Universidade Estadual de Campinas
Instituto de Computação

Augusto Fernandes Ribas Queiroz

Secure code execution using PUF authentication

Execução segura de códigos utilizando PUF para autenticação

CAMPINAS
2019

Augusto Fernandes Ribas Queiroz

Secure code execution using PUF authentication

Execução segura de códigos utilizando PUF para autenticação

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

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

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

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

CAMPINAS
2019

Universidade Estadual de Campinas
Instituto de Computação

Augusto Fernandes Ribas Queiroz

Secure code execution using PUF authentication

Execução segura de códigos utilizando PUF para autenticação

Campinas, 28 de março de 2019

Dedicatória

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

Resumo

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

Abstract

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

List of Figures

4.1 The CSHIA architecture. 22
4.2 Bus Handler black box 24
4.3 Bus Handler state machine 25
4.4 Security engine black box 27
4.5 Top of the system 28
4.6 Key generation on CSHIA. 30
4.7 Fuzzy Extractor actions during the enrollment and recovery procedure. . . 31

List of Tables

3.1 Summary of Related Works in comparison with CSHIA. 18

4.1 Ports of the security handler 24
4.2 Ports of the security engine 27

5.1 CSHIA FPGA implementation configuration. 35

Contents

1 Introduction **11**
1.1 Contributions 12
1.2 Publications 13
1.3 Organization of the dissertation 13

2 Fundamental concepts **14**
2.1 Physical Unclonable Functions - PUFs 15
2.2 Security Properties 15
2.2.1 Authenticity 15
2.2.2 Integrity 15
2.2.3 Secrecy 16

3 Related Work **17**

4 CSHIA Architecture **21**
4.1 Components of the Architecture 22
4.1.1 Bus Handler (BUS-HDLR) 23
4.1.2 Security Engine 26
4.1.3 Memory Management Unit (PMMU) 29
4.1.4 PTAG Memory 29
4.2 Operation Modes 29
4.2.1 Runtime Phase 29
4.2.2 Enrollment Phase 30

5 CSHIA Prototype **33**
5.1 Hardware setup 34
5.2 GAISLER Tools 36

6 CSHIA Evaluation **37**

7 Conclusion and Future Work **38**
7.1 Conclusion 38
7.2 Future Work 38

Bibliography **39**

Chapter 1

Introduction

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

Due to the stringent nature of available resources of embedded systems, software solutions for code and data integrity do not fit best. In addition, software authenticity would involve a third party certification authority. Therefore, hardware solutions are desirable for such systems. A myriad of hardware solutions for code and data authenticity and integrity have been proposed in the literature ([**?**,3,9,21]), however, some of those solutions target high-end embedded systems or more powerful configurations, requiring at least a two-level cache in the processor for their performance overhead not to be prohibitive. Other approaches need modifications on the Instruction Set Architecture (ISA) or processor datapath, leading to complete redesign of code, compilers, operating systems, among others. Moreover, not all solutions provide integrity and authenticity.

Recently, an architecture aiming at code/data authenticity and integrity was proposed in [8]. 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 CSHIAover the previous hardware solutions are that it does not require changes in the ISA or datapath, being adaptable to most of embedded system architectures while providing complete software compatibility, and also using a separate bus for the tag memory, which gives to designers freedom to match timing requirements to hide verification overhead.

Basing on Gaisler's Leon3 [1] FPGA implementation, this work presents a proof-of-concept of CSHIA. The main goal of our implementation was to improve the original version of the architecture and add more flexible design choices. Besides presenting an in depth description of the hardware implementation, the design tradeoffs and the integration between the architecture and a real processor.

1.1 Contributions

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

1.2 Publications

The contribuitions of this work were published in the following conferences

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

1.3 Organization of the dissertation

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

Chapter 2

Fundamental concepts

2.1 Physical Unclonable Functions - PUFs

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

2.2 Security Properties

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

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 (an enrollment phase). On running time, this key (or set of keys) is used to regenerate tags. Only a correct key value will be able to verify what was installed during manufacture. Therefore, an instance will not accept code or data that was not tagged using its own keys.

