

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

Caio Hoffman^{a,*}, Augusto F. R. Queiroz^a, Diego F. Aranha^{a,b}, Mario L. Côrtes^a, Guido Araujo^a

^a*Institute of Computing, University of Campinas, Brazil*

^b*Department of Engineering, Aarhus University, Denmark*

Abstract

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

NOT PUBLISHED IN PEER-REVIEWED VENUE

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

1. Introduction

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

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

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

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

*caio.hoffman@ic.unicamp.br

average 5.77 % reduction on performance. Finally, we discuss how the CSHIA flexible design is a very attractive solution for embedded systems when compared to state-of-the-art architectures.

Overall, this work's contributions are:

- A flexible FPGA implementation which allows designers to easily change an architecture security option between Timestamps and Merkle Tree.
- A secure architecture that deploys PUF-based keys with a robust and careful implementation of Fuzzy Extractor.
- A open-source and easy to run VHDL extension of Leon3's processor.

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

2. Security Issues in Embedded Systems

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

2.1. Threat Model

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

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

tampering with the memory contents, the attacker successfully changes the instructions/data the processor will receive when performing read operations.

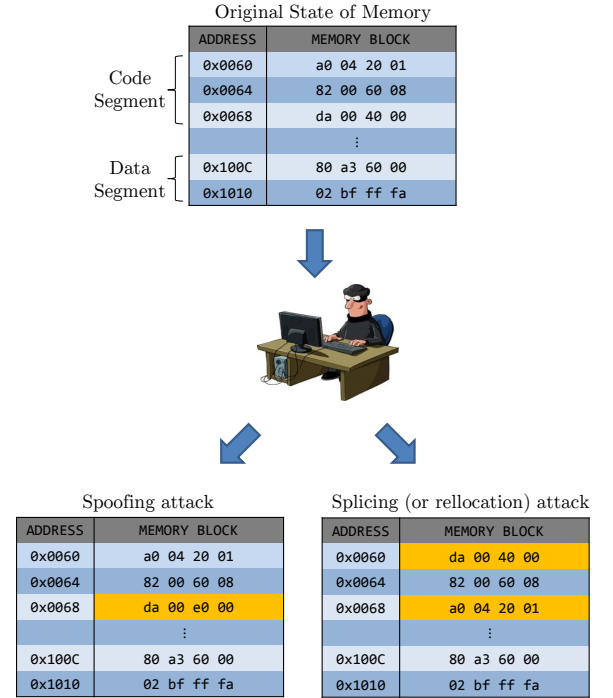


Figure 1: Spoofing and Splicing (or Relocation) attacks.

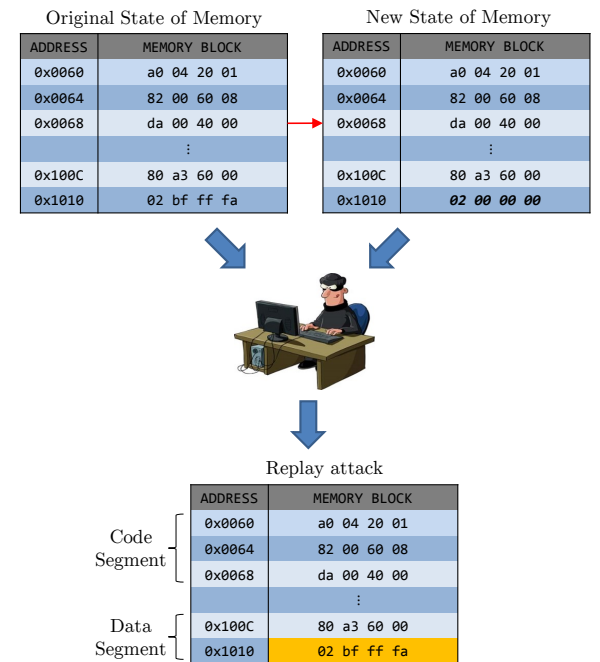


Figure 2: Replay attack.

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.

Previously the introduction of electronic Physical Unclonable Functions (PUFs) [7], these keys had to be inserted into systems before they were made available to customers. To do so, keys were stored on chip using non-volatile memories and the manufacturer/vendor controlled the uniqueness of the keys in each instance. The main downsides of storing key permanently include: facilitating physical attacks [8], and possibly increasing costs of production since it may demand integration of different technologies on the same chip.

Recently, PUFs have been employed to generate secret keys. PUFs are physical functions created to mimic random functions. Their inputs, called challenges, and outputs, called responses, are designed to have a unique relation for every PUF instance. This is achieved by leveraging on imperfections resulted from fabricating electronic devices. In regard to authenticity, the main advantage of using PUFs as key generators is that they can produce keys on running time, on-chip memories are not needed for key storage, and they are unclonable. That means that even the manufacturer itself cannot produce two PUF instances that will have the same the set of Challenge-Response Pairs (CRPs) [7].

2.2.2. Integrity

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

Integrity can also be considered at the instruction sequence level, which we refer as Control-Flow Integrity (CFI). Hardware solutions for control-flow integrity usually require deep integration between hardware and software [9], that can result

not only in changing the Instruction Set Architecture (ISA) and/or the tool-chain, but also the processor's data path, as proposed in [10, 11]. Even though the CFI protection is welcomed, due to the focused nature of embedded systems, many applications cannot afford the performance penalties and storage overhead inherently of this solution. For instance, in applications where user inputs is limited and I/O involves fixed amounts of data, an attacker has very little room to employ a buffer overflow or similar attacks prevented by CFI. However, integrity verification regarding blocks of code and data (as mentioned above) can avoid a variety of situations that go beyond runtime attacks. For example, if an embedded system is unwatched, an attacker can upload a malicious code or modify the data in the external memory even if the system is not running. Integrity verification can prevent and indicate these violations before they reach the processor.

2.2.3. Secrecy

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

2.3. Analysis of Countermeasures

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

2.3.1. Preventing Spoofing Attacks

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

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

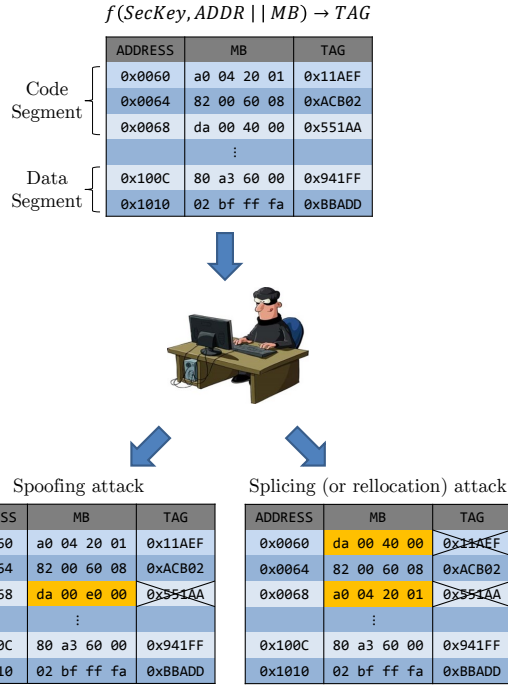


Figure 3: Preventing Spoofting and Splicing (or Relocation) attacks.

