Universidade Estadual de Campinas
Instituto de Computação

Augusto Fernandes Ribas Queiróz

Implementation of a secure code execution architecture using PUFs

Implementação de uma arquitetura para execução segura de código utilizando PUFs

CAMPINAS
2019

Augusto Fernandes Ribas Queiróz

Implementation of a secure code execution architecture using PUFs

Implementação de uma arquitetura para execução segura de código utilizando PUFs

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

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

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araújo

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

CAMPINAS
2019

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

Q32i

Queiroz, Augusto Fernandes Ribas, 1989-
Implementation of a secure code execution architecture using PUFs / Augusto Fernandes Ribas Queiroz. – Campinas, SP : [s.n.], 2019.

Orientador: Guido Costa Souza de Araújo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. FPGA (Field Programmable Gate Array). 2. Computadores - Medidas de segurança. 3. Hardware - Arquitetura. I. Araújo, Guido Costa Souza de, 1962-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Implementação de uma arquitetura para execução segura de código utilizando PUFs
Palavras-chave em inglês:
Field programmable gate arrays
Computer security
Hardware - Architecture
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Bruno de Carvalho Albertini
Sandro Rigo
Data de defesa: 19-07-2019
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-9137-796X
- Currículo Lattes do autor: http://lattes.cnpq.br/6140950733861869

Universidade Estadual de Campinas
Instituto de Computação

Augusto Fernandes Ribas Queiróz

Implementation of a secure code execution architecture using PUFs

Implementação de uma arquitetura para execução segura de código utilizando PUFs

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo
  Universidade Estadual de Campinas
- Prof. Dr. Sandro Rigo
  Universidade Estadual de Campinas
- Prof. Dr. Bruno de Carvalho Albertini
  Universidade de São Paulo

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, 19 de julho de 2019

Dedicatória

Concluir um mestrado não é uma tarefa fácil, formalmente exige o cumprimento de créditos a realização da pesquisa e apresentação dos resultados, mas é muito mais do que isso, é necessária tranquilidade e um ambiente seguro e confortável para que todas essas tarefas possam ser realizadas. Dedico essa dissertação a todos que de alguma forma colaboraram, ao meu falecido pai Alexandre Queiroz quem me deu toda a base para que pudesse estar aqui hoje, à minha mãe e irmão, Célia e Alexandre que sempre me deram apoio e se fizeram presentes. Dedico também a todos os professores do ensino básico em diversas escolas, graduação na UFMS e mestrado na UNICAMP, todos de altíssimo nível e que dedicaram seu tempo para que hoje eu estivesse capacitado para concluir este trabalho. Em especial agradeço à aqueles que participaram ativamente deste trabalho e o tornaram possível, meus orientadores Professores Doutores Guido Araújo e Mario Côrtes, ao professor Dr. Diego Aranha e ao colega Dr. Caio Hoffman.

Resumo

As técnicas padrões de design para proteger a execução de código são baseadas em mecanismos criptográficos bem conhecidos e em recursos de (micro) arquitetura para codificar transações de barramento ou isolar o código seguro em plataformas confiáveis, entre outras. Embora essas técnicas geralmente forneçam níveis adequados de segurança, a maioria delas é ineficiente, consideravelmente impacta o projeto da (micro) arquitetura, requer mudanças extensas na cadeia de ferramentas de programação ou é tão complicada que pode criar brechas de segurança inesperadas. Com o objetivo de resolver esses problemas de segurança na execução de códigos, a Segurança de Computadores por Autenticação Intrínseca ao Hardware (CSHIA) foi proposta para autenticar todos os blocos de uma memória externa usando uma chave exclusiva extraída de Funções Físicas não Clonáveis (PUFs). Com base na implementação em FPGA do processador Leon3 da Gaisler, este trabalho apresenta uma prova de conceito do CSHIA, apresentando os detalhes e uma descrição detalhada da implementação do hardware, os compromissos do design e a integração entre a arquitetura e um processador real. Mostramos os recursos do FPGA, uma avaliação de desempenho com benchmarks padrão da indústria e estimativas de energia e área. A versão final do CSHIA forneceu um design robusto e melhoria de segurança para o processador selecionado, à custa de $2,76\%$ a $5,77\%$ de sobrecarga de desempenho, dependendo da solução adotada com um aumento da área lógica de 34% para a configuração selecionada. A implementação final do CSHIA tornou-se uma plataforma altamente configurável que oferece várias opções de design e recursos de segurança a um usuário final, onde este trabalho contribuiu para fornecer um chassi que pode ser usado por qualquer sistema AMBA2.

Abstract

Standard design techniques to secure code execution are based on well-known cryptographic mechanisms and (micro) architecture features to encode bus transactions, or isolate secure code into trusted platforms, among others. Although such techniques usually provide proper levels of security, most of them are either inefficient, considerably impact processor (micro) architecture design, require extensive changes in the programming toolchain, or are so complicated that may create unexpected security loopholes. Aiming to address this security issues in code execution the Computer Security by Hardware-Intrinsic Authentication (CSHIA) was proposed to provide authenticity by authenticating all memory blocks of an external memory using a unique key extracted from Physical Unclonable Functions (PUFs). Based on Gaisler's Leon3 FPGA implementation, this work presents a proof-of-concept of CSHIA, presenting the details and an in-depth description of the hardware implementation, the design tradeoffs, and the integration between the architecture and a real processor. We show the FPGA resources, a performance evaluation with industry standard benchmarks and power and area estimations. The final CSHIA version provided a robust design and security improvement to the selected processor at the expense of 2.76% to 5.77% of performance overhead depending on the solution adopted with logic area overhead of 34% for the selected configuration. The final CSHIA implementation became a highly configurable platform that offers several design choices and security features to an end user, where this work contributed to provide a chassis that can be used by any AMBA2 system.

List of Figures

2.1 One arbiter PUF that receives a challenge X and produces the output Y. . . 16
2.2 A simplistic example of a SRAM PUF , showing the SRAM cell logic circuit inside an SRAM block. 17
2.3 Houde *et al.* model for evaluation of the goals of prototypes. 19

4.1 The CSHIA architecture flow since the proposal in [19] until the final evaluation. 23
4.2 The CSHIA basic architecture. 24
4.3 The CSHIA expanded architecture. 25
4.4 Typical AMBA2 system , with a CPU , DMA and other low bandwidth peripherals. 26
4.5 Overview of the AMBA2 Organization, with the arbiters masters and slaves distribution. 27
4.6 AHB basic transfer with one cycle for address and control and one or many for data. 28
4.7 AHB master interface, with control and data signals. The HBUSSREQ and HGRANT signals will be used take control over the bus. 28
4.8 AHB slave interface, where the HSEL signal will indicate when to start transfers. 29
4.9 AHB arbiter interface with one HGRANT and one HBUSSREQ for each master. 29
4.10 AHB Decoder interface 30
4.11 Bus Handler interface 32
4.12 Bus Handler state machine. The blue text indicate the condition necessary for the to transition to happen, the red text indicates the BUS-HDLR actions. 33
4.13 Requesting grant with no wait states 34
4.14 Requesting grant with wait states 34
4.15 Incremental burst transfers with halfword and word. 35
4.16 Incremental write burst of words. 36
4.17 Security engine interface. 36
4.18 SEC-ENG state machine. The blue text indicate the condition necessary for the to transition to happen, the red text indicates the SEC-ENG actions. 37
4.19 PTAG read and validate on SEC-ENG. 39
4.20 PTAG read with Merkle Tree 39
4.21 PTAG write on SEC-ENG. 40
4.22 PTAG write on sec engine with Merkle Tree. 40
4.23 Fuzzy Extractor actions during the enrollment and recovery procedure. . . 42
4.24 Key generation on CSHIA. 43

5.1 CSHIAintegrated with the DE2-115 kit ,with Leon3 , DSU and all presented security components. 44
5.2 GRMON Interface 47

List of Tables

3.1 Summary of Related Works in comparison with CSHIA. 22

4.1 Description of the BUS-HDLR ports 32
4.2 Description of the SEC-ENG ports. 37

5.1 CSHIA FPGA implementation configuration. 45

6.1 Coverage of data segment in benchmarks. 50
6.2 Performance overhead in % of the evaluated CSHIA instances in comparison of running times in *Leon3 Baseline*. 51
6.3 Area and power for CSHIA implementation without considering instruction and data cache memories of the processor. 52

A.1 **ptag_sec_req_type** content description 60
A.2 **ptag_sec_val_type** content description 60
A.3 **ptag_mreq_type** content description 60
A.4 **ptag_mresp_type** content description 61
A.5 **ahb_mst_in_type** content description 61
A.6 **ahb_mst_out_type** content description 61

Contents

1 Introduction **13**

1.1 Contributions . . . 14

1.2 Organization of the dissertation . . . 14

2 Fundamental concepts **15**

2.1 Physical Unclonable Functions - PUFs . . . 15

2.1.1 PUF Types . . . 15

2.1.2 PUF as a Cryptographic Key Generator . . . 16

2.2 Security Properties . . . 17

2.2.1 Authenticity . . . 17

2.2.2 Integrity . . . 18

2.2.3 Secrecy . . . 18

2.3 Prototypes . . . 18

3 Related Work **20**

4 CSHIA Architecture **23**

4.1 Overview . . . 24

4.2 AMBA2 . . . 25

4.2.1 AMBA2 AHB operation . . . 26

4.2.2 AMBA2 Components . . . 28

4.3 CSHIA Components . . . 31

4.3.1 Bus Handler (BUS-HDLR) . . . 31

4.3.2 Security Engine . . . 36

4.3.3 PTAG Memory Management Unit (PMMU) . . . 40

4.4 Operation Modes . . . 40

4.4.1 Enrollment Phase . . . 41

4.4.2 Runtime Phase . . . 42

5 CSHIA Prototype **44**

5.1 Hardware setup . . . 44

5.2 GAISLER Tools . . . 46

5.2.1 Overview . . . 46

5.2.2 Debug . . . 47

6 CSHIA Evaluation **49**

6.1 Evaluation Goals . . . 49

6.2 Experimental Setup . . . 49

6.3 Performance Analysis . . . 50

6.4 Area and Power Estimates 51

7 Conclusion **53**

Bibliography **55**

A Type Description **60**

Chapter 1

Introduction

The demand for code/data integrity and authenticity has steadily increased. The broad spectrum of known attacks currently poses a threat to a variety of embedded systems that need constant protection against tampering. An excellent example of such systems are the ones that are equipped with large external non-volatile memories to store software and data, like voting machines, smart metering devices and employee attendance control systems. These systems need to provide integrity and authenticity, but usually not secrecy or confidentiality, in order to be easily audited by governmental authorities and independent experts.

Due to the stringent nature in available resources of embedded systems, software solutions for code and data integrity are not efficient due to their impact on the performance and power consumption of the system. Besides, software authenticity could involve a third-party certification authority, considerably increasing the complexity of the final poduct, thus making hardware a potentially useful way to solve such a problem. A myriad of hardware solutions for code and data authenticity and integrity have been proposed in the literature ([11,21,36,40]), however, some of these, target high-end embedded systems or more robust configurations.

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 [19]. 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 the embedded system architectures while providing complete software compatibility, it also uses a separate bus for the tag memory, which gives designers the freedom to match timing requirements so as to hide verification overhead.

The CSHIA architecture original proposal had security and viability evaluation, but how can one implement such a system using industry standard tools and IPs? This ques-

tion drove this work where we show the implementation details, evaluate the results with well-known benchmarks showing the performance overheads, and synthesize it presenting the reports of the final implementation.

1.1 Contributions

The main contributions to this work are the following:

(a) it provides the first hardware implementation of the CSHIA architecture ,using LEON3 processor;

(b) it analyses the trade-offs of the resulting architecture, performance in benchmarks and provide area and power estimations.

1.2 Organization of the dissertation

This work is organized as follows, Chapter 2 introduces the necessary concepts needed for this work. A review of the related work is presented in the chapter 3. The CSHIA architecture implementation is described in Chapter 4. Chapter 5 details the prototype and all implementation requirements. The evaluation of the prototype is presented in Chapter 6 ans Chapter 7 concludes this work.

Chapter 2

Fundamental concepts

To better understand this work, some basic concepts need to be clarified. It is essential to understand the role of each component, so the goal of this chapter is to understand how a physical function can help us achieve a robust architecture. Using a PUF as a mean to achieve physical security objectives that are secure key generation and storage, and then using this key together with cryptographic primitives to achieve information security objectives that are data integrity and authentication.

This chapter will present the PUFs in Section 2.1, also show basic examples to provide a better understanding of these functions and how we can use it to generate cryptographic keys. In Section 2.2, a hardware approach of how to achieve information security objectives are introduced, and finally in Section 2.3, what is a prototype and how we can model it to answer our design questions.