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

2.2.2 Integrity

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

Integrity can also be considered at the instruction sequence level, which we refer as Control-Flow Integrity (CFI). Hardware solutions for control-flow integrity usually require deep integration between hardware and software [4], 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 [6,10]. Even though the CFI protection is welcomed, due to the focused nature of embedded systems, many applications cannot afford the performance penalties and storage overhead inherently of this solution. For instance, in applications where user inputs is limited and I/O involves fixed amounts of data, an attacker has very little room to employ a buffer overflow or similar attacks prevented by CFI. However, integrity verification regarding blocks of code and data (as mentioned above) can avoid a variety of situations that go beyond runtime attacks. For example, if an embedded system is unwatched, an attacker can upload a malicious code or modify the data in the external memory even if the system is not running. Integrity verification can prevent and indicate these violations before they reach the processor.

2.2.3 Secrecy

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

Chapter 3

Related Work

Qualitative analyses of PUFs have already been done in the literature [11] motivated by several applications such as cryptographic key generation [2, 17] and true random number generation [7, 12]. Unlike those works, which aim at evaluating the quality of a standalone PUF-inspired mechanism, this work focus on proposing and analyzing a PUF-based micro-architecture mechanism to enable secure code execution.

Most of the preliminary work on secure code execution aimed at keeping instructions and data secure from scrutiny, by using mechanisms like bus encryption. In [5], 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 [15].

AEGIS, the secure processor proposed by Suh *et al.* in [18], employs PUFs as a cryptography primitive to uniquely authenticate code and data in order to prevent both software and physical attacks. They present a toolchain for developing secure software for their architecture which includes a secure operating system to manage different levels of memory protection. Although the presented toolchain does not require modifications in the processor architecture, it demands extensive changes in the SoC architecture, in addition to changes in the compiler and operating system. Moreover, AEGIS *does not* ensure full-time security from power-on to power-off; i.e. the system runs unprotected until the security kernel loads the system. In addition, physical attacks were neither evaluated nor simulated. Different circuits used in AEGIS, like PUFs and post-processing schemes for key extraction such as Fuzzy Extractors, have been successfully attacked with side-channel [13, 20] and semi-invasive attacks [19]. While semi-invasive attacks are hard to repeal, side-channel attacks have few known countermeasures [14] that can be easily adopted.

In 2009, Vaslin *et al.* proposed a security approach for off-chip memory in embedded microprocessors [21]. Vaslin *et al.* used the One-Time-Pad (OTP) scheme to provide

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

Work	Target Architecture	Most Positive Feature	Downside
AEGIS	High-End embedded systems and above	A complete solution	Integration with standard products can be difficult due to modification imposed to the whole toolchain.
[21]	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.
[3]	Embedded Systems	Security is based on public-key cryptography.	No performance evaluation.
[16]	MPSoC	First PUF based secure architecture for multiple cores.	Does not estimate area and power increment in regard to the baseline system.
CSHIA	Embedded Systems	Design Flexibility.	Does not provide concrete estimative of area and power.

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

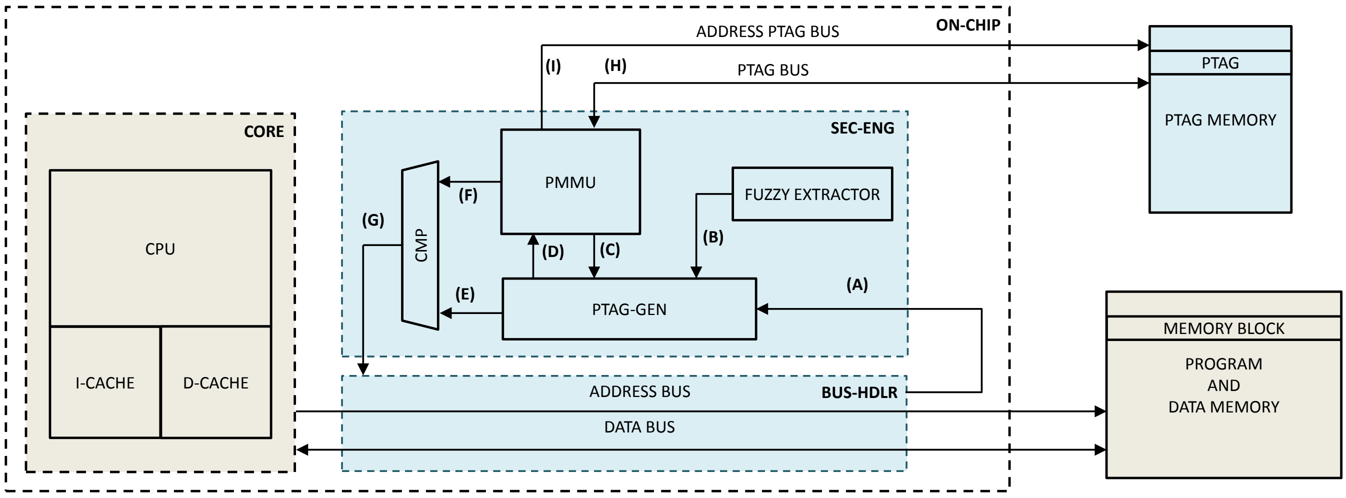
Bobade and Mankar presented in [3] a secure architecture for embedded system. Their architecture provides integrity and secrecy through an Elliptic Curve Cryptographic engine. The main difference regarding the others architectures presented here is that they use the timestamps as private keys. Thus, cache lines are encapsulated with their address and time stamp (for integrity verification purpose), and then encrypted with the public key to be stored in external memory. As the timestamps are stored in an internal memory, the decryption can be done with reprocessing the pair private/public key and the integrity is ensured by the correct decryption of the triad encapsulated: data, address, and time stamp. Although Bobade and Mankar synthesized their architecture for a FPGA, they only simulated the architecture and did not use any benchmark. Nonetheless, they computed the overhead of slices and LUTs over their baseline processor, which was over 76%. Memory overhead was 25%. In addition, they estimated power increment over baseline. Despite the dynamic power more than doubled in all processor's frequency simulated, the static was kept stable.