Hence, while timestamps need an on-chip memory, a root tag only requires a memory element of the size of a register.

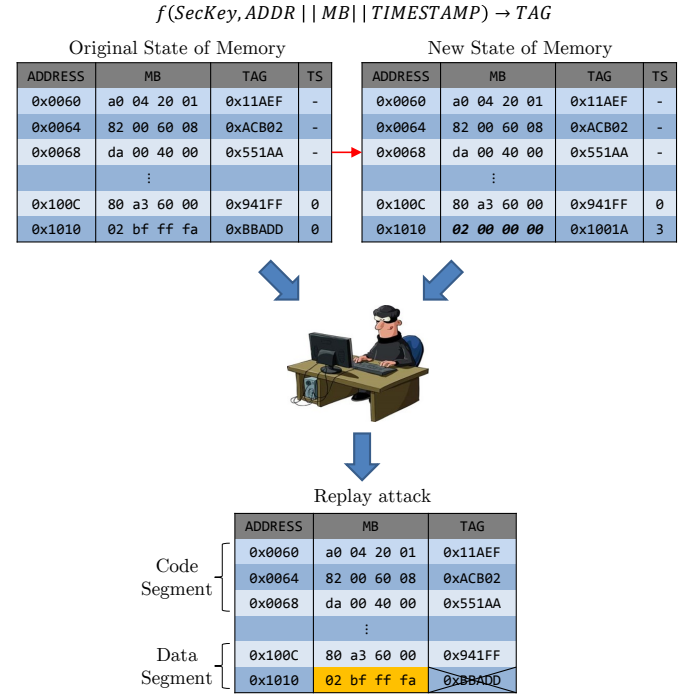


Figure 4: Replay attack prevention using Timestamps.

2.3.2. Preventing Splicing or Relocation Attacks

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

2.3.3. Preventing Replay Attacks

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

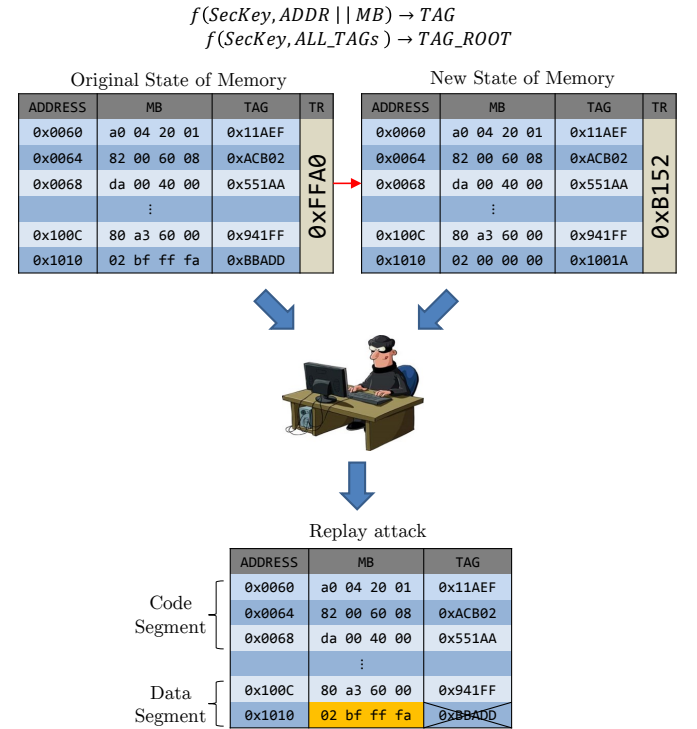


Figure 5: Replay attack prevention using Merkle Tree.

It is worth noticing that a root tag can degrade performance

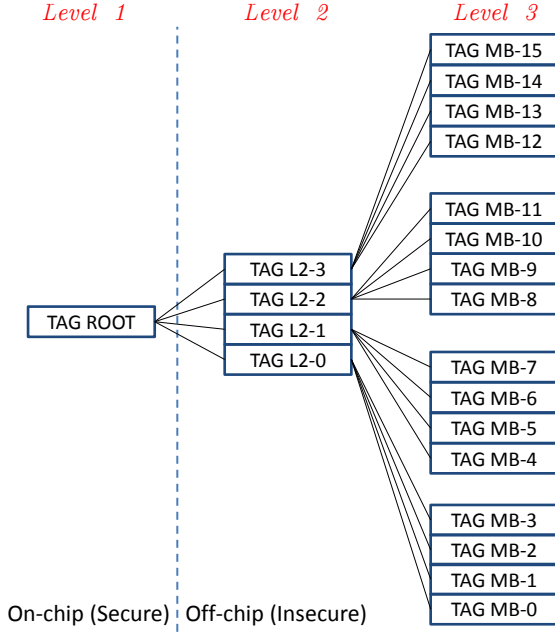


Figure 6: Merkle Tree structure to 16 memory blocks.

significantly, to a point in which can be impractical. To minimize such impact on performance when using a root tag, a tree structure can be applied. This tree is known as Merkle Tree. The literature has proposed a vast number of approaches of how to implement this tree, and details can be found in [12, 13]. The tree is constructed by grouping tags in chunks and creating tags for them. By recursively repeating this procedure until only one tag is left, we obtain the root tag. Figure 6 illustrates this process, in which a chunk is composed of 4 tags of memory blocks, and the next level of the tree has 4 tags that originated from each descendant chunk. The root tag results from the chunk created by the tags in the second level. Notice that all intermediate tags between the root tag and the leaves may be stored off-chip. The root tag is the only one that cannot be exposed.

Even though the tree seems to be an elegant solution to reduce performance penalties of the root tag solution, usually it is not enough. Because every read and write of memory blocks will require verification and update of the nodes in the leaf-root path, an on-chip cache for nodes of the Merkle Tree can be implemented to avoid constant complete walks from leaves all the way to the root. More details can be found in [14, 15]. Additionally, optimizations have been proposed [16] to reduce tree height, which can lead to even better performance. Finally, Merkle Tree can present consistency issues due to interruptions, like power outages, attacks, etc., during updating nodes of the tree. Some approaches have tried to solve this issue [17], however, to the best of our knowledge, the usage of cache still brings challenges to consistence that have not been addressed yet.