2.1 Physical Unclonable Functions - PUFs

It is well known that every time an Integrated circuit(IC) is fabricated there are small random process variations, imperfections that make every path in a design unique, for a design where the goal is to have always the same behavior for all ICs, these imperfections are measured and errors are avoided by creating timing constraints that will guide the Electronic Design and Automation(EDA) tools to generate error-free paths and deliver a secure robust design. PUFs are physical functions that instead of avoiding, take advantage of this inherent imperfections to mimic random functions. Their inputs, called challenges, and outputs, called responses, are designed to have a unique relationship for every PUF instance. It is possible to explore IC process variations in several ways [26] in the next sessions we describe possible implementations

2.1.1 PUF Types

It is possible to explore IC process variations in several ways, a great and simple example to understand the behavior of these functions is an arbiter PUF, pictured in Figure 2.1. This PUF is composed by a set of crossbar switches and one arbiter, the flip-flop. The idea behind this construction is to build two paths with the same layout length and compute the relative delay between these two paths given a challenge X of size N. For every switch

n, if $X[i]$ is one both paths will pass through the arbiter, if its zero the inputs will be switched in the output changing the delay of booth paths. To evaluate the output(Y), the challenge X and a rising signal are provided to both paths, Y is one if the signal to the latch data input (D) is faster, and zero otherwise. To get a response of size Z, this PUF construction can be instantiated Z times.

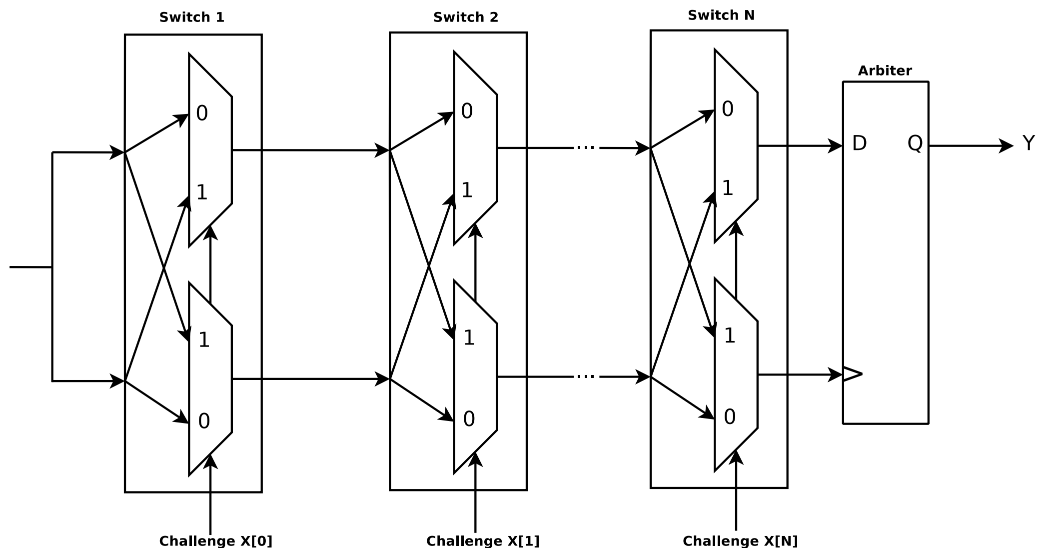


Figure 2.1: One arbiter PUF that receives a challenge X and produces the output Y.

Another PUF type is the Static Random-Access Memory (SRAM) PUF [25], in this model, each SRAM cell within an SRAM block can be considered a PUF. The typical cell implementation, shown in Figure 2.2 is built from two cross-coupled inverters at its core,in a logic sense, this circuit has two stable values, and by residing in one of both states the cell stores one binary digit. The operation principle of an SRAM PUF is based on the transient behavior of an SRAM cell when it is powered up, i.e., when its supply voltage Vdd comes up. The circuit will evolve to one of its operating points, but it is not immediately clear to which one. In Figure 2.2 the inverters I_1 and I_2 have its drive strength determined by the process variations when this memory was manufactured, these inverters will compete to achieve stability. When one of the inverters is significantly stronger than the other one, the preferred initial operating point will be a stable state, and the preference will be very distinct, i.e., such a cell will always power-up in the same stable state, but which state this is ('0' or '1'), is randomly determined for every cell.

So in a PUF that uses an SRAM, the challenge is the row and column addresses, the output is the state of the cell after power up.

2.1.2 PUF as a Cryptographic Key Generator

When a system needs a key in hardware for any purpose, such as encryption, authentication, or any other application, this key needs to be generated and stored in hardware [33]. The main advantage of using PUFs as key generators is that they can produce keys at running time, this way, on-chip memories are not needed for key storage. Another benefit is that they are unclonable, meaning that even the manufacturer itself cannot produce two PUF instances that will have the same set of Challenge-Response Pairs (CRPs) [16].

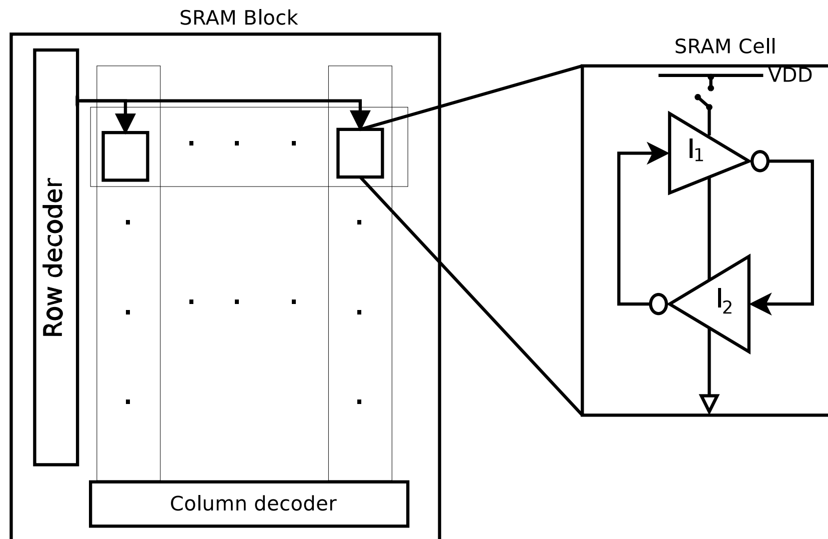


Figure 2.2: A simplistic example of a SRAM PUF , showing the SRAM cell logic circuit inside an SRAM block.

2.2 Security Properties

All information held and processed by an entity is subjected to threats of attack, error, and several other vulnerabilities, to protect information assets one need to define, achieve, maintain, and improve information security. In order to build a robust system, a designer has at his disposal mechanisms that implements three security properties: authenticity which is the property that an entity is what it claims to be, integrity that is accuracy and completeness property, and secrecy that is the property of hiding information. 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 sections, we discuss hardware implementation of those security features.

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 solution, a key (or unique set of keys) is determined for each instance. Code and/or data are tagged using these keys during manufacturing. At run time, this key (or set of keys) is used to regenerate tags. Only a correct key value will be able to verify what was installed during manufacture. Therefore, an instance will not accept code or data that was not tagged using its own keys.

Before the introduction of electronic PUFs [16], these keys had to be inserted into the system before they were made available to the users. To do so, keys are 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 [31], and possibly increasing costs of production since it may demand integration of different technologies on the same chip.

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. 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, memory pages, or memory blocks.

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 [12], 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 [17, 23]. Even though CFI protection is welcomed, many embedded system applications cannot afford the performance penalties and storage overhead inherently of this solution. For instance, in applications where user inputs are 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 run time attacks. For example, if an embedded system is unwatched, an attacker can upload 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 run these instructions and data only after decryption. Therefore, the major drawback of using encryption is the performance overhead that highly depends on which cryptographic primitive is employed [34]. Also, 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 [14].

2.3 Prototypes

When one comes up with an idea, it can take several steps to test its viability and come to a final product, sketches, drawings, and possibly prototypes. A prototype is an initial model built to test a design [2]. Today complex systems can have numerous combinations of software, hardware, user interactions, and visual among other possible components of this hypothetical system.

Houde *et al.* [22] proposes a simple model to understand what prototypes prototype, in this model, a designer can evaluate its goals and evaluate the prototype. Figure 2.3 shows Houde's model where the dimensions can be interpreted as follows, bearing in mind that one artifact is the system being designed.

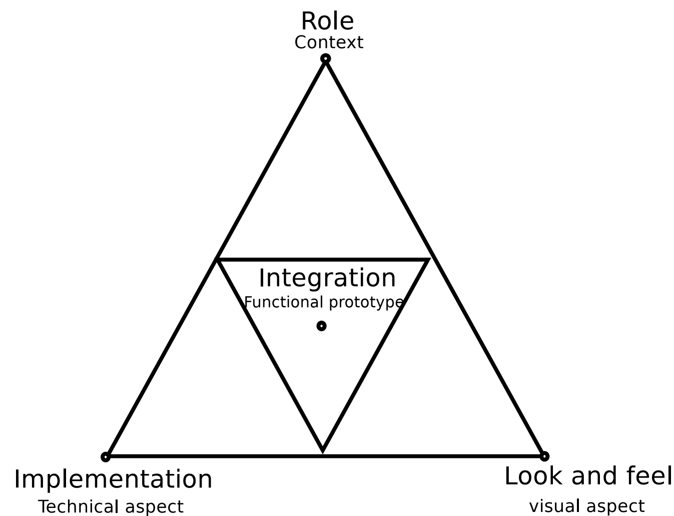


Figure 2.3: Houde *et al.* model for evaluation of the goals of prototypes.

- Role - questions about the functions of the final artifact in a user's life and this dimension will represent the context of it.
- Look and feel - denote questions about the sensory experience of an artifact, and this dimension will require a real user experience.
- Implementation - refer to techniques and components required for an artifact to accomplish its function. This dimension requires a working system to be built. Prototypes in this dimension are usually built to answer technical questions about the artifact and identify performance issues.
- Integration - It refers to a full experience of the design, it should be functional with the correct context and as close as possible to what it will look to a user.

Today we have cycle accurate computer simulations that allow individuals to test hardware even without physical instances, prototypes are still useful, though, for testing functionality, safety, and commercial potential.

Chapter 3

Related Work

Qualitative analyses of PUFs have already been done in the literature [24] motivated by several applications such as cryptographic key generation [10,35] and true random number generation [18, 25]. Unlike those works, which aim at evaluating the quality of a standalone PUF-inspired mechanism, most of the preliminary work on secure code execution aimed at keeping instructions and data secure from scrutiny, by using mechanisms like bus encryption. In [13], Elbaz *et al.* performed a comprehensive survey of bus encryption, where they describe many possible ways of using cryptographic algorithms in SoC architectures, so as to ensure that no malicious instruction/data would be executed by the CPU. The major shortcoming of these solutions is the usage of on-chip secret key storage in non-volatile memories, which enable off-line key recovery attacks [30].

AEGIS, the secure processor proposed by Suh *et al.* in [37], employs PUFs as a cryptography primitive to uniquely authenticate code and data in order to prevent both software and physical attacks. They present a toolchain for developing a secure software for their architecture which includes a secure operating system to manage different levels of memory protection. Although the presented toolchain does not require modifications in the processor architecture, it demands extensive changes in the SoC architecture, in addition to changes in the compiler and operating system. Moreover, AEGIS *does not* ensure full-time security from power-on to power-off; i.e. the system runs unprotected until the security kernel loads the system. In addition, physical attacks were neither evaluated nor simulated. Different circuits used in AEGIS, like PUFs and post-processing schemes for key extraction such as Fuzzy Extractors, have been successfully attacked with side-channel [27, 39] and semi-invasive attacks [38]. While semi-invasive attacks are hard to repeal, side-channel attacks have few known countermeasures [28] that can be quickly adopted.

In 2009, Vaslin *et al.* proposed a security approach for off-chip memory in embedded microprocessors [40]. 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 needs not only an internal timestamp memory but also a CRC32 memory. That led to 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.

Bobade and Mankar presented in [11] a secure architecture for embedded system. Their architecture provides integrity and secrecy through an Elliptic Curve Cryptographic engine. The main difference regarding the other 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 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 FPGAS, 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%. Also, 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 [32] proposed a Multi-Processor System-on-Chip that provides memory integrity and authenticity through PUFs. The proposed architecture innovates by targeting multi-processors. One key difference in 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.* argues that CSHIA needs profound modifications in SoC and CPU. However, we believed that this work demonstrates that only minor modification is needed and they are all transparent to the core and does not affect how it works. It is also essential to notice that the authors used a similar Code-offset Fuzzy Extractor CSHIA had employed initially, which is less secure than the one used in CSHIA in terms of entropy reduction 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 3.1 presents a summary of the advantages and drawbacks 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, among others. However, a qualitative analysis of design choices can still be done, as discussed in Chapter 4. For instance, PUFs have continuously been claimed to be a better solution for key generation than storing on-chip key. In that regard, CSHIA is more advantageous than those that did not use them. All the mentioned related works have a higher abstraction level, in this work, we disclose the implementation details and design tradeoffs of CSHIA.

Table 3.1: Summary of Related Works in comparison with CSHIA.

Work	Target Architecture	Advantages	Drawbacks
AEGIS [37]	High-End embedded systems and above	A complete solution	Integration with standard products can be difficult due to modification imposed to the whole toolchain.
[40]	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.
[11]	Embedded Systems	Security is based on public-key cryptography.	No performance evaluation.
[32]	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 estimate of area and power.

Chapter 4

CSHIA Architecture

The CSHIA architecture was build over an original work in [19], where the security evaluation and the viability of such solution were pondered, although the work contained an original architecture, the implementation details that would make it feasible were still to be considered. As illustrated in Figure 4.1, this first theoretical proposal was matched with a platform that would make the implementation possible; this is described in Chapter 5. With a physical platform, the creation of new interfaces was possible, and the design tradeoffs evaluated. With the end of the design phase, the first CSHIA basic architecture was ready and, later expanded to receive security upgrades, the goal of this Chapter is to describe in details the CSHIA architecture and highlight the contribution of this work and external components.

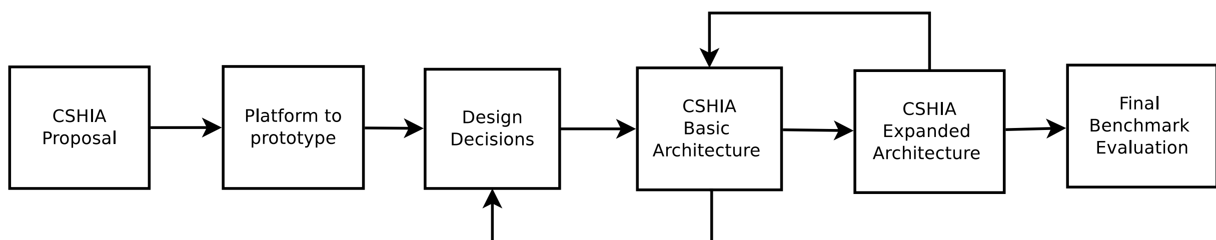


Figure 4.1: The CSHIA architecture flow since the proposal in [19] until the final evaluation.

This Chapter is organized to introduce the components as they are needed to understand each part of the system. First, a macro look of the CSHIA architecture in Section 4.1 to introduce the building blocks of this implementation, then, since this architecture will be interconnected using the AMBA2 bus, Section 4.2 describes the main components of this bus and how they work together to provide high-speed interconnections. Section 4.3 shows in detail all the components that are in the scope of this work and include the description and waveforms of the signals to explain how they interact with each other and, finally how CSHIA implementation works and in the security point of view how it accomplishes the security goals that are described in Section 4.4.

4.1 Overview

CSHIA was originally proposed in [19] as an architecture for IoT. However, we believe that CSHIA fits in a broader class of embedded system applications that can benefit from its nice security features. Many embedded system applications do not need secrecy/confidentiality, but strongly require code and data authenticity and integrity. Using the

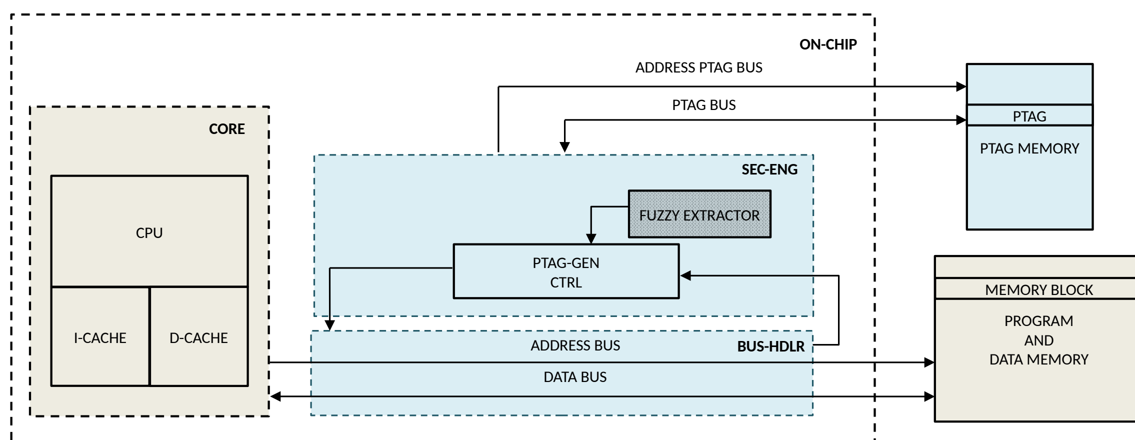


Figure 4.2: The CSHIA basic architecture.

original work with some architectural elements modified to provide stronger security features, the first Leon3 FPGA based implementation of CSHIA was realized. Figure 4.2 illustrates the first CSHIA implementation, where the full functionality originally proposed was available and indicated in light blue. All the interfaces, control, and design choices are part of this work, the security component hatched in the Figure, the FUZZY EXTRACTOR which is responsible for extracting and store the security key is an external IP datailed in [20].

This basic architecture was further evaluated and expanded by Caio Hoffman in [20] to be robust against replay attacks, for this, the architecture illustrated in Figure 4.3 was implemented and extra security components were added, the PTAG cache, a timestamp generation and control and a security feature shown as MERKLE TREE control. Together these new components that are not part of this work provide a more robust solution that is evaluated in the next sessions.

The following sections focus on presenting the CSHIA implementation components and how they work to provide authenticity and integrity, also an introduction of the AMBA2 protocol is provided to understand the proposed architecture better. Figure 4.3 illustrates the basic components required for CSHIA to work:

- One core that in this implementation is a Leon3 processor;

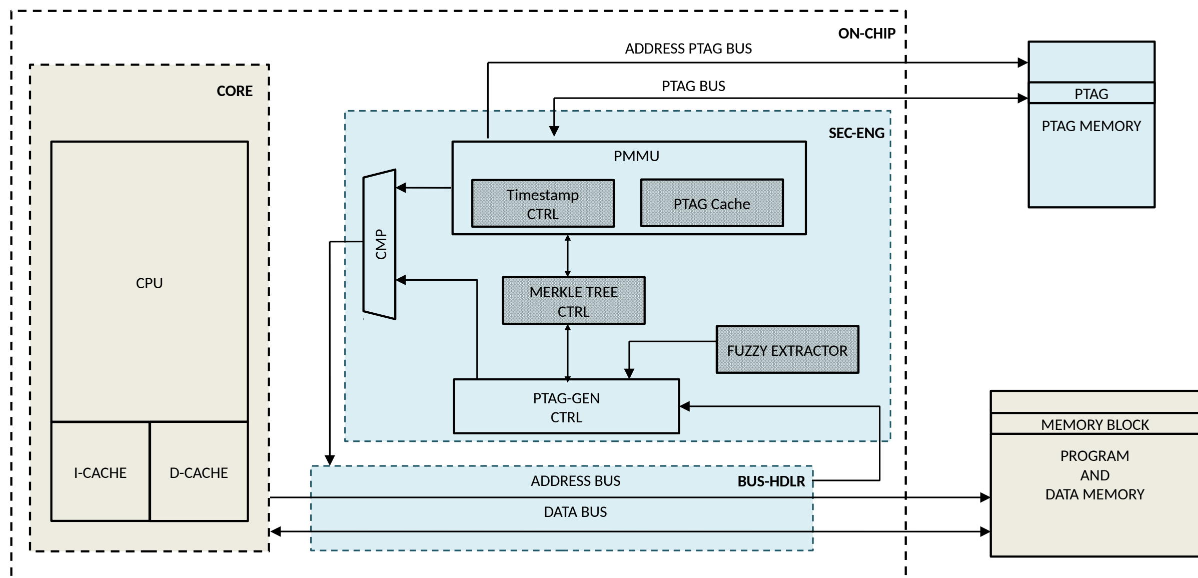


Figure 4.3: The CSHIA expanded architecture.

- The CSHIA components:
  - PTAG Memory Management Unit (PMMU)
  - Bus Handler (BUS-HDLR)
  - Security Engine (SEC-ENG)
- One external memory that contains instructions and data;
- One interconnection bus using AMBA2.

4.2 AMBA2

To interconnect complex systems, one can create customized protocols to fit the needs of the design or use industry standard protocols to save design and verification time like AMBA2 which is a protocol that enables fast interconnections of components and is broadly used. This protocol is based on masters slaves and arbiters so, the goal of the this section is to describe these components and how they work together to make sure all components communicate properly.

AMBA2 is a flavor of the Advanced High-performance Bus (AHB), where on-chip memory and other peripherals also reside. This bus provides a high-bandwidth interface between the elements connected to it, also, located on the bus is a bridge to the lower bandwidth APB, where most of the peripheral devices in the system are located. Figure 4.4 exemplify a traditional AHB utilization.

AMBA2 AHB implements the features required for high-performance, high clock frequency systems including:

- burst transfers
- split transactions
- single-cycle bus master handover
- non-tristate implementation
- wider data bus configurations (64/128 bits).

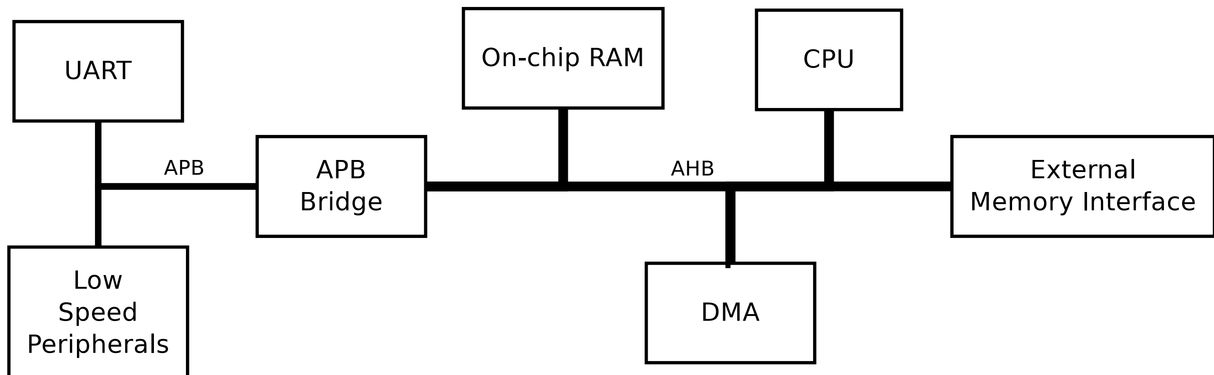


Figure 4.4: Typical AMBA2 system , with a CPU , DMA and other low bandwidth peripherals.

4.2.1 AMBA2 AHB operation

The AMBA2 AHB operation relies on basic entities to work so, to understand the previously described AMBA2 components, we need to go one step back and see what the bus sees. Anything connected to the AHB bus is either a master or a slave, as depicted in Figure 4.5, where, for instance, the CPU is an AMBA2 master, and the on-chip RAM is a slave. Who decides the priorities and decode all access is the AMBA2 arbiter. Masters can perform read and write requests while slaves need to answer when requested, the arbiter will guarantee a fair execution of those requests. This section will show the operations of an AMBA2 system.

Before an AMBA2 AHB transfer, from now on just referred to as transfer, can commence the bus master must be granted access to the bus. This process is started by the master asserting a request signal to the arbiter. Then the arbiter indicates when the master will be granted use of the bus. A granted bus master starts a transfer by driving the address and control signals. These signals provide information on the address, direction and width of the transfer, as well as an indication if the transfer forms part of a burst. Two different forms of burst transfers are allowed:

- incremental bursts, which do not wrap at address boundaries
- wrapping bursts, which wrap at particular address boundaries.

A write data bus is used to move data from a master to a slave, during a read data bus is used to move data from a slave to a master. Every transfer consists of:

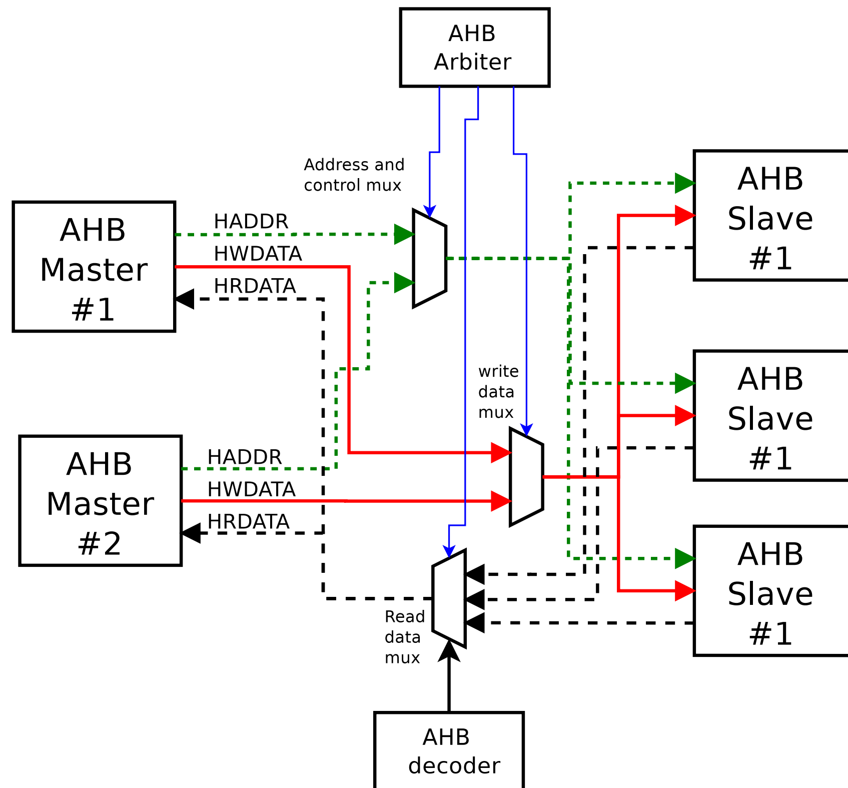


Figure 4.5: Overview of the AMBA2 Organization, with the arbiters masters and slaves distribution.

- an address and control cycle
- one or more cycles for the data.

In the address and control cycle, the address cannot be extended, and therefore, all slaves must sample the address during this time. The data, however, can be extended using the HREADY signal. When LOW these signal causes wait states to be inserted into the transfer and allow extra time for the slave to provide or sample data, as can be seen in Figure 4.6. During a transfer, the slave shows the status using the HRESP response signal where the following values are used:

- OKAY - The OKAY response is used to indicate that the transfer is progressing normally and when HREADY goes HIGH this shows the transfer has completed successfully.
- ERROR - The ERROR response indicates that a transfer error has occurred and the transfer has been unsuccessful.
- RETRY and SPLIT - Both the RETRY and SPLIT transfer responses indicate that the transfer cannot complete immediately, but the bus master should continue to attempt the transfer. In normal operation, a master is allowed to complete all the transfers in a particular burst before the arbiter grants another master access to the bus. However, in order to avoid excessive arbitration latencies, it is possible for the

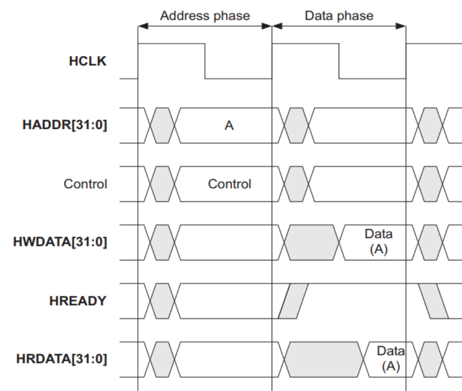


Figure 4.6: AHB basic transfer with one cycle for address and control and one or many for data.

arbiter to break up a burst and in such cases, the master must re-arbitrate for the bus in order to complete the remaining transfers in the burst.

4.2.2 AMBA2 Components

As described in Section 4.2.1 master, slaves, and arbiters are the main components of an AMBA2 system, understanding their behavior in detail will help to understand the general behavior of the system. These AMBA2 components are a subset of the entire AMBA2 features, the focus of this Section is to provide the basis to understand the signals and work required for the CSHIA implementation, for a full reference please check [1].

Masters

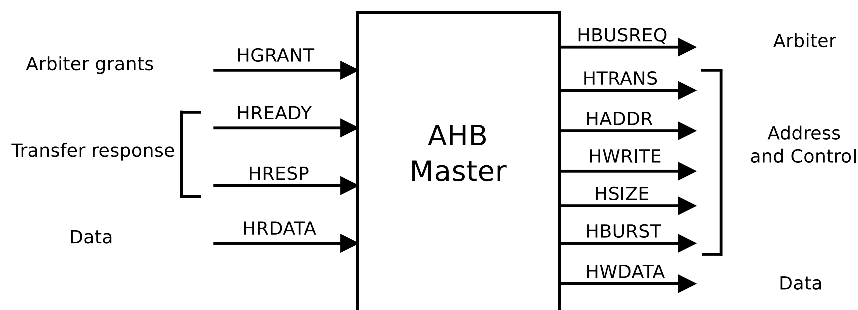


Figure 4.7: AHB master interface, with control and data signals. The HBUSSREQ and HGRANT signals will be used take control over the bus.

An AHB bus master has the most complex bus interface in an AMBA2 system, and these interfaces are depicted in Figure 4.7. The master contains one direct interface with the arbiter to receive and request the bus grant, a group of control signals to control the flow and duration of the transfer and to interfaces for reading and write data. Typically an AMBA2 system designer would use predesigned bus masters and therefore would not need to be concerned with the detail of the bus master interface.

Slaves

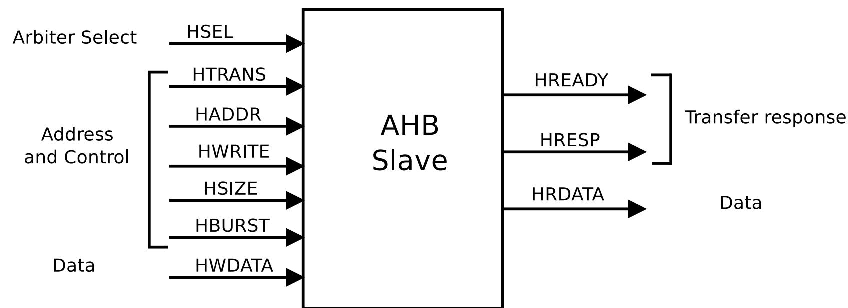


Figure 4.8: AHB slave interface, where the HSEL signal will indicate when to start transfers.

An AHB bus slave, as depicted in Figure 4.8, uses almost the same signals as the master, the difference is that the slave never controls the bus, so, the interface with the arbiter is being selected as active and respond to transfers initiated by bus masters within the system. The slave uses an HSEL select signal from the decoder to determine when it should respond to a bus transfer. All other signals required for the transfer, such as the address and control information, will be generated by the bus master.

Arbiter

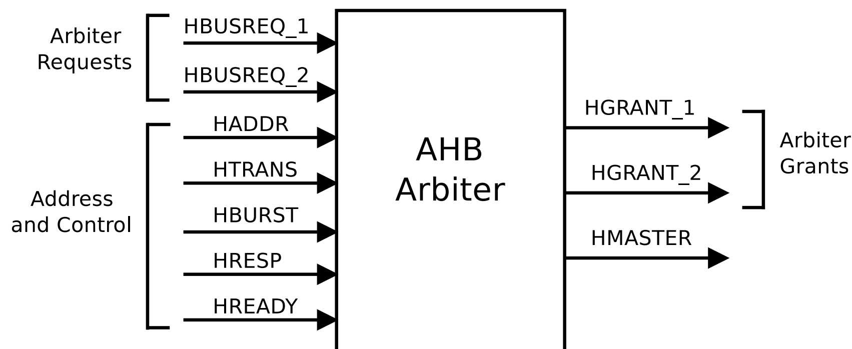


Figure 4.9: AHB arbiter interface with one HGRANT and one HBUSSREQ for each master.

The role of the arbiter in an AMBA2 system is to control which master has access to the bus. As can be seen in Figure 4.9 every bus master has a REQUEST / GRANT interface to the arbiter and the arbiter uses a prioritization scheme to decide which bus master is currently the highest priority master requesting the bus. The detail of the priority scheme is not specified and is defined for each application. It is acceptable for the arbiter to use other signals, either AMBA2 or non-AMBA2, to influence the priority scheme that is in use. The address and control signals are used by the arbiter to route the HWDATA and the HADDR from the granted master to the selected slave as illustrated in Figure 4.5.

Arbitration

The arbitration process needs specific control signals, despite the ones described below, two more signals are used in the AMBA2 system, HLOCK and HSPLIT, the first to indicate that the master wants exclusive access to the bus, the second to restart transactions when they are interrupted.

- HBUSREQ - The bus request signal is used by a bus master to request access to the bus. Each bus master has an HBUSREQ signal to the arbiter and there can be up to 16 separate bus masters in any system.
- HGRANT - The grant signal is generated by the arbiter and indicates that the appropriate master is currently the highest priority master requesting the bus, taking into account locked transfers and SPLIT transfers. A master gains ownership of the address bus when HGRANT is HIGH and HREADY is HIGH at the rising edge of HCLK.
- HMASTER - The arbiter indicates which master is currently granted using the HMASTER signal and, this same signal can be used to control the central address and control multiplexer, automatically routing the correct HDATA and control signals to the correct slave. The master number is also required by SPLIT-capable slaves so that they can indicate to the arbiter which master can complete a SPLIT transaction.

An Example of a master requesting the bus control is shown in Figure 4.13 where the arbiter grants the access after a few waiting cycles.

Decoder

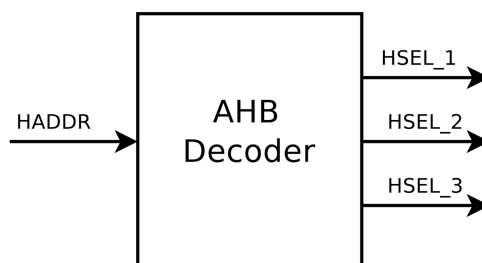


Figure 4.10: AHB Decoder interface

The decoder in an AMBA2 system is used to perform a centralized address decoding function, which improves the portability of peripherals, by making them independent of the system memory map. The Idea is that the decoder will snoop the address and select with slave will respond to the transaction, also, as shown in figure 4.5 the decoder will indirectly control a demultiplexer that will route the signals from the correct slave to the granted master.

4.3 CSHIA Components

As Section 2.2.2 discussed, the main resource to provide authentication and integrity are tags. Since CSHIA uses PUF-based keys to generate tags, we called them PUF-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 a one-to-one relationship between programs and instances, providing authenticity. To handle PTAGs, three main components are added to conventional embedded system architecture. They are: The PTAG Memory; the Bus Handler (BUS-HDLR); and the Security Engine (SEC-ENG). Figure 4.2 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 (on behalf of the processor) to main memory so as form complete memory blocks that are necessary to generate PTAGs.

SEC-ENG has three major sub-components. 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 to decode internal addresses of PTAGs to physical addresses of PTAG Memory. This section explains in detail the CSHIA components that are contributions of this work, providing the signal description, the internal behavior, and the waveforms of all the essential operations.

4.3.1 Bus Handler (BUS-HDLR)

To implement the CSHIA architecture, all the security operations described in 4.4 will be performed using a set of words, ideally an entire cache line that the processor might request but that can be multiple cache lines or any other combination necessary to be in the format of the SEC-ENG input, this set of words required to calculate the PTAGs from now on will be referred as SEC Line. The BUS-HDLR has three main functions, monitoring the processor request and respond it when necessary, assemble a SEC Line and prevent the processor the execute unsafe or unverified instructions as well as don't let it write in the bus any unsafe operation. In this context any instruction or data requested written by the processor that was not verified by the SEC-ENG is considered unsafe.

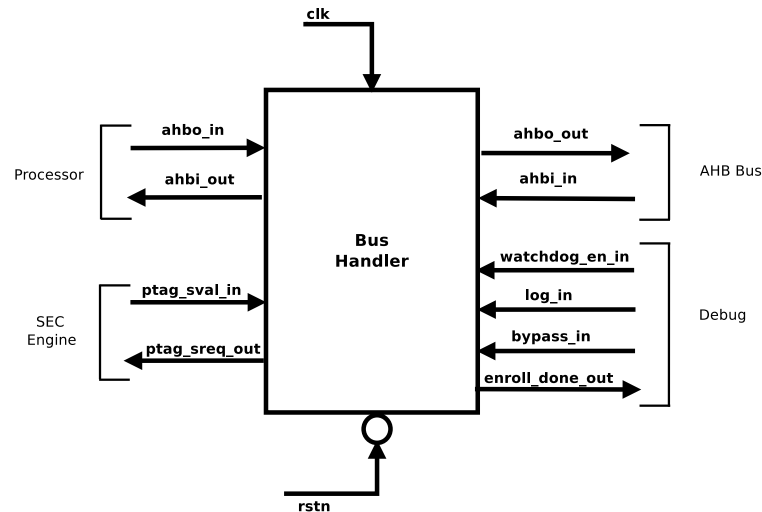


Figure 4.11: Bus Handler interface

Block Diagram

Signal Description

The inputs and outputs of this block can be split in four interfaces, the signals ahbo_in and ahbi_out the interface with the processor, ahbo_out and ahbi_in the interface with the bus, ptag_sval_in and ptag_sreq_out with the security engine and control and debug signals as described in Table 4.1. The description of each type can be found in Appendix A.

Port	in/out	Type	Description
clk	in	std_ulogic	system clock
rstn	in	std_logic	negated rset
ptag_sreq_out	out	ptag_sec_req_type	security check request
ptag_sval_in	in	ptag_sec_val_type	security check response
ahbi_in	in	ahb_mst_in_type	AHB input from bus
ahbi_out	out	ahb_mst_in_type	AHB output to processor
ahbo_in	in	ahb_mst_out_type	AHB input from processor
ahbo_out	out	ahb_mst_out_type	AHB output to BUS
std_logic bypass_in	out	std_logic	bypass input
log_in	in	std_logic	log bus activities
watchdog_en_in	in	std_logic	enable a watchdog
enroll_done	out	std_logic	enrollment phase status

Table 4.1: Description of the BUS-HDLR ports .

Functional Description

As shown in Figure 4.2 the processor will see the BUS-HDLR as the bus, and by the bus as the processor, assembling SEC Lines and constantly sending those lines to SEC-ENG. To store the SEC Lines, an internal buffer with a configurable size is used; this is the

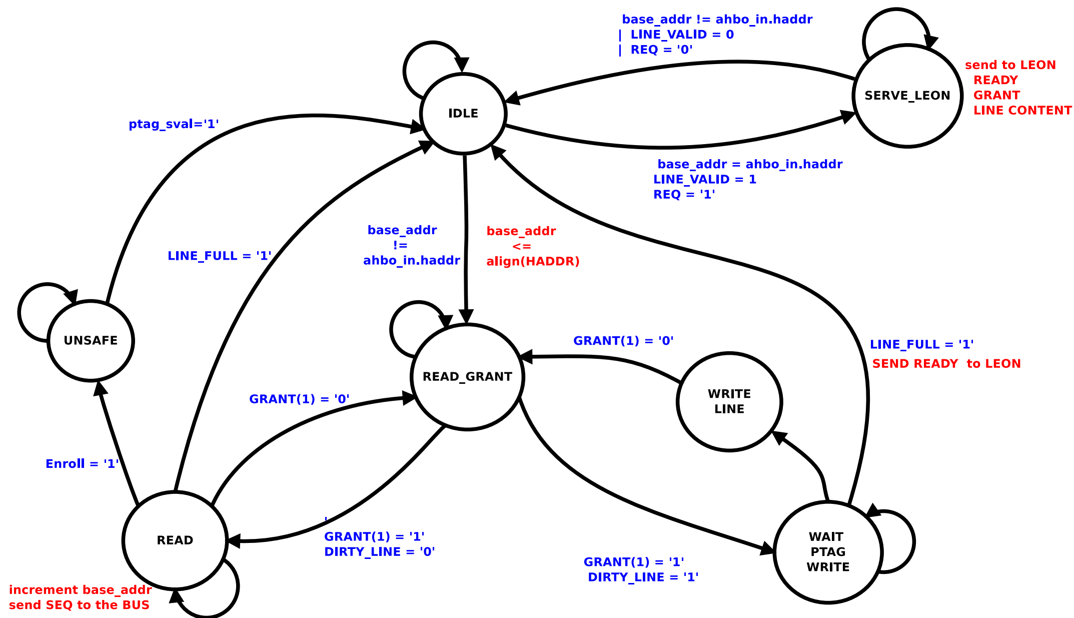


Figure 4.12: Bus Handler state machine. The blue text indicate the condition necessary for the to transition to happen, the red text indicates the BUS-HDLR actions.

BUS-HDLR SEC Line buffer and will be from now on referred to as BHS Buffer. When the processor requests the bus, the BUS-HDLRwill assert the grant, and depending on the state of the BHS Buffer, load new SEC Lines from the bus or send one from the BHS Buffer. The BUS-HDLR is implemented using a state machine which gives the block stability to assemble the SEC Line and to control the flow to the processor and to SEC-ENG. Since this state machine controls all BUS-HDLR operations the functional description of this block can be explained using the state transitions of Figure 4.12 and the following state description:

- **IDLE**

  The system stays in this state until the processor asserts HBUSSREQ, at this time the BUS-HDLR, on behalf of the arbiter, will answer these requests according to one of these scenarios:

  1. The BHS Buffer contains the address requested by the processor in a SEC Line - In this case, the processor can start the transaction in the SERVE LEON state.
  2. The address requested is not in the BHS Buffer - In this scenario, the BUS-HDLR will get the grant in the READ GRANT state and evaluate if its a read or a write, then assemble a new SEC Line.
  3. The processor requested a line that was not verified or verified incorrectly by SEC-ENG - In this case, the system will halt because a security flaw was detected.

- **READ GRANT**

  This state is required for any transaction in the bus, it requests the bus grant for the arbiter, asserting the HBUSSREQ signal on behalf of the processor before beginning

the transaction. The arbiter can answer in two possible ways, with or without wait states, for the first, illustrated in Figure 4.13, the HBUSSREQ is asserted in time 2, by the time the arbiter responds, at time 3, HGRANT and HREADY are asserted indicating that the transaction can start in the next cycle. For the second, depicted in figure 4.14, The HBUSSREQ is asserted on time 2, but the HREADY signal is only asserted on time 5 delaying the start of the transactions to time 7, in situations like this the HREADY signal regulates all the traffic.

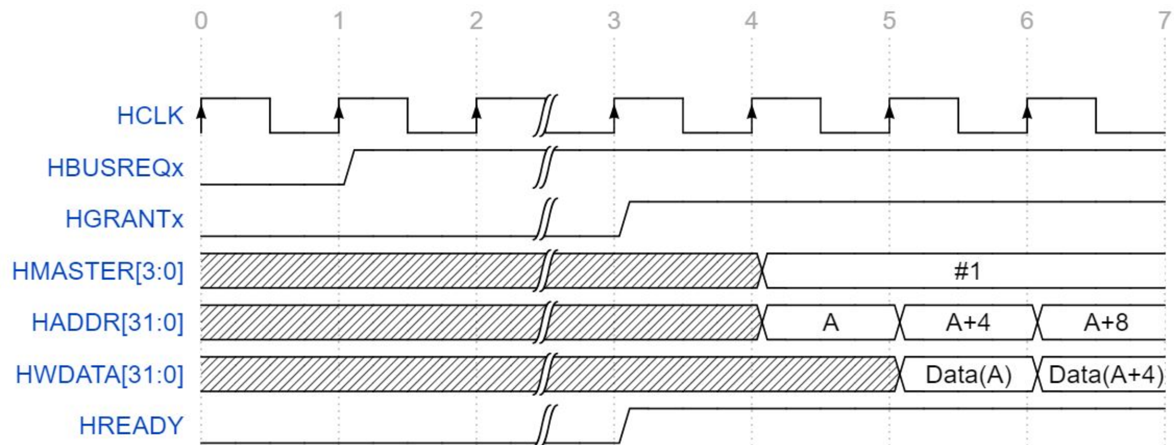


Figure 4.13: Requesting grant with no wait states

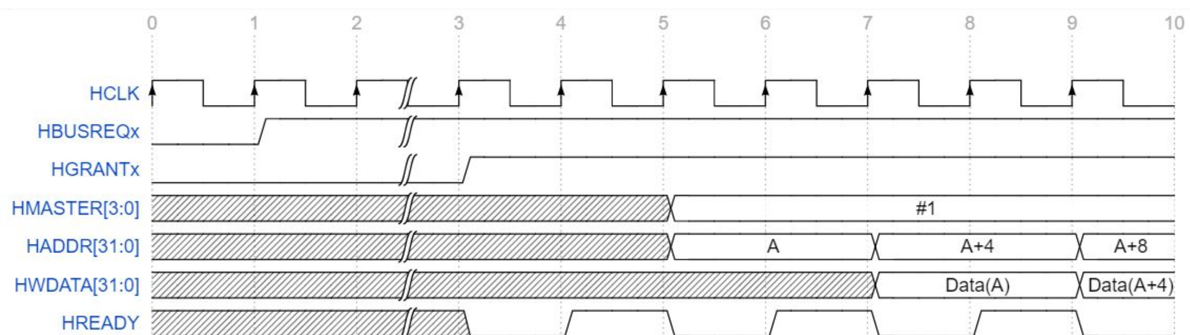


Figure 4.14: Requesting grant with wait states

Once the arbiter asserts HGRANT, the BHS Buffer is checked, and one of the positions is selected to be replaced, if the chosen SEC Line contains any updated value by the processor, then it needs to be written in the memory before loading a new SEC Line. In this is the case, BUS-HDLR will send it to the SEC-ENG to calculate a new PTAG before writing the line in the memory, then change to WAIT PTAG WRITE state while the security operations are done. If no changes were made in the SEC Linethen a new one is loaded in the READ LINE state.

- **READ LINE**

  A new SEC Linewill be loaded from the main memory into the BHS Bufferto start a transaction after the bus is granted and all the control signals are in place, as described in Section 4.2. The Figure 4.15 illustrates two different incremental read

transfers, the first starts on time 2 indicated by the first HTRANS=NONSEQ, with halfword size reading positions 0x20 and 0x22, the second transaction starts at time 4 where a small delay is exemplified to show how the HREADY controls the bus operations, and then three words are transferred to the master. Once an entire SEC Lineis transferred, the line is sent to SEC-ENG and the state is set to IDLE again, where the system will halt until the execution is considered secure. After a power cycle, all the instructions are read and tagged, in this case, the execution is not safe, so the state is set to UNSAFE before going back to IDLE. This process is explained in Section 4.4.

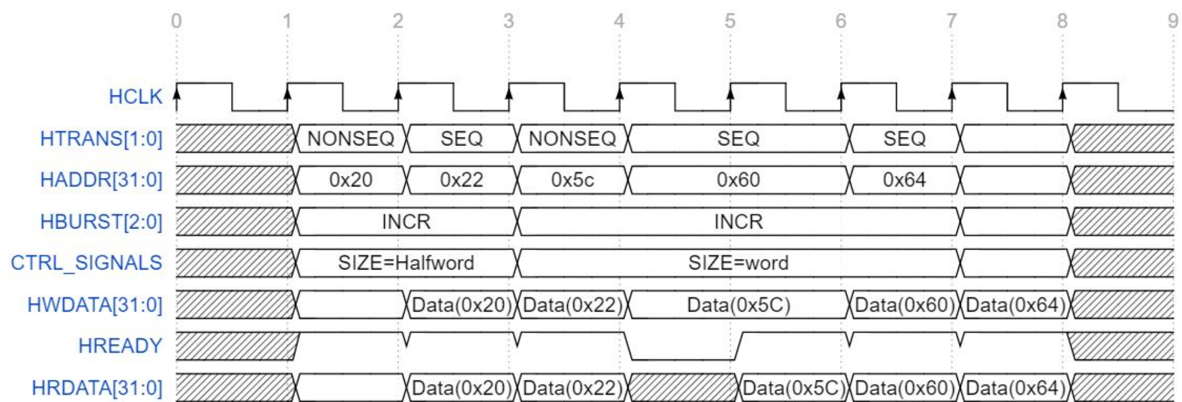


Figure 4.15: Incremental burst transfers with halfword and word.

- **WAIT PTAG WRITE** When a SEC Lineneed to be written, first is sent to the SEC-ENG to calculate the respective PTAG and store in the PTAG Memory, this process can take a different number of cycles depending on the features used in the SEC-ENG, this process is explained in Section 4.3.2. After the PTAGis calculated the line can be written in the state WRITE LINE.

- **WRITE LINE** The write process is much similar to the read, as Figure 4.16 shows the difference is the HWRITE signal that is asserted during the entire transfer. After the write is completed the state is set to IDLE and the BUS-HDLR is ready to send respond the processor requests.

- **SERVE LEON**

  On this state all LEON requests read or write are executed, the BUS-HDLR will assert the HGRANT and starting acting like the bus until there is data in the BHS Buffer. The transfers towards the processor are the same as described before in Figures 4.15 and 4.16, when the processor requests any address that is not on the BHS Buffer, the state is set to IDLE.

- **UNSAFE** After a power cycle all the instructions are read and tagged in a process called enrollment that can take a different number of cycles depending on the features used in the SEC-ENG, the enrollment is explained in Section 4.4. After a SEC Lineis considered secure during the enrollment phase, the state goes to IDLE.

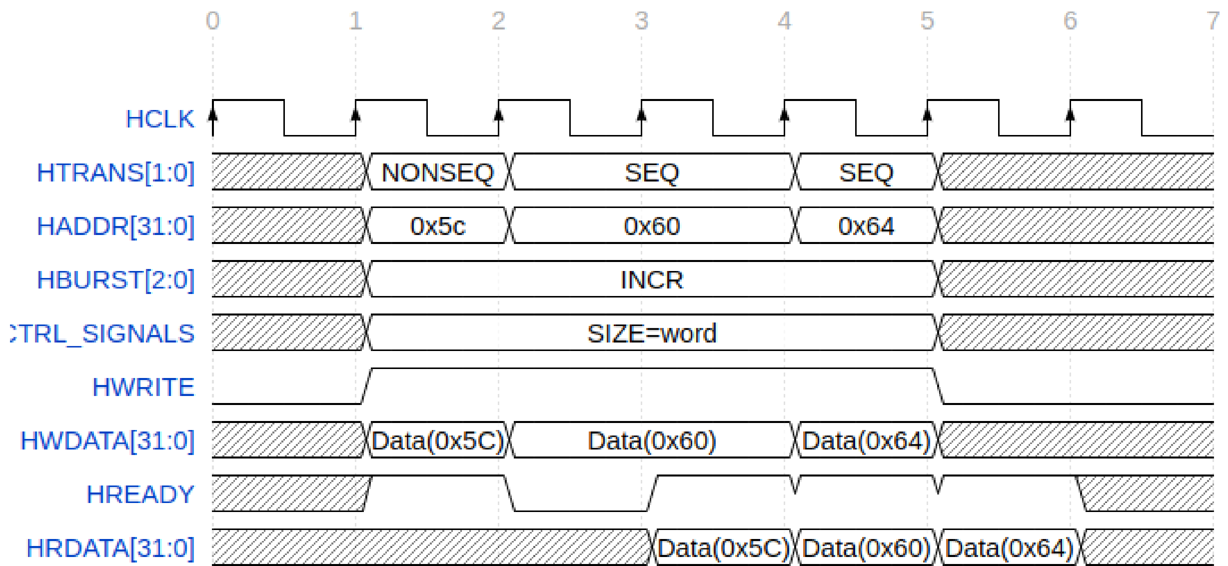


Figure 4.16: Incremental write burst of words.

4.3.2 Security Engine

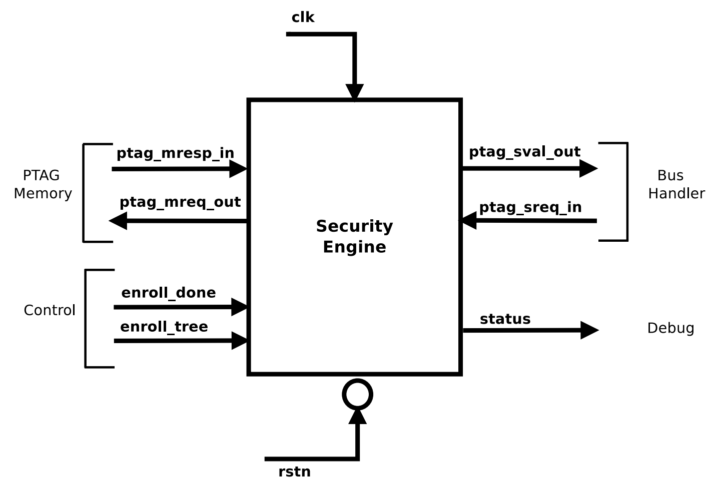


Figure 4.17: Security engine interface.

This block is responsible for the security part of CSHIA, as shown in Figure 4.17 it contains one interface toward the PTAG Memory to read and write PTAGs, one interface towards the BUS-HDLR and also control and debug signals described in Table 4.2. The SEC-ENG has three main functions, extract the key from the PUF, generate and validate PTAGs and make the PTAG Memory operations transparent to BUS-HDLR and the processor.

Signal Description

Port	in/out	Type	Description
clk	in	std_ulogic	system clock
rstn	in	std_logic	negated reset
ptag_sreq_in	in	ptag_sec_req_type	security check request
ptag_sval_out	out	ptag_sec_val_type	security check response
ptag_mreq_out	out	ptag_mreq_type	ptag memory request
ptag_mresp_in	in	ptag_mresp_type	ptag memory response
enroll_done	in	std_logic	enroll phase status
status	in	std_logic	internal state

Table 4.2: Description of the SEC-ENG ports.

Functional Description

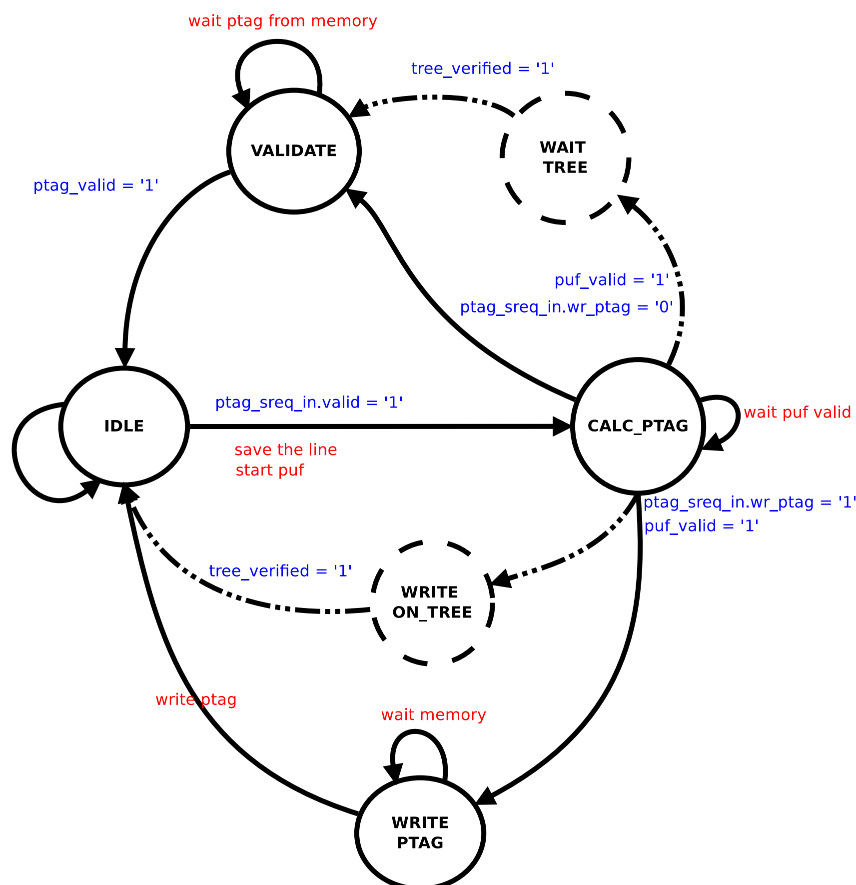


Figure 4.18: SEC-ENG state machine. The blue text indicate the condition necessary for the to transition to happen, the red text indicates the SEC-ENG actions.

The SEC-ENG is the core of CSHIA, here all the security features take place. This block has the following features:

1. Extract the cryptographic key from the SPUF

2. Create the PTAGs for each SEC Line
3. Validate PTAGsan inform the BUS-HDLR that the execution is secure
4. Control the PTAG Memory

The implementation of SEC-ENG uses a state machine illustrated in Figure 4.18 to synchronize the internal components so, the state machine transactions will be used to explain how SEC-ENG works. The initial state where IDLE, VALIDATE,CALC PTAG and WRITE PTAG, after the CSHIA expansion to add a Merkle Tree two more states were added, WRITE ON TREE and WAIT ON TREE to cooperate with the latency of the tree operations. There are two possible configurations for the SEC-ENG, with or without a Merkle Tree to provide extra protection and improve the integrity of the system this states are explained bellow.

- **IDLE**

  The SEC-ENG stays in the IDLE state until a valid SEC Line is sent by the BUS-HDLR when the line arrives either is to write or validate a PTAG, for both cases, the line will be registered, and a PTAG will be generated so that the next state will be CALC PTAG.

- **CALC PTAG** After a SEC Line is registered in IDLE state, it is used to generate a PTAG, a process that can take many clock cycles. After the PTAG is ready, the initial controls signal from the BUS-HDLR are evaluated, and the state is set to either VALIDATE for a read operation or WRITE PTAG if this is a write operation. In the extended CSHIA the next state is set to either WAIT TREE for a read operation or WRITE ON TREE if this is a write operation. For all possible cases, if the operation was a read, a request is made to the PMMU to fetch the previously calculated PTAG of the equivalent SEC Line.

- **VALIDATE** Since in this state the PTAG from the PTAG Memory and the newly generated one from the CALC PTAG state are ready, the comparison between the two values can be done, and the result as a security response to the BUS-HDLR can be sent. Then the system goes back to IDLE.

- **WRITE PTAG** This state signalizes to the PMMU that the PTAG from the CALC PTAG state can be written in the PTAG Memory and confirmation to the BUS-HDLR is sent.

- **WRITE ON TREE** When the Merkle Tree is used, this state will utilize the CSHIA extensions to write the PTAG on the memory. It requires a different set of operations that is out of this work scope and after the write is done the next state is IDLE again.

- **WAIT TREE** After a PTAG is calculated in a read request, if the Merkle Tree is used, the system will need time to retrieve the current PTAG from the tree. This state will wait for it, and the next state will be VALIDATE.

SEC-ENG operation

The SEC-ENG operations described in the previous sections can be better visualized as a continuous operation, so every time a new SEC Line is loaded it will be called a validate operation and every time it is written it will be called a write operation. The validate and the write transactions are described next.

- **Validate PTAG -** As can be seen in Figure 4.19 on time 2 the BUS-HDLR sends a SEC Lineto the SEC-ENG, the line is registered and the generation of the PTAG starts, when the ptag_ready signal is asserted, the equivalent PTAG for the SEC Line is requested from the PTAG Memory oin time 5. On time 6 in one cycle the comparison is made and if the PTAGs match a line_secure flag is asserted and the BUS-HDLRcan continue to serve the processor.

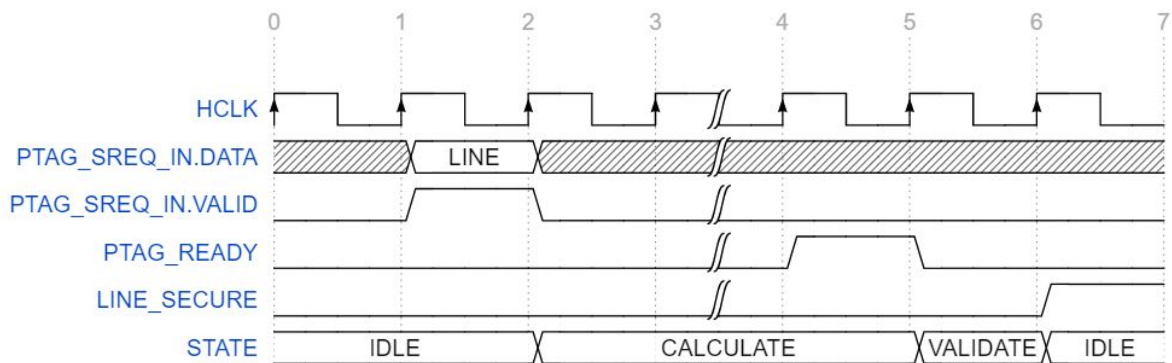


Figure 4.19: PTAG read and validate on SEC-ENG.

This process is slightly modified in the extended CSHIA, as can be seen in Figure 4.20, on time 5 the new PTAG will be compared to the content of the tree so, the new state WAIT TREE is used to wait for this operation and then on time 8 the comparison is made and the SEC-ENG can proceed.

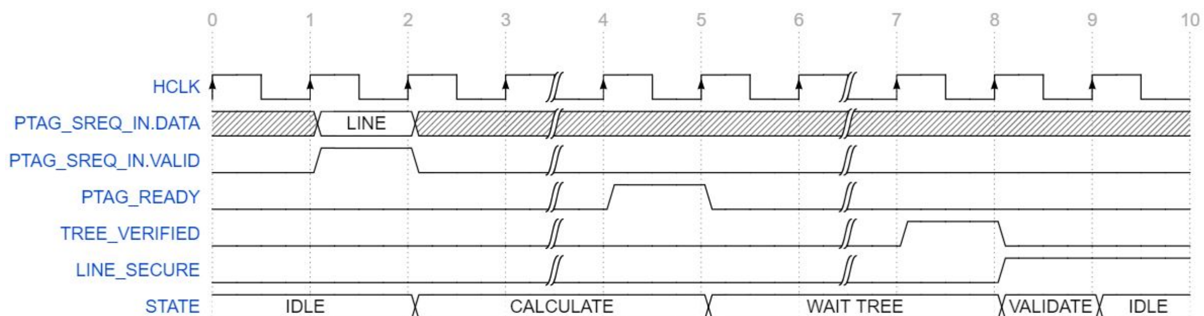


Figure 4.20: PTAG read with Merkle Tree

- **Write PTAG -** for a write operation, like in the example from Figure 4.19, the SEC Line that come from BUS-HDLR on time 2 together with a write request are registered on IDLE state. Next a new PTAG is generated in the CALC PTAG state, after the PTAG is ready and the ptag_ready signal is asserted on time 5, a

request to the PMMU is made to write the new PTAGin the PTAG Memoryon time 6, finally the state is back to IDLE and a line_secure flag is asserted on time 7 so the BUS-HDLR can continue its operation.

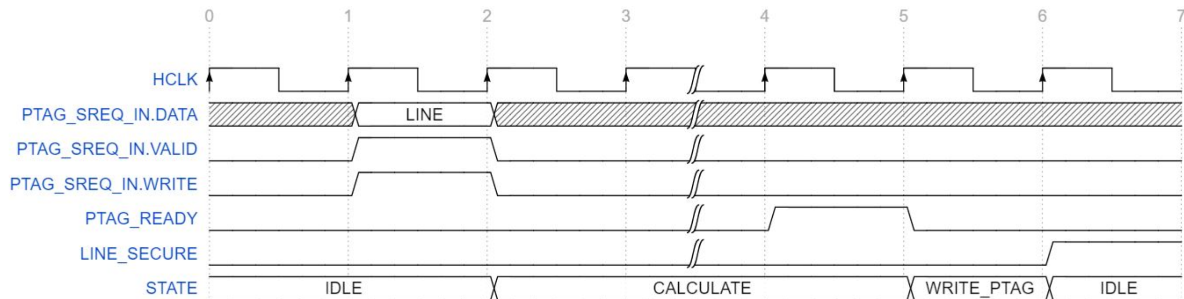


Figure 4.21: PTAG write on SEC-ENG.

In the extended CSHIA an extra step is needed as depicted in Figure 4.22 on time 5 a write is performed on the Merkle Tree and after this write is complete on time 8 the SEC-ENG can continue its operations.

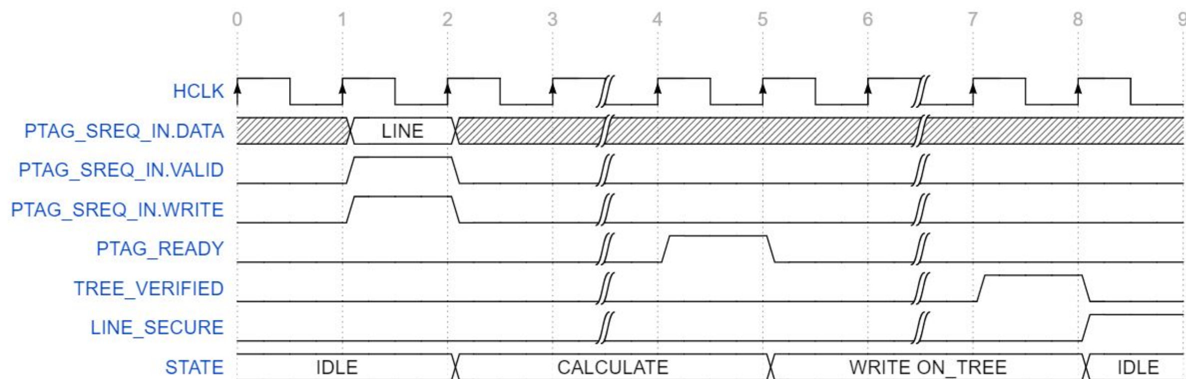


Figure 4.22: PTAG write on sec engine with Merkle Tree.

4.3.3 PTAG Memory Management Unit (PMMU)

The main functions of the PMMU are to store and request PTAGs from the PTAG Memory and also to decode internal addresses of PTAGs to physical addresses of PTAG Memory. This block is required every time the BUS-HDLR needs to execute a security check, it needs to provide the PTAG from the equivalent SEC Lines to be compared in the VALIDATE state of the SEC-ENG and write the newly generated PTAGswhen requested in the WRITE PTAG state of the SEC-ENG. Other blocks of CSHIA are agnostic to the PMMUoperation since it controls the PTAG Memory that is not connected to the bus.

4.4 Operation Modes

The previous Sections presented the CSHIA and AMBA2 components, now in a functional point of view CSHIA can be split into two phases, enrollment and runtime. The enrollment

phase described in Section 4.4.1 is the initial and crucial step ensure that the security features works correctly, the runtime phase is the regular operation, where the code will be executed and protected against attacks as described in Section 4.4.2.

4.4.1 Enrollment Phase

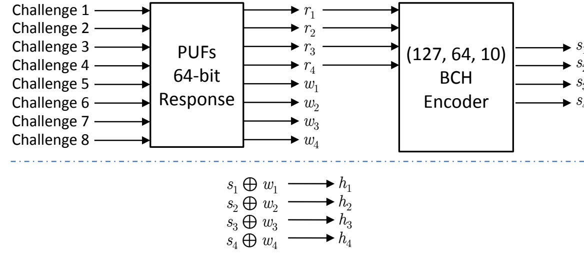
In order to ensure authenticity and integrity, an initial procedure has to be conducted by the manufacturer/vendor. This enrollment procedure will activate the Fuzzy Extractor to to extract the secret key from PUFs. Once that is done, the BUS-HDLR brings all memory blocks for tag generation. Next, this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

Key Extraction

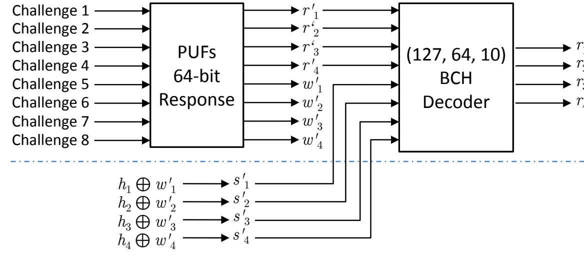
PTAG Generator implements a Pseudo-Random Function (PRF), which is a primitive cryptographic very similar to a hash function with a significant difference: the input processing is based on a secret key. In order to provide uniqueness to every CSHIA instance this key has to be unique. As aforementioned, 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 the imperfection of the fabrication process, external factors such as temperature variation, voltage variation, etc., can interfere with their responses. Thus, varying responses to challenges during the lifetime of devices. In order to provide consistency 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 [9].

There are multiple ways of implementing a Fuzzy Extractor. Originally, CSHIA was proposed using a Code-offset FE, which is well-known to reduce the entropy of extracted keys [7]. To strengthen the CSHIA design, we now use an adapted version of the Index-based Syndrome (IBS) FE proposed by Yu and Devadas in [42]. Figure 4.23 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 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 a secret key or derive the key).

To fully explain Figure 4.23 a, the chosen parameters are detailed. First, CSHIA incorporates PUFs that produce 64-bit responses. These PUFs will be responsible for generating 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 4.23 a depicts, there are four-bit strings r_i, which are compounded two by two and fed to the PRF (Figure 4.24). Such combinations were specifically designed to match the PRF chosen for CSHIA, the SipHash [8], which has an output of 64 bits and uses a 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 4.24 are



(a) Fuzzy Extractor during key extraction.



(b) Fuzzy Extractor during key regeneration.

Figure 4.23: Fuzzy Extractor actions during the enrollment and recovery procedure.

replacing addresses for input of the PTAG-GEN. 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.

Full Memory Protection

The Enrollment Phase proceeds to tag the memory range the manufacturer/vendor specified during design. Now that PTAG-GEN has a 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.

4.4.2 Runtime Phase

After the enrollment phase, CSHIA instances are ready for distribution. 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 more straightforward. Once any memory block that reached the processor was verified for integrity and authenticity, BUS-HDLR can

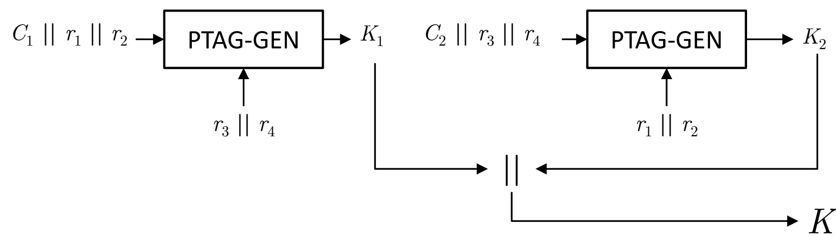


Figure 4.24: Key generation on CSHIA.

serve the cache line to SEC-ENG that uses PTAG-GEN to compute a new PTAG and PMMU sends that PTAG to PTAG Memory. 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.

Key Regeneration

During the enrollment, there were eight 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 4.23 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 4.24 shows.

Chapter 5

CSHIA Prototype

To test the CSHIA architecture we utilized industry standard IPs provided with no cost for academic purposes, the target processor to receive the security updates was the Leon3, any processor with an AMBA2 interface could be used since the goal of this work is to test how easy the implementation fits an already existing design and the impact on the performance. This chapter shows the prototype hardware setup in Section 5.1 giving details of the design kit used for this implementation and Leon3, the processor selected to host CSHIA, and in Section 5.2 the tools used to make this implementation work and its features are presented.

5.1 Hardware setup

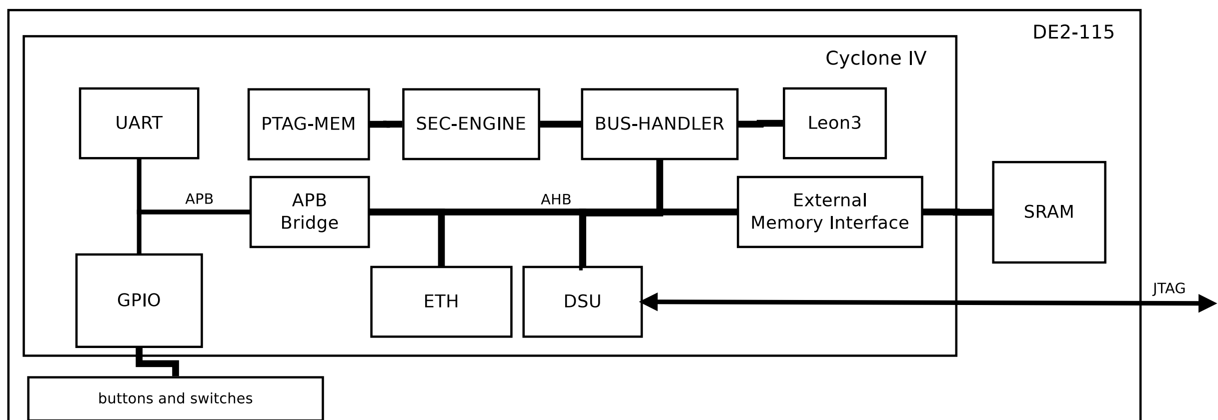


Figure 5.1: CSHIAintegrated with the DE2-115 kit ,with Leon3 , DSU and all presented security components.

We chose the Leon3 platform from Cobham Gaisler [4] 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 4.3 and better visualized in Figure 5.1. Leon3's processor (the core) is connected through the main memory by a AMBA2 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.

Table 5.1: 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

We implemented CSHIA in an Altera FPGA Development Kit DE2-115. The parameters of the processor and CSHIA are in Table 5.1. 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 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, due to the simplicity of design 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.

Continuing to look at Table 5.1, 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.

5.2 GAISLER Tools

Together with Leon3 Gaisler also provides simulation and debug tools, the simulators to test software while the hardware is being developed and the debug software named GRMON to load and debug programs directly on hardware. Whit GRMON one can access all masters and slaves on the bus and perform debug operations like inserting breakpoints for instance. This section gives us an overview of the GRMON features and an explanation about how the debug works.

5.2.1 Overview

GRMON [15] is a general debug monitor and control software that can be used after one SoC design, using GRLIB IP Library cores is loaded to an FPGA or after being manufactured as an ASIC. With the GRMON console, it is possible to download and execute Leon3 applications, and it also includes the following features:

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

5.2.2 Debug

The GRMON debug monitor is intended to debug SOCs designs based on the LEON processor. The monitor connects to a dedicated debug interface on the target hardware, through which it can perform read and write cycles on the on-chip bus (AHB).LEON3 also supports JTAG, ethernet and spacewire (using the GRESB ethernet to spacewire bridge) debug interfaces. On the target system, all debug interfaces are realized as AHB masters with the debug protocol implemented in hardware. There is thus no software support necessary to debug a LEON system, and a target system does in fact not even need to have a processor present.

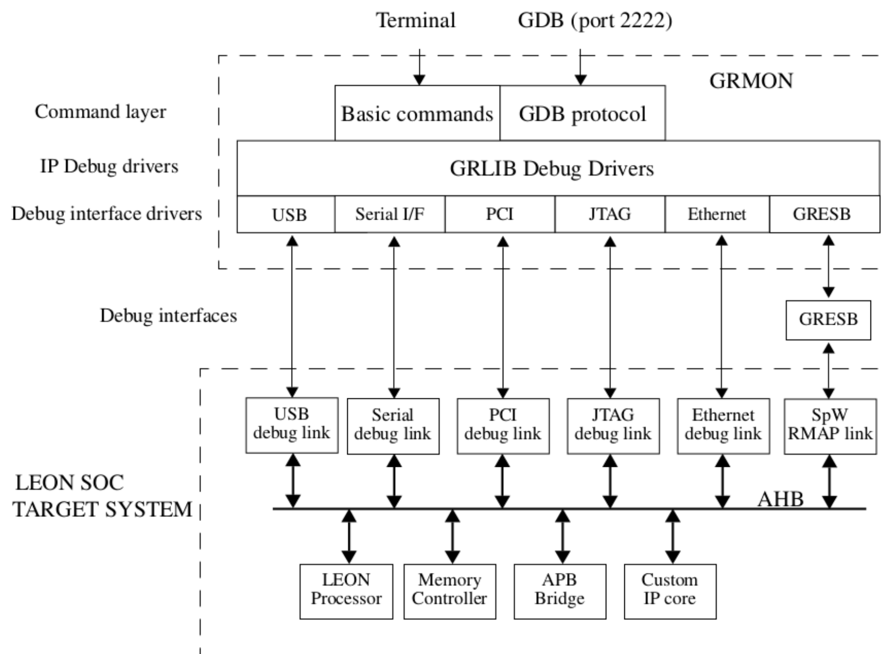


Figure 5.2: GRMON Interface

GRMON can operate in two modes: command-line mode and GDB mode. In command-line mode, GRMON commands are entered manually through a terminal window. In GDB mode, GRMON acts as a GDB gateway and translates the GDB extended-remote protocol to debug commands on the target system. As illustrated in Figure 5.2 GRMON is implemented using three functional layers: command layer, debug driver layer, and debug interface layer. The command layer consist of a general command parser which implements commands that are independent of the used debug interface or target system. These commands include program downloading and flash programming. The debug driver layer implements custom commands which are related to the configuration of the target system. GRMON scans the target system at startup, and detects which IP cores are present and how they are configured. For each supported IP core, a debug driver is enabled which implements additional debug commands for the specific core. Such commands can consist of memory detection routines for memory controllers, or program debug commands for the LEON processors. The debug interface layer implements the debug link protocol for each supported debug interface. The protocol depends on which interface is used, but

provides a uniform read/write interface to the upper layers. Which interface to use for a debug session is specified through command-line options during the start of GRMON.

Chapter 6

CSHIA Evaluation

The hardware setup described in Chapter 5 was evaluated to measure the impact of the implemented architecture on the original provided by GAISLER. The evaluation was performed in two ways, performance, and resources(Area and Power). This chapter describes the goals of this evaluation, the experimental setup, and results. First, it describes benchmarks and experiments configuration in Section 6.2. Then, it presents experimental results on performance in Section 6.3, also in the area and power estimates discussed in Section 6.4.

6.1 Evaluation Goals

When evaluating a hardware design, two particular metrics are of interest, the performance of that design and the resource consumption [41]. In this work, the goal was to use industry standard benchmarks to measure performance, in this case, the MiBench suite, and for the resources estimate area and power as close as possible to the final result.

As discussed in the previous chapters, the intention of this evaluation is to measure the CSHIA extended implementation since it provides a more robust solution, the evaluations of the next sections include all the previously described components that are part of this work.

6.2 Experimental Setup

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 [5]: `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. 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.

Table 6.1: 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

As Section 5.1 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 6.1 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 were able to cover a small portion of `.heap` segment.

Each benchmark was run in eight different instances of CSHIA in [20](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 a solution against replay attacks. Since this works only evaluate the CSHIA implementation and not the tradeoffs of the CSHIA extended security features, we will compare the (1) and (2) with (3) *CSHIA-MT*-64x2-LRU, an instance of CSHIA with a Merkle Tree and PTAG Cache of 64 lines and 2 sets.

6.3 Performance Analysis

Table 6.2 shows our results. The first conclusion is that *CSHIA-TS* performs better than the instance using the 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 AMBA2 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`.

Table 6.2: Performance overhead in % of the evaluated CSHIA instances in comparison of running times in *Leon3 Baseline*.

Benchmarks	*CSHIA-TS*(%)	*CSHIA-MT* instance(64x2-LRU)(%)
qsort_small	3.77	9.90
qsort_large	0.05	0.05
bitcount_small	0.00	0.00
bitcount_large	2.43	0.00
sha_small	8.31	16.55
sha_large	1.78	4.75
search_small	0.10	0.00
search_large	0.00	0.01
fft_small	0.00	1.07
fft_small_inv	0.92	0.00
dijkstra_small	6.40	14.09
dijkstra_large	7.35	16.90
basicmath_small	1.73	1.73
susan_small	1.37	2.73
susan_large	7.23	18.72
Average	**2.76**	**5.77**

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.

6.4 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 [29] to compute our estimations. For that, we used well-known open tools like CACTI 5.3 [3] for cache memories estimative of power and area, and Ahmed *et al.*'s work [6] 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 primary goal is to estimate the 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.

Through 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 6.3: 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

We used CACTI to estimate how the timestamp memory and PTAG Cache affects the design. From Table 5.1, the total timestamp memory size was 2 bytes $\times$ 2^{14} (or 32 KB). Even though CACTI does do 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 two sets, all estimations are summarized in Table 6.3.

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. In the security side, 16-bit timestamps 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.

Chapter 7

Conclusion

PUFs are physical security primitives which enable trust in the context of digital hardware implementations of cryptographic constructions, in particular, they can initiate physically unclonable and secure key generation and storage. In this dissertation, the implementation of CSHIA, a platform that utilizes PUFs to achieve information security objectives that are data integrity and authentication is presented.

We described in detail the building blocks of this implementation and the way this system can work at block level together with the impact of this implementation in comparison to the non-secure system.

In Chapter 2, we introduced the basics of PUFs and the information security goals of this work, in order to make clear what we want to achieve. The importance of authentication and integrity was highlighted and examples of PUFs in order to give useful examples of how they work and fit in the context of this work.

A review of the related work was presented in chapter 3, where similar architectures implemented security features to provide similar features as CSHIA, but in this kind of work, it is difficult to have a fair comparison between the security features and the implementation details, mostly because of FPGA limitations and the way synthesis tools deal with different designs. So the characteristics of each work were resumed and compared with strong and weak points to make it easier to see where CSHIA fill the gaps of other works, presenting performance, area, power estimations and now implementations details of the how the security blocks work on hardware.

The CSHIA architecture implementation was described in Chapter 4, with an extensive description of the AMBA protocol and all the steps required to get the results achieved, the initial CSHIA architecture and which components were added to extend CSHIA to make it more robust and improve integrity. The contributions of this work were described and an in-depth explanation of how the building blocks of this implementation work with timing diagrams and interfaces explained.

Chapter 5 presented the details of the prototype, and how using the Leon3 processor with our tailored security IPs in a DE2-115 FPGA development kit we could sustain the original 50MHz, and due to the limitations of the kit, we covered 512Kb of memory. The evaluation of the prototype was presented in Chapter 6 were we show the selected benchmarks together with the total coverage and performance of the CSHIA extended implementation where without the Merkle Tree the average performance degradation is

2.76% and with it goes to 5.77% in exchange of a more robust design. Analyzing area and power the CSHIA logic overhead achieved 34% mostly due to the BHS Buffer that scales when accessing big SEC Lines.

The contributions of this work were submitted to the following venues

- TETC-2017 - IEEE Transactions on Emerging Topic in Computing - Achieving Code Authenticity through Hardware Intrinsic Feature
- MicPro-2018 - Microprocessors and Microsystems: Embedded Hardware Design - Implementing a Secure Architecture for Code and Data Authenticity and Integrity in Embedded Systems

Bibliography

[1] AMBA2. https://static.docs.arm.com/ihi0011/a/IHI0011A_AMBA_SPEC.pdf. Web Site. Accessed: 26-May-2019.

[2] Blackwell, amy hackney, and elizabeth manar, editors. "prototype." uxl encyclopedia of science, 3rd ed., uxl, farmington hills, mi, 2015. science in context. link.galegroup.com/apps/doc/ENKDZQ347975681/SCIC?u=dclib_main&sid=SCIC&xid=0c8f739d. Web Site. Accessed: 27 June 2019.

[3] 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.

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

[5] 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.

[6] S. Z. Ahmed, J. Eydoux, L. Rouge, J. B. Cuelle, G. Sassatelli, and 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*, pages 184–189, April 2009.

[7] 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.

[8] Jean-Philippe Aumasson and Daniel 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.

[9] 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.

[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] Sunil D Bobade and Vijay R Mankar. Developing configurable security algorithms for embedded system storage. *International Journal of Computer Science and Telecommunications*, 6(7), July 2015.

[12] 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.

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

[14] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B. Lee, Nachiketh Potlapally, and Lionel Torres. Transactions on computational science iv. chapter Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines, pages 1–22. Springer-Verlag, Berlin, Heidelberg, 2009.

[15] COBHAM Gaisler. Grmon user manual, 2018. accessed 06/28/2018, `https://www.gaisler.com/doc/grmon2.pdf`.

[16] 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.

[17] Olga Gelbart, Paul Ott, Bhagirath Narahari, Rahul Simha, Alok Choudhary, and Joseph Zambreno. Codesseal: Compiler/fpga approach to secure applications. In Paul Kantor, Gheorghe Muresan, Fred Roberts, Daniel D. Zeng, Fei-Yue Wang, Hsinchun Chen, and Ralph C. Merkle, editors, *Intelligence and Security Informatics*, pages 530–535, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[18] Anthony Van Herrewege, Vincent van der Leest, André Schaller, Stefan Katzenbeisser, and Ingrid Verbauwhede. Secure prng seeding on commercial off-the-shelf microcontrollers. *IACR Cryptology ePrint Archive*, 2013:304, 2013.

[19] 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.

[20] Caio Hoffman. *Computer Security by Hardware-Intrinsic Authentication*. PhD thesis, University of Campinas, 1 2019.

[21] 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.

[22] Stephanie Houde and Charles Hill. What do prototypes prototype. *Handbook of HumanComputer Interaction*, 1997.

[23] A. K. Kanuparthi, M. Zahran, and R. Karri. Architecture support for dynamic integrity checking. *IEEE Transactions on Information Forensics and Security*, 7(1):321–332, Feb 2012.

[24] 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.

[25] 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.

[26] Roel Maes. *Physically Unclonable Functions: Constructions, Properties and Applications (Fysisch onkloonbare functies: constructies, eigenschappen en toepassingen)*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2012.

[27] 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.

[28] Dominik Merli, Frederic Stumpf, and Georg Sigl. Protecting puf error correction by codeword masking.

[29] M. Nemani and 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):697–713, Jun 1999.

[30] Ahmad-Reza Sadeghi and David Naccache, editors. *Towards Hardware-Intrinsic Security - Foundations and Practice*. Information Security and Cryptography. 2010.

[31] Ahmad-Reza Sadeghi and David Naccache. *Towards Hardware-Intrinsic Security: Foundations and Practice*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Shahin Tajik, Enrico Dietz, Sven Frohmann, Helmar Dittrich, Dmitry Nedospasov, Clemens Helfmeier, Jean-Pierre Seifert, Christian Boit, and Heinz-Wilhelm Hübers. Photonic side-channel analysis of arbiter pufs. *Journal of Cryptology*, pages 1–22, 2016.

[40] 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).

[41] Wayne Wolf. A decade of hardware/software codesign. *Computer*, 36(4):38–43, April 2003.

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

Appendix A

Type Description

Signal	Type	Description
cache_line	std_logic_vector	the entire cach line
base_addr	std_logic_vector	base physical address of the line
valid	std_logic	asserted if the line is valid
wr_ptag	std_logic	asertedif the calculated ptag will be writen in th e PTAG mem

Table A.1: **ptag_sec_req_type** content description

Signal	Type	Description
ptag	std_logic_vector	the calculated ptag
valid	std_logic	asserted when ptag is valid
line_secure	std_logic	asserted if the provided line is secure
ready	std_logic	asserted when ready to receive a new line

Table A.2: **ptag_sec_val_type** content description

Signal	Type	Description
we	std_logic;	active high write enable
address	std_logic_vector	memory address
data	std_logic_vector	data to be writen

Table A.3: **ptag_mreq_type** content description

Signal	Type	Description
data	std_logic_vector	PTAG read from memory

Table A.4: **ptag_mresp_type** content description

Signal	Type	Description
hgrant	std_logic_vector	bus grant
hready	std_ulogic	transfer done
hresp	std_logic_vector	response type
hrdata	std_logic_vector	read data bus
hcache	std_ulogic	cacheable
hirq	std_logic_vector	interrupt result bus
testen	std_ulogic	scan test enable
testrst	std_ulogic	scan test reset
scanen	std_ulogic	scan enable
testoen	std_ulogic	test output enable

Table A.5: **ahb_mst_in_type** content description

Signal	Type	Description
hbusreq	std_ulogic	bus request
hlock	std_ulogic	lock request
htrans	std_logic_vector	transfer type
haddr	std_logic_vector	address bus (byte)
hwrite	std_ulogic	read/write
hsize	std_logic_vector	transfer size
hburst	std_logic_vector	burst type
hprot	std_logic_vector	protection control
hwdata	std_logic_vector	write data bus
hirq	std_logic_vector	interrupt bus
hconfig	ahb_config_type	memory access reg.
hindex	integer	diagnostic use only

Table A.6: **ahb_mst_out_type** content description