Recently, Sepulveda, Wilgerodt, and Pehl in [16] has proposed a Multi-Processors System-on-Chip that provides memory integrity and authenticity through PUFs. The proposed architecture innovates by targeting multi-processors. They also used SipHash to provide integrity tags to memory blocks to protect against all three major threats we have discussed before. One key difference on their replay attack solution is that they use session tokens instead of timestamps. While that is an innovative way, it may not be sufficient to protect against replay attacks, since tokens are updated during idle periods and booting time. Thus, in a long period of execution, in which a specific memory block can be written back multiple times to memory, an attacker might mount a replay attack. One interesting point is that Sepulveda *et al.*argue that CSHIA needs deep modifications in SoC and CPU. However, we believed that this work demonstrates that only minor modification are needed and they are all transparent to the core and does not affect how it works. It is also important to notice that the authors used a similar Code-offset Fuzzy Extractor CSHIA had originally employed, which, as we discussed in the previous section, is less secure than the one used in CSHIAin 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 most positive feature and downside of CSHIA and related works. A fair comparison of performance among the works is quite hard to be performed, due to a variety of benchmarks, baseline cores, choice of platforms, etc. However, a qualitative analysis over design choices can still be done. For instance, PUFs have been constantly claimed to be a better solution for key generation than storing on-chip key. In that regard, CSHIAis 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 tradoffs of CSHIA.

Chapter 4

CSHIA Architecture



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

Figure 4.1: The CSHIA architecture.

CSHIA was originally proposed in [8] as an architecture for IoT. However, we believe that CSHIA fits in a variety of embedded system applications that can benefit from its architectural design decisions. As we stated before, many embedded system applications do not need secrecy/confidentiality, but strongly require code and data authenticity and integrity. Using the original work as base, we modified some elements to provide stronger security features, as well as make CSHIA adaptable to a FPGA implementation. This section focus on presenting our CSHIA main architectural components and how they work to provide authenticity and integrity.

4.1 Components of the Architecture

As Section 2.2.2 discussed, the main resource to provide 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 one-to-one relatioship between programs and instances, providing authenticity. To handle PTAGs, three main components are added to a conventional embedded system architecture. They are: The PTAG Memory; the Bus Handler (BUS-HDLR); and the Security Engine (SEC-ENG). Figure 4.1 shows this design and how components communicate between themselves.

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

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

4.1.1 Bus Handler (BUS-HDLR)

This block has three main functions, monitoring the processor requests and responde then when necessary, assemble a line that can be multiple cache lines or any other combination necessary to be in the format of the security engine input and prevent the processor the execute unsafe or unverified instructions as well as don't let it write in the bus any unsafe operation.

Block Diagram

Signal Description

The inputs and outputs of this block can be split in three interfaces and control being the signals ahbo_in and ahbi_out the inteface with the leon3 processor, ahbo_out and ahbi_in the interface with the bus , ptag_sval_in and ptag_sreq_out with the security engine and control the signals clk ,reset_n and bypass_in. The description of each type can be found in the Appendix **??**.

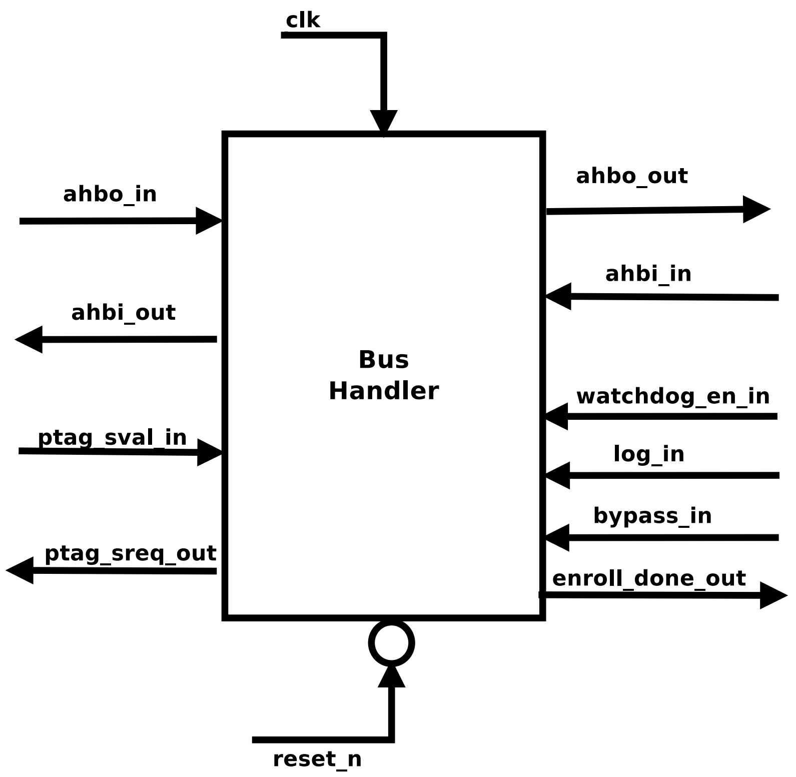


Figure 4.2: Bus Handler black box

Port	in/out	Type	Description
clk	in	std_ulogic	system clock
rstn	in	std_logic	negated rset
bypass_in	out	std_logic	bypass input
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

Table 4.1: Ports of the security handler

Functional Descricption

Since this one state machine control all the operation of the security handler the functional description of this block can be explained using the state transictions of Figure 4.3 and the following state description:

- **IDLE**

  The system stay in this state until the processor signalize a request, givem the request there are three options, the line contains the content requested by the processor, in this case the request can be responded in the SERVE LEON state, the

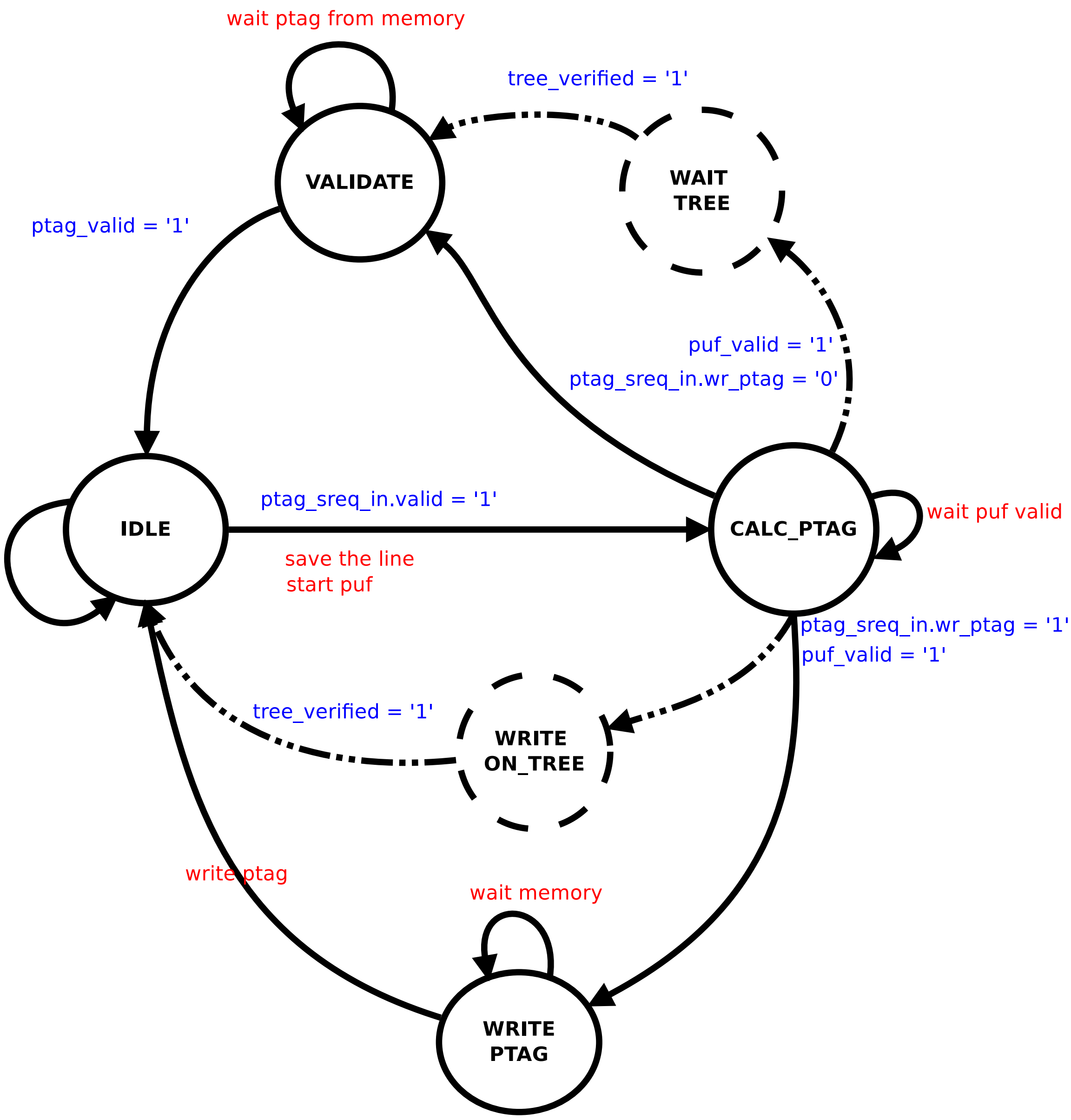


Figure 4.3: Bus Handler state machine

line doesn't contain the content required so a new line must be loaded in the READ GRANT state or the current line is not safe and the system will halt going to UNSAFE state.

- **READ GRANT**

  This state is required for any transaction in the bus, it requests the bus grant for the arbiter prior to begin the request, once the arbiter gives the grant the line status is checked and if the line is dirty(the processor update any value in the line) then it needs to be written in the meomory before loading a new line, if this is the case then the system goes to WRITE LINE state, if no changes were made in the line then a new line is loaded in the READ LINE state. The steps needed to aquire the bus grant are shown in section ??.

- **READ LINE**

A new line will be loaded from the main memory to the local buffer,the steps needed to execute a read or write operation is shown in ??, so a counter is started and counts the number of words the bus provided, so when the line is loaded the line status is changed to valid and the system goes back to IDLE state. When the line is ready then a request is sent to the security engine to check if the line is safe and the system will freeze until the integrity of the line is confirmed.

- **WRITE LINE**

  Once the line is dirty and need to be written in the main memory, a counter is starded, similar to the read process but it now counts each time the write request is confirmed by the bus, when all words were written the line sirty status is now changed to not dirty and the system goes to IDLE. In the process of writing the line to memory, in parallel a request is made to the security engine to calculated the new PTAG related tohis line and store it in the PTAG memory.

- **SERVE LEON**

  On this state all LEON requests read or write are executed, if the content needed is in the line. when any operetaion requires a word that is not in the local line buffer then the system goes back to IDLE state.

- **UNSAFE**

  When a security check fails on the IDLE state the system uses this as a trap state to halt.

subsection

Operation

4.1.2 Security Engine

This block is responsable for the interface with the external memory wich stores the physical tags values for each sets of challenges, and for calculatng this values using one or more PUFs, the PUF description is in section XX.

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

Table 4.2: Ports of the security engine

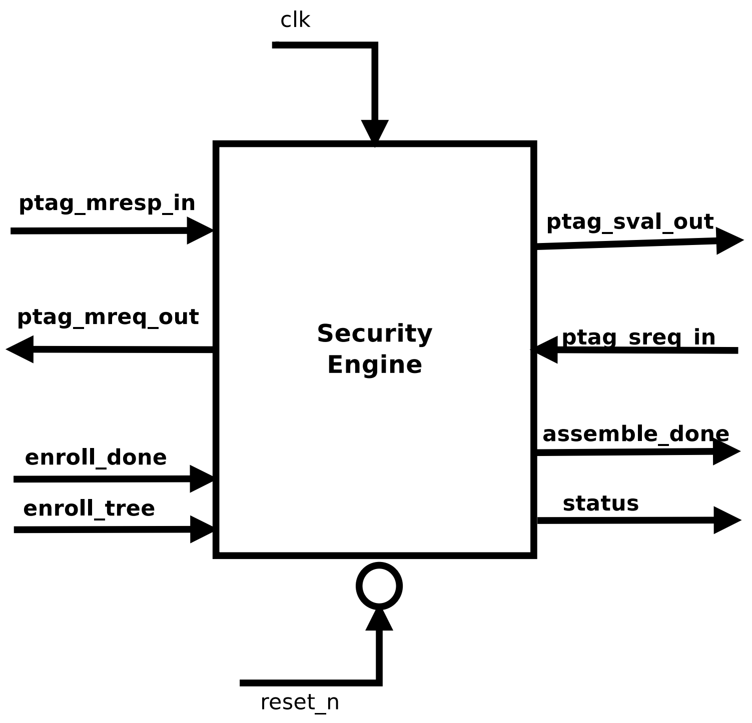
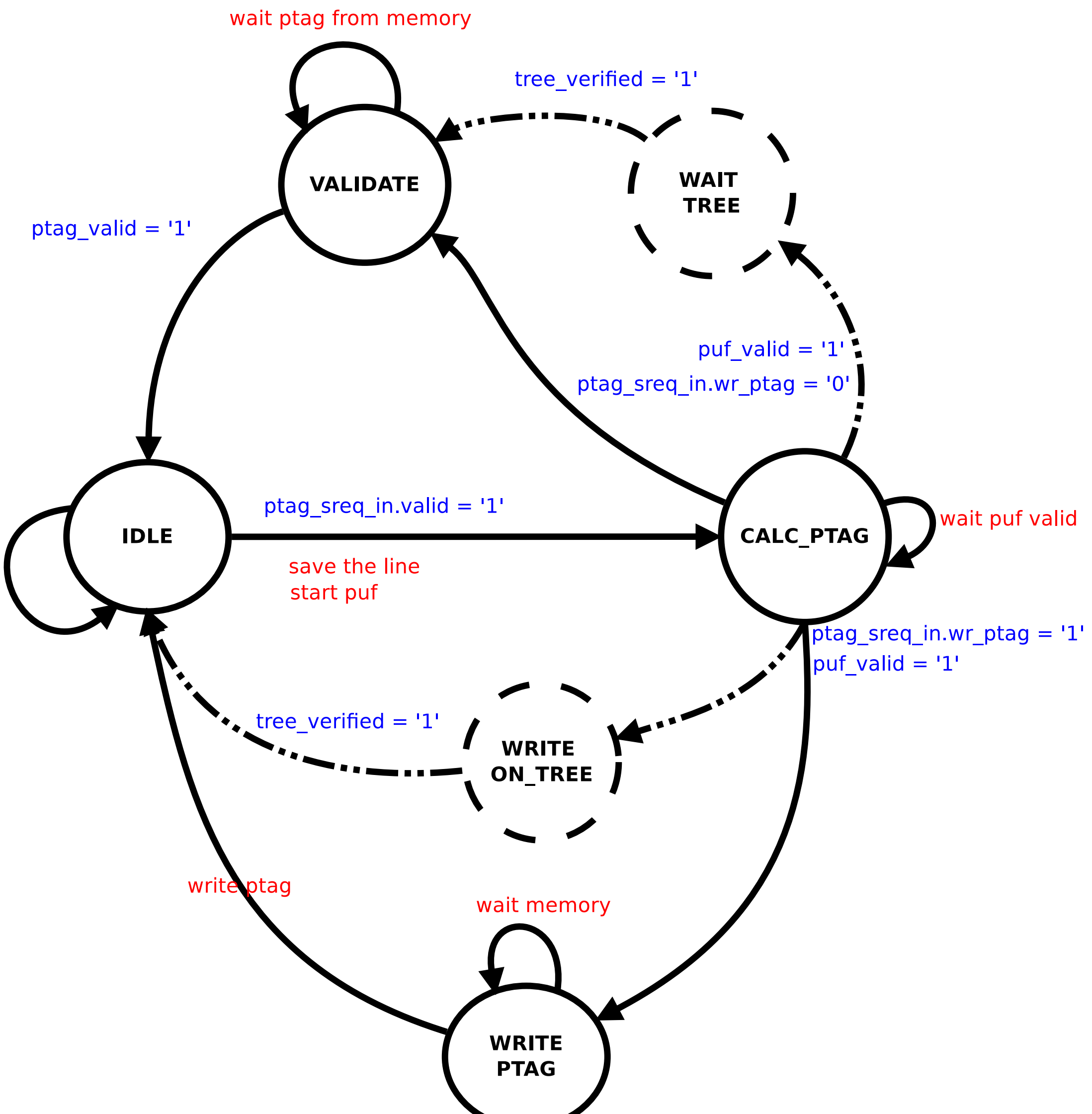


Figure 4.4: Security engine black box

Functional Descricption



The functional operation of this block consists in two basic actions, given a challenge calulate the PTAG using the internal PUF and if this is a check operation the compare the calculated value with the equivalent from the PTAG memory, if its a write operations than just store the calculated value in the PTAG memory.the state machine needed to complete this task is described bellow.

- **IDLE**

  The security engine stays in IDLe until a valid challenge is siganlized, then the input is registered and a request for the PUF is sent to calculate tha PTAG , as well as read operation to the PTAG memory in case of a check operation.

- **CALC PTAG** When a new challenge comes to be checked or written, a new ptag is calculated using the internal PUF, after the calculation is ready the controller goes either to VALIDATE state if this is a check operation or to WRITE PTAg state if this is a write operation.

- **VALIDATE** Since in this state the PTAG from the PTAG memory and the calculated from the PUF are ready this state compares the values and send the result as a security response to the security handler. Then the system goes back to IDLE.

- **WRITE PTAG** This state uses the memory interface to write the calculated PTAG in the PTAG memory and send a confirmation through the security response to the security handler.

4.1.3 Memory Management Unit (PMMU)

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

4.1.4 PTAG Memory

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

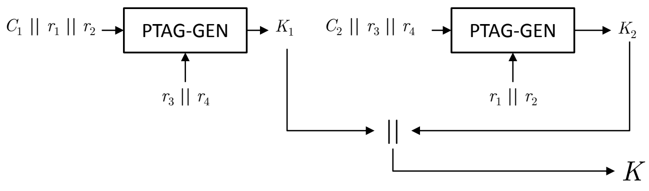


Figure 4.6: Key generation on CSHIA.

4.2 Operation Modes

4.2.1 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 simpler. Once any memory block that reached the processor was verified for integrity and authenticity, BUS-HDLR can serve the cache line to SEC-ENG that uses PTAG-GEN to compute a new PTAG and PMMU sends that PTAG to PTAG Memory. 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 8 challenges selected to produce four r_i and four w_i values. These challenges and helper data can be exposed off-chip and stored in PTAG Memory if the designer chooses to do so. The recovery process of the secret key can be seen in Figure 4.7 (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 4.6 shows.

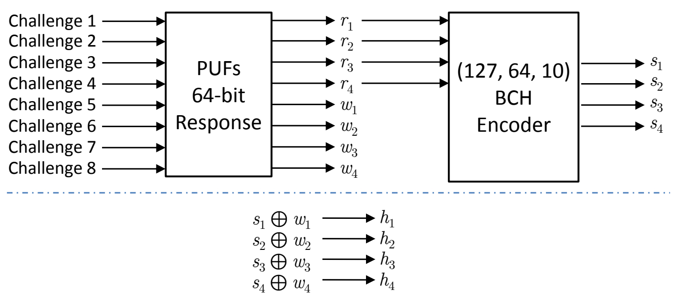
4.2.2 Enrollment Phase

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

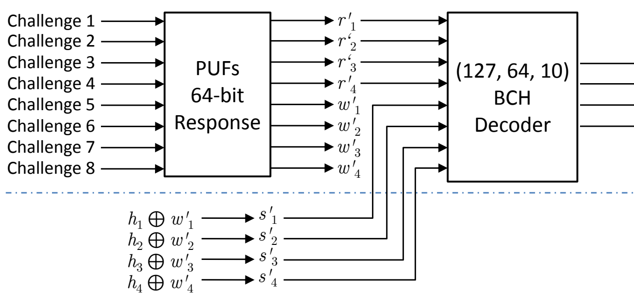
Key Extraction

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

There are multiple ways of implementing a Fuzzy Extractor. Originally, CSHIA was proposed using a Code-offset FE, which is well-known to reduce entropy of extracted keys [?]. To strengthen the CSHIA design, we now use an adapted version of the Index-based Syndrome (IBS) FE proposed by Yu and Devadas in [?]. Figure 4.7 (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 secret key or derive the key).


Challenge 1
Challenge 2
Challenge 3
Challenge 4
Challenge 5
Challenge 6
Challenge 7
Challenge 8
PUFs 64-bit Response
r_1 r_2 r_3 r_4 w_1 w_2 w_3 w_4
(127, 64, 10) BCH Encoder
s_1 s_2 s_3 s_4
$s_1 \oplus w_1 \rightarrow h_1$
$s_2 \oplus w_2 \rightarrow h_2$
$s_3 \oplus w_3 \rightarrow h_3$
$s_4 \oplus w_4 \rightarrow h_4$

(a) Fuzzy Extractor during key extraction.


Challenge 1
Challenge 2
Challenge 3
Challenge 4
Challenge 5
Challenge 6
Challenge 7
Challenge 8
PUFs 64-bit Response
r'_1 r'_2 r'_3 r'_4 w'_1 w'_2 w'_3 w'_4
(127, 64, 10) BCH Decoder
$h_1 \oplus w'_1 \rightarrow s'_1$
$h_2 \oplus w'_2 \rightarrow s'_2$
$h_3 \oplus w'_3 \rightarrow s'_3$
$h_4 \oplus w'_4 \rightarrow s'_4$

(b) Fuzzy Extractor during key regeneration.

Figure 4.7: Fuzzy Extractor actions during the enrollment and recovery procedure.

To fully explain Figure 4.7 (a), the chosen parameters are detailed. First, CSHIA incorporates PUFs that produce 64-bit responses (more details in the next section). These PUFs will be responsible to generate each string r and w that are 64 bits long. To match the length of r and w, CSHIA has a (127, 64, 10)-BCH ECC. As Figure 4.7 (a) depicts, there are four bit strings r_i, which are compounded two by two and fed to the PRF (Figure 4.6). Such combinations were specifically designed to match the PRF chosen for CSHIA, the SipHash [?], which has an output of 64 bits and uses key of 128 bits. Therefore, the first pair of bit strings r_i is concatenated with a constant and processed by the PRF using the second pair of bit string r_i as key. That generates a hash K_1. Then, inverting their places and concatenating the second pair with a different constant, a hash K_2 is obtained. Concatenating K_1 with K_2 results in K which is the secret key of CSHIA. Notice that C_1 and C_2 in Figure 4.6 are replacing addresses for input of the PTAG-GEN. Further details of security will be given in the following sections, however, one can notice that assuming that each bit string r_i has at least half of their length of entropy, each part of the key will have full entropy. Hence, the key has full entropy.

Full Memory Protection

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

Chapter 5

CSHIA Prototype

5.1 Hardware setup

We chose the Leon3 platform from Cobham Gaisler [1] 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 4.1 in Section **??**. Leon3's processor (the core) is connected through the main memory by a AMBA Bus version 2.0. In our modification, the processor's I/O master bus connects it to BUS-HDLR, which then provides a new I/O master bus for the rest of the components in the platform. Thus, BUS-HDLR is transparent to all components of the platform, even the core. One of the components that is specific of Leon3's platform is the Debug Support Unit (DSU), which allows a designer using a debug host (such as a computer) to connect to development kits running Leon3. Through the debugging connection, a program can be loaded to the FPGA memory, started, paused, among other useful functions.

We implemented CSHIA in an Altera FPGA Development Kit DE2-115. The parameters of the processor and CSHIA are in Table 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.

Due to the high utilization of internal memory, timestamp memory became limited to

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

2^{14} 16-bit words to cover the 512 KB of data memory. This represents 5.4% of the total 588 KB main memory coverage. The PTAG Memory utilization for this solution was up to 147 KB (code and data), or 25% overhead. Table 5.1 also shows the PTAG cache's configuration. Since this cache is an internal memory as well, it was limited to 4 KB. That limitation did not prevent us of evaluating the cache in multiple configurations. We evaluated this cache in different configurations of lines, set associativity, and replacement policies. Finally, as previously discussed, PMMU requires a buffer to stall a PTAG cache write or read while an eviction is required. We calculated that this buffer needs to be at most 2 times the number of lines in PTAG Cache.

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

5.2 GAISLER Tools

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

Chapter 6

CSHIA Evaluation

Chapter 7

Conclusion and Future Work

7.1 Conclusion

7.2 Future Work

Bibliography

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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).