Therefore, the dilemma is: timestamps are likely to provide lower performance penalties since verification is simple, but they can require larger non-volatile on-chip memories. On the other hand, Merkle Tree may only need a non-volatile on-chip register, but in order to reduce performance slowdown a cache

should be added, which will adversely affect dynamic power and area. Further details in the downsides of both solutions are discussed in depth in the next sections.

2.4. The Matter of Confidentiality

Although confidentiality mitigates some attacks by simply concealing the memory contents, many embedded system applications do not need it. For instance, applications that use open source programs and/or have data that must be audited at some point. In general, encryption primitives are slower than authentication due to their reversible nature. Different secure architectures that implement confidentiality using AES report memory block encryption latency between 13 [18] to 20 cycles [3] without estimating real impact on area and power. In [19], a 16-byte block is assumed to be encrypted in 10 cycles using AES with an estimated area of 25,000 gates. However, a Pseudo-Random Function (PRF) such as SipHash [20], can be implemented in 7,900 gate-equivalent elements capable of processing 20 bytes in 5 cycles. Therefore, embedded systems can have authenticity and integrity with very low performance penalties, when secrecy is not needed. Next, we present the CSHIA architecture, which assuming the threat model and security issues and solutions discussed in this section, provides authenticity and integrity for code and data.

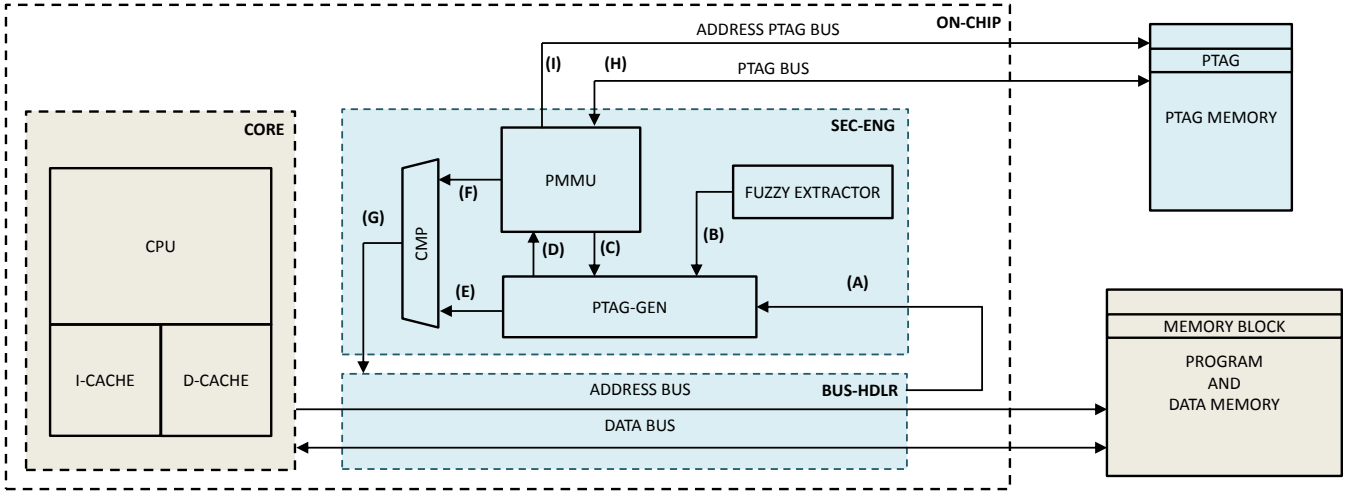
3. CSHIA

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

3.1. Components of the Architecture

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

PTAG Memory is an external memory and has its own buses. This architectural decision gives freedom to designers who can choose bus width, frequency, address space, etc. Because the processor is not aware of any additional component of CSHIA,



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

Figure 7: The CSHIA architecture.

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 is responsible to extract the PUF-based key and provide it to PTAG-GEN. Finally, we have the PTAG Memory Management Unit (PMMU). The main functions of PMMU are to store and request PTAGs from 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 using a cache memory, the PTAG Cache, to speed up these tasks.

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.

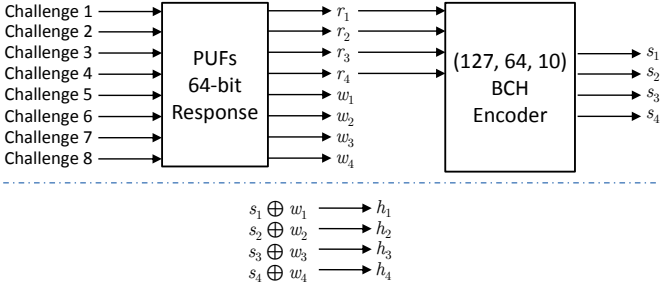
For write operations, the process is simpler. Once any memory block that reaches the processor was verified for integrity and authenticity, BUS-HDLR can serve the cache line to SEC-ENG that uses PTAG-GEN to compute a new PTAG and PMMU sends that PTAG to PTAG Memory. We can see PMMU for now as a black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

3.2. Enrollment Phase

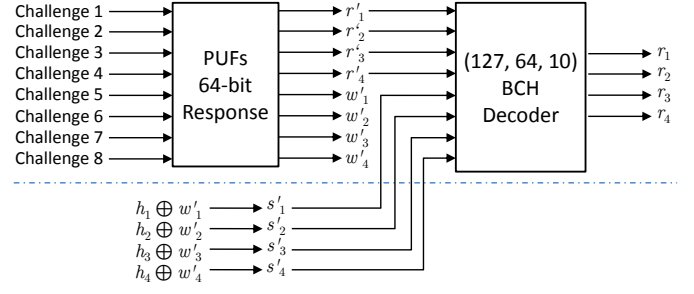
In order to ensure authenticity and integrity, an initial procedure has to be conducted by the manufacturer/vendor. This enrollment procedure will activate the Fuzzy Extractor to extract the secret key from PUFs. Once that is done, the BUS-HDLR brings all memory blocks for tag generation. Next, we detail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

3.2.1. Key Extraction

PTAG Generator implements a Pseudo-Random Function (PRF), which is a primitive cryptographic very similar to a hash



(a) Fuzzy Extractor during key extraction.



(b) Fuzzy Extractor during key regeneration.

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

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 secure sketches comprised of an extraction algorithm and a recovery procedure. Becker provides a solid review and formal definitions in [21].

There are multiple ways of implementing a Fuzzy Extractor. It is common to employ in FE designs an Error Correction Code (ECC) (n, k, t) , where n is the codeword length, k is the data length, and t is the maximum amount of errors that can be corrected. Originally, CSHIA was proposed using a Code-offset FE, which is well-known to reduce entropy of extracted keys [22] since, despite the key having n bits, an attacker only needs to figure out k bits. To strengthen the CSHIA design, we looked at Pin Sketches [23] and Index-based Syndrome (IBS) FE proposed by Yu and Devadas in [24]. Pin Sketches deploy an ECC but make only the syndrome $(n - k)$ bits public. That is very secure because a random data generates a random syndrome value [25], and thus figuring out the data from the syndrome is computationally hard. On the other hand, the IBS FE encodes the whole codeword using indexes that are extracted from PUFs responses. The most simple index would be the one in which a PUF response 0 represents a negative index value and 1 represents a positive index value. This simplification is similar to the XOR operation used to produce a Code-Offset. Yu and Devadas on their proposal use a set of positive and negative values extracted from PUFs and choose the minimum value of the set for a key bit 0 and maximum value for key bits 1.

The implementation of Fuzzy Extractors is illustrated in Figures 8 and 9. As mentioned previously, FE are comprised of an extraction algorithm (Figure 8 (a)) and recovery procedure (Figure 8 (b)). In Figure 8 (a) the process of key extraction of CSHIA's FE consists in extracting 8 bit strings from PUFs. The

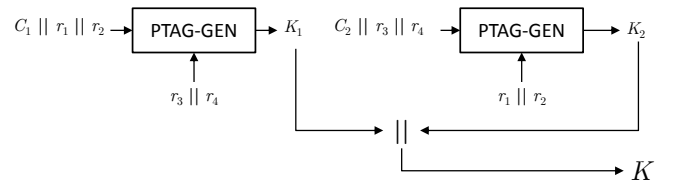


Figure 9: Key generation on CSHIA.

strings r_i will be employed in the key generation. The strings w_i will be used to encode the syndromes s_i derived from r_i . We use (n, k, t) -BCH to generate each s_i that is combined in with w_i in a one-time pad encryption. Each combination results in a helper data h_i , which can be externally exposed and will not leak information about r_i , which is the critical information for the system.

In terms of parameters, CSHIA incorporates 64 APUFs to produce 64-bit responses. To each generate each 64-bit word, 8 challenges are used. To match the length of r and w , CSHIA has a $(127, 64, 10)$ -BCH ECC. Once all helper data are generated, the bit strings r_i are compounded two by two and fed to the PRF, as Figure 9 shows. Such combinations were specifically designed to match the PRF chosen for CSHIA, the SipHash [20], which has an output of 64 bits and uses key of 128 bits. Therefore, the first pair of bit strings r_i is concatenated with a constant and processed by the PRF using the second pair of bit string r_i as key. That generates a hash K_1 . Then, inverting their places and concatenating the second pair with a different constant, a hash K_2 is obtained. Concatenating K_1 with K_2 results in K which is the secret key of CSHIA. Notice that C_1 and C_2 in Figure 9 are replacing addresses for input of the PTAG-GEN. Further details of security will be given in the following sections, however, one can notice that assuming that each bit string r_i has at least half of their length of entropy, each part of the key will have full entropy. Hence, the key has full entropy.

3.2.2. Full Memory Protection

The Enrollment Phase proceeds to tag the memory range the manufacturer/vendor specified during design. Now that PTAG-GEN has 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, however, depending on the solution against replay attacks a designer chooses, PTAG-GEN is used differently.

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

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

3.3. Runtime Phase

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

3.3.1. Key Regeneration

During the enrollment there were 8 challenges selected to produce four r_i and four w_i values. These challenges and helper data can be exposed off-chip and stored in PTAG Memory if

the designer chooses to do so. The recovery process of the secret key can be seen in Figure 8 (b). After using the challenges and all helper data, the syndromes are recovered. Due to inconsistent nature of PUFs, the fuzzy extractor actually recovers bit-flipped versions w'_i and r'_i , what leads the BCH decoder to receive r' and s' . Once bit-flips in r'_i values are corrected, the FE uses all r_i to regenerate the secret key as Figure 9 shows.

3.3.2. Runtime Protection using Timestamps

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

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

3.3.3. Runtime Protection using Merkle Tree

In CSHIA, the basic difference between using timestamps and Merkle Tree relies on the work PMMU has to do. For a solution based on Merkle Tree, PMMU has a small PTAG cache memory that only caches PTAGs of data memory block and its ancestors in the tree. Nonetheless, the process in which the PMMU makes PTAGs available becomes very complex. For the sake of simplicity, we provide a flowchart of the PMMU's work in Figure 10. The same process described for read and write in the runtime protection using timestamps is also valid here.

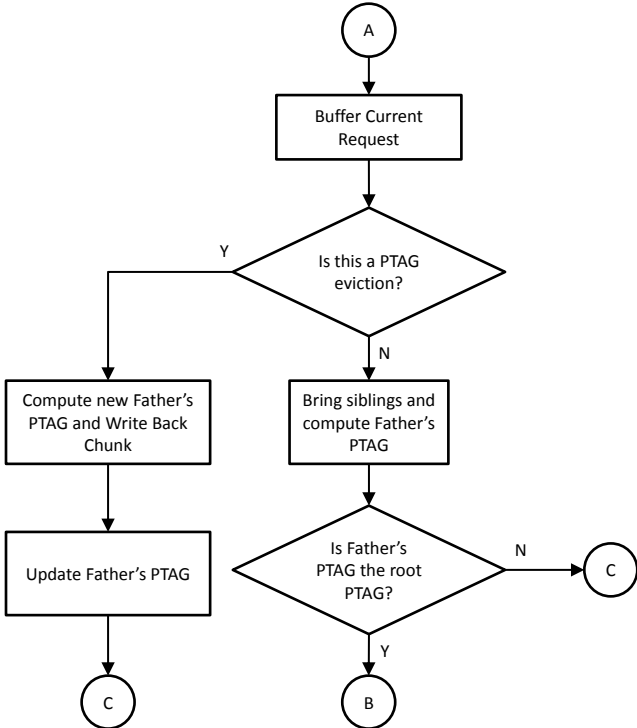
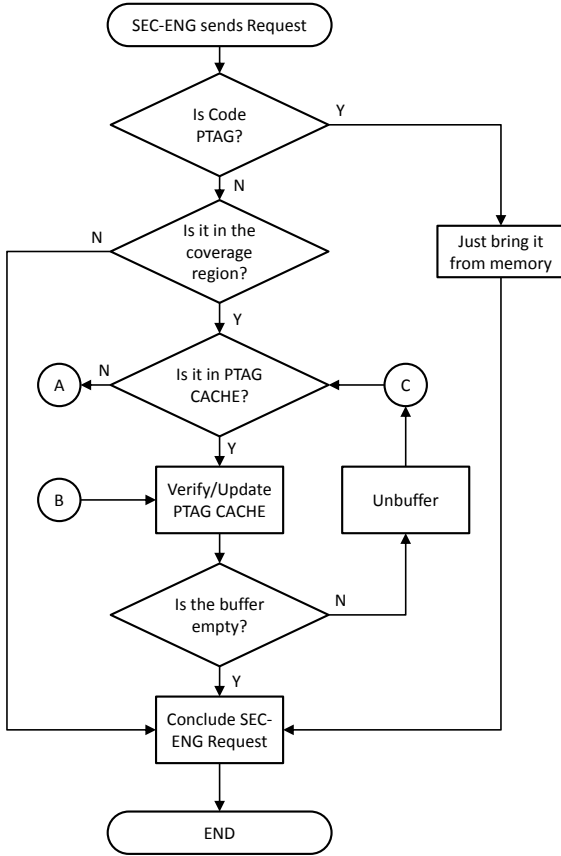


Figure 10: Flowchart of the PTAG Memory Management Unit's algorithm for verification and update of the PTAG Cache for the Merkle Tree implementation.

That means that the work of BUS-HDLR during processor's read/write requests is the same.

To summarize Figure 10, PMMU has a buffer to solve deadlocks when replacing dirty chunks of PTAGs in the cache. Most of those deadlocks happen because right after evicting a chunk, its ancestor has to be updated and thus the current write or verification has to be stalled. For intelligibility, we omit further details of implementation, they will be found in our source code and corresponding documentation about of the CSHIA implementation [26]. Finally, information about caching Merkle Tree nodes can be found in [14].

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

4. Implementation

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

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

We implemented CSHIA in an Altera FPGA Development Kit DE2-115. The parameters of the processor and CSHIA are in Table 1. The Altera's kit allows the processor to run at 50 MHz. The total amount of SDRAM memory dedicated to Leon3 is 128 MB. As convention, all programs starts by its .text segment (code) at the address 0x40000000. We set .data segment (data) to start at 0x40013000, or at 0x40023000, depending on the size of the code segment. As described in the previous section, BUS-HDLR has a buffer that stores memory words. When these words form a memory block, it is handed to SEC-ENG. We set the size of this buffer to 4 cache lines, which gives a total of 128 bytes. The 128-bit SipHash's key is extracted from 64 Arbiter PUFs (APUFs). Although any PUF could be used, for the sake of design simplicity, we chose the APUF as a proof of concept. Each APUF has a 64-bit challenge input. PTAG generation lasts 10 cycles, between SEC-ENG request and PTAG-GEN reply. Our SipHash

Table 1: CSHIA FPGA implementation configuration.

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

implementation is an adaption of the VHDL version available in [27].

Continuing to look at Table 1, the PTAG Memory uses internal memory of the FPGA. This option arose due to limiting options available in the kit. Because we wanted to design a 64-bit bus memory, no better option than internal memory was available. The SRAM of the kit only allowed 16-bit words. We also could not increase the frequency of the SRAM using PLLs since its maximum frequency was limited to 125 MHz, and, to simulate a 64-bit bandwidth, we would need a SRAM operating at at least 200 MHz. The option for FPGA internal memory limited our coverage to a maximum of 512 KB of data memory. The PTAG Memory’s overhead was $\approx 36\%$ (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 2¹⁴ 16-bit words to cover the 512 KB of data memory. This represents 5.4% of the total 588 KB main memory coverage (code and data). The PTAG Memory’s overhead for this solution was 25% overhead. Table 1 also shows the configurations of PTAG Caches. 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 to support the Merkle Tree implementation. Being able to switch between those modes only using switch keys of the development key helped us in debugging and evaluating the performance of the architecture.

5. Experiments and Results

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

5.1. Experimental Setup

The DSU connects the Leon3 platform to a computer through a debugging program called GRMON. Using GRMON, we are able to load programs, measure runtime, insert breakpoints, and set some Leon3 parameters. As benchmarks, we chose nine programs from the MiBench suite [28]: basicmath; bitcount; dijkstra; fft; fft_inv; qsort; sha; susan; stringsearch (or just search for short). These benchmarks were either executable without input files or easily modified to run without them, since we could not load input file through GRMON. Thus, for some benchmarks we incorporated input files in their data segment, and these modifications were evaluated against reference outputs. MiBench usually provides two types of inputs: small and large. We ran both inputs for most of the benchmarks, except by basicmath, fft, and fft_inv. The large inputs of these programs did not affect the size of the data segment and yet most of their run time was dominated by printing their outputs over GRMON.

As Section 4 discussed, the CSHIA implementation is able to cover up to 512 KB of data. This was enough for most of

Table 2: Coverage of data segment in benchmarks.

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

the benchmarks except by the large inputs of `qsort` and `sha`, as Table 2 shows. Only the `.data` and `.bss` segments of the programs were covered. We did not have enough memory to reach the beginning of the `.stack` segment and we would only be able to cover a small portion of `.heap` segment.

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

5.2. Performance Analysis

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

They presented similar behavior of the `sha` benchmarks, despite CSHIA almost entirely covers the data segment of `qsort_large`.

The results for *CSHIA-MT* show that increasing the number of sets and reducing the number of cache lines seems to provoke a performance penalty. The average performance of *CSHIA-MT-16x8-ALAP* and *CSHIA-MT-16x8-LRU* are worse than *CSHIA-MT-64x2-ALAP* and *CSHIA-MT-64x2-LRU*, respectively, with 7.05% and 5.99% against 5.99% and 5.77%. Therefore, cache memory for *CSHIA-MT* is likely to improve performance if a designer chooses to add more cache lines than sets. For cache replacement policies, we can observe in Table 3 that LRU has better overall performance than ALAP. These numbers do not invalidate the results of Su *et al.* in [15], but we would recommend pure LRU as replacement policy.

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

5.3. Area and Power Estimates

For area and power estimates we used Altera’s Quartus synthesis report and Cadence Genus. Some changes in design were needed. For instance, we made separated VHDL code for each CSHIA instance. Besides, we noticed that our buffer for the Merkle Tree cache was not been synthesized as memory, but registers instead. That made an enormous difference between *CSHIA-MT-16x8* and *CSHIA-MT-64x2* estimates, favoring significantly the first and thus leading to an incorrect analyses. Because of that we set that buffer as two lines for these estimates. In the following, we discuss each estimate.

5.3.1. Altera Estimates

We separated these estimates into the main components. In terms of area, Quartus gives us the total of combinational and

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

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

Table 4: FPGA area and power estimates of CSHIA.

Instance	Logic Cells	Static Power (mW)	Dynamic Power (mW)
<i>Leon3 Baseline</i>	29240	55.17	240.64
<i>Common CSHIA Components</i>			
BUS-HDLR	≈ 5153	-	≈ 18.34
SEC-ENG			
FE + APUF	≈ 4034	-	≈ 16.87
PTAG-GEN	≈ 2348	-	≈ 10.71
PTAG Memory	-	-	≈ 63.84
<i>CSHIA-TS</i>			
SEC-ENG	6930	-	40.27
TS Memory	-	-	8.16
Total	40659	62.48	≈ 364.43
<i>CSHIA-MT-64x2</i>			
SEC-ENG	21642	-	105.48
Cache	9635	-	46.59
Total	55193	64.11	≈ 434.97
<i>CSHIA-MT-32x4</i>			
SEC-ENG	21801	-	125.23
Cache	9796	-	65.45
Total	55318	64.11	≈ 454.21
<i>CSHIA-MT-16x8</i>			
SEC-ENG	22768	-	171.98
Cache	10754	-	107.74
Total	56275	64.11	≈ 496.69

Table 5: CSHIA’s area and power estimates from Cadence Genus.

Instance	Area mm ²	Static Power (mW)	Dynamic Power (mW)
<i>Leon3 Baseline</i>	0.34	0.17	14.33
<i>Common CSHIA Components</i>			
BUS-HDLR	0.01	0	0.30
SEC-ENG			
Fuzzy Extractor	≈ 0.04	≈ 0.01	≈ 1.26
APUFs	≈ 0.02	≈ 0	≈ 0.89
PTAG-GEN	≈ 0.01	≈ 0	≈ 0.11
<i>CSHIA-TS</i>			
SEC-ENG	0.05	0.01	8.40
TS Memory	0.03	-	7.07
Total	0.43	0.18	23.03
<i>CSHIA-MT-64x2</i>			
SEC-ENG	0.24	0.09	8.28
Cache	0.18	0.08	6.49
Total	0.59	0.26	22.91
<i>CSHIA-MT-32x4</i>			
SEC-ENG	0.24	0.09	8.58
Cache	0.18	0.08	6.85
Total	0.59	0.26	23.21
<i>CSHIA-MT-16x8</i>			
SEC-ENG	0.24	0.09	9.27
Cache	0.18	0.08	6.59
Total	0.59	0.26	23.90

register logic cells used in FPGA, and we consolidated those number into one. The estimates are shown in Table 4. We can see that all CSHIA instances have noticeable cell number and power increment in regard to the baseline. However, our main question in this analysis is to identify which configuration presents the best trade-off among high performance, low area, and power consumption. Based on the results, *CSHIA-MT* is the one that provides it. Of course, FPGA estimates are taking PTAG Cache and Timestamp memory as the same kind, which in real implementations is probably not the case.

5.3.2. Cadence Genus Estimates

In these estimates we used 45nm Generic Standard Cells library without SRAM libraries. Thus, cache memories are estimated as registers. This scenario favours the *CSHIA-TS* configuration because we cannot compute area and power for the internal memory. To do that we used CACTI 6.5 [29]. It is important to notice that CACTI 6.5 failed in estimating the PTAG Caches due to them being too small. Observing Table 5, the conclusions drawn from Table 4 do not change significantly. Adding PTAG Caches increases power consumption and area. Even though those estimates did not use SRAM standard cells library, cache memories contribute to noticeable increments on power and area [30].

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, *CSHIA-TS* would be the best solution for low-power applications. Of course, that would only apply to this specific memory size we evaluated. As we will discuss in the next section, 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 on-chip power and area. Nonetheless, higher penalties in performance would happen. Among all *CSHIA-MT* instances, we would definitely indicate *CSHIA-MT-64x2-LRU* as the best choice due to its performance and smaller area and power overheads.

Possible improvements in area and energy can come from increasing the length of memory blocks. For instance, changing memory blocks from 256 to 512 bits would reduce the number of PTAGs and consequently the timestamp memory size. The impact on PTAG Caches would need further evaluation though. The obvious drawback is performance reduction, as it would take longer to generate PTAGs using SipHash. One will be able to conduct such investigations with our FPGA implementation.

6. Security Analysis

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

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

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

6.1. Brute-force/Forgery Attacks

Scenarios where an attacker tries to bypass CSHIA integrity verification by inserting manipulated PTAGs have a small probability of being successful. PTAGs cannot be directly read by any program and thus, unless the PTAG Memory is externally inspected and manipulated, or reverse-engineered, an attacker cannot modify its contents. A tampered PTAG Memory poses the same threat as tampered PTAGs inserted into the PTAG Buses. Tampered PTAGs will only match tampered memory blocks if the attacker has the same PUF-based key of the CSHIA instance he/she is trying to attack. This can only happen if he/she guesses the key which has a probability of 1 in 2^{128} of happening, or if the attacker guesses PTAGs correctly for the tampered memory blocks. That has a probability of 1 in 2^{64} for an individual memory block. Although, this value does not match the standard security level for personal computers, servers, and high performance systems (see NIST report in [31]), they can significantly decrease threats to embedded systems, requiring more resourceful attackers and reducing chances of successful attacks.

6.2. Modeling attacks

The seminal work of Rühmair *et al.* [32] showed that APUF and its variations are easy to model using machine learning. One would argue that the FPGA implementation of CSHIA is weakened due to the usage of APUFs to generate the PRF's secret key. However, the key point in which CSHIA relies is that there is no PUF response available to an attacker. Although the challenges that are used to extract the key and its syndrome can be made externally available, they do not provide enough information to create a model of an APUF. Becker *et al.* state that the number of CRPs to model an APUF is approximately $0.5 \cdot (k+1) \cdot \epsilon^{-1}$, where k is the number of stages of APUF and ϵ is the error rate [33]. Thus, exposing only 8 challenges without responses, as shown in Section 3, does not enable accurate modeling. However, Becker *et al.* showed that IBS Fuzzy Extractor can result in entropy loss for the key. That is discussed in the next subsection. Finally, modeling PTAGs generation is not possible because SipHash was evaluated to be indistinguishable from a uniformly random function [20]. Therefore, such model would have success equivalent to brute-force attacks.

6.3. Fuzzy Extractor Attacks

Assuming that an attacker would not be granted access to the system during its enrollment phase, CSHIA's FE is secure. Let us break down a variety of attacking scenarios:

Exposing Challenges and Encoded Syndromes. An important matter in this regard is the number of challenges exposed. Becker *et al.* show that exposing challenges can enable an attacker to recover bits of the key [33]. However, our Fuzzy Extractor does not directly or indirectly expose bits of the key. What Becker *et al.* explores from the proposed IBS of Yu and Devadas [24] is the availability of the encoded *codeword* (not only the syndrome) which has embedded bits of the key. Our Fuzzy Extractor exposes the encoded syndrome only. In fact, there is no need in exposing the key anyhow [23]. Thus, we derived from IBS the concept of encoding with positive and negative numbers, but it can be reduced to the Code-Offset technique when only + and - are used as encoders. Recalling that code-offsetting using a unique and random binary word is equivalent to a one-time pad encryption, which is information-theoretically secure [34]. Of course, if an adversary is still able to deploy Becker *et al.*'s attack [33] and unveil all syndrome values, he/she can compute all 2^{64} bit strings and their syndromes and match them with those he/she unveiled. However, from Becker *et al.*'s results, we believe that could be hard. The attacker would have only 4 encoded bits per PUF to model since CSHIA uses only 4 challenges in 63 APUFs to generate 63-bit of encoding. Any significant result for 4 secret bits in Becker *et al.*'s work uses more than 4 challenges. In conclusion, even if an attacker finds the syndromes, which we believe is hard to do, there is still an attack complexity of 2^{64} .

Entropy Loss. Using a $BCH(n, k, t)$ in a FE leads, in general, to $n - k$ bits of entropy loss [35]. This happens because attackers may generate every possible 2^k bit string, and their $(n - k)$ -bit syndrome, to extract each possible key from the helper data, thus resulting in 2^k attack complexity. Now, assuming that all CSHIA's PUFs are unique at each instance, further entropy loss due to bit error or correlation analyses [35] cannot happen in the CSHIA' FE, as CRPs are unavailable to adversaries. What is more, if the attacker cannot find the syndrome values from the helper data to figure out the key, which we argued before would not be as easy as Becker *et al.* showed, the attacker has to test 2^k keys among 4 combinations of key generation. This is the case because our key generation uses four k -bit strings (see Figure 9). If we consider that t bits of minimum distance between bit strings enable error correction, an attacker has to actually produce $k - t$ bits for each part of key, or $64 - 10 = 54$ bits in our case. So, without any knowledge of the syndromes, the attacker has to test $(2^{k-t}) \cdot (2^{k-t} - 1) \cdot (2^{k-t} - 2) \cdot (2^{k-t} - 3)$ possibilities. That is harder than trying to guess the 128-bit SipHash key.

Fault Injection. Some known fault injection attacks on our Fuzzy Extractor can have reduced effectiveness. First, the attacker can flip bits of the helper data to perform a side-channel

attack. Merli *et al.* [36] showed how that could be done to recover the key. In our FE design, such technique would only be effective to recover the syndrome. As key bits are not directly affected by helper bit-flips. If syndromes are unveiled, the attack complexity falls to 2^k (that is, 2^{64}) since the attacker only needs now to match each of the 2^k syndromes with those they found. Nonetheless, this is harder than the traditional Code-Offset FE offers. In a second maneuver, the attacker can flip helper data bits to apply a side-channel attack in the BCH decoding [37]. However, as pointed out by Karakoyunlu and Sunar, it is possible to implement BCH decoders to resist this side-channel attack. Then again, we must clarify that this is not a claim of side-channel attack mitigation. Lastly, a different fault injection attack has been recently described by Becker [21]. This attack consists in changing the helper data in order to force BCH decoders to fail and set bits in the codeword. To succeed, such an attack depends on the ECC design. In our VHDL BCH implementation, we performed waveform simulation of 10 bit-flips or more in the syndrome, and data outputs showed no modification. Therefore, this attack would not be successful. The FPGA CSHIA implementation uses a VHDL code extracted from the VHDL BCH generator available in [38].

6.4. Memory Integrity Attacks

We discussed in Section 2 the three main attacks against main memory: *spoofing*, *splicing* or *relocation*, and *Replay* attacks. The capability of CSHIA preventing spoofing and splicing attacks depends on the length of PTAGs. An attack is only successful if the attacker guesses a PTAG, which has probability 1 in 2^{64} of happening.

For replay attacks, we showed two solutions. The first solution was the *CSHIA-TS* instance, which had 16-bit timestamps for each data memory block. That means that after 2^{16} memory writes of a specific memory block its timestamp would be reset, enabling a replay attack. Recall that our integrity protection covered 512 KB of data, which demanded a 32-KB timestamp memory. If we, for instance, set timestamps to 64 bits, we would need 128 KB of internal memory. However, notice that 512 KB did not cover all the data memory of some programs. For instance, the *sha_large* had more than 3 MB of data memory, to cover this with 64-bit timestamps the non-volatile timestamp memory size goes over 512 KB. In this scenario, it is possible that the analyses of the trade-off between security, performance, area, and power favours *CSHIA-MT*.

For our tested instances of *CSHIA-MT*, the 64-bit PTAGs results in 2^{32} attack complexity for replay attacks. This reduction is due to Birthday Paradox [39]. It is known that for any collision-resistant hash function, an attacker can find a collision (with a coin toss probability) between two values if he/she collects at least square root of the total possible outcomes of the function. Although a PRF should not have this problem if the key is changed periodically, because CSHIA's secret key is fixed for its lifetime, the PRF is reduced to an unknown hash family. Hence, to find a collision in CSHIA FPGA implementation with 50% probability, an adversary would need to collect 2^{32} memory blocks of 32 bytes written back to the *same memory address*. This means that, in the average case scenario, the

attacker might find a collision after collecting 32 GB of data of one specific memory address, besides collecting all PTAGs in the tree that relate to this address as well. And it is crucial to notice that, in this scenario, the attacker does not choose values, the 50% probability regards to two random values found in the 32 GB data collected. This attack is unequivocally feasible, but requires significant resources.

Finally, of course, using 128-bit PTAGs we can increase CSHIA robustness against replay attacks, but to keep the same size of PTAG Memory in relation to the main memory, memory blocks and chunk of PTAGs would have to be 512 bits. We discussed such an impact on design in the previous section. Then again, 128-bit PTAGs would need another PRF since SipHash does not meet this specification.

7. Related Work

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

In 2003, Yang, Zhang, and Gao [40] proposed an improved version of XOM, an architecture for digital copyright protection. The architecture provides authenticity through a pairwise private/public key. Every instance of a XOM architecture has a unique private key. Secrecy is provided by encrypting software using specific symmetric keys chosen by the software vendor. These keys are encrypted using XOM's public key and therefore only the instance that has the correspondent private key will be able to execute the software. XOM provides integrity protection by hashing memory blocks, but it only prevents spoofing and splicing. Replay attacks are left uncovered. The differential of XOM is to isolate programs in compartments, which have their own tags and keys. Due to this isolation, new instructions had to be added to the processor in order to relax constraints of the architecture. Enabling data share between isolated programs would be an example of that. From the point of view of targeted market, XOM is suitable for very high end embedded systems or above since they simulated their architecture using a processor capable of out-of-order execution and XOM's performance highly depends on the existence of second-level cache (L2) and its size. Finally, area and power overheads are not estimated, implementation is through simulation, and the averaged performance slowdown is 16.76% on the tested benchmarks, however, they were able to reduce this slowdown to 1.28% implementing an additional cache memory.

The first architecture that proposed to use PUFs for key generation was AEGIS. The work presented by Suh *et al.* [1] is a tamper-resistant and tamper-evident architecture. That is, it hinders tampering threats, and even if an attacker successfully overpasses security, it can still indicate that. AEGIS is a complete solution in which not only new instructions are provided, but also system calls, security modes, and different divisions of memory into new regions. It can securely run even under an untrusted operating system. AEGIS provides a Merkle Tree to prevent replay attacks and uses a small cache to store nodes and reduce performance penalties. All this security comes with

downsides such as an almost 100% area increase (in comparison to the baseline system), and modifications of the entire toolchain: compilers, operating systems, and even programs. Due to the complexity of the architecture, targeted systems are preferable high-end embedded systems or above. Although the authors are not very clear about overall performance penalties of the architecture, when they used their full protection mode, a 32-KB instruction/data cache, and 16 KB Merkle Tree cache, the worst benchmark performance overhead was 3.3%. However, when the architecture configuration is 4 KB instruction/data cache and 2 KB Merkle Tree cache, the same benchmark has a performance overhead of 73.1%.

Rogers, Milenković, and Milenković presented in 2007 a secure architecture [41]. Different from the previous works described above, they truly focused on embedded system since they assume that processors would not have second-level cache memory (L2), but would present a separated L1 into data and instruction memories. Their architecture provides integrity and secrecy for memory blocks of instructions, and data integrity is not discussed. The architecture uses virtual address to compound encrypted blocks, in order to thwart splicing attacks, they came up with a interesting solution of encrypting an unique PUF-generated (or thermal-noise generated) key together with the program this key authenticates. Thus, when two programs present a collision between virtual addresses, an attacker will not be able to switch programs because their key will be different. Although their architecture was only simulated, they estimated a power consumption overhead, which reached as high as twice the baseline's. In terms of performance penalties for the tested benchmarks, the results also were very detrimental for a small instruction L1, achieving an overhead of 2 times greater than baseline's performance, and becoming negligible for a 8 KB instruction cache. At last, for a standard memory block of 256 bits, their storage overhead reached 50% of the main memory, which is high.

In 2009, Vaslin *et al.* proposed a security approach for off-chip memory in embedded microprocessors [2]. Vaslin *et al.* used the One-Time-Pad (OTP) scheme to provide integrity and secrecy. Their architecture encrypts timestamp, 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, FPGA area overhead almost tripled.

FEDTIC, the 2010 work of Hong and Guo [3], is an architecture for integrity verification and secrecy for embedded systems. Their main contribution is a unique engine that uses one AES hardware instance for encryption, decryption, and tagging of cache lines. Stamps are used to prevent replay attacks. However, instead of a one-time-pad scheme, Hong and Guo used AES in output feedback mode which allowed them to use

shifted encrypted blocks to compose a tag. In terms of achievements, for a 512 bit cache lines, their external memory overhead was less than 7% and for the tested benchmarks a maximum internal memory for timestamps needed was 5 KB. We should notice that this internal memory is not a cache and thus will increase with the program size. The average performance penalty was 7.6% and the maximum 30.72%. Their evaluation used a combination of simulation and FPGA implementation.

Bobade and Mankar presented in [4] a secure architecture for embedded system. Their architecture provides integrity and secrecy through an Elliptic Curve Cryptographic engine. The main difference regarding the other architectures presented here is that they use the timestamps as private keys. Thus, cache lines are encapsulated with their address and timestamp, 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 reprocessing the pair private/public key and the integrity is ensured by the correct decryption of the triad encapsulated: data, address, and timestamp. 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 has proposed SEPUFSoC a Multi-Processors System-on-Chip that provides memory integrity and authenticity through PUFs[42]. The proposed architecture innovates by targeting multi-processors. They also used SipHash to provide integrity tags to memory blocks and protection 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 long periods 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 believe that this work demonstrates that only minor modifications are needed and they are all transparent to the core and do not affect how it works. It is also important to notice that the authors chose a similar Code-offset Fuzzy Extractor which CSHIA had originally employed. As previously discussed, it is less secure than the one we currently propose because it reduces entropy of the key. Finally, they estimated area and power of the components of their architecture. And performance evaluation presented a worst case scenario of 25.4 %.

Table 6 presents a summary of most positive features and downsides of CSHIA and related works. A fair comparison of performance among the works is quite hard to achieve, due to a variety of benchmarks, baseline cores, overhead metrics, etc. However, a qualitative analysis of design choices can still be done. For instance, PUFs have been constantly claimed to be

Table 6: Summary of Related Works in comparison with CSHIA.

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

a better solution for key generation than storing on-chip key. In that regard, our implementation is more advantageous than those that did not use them. Moreover, we carefully analyzed major threats presented in the literature in order to propose a secure employment of a PUF-based key. Because embedded system applications can have a very specific nature, our concern since the beginning was to propose a flexible architecture, which is characterized by its additional and independent PTAG Bus and Memory that allow CSHIA's performance to have low overhead with integrity verification of code and data memory blocks. Furthermore, an architecture that can work with Timestamps or Merkle Tree without the need of deep changes or redesign is new in the literature, to the best of our knowledge. Thus, given our results and comparing what related works have to offer, we believe that we presented a solid solution for the security of embedded systems.

8. Conclusions

The main goal of this work was to present a proof-of-concept implementation of CSHIA, evaluating performance, area and power. Throughout the text, we described an in-depth implementation and evaluation of the architecture. We achieved a flexible solution for embedded systems that allows designers to: (1) specify a bus and memory configuration that provides integrity; (2) select any PUF design that meets security constraints; (3) choose between two countermeasures against replay attack. All that being completely transparent from other components in the chip. In terms of performance, CSHIA Timestamp instance showed an average 2.76% performance overhead, while the fastest CSHIA instance with Merkle Tree showed an average 5.77% performance penalty. