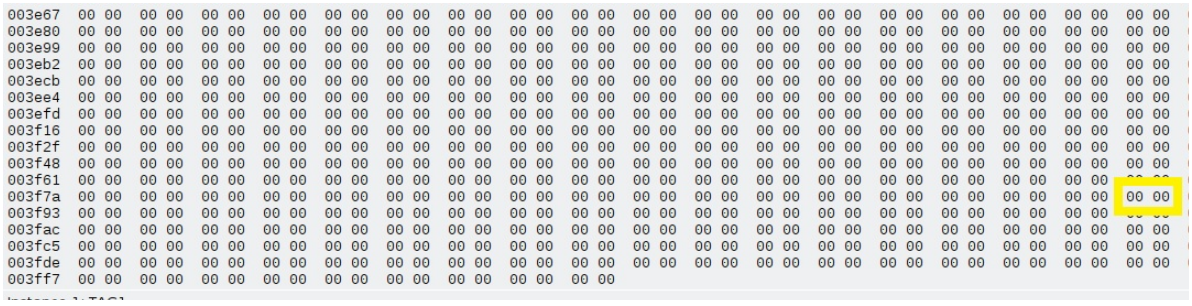


Figure D.9: Initial values of the timestamp memory before running `sha`.

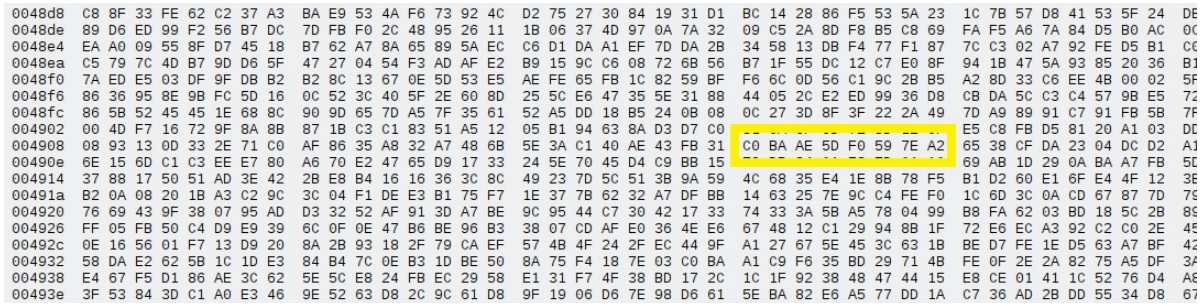


Figure D.10: Initial values of PTAG Memory before running `sha`.

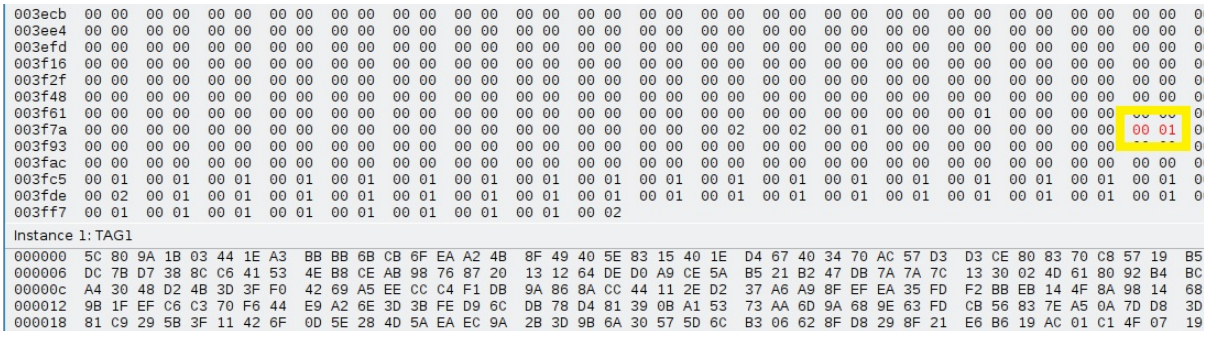


Figure D.11: Timestamp memory status at the breakpoint.

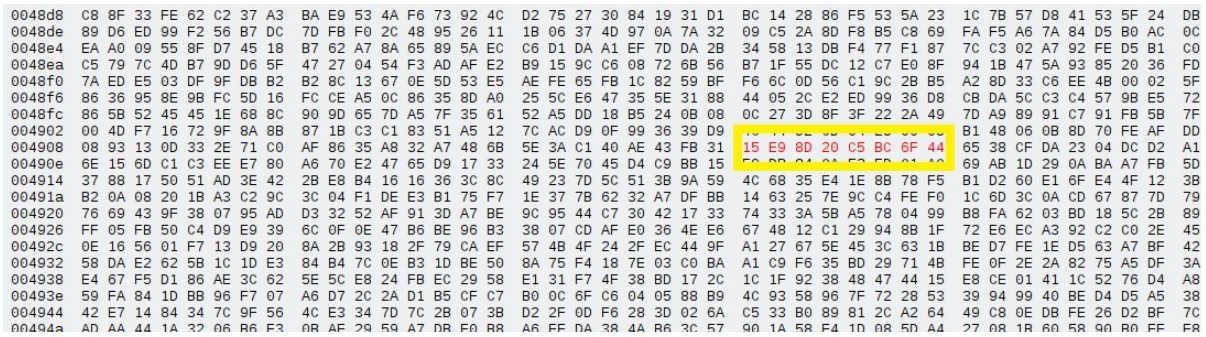


Figure D.12: PTAG Memory status at the breakpoint.

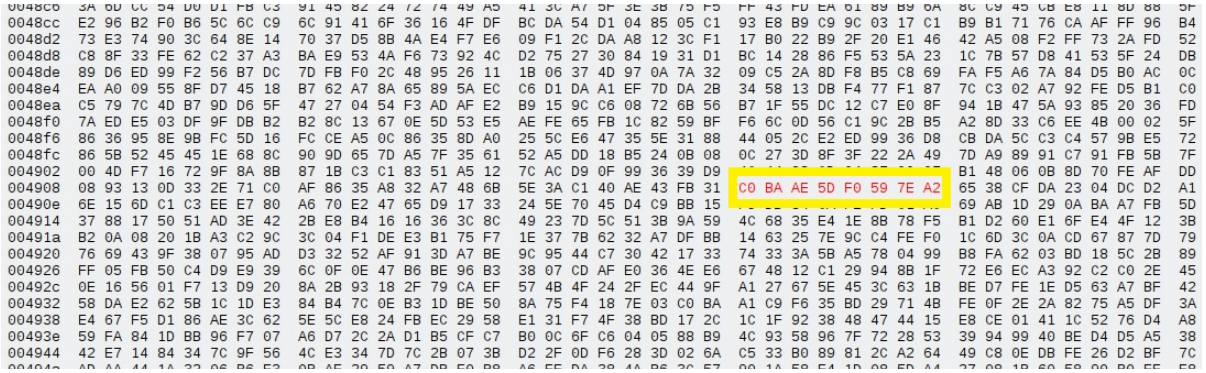


Figure D.13: Modifying PTAG Memory before continuing to execute `sha`.

```
1  grmon3> bp delete
2  grmon3> cont
3  Stopped (tt = 0x00, )
4  0x40005740: 4000069c  call  0x400071B0  <_vfprintf_r+2556>
5  SIGHUP
6  grmon3> mem 0x40092160
7  0x40092160  00000000  00000000  00000000  00000000     ................
8  0x40092170  00000000  00000000  00000000  00000000     ................
9  0x40092180  400048fc  400048a4  4000484c  40004830     @.H.@.H.@.HL@.H0
10 0x40092190  00000000  00000000  00000000  00000000     ................
11 0 0 0 0 0 0 0 0 1073760508 1073760420 1073760332 1073760304 0 0 0 0
```

Figure D.14: Forcing null words into the *CSHIA-TS* memory. Then, after deleting the breakpoint, `sha` is continued and stalls.

D.3 Attacking *CSHIA-MT*

Attacking the CSHIA Merkle Tree instance (*CSHIA-MT*) is significantly more complicated. First, an attacker has to roll back more than one PTAG in order to make the Merkle Tree consistent in memory. Second, tampering with one memory block has a high chance of producing an unsuccessful attack because in the tree structure ancestors have PTAGs that are related not only to the PTAG of the tampered memory block but also to other memory blocks. Third, PTAG Cache hinders an attack because it may happen that some PTAGs of the Merkle Tree, which an attacker needs to roll back, are in the cache, and thus he/she cannot modify their current value. Remembering that our threat model

considers that PTAG Cache lies in on-chip and so is inaccessible to attackers. Finally, the root PTAG is not reachable by attackers, thereby he/she will ultimately fail in any attempt of rolling back a previous memory state because they cannot change the root.

As presented in "*CSHIA Implementation*" in Chapter 2, we divided PTAG Memory into two: a PTAG Memory for PTAGs of code and data memory blocks; and a PTAG Memory for Merkle Tree PTAGs. For the sake of this attack, we reduced the PTAG Cache to 2 sets of 4 lines. That was needed to increase the number of PTAGs that are written back to PTAG Memory. Once CSHIA's is programmed in the FPGA through Quartus and the commands in Figure D.2 and D.3 are run, we need to press and release `KEY2` and then press and release `KEY3` in this specific order. After that, we set `SW17` to high. Figure D.15 shows `KEY2` and `KEY3` highlighted by orange and lightblue frames, respectively, besides highlighting `SW0`, `SW1`, `SW17` positions in yellow frames.

Figures D.16, D.17, D.18, D.19, and D.20 exhibit the initial values of the PTAGs related to the memory block that starts at `0x40092160`. Figure D.16 refers to the memory block PTAG and the other figures exhibits PTAGs of the Merkle Tree. The only PTAG that is not visible in those figures is the root. After setting the breakpoint and running `sha` (Figure D.5), we reach the breakpoint[2]. Figures D.21, D.22, D.23, D.24, and D.25 present the PTAG Memory status at the breakpoint. We can notice that if one rolls back the PTAGs highlighted in Figure D.25, he/she will also need to roll back all values in red in Figures D.24, D.23, and D.22 because they are used during the computation of the PTAGs displayed by Figure D.25. Consequently, the attacker will need to roll back all memory blocks related to the PTAGs that are in red in Figure D.21. We tried to proceed this attack by rolling back all PTAGs and only modifying the memory word at `0x40092160` (Figure D.6). But that did not work, as expected, and resulted in the processor hanging up in some non-specific part of the code, as Figure D.26 shows. To make this attack fail at the verification of the root PTAG, we would need to restore all memory blocks to their initial value beside rolling back all PTAGs. Since that would require a huge log to document each step, we chose not to display it.

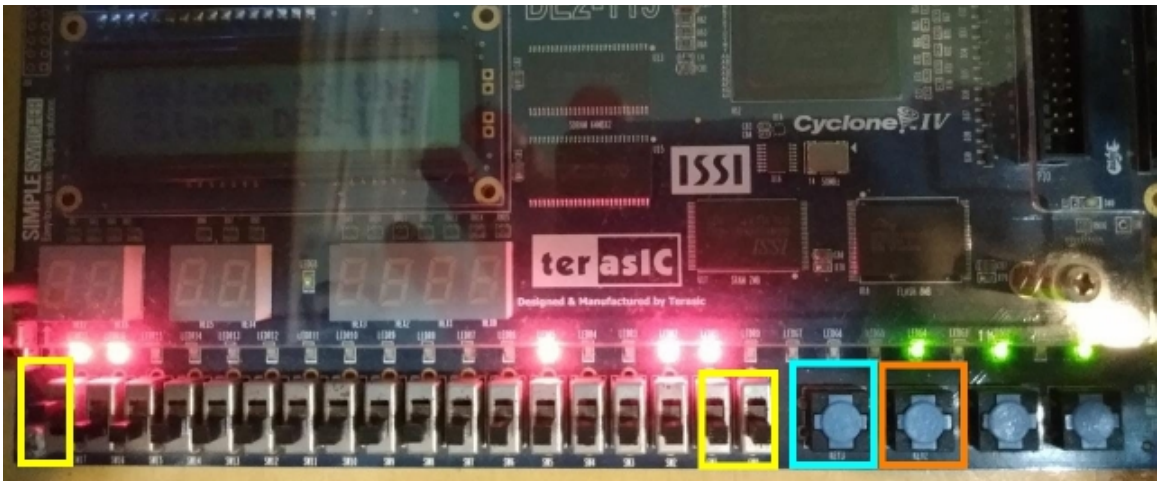


Figure D.15: Configuring CSHIA to Merkle Tree mode in Altera's FPGA DE2-115.

[2]Again, achieving those values in Figure D.5 can take multiple stops at the breakpoint.

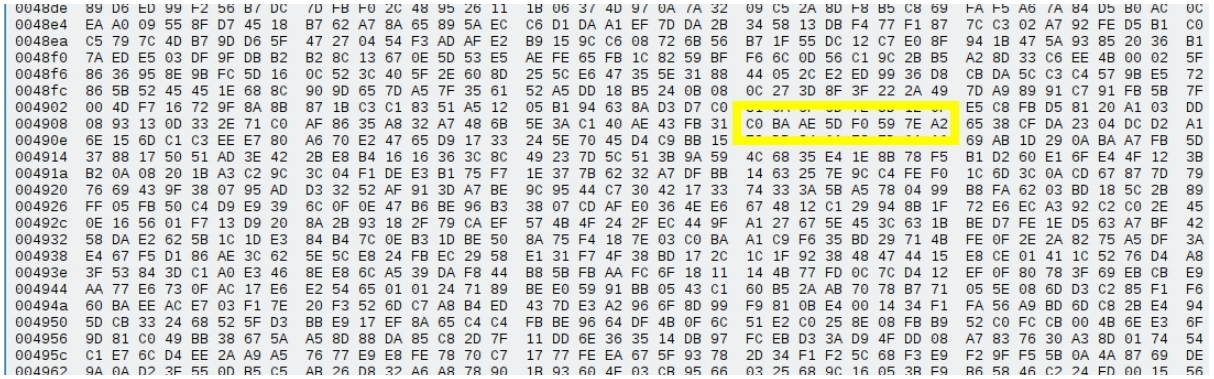


Figure D.16: Initial values of PTAGs of data memory blocks.

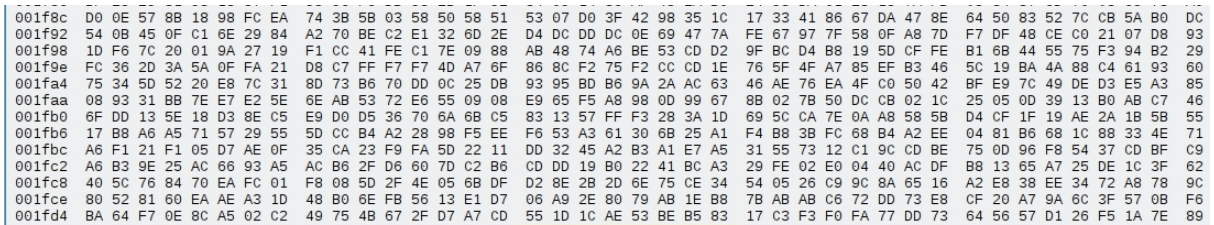


Figure D.17: First level of the Merkle Tree in PTAG Memory before running `sha`.

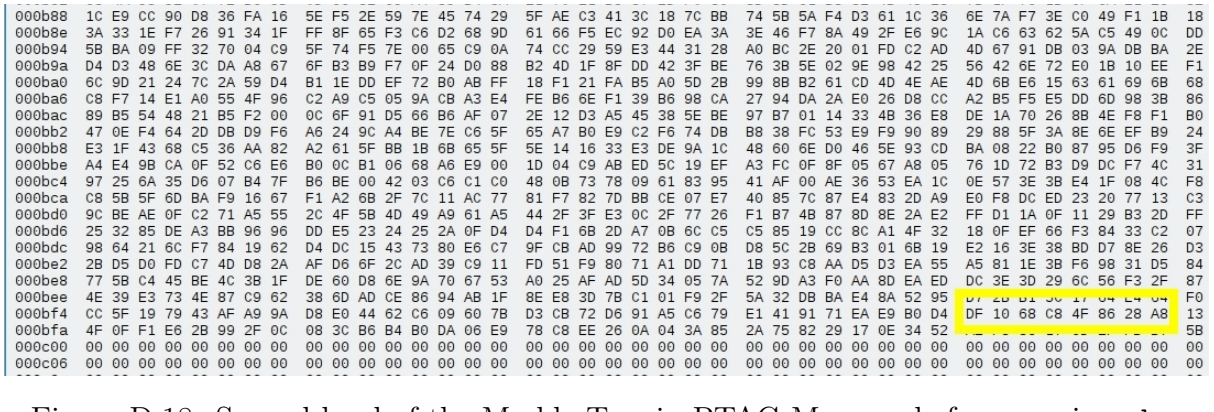


Figure D.18: Second level of the Merkle Tree in PTAG Memory before running `sha`.

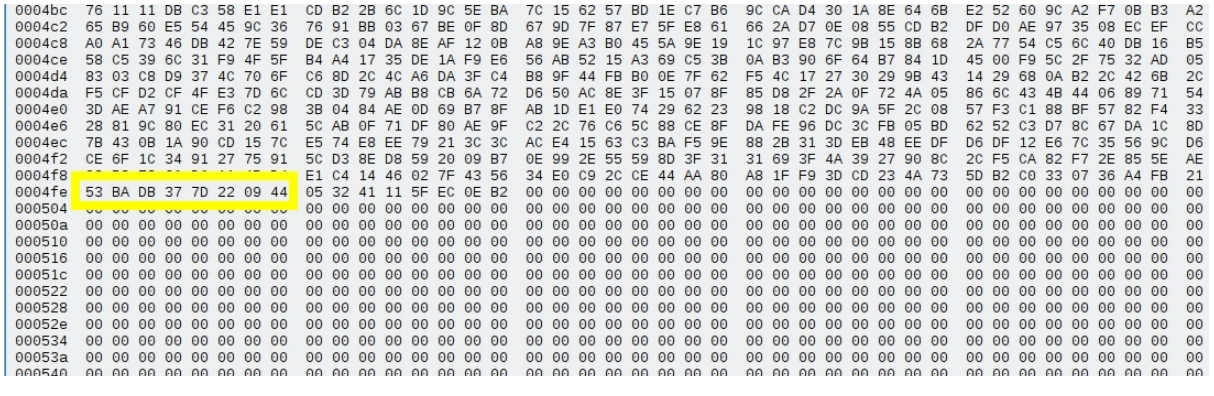


Figure D.19: Third level of the Merkle Tree in PTAG Memory before running `sha`.

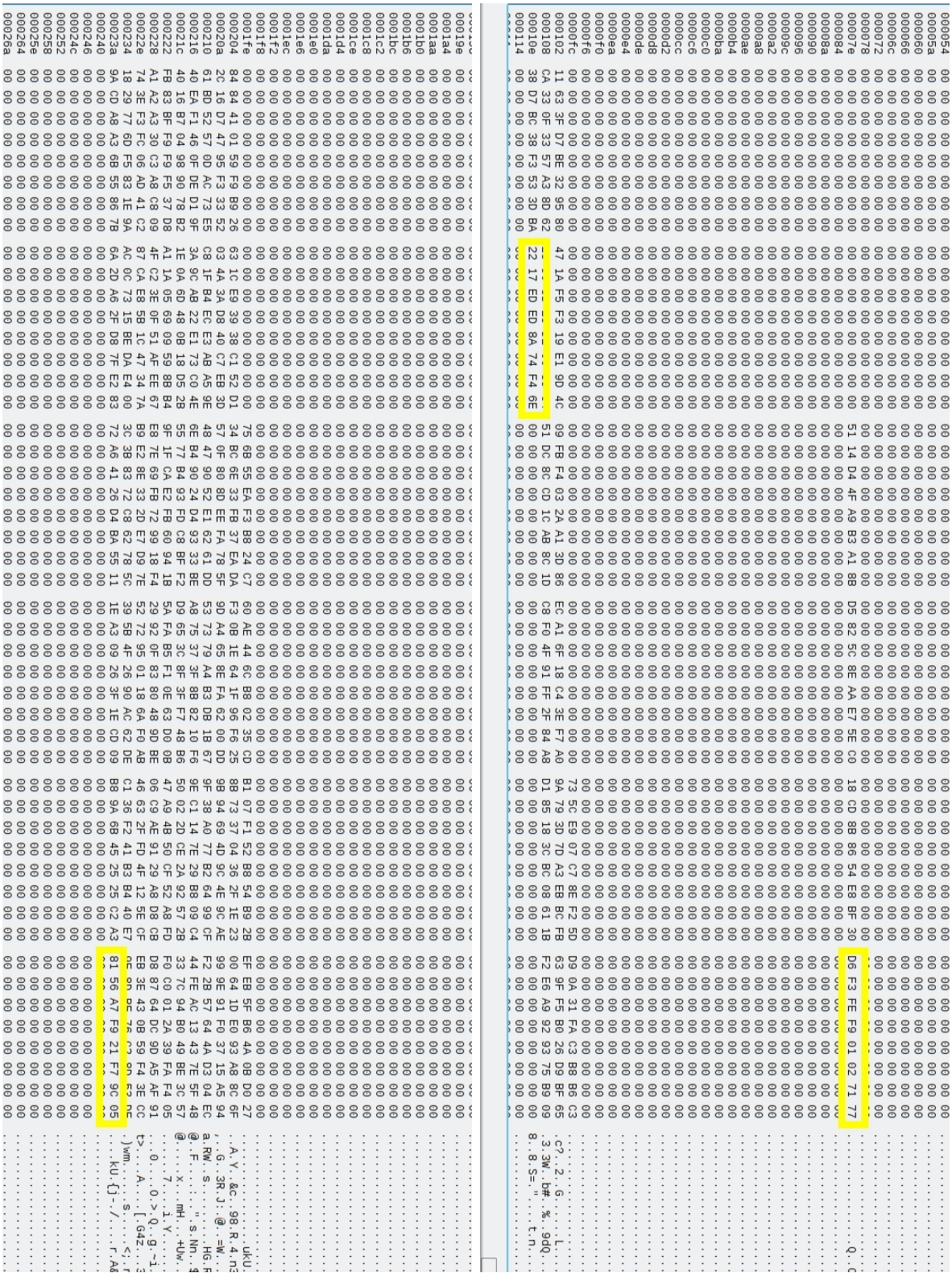


Figure D.20: Fourth, fifth, and sixth levels of the Merkle Tree in PTAG Memory before running `sha`.

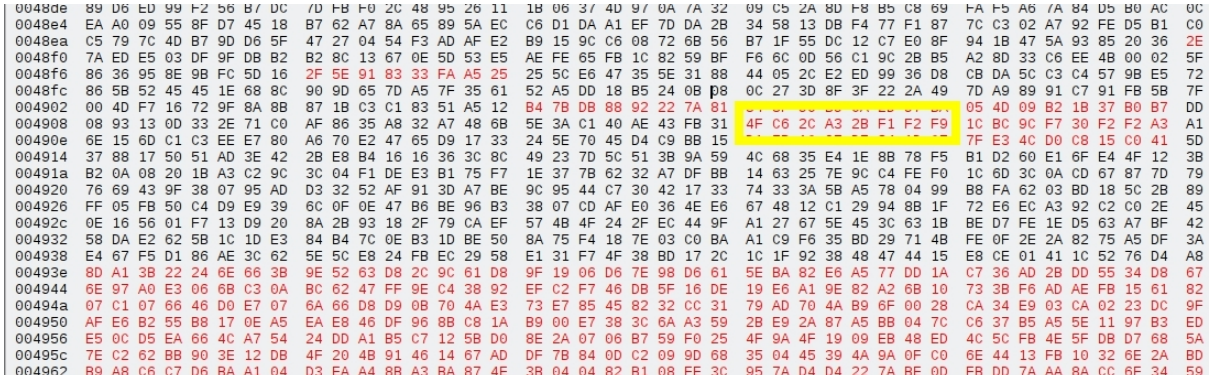


Figure D.21: Values of PTAGs of data memory block at the breakpoint.

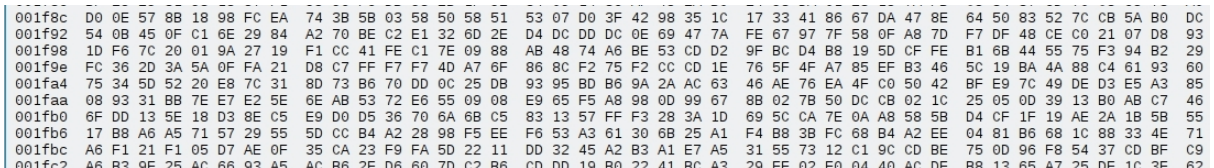


Figure D.22: First level of the Merkle Tree in PTAG Memory at the breakpoint.

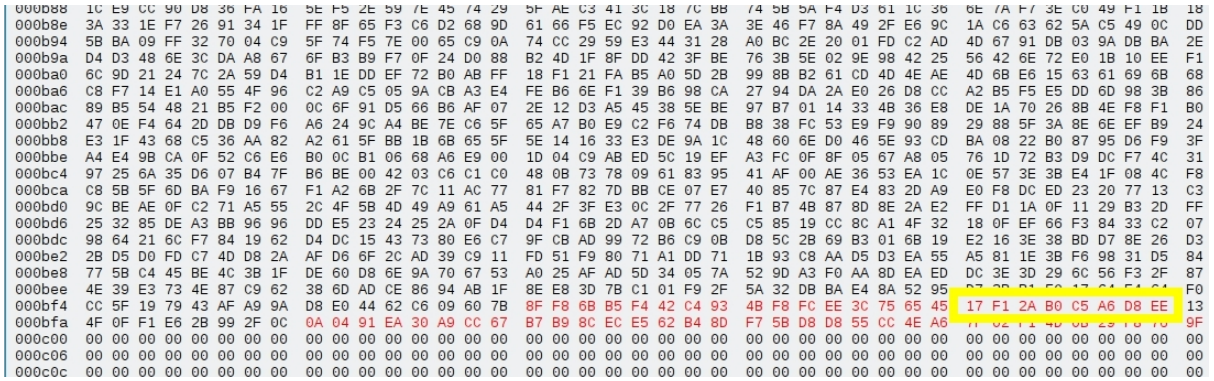


Figure D.23: Second level of the Merkle Tree in PTAG Memory at the breakpoint.

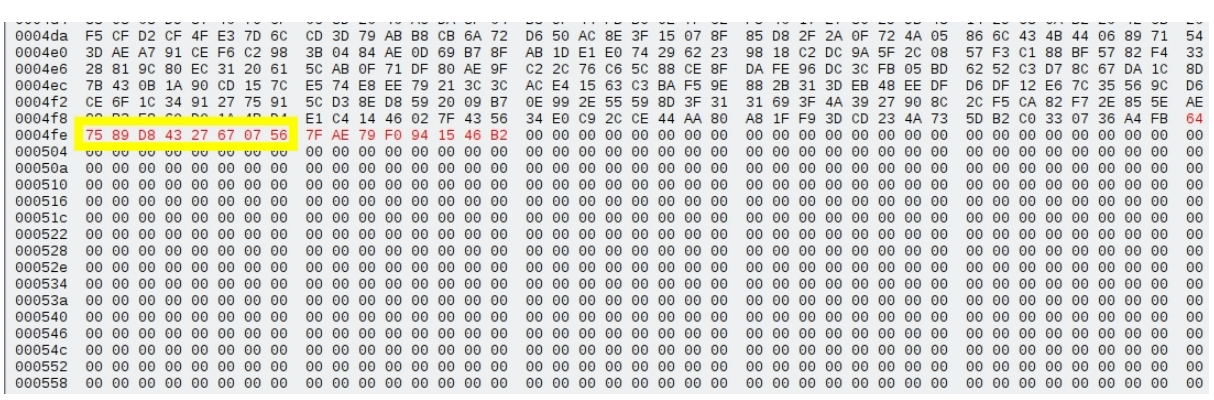


Figure D.24: Third level of the Merkle Tree in PTAG Memory at the breakpoint.

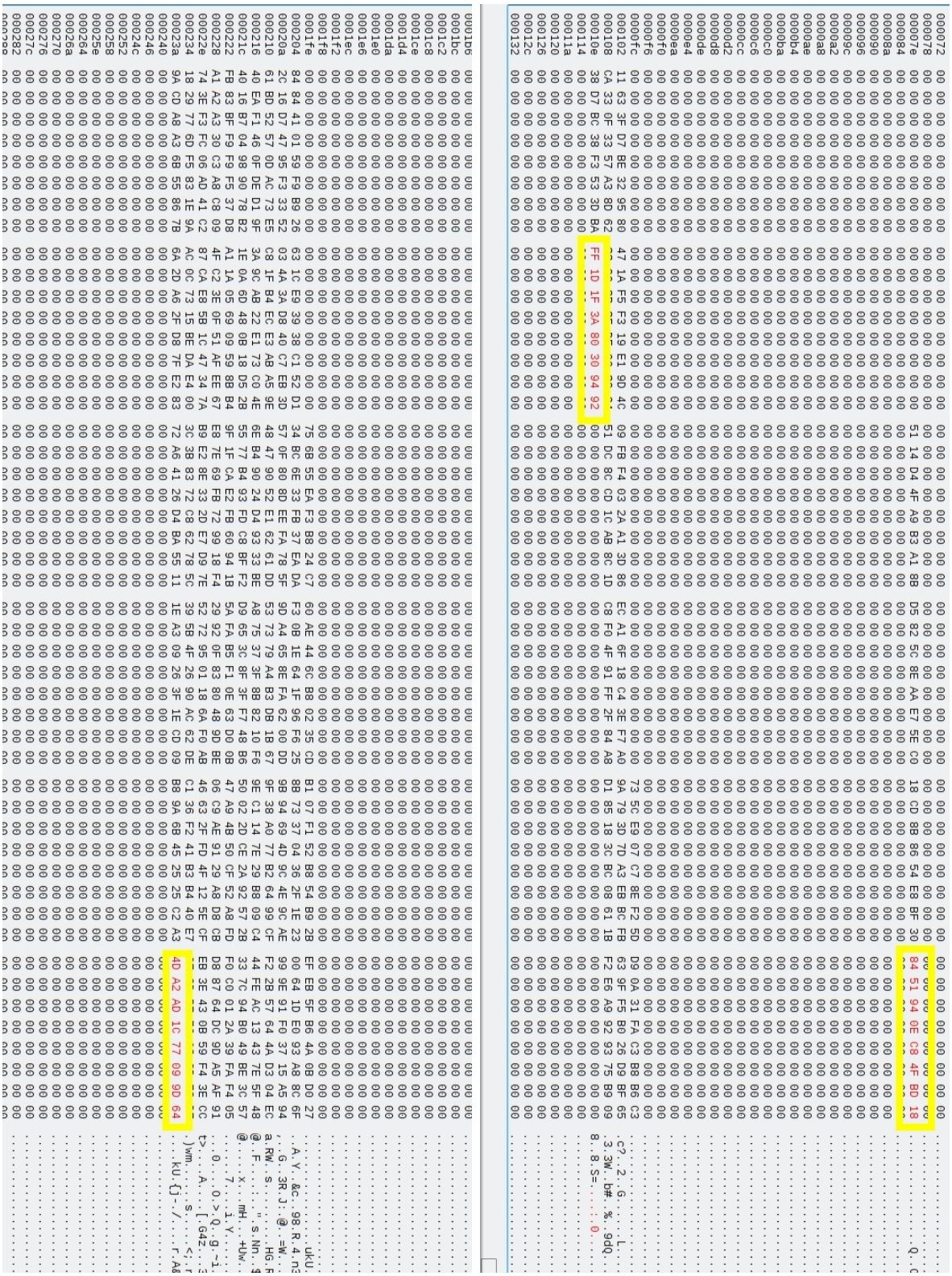


Figure D.25: Fourth, fifth, and sixth levels of the Merkle Tree in PTAG Memory at the breakpoint.

```
grmon3> mem 0x40092160
0x40092160  00000000  00000000  00000000  00000000    ................
0x40092170  00000000  00000000  00000000  00000000    ................
0x40092180  400048fc  400048a4  4000484c  40004830    @.H.@.H.@.HL@.H0
0x40092190  00000000  00000000  00000000  00000000    ................
0 0 0 0 0 0 0 0 1073760508 1073760420 1073760332 1073760304 0 0 0 0
grmon3> cont
Stopped (tt = 0x00, )
0x4000c97c: 81c7e008  ret    <___st_pthread_mutex_unlock+36>
SIGHUP
```

Figure D.26: Forcing null words into the *CSHIA-MT* memory. Then, after deleting the breakpoint, `sha` is continued and stalls.

D.4 Summary

This section presented some attacking scenarios to explore CSHIA's countermeasures against replay attacks. Our goal was to elucidate the steps one would need to take in order to apply these attacks in the CSHIA prototype. In addition, we would like to make clear that these attacks do not prove that the current CSHIA prototype is unbreakable or may lack vulnerabilities. Those who find loopholes and achieve sucessful attacks are very welcomed to share with us, helping us to improve CSHIA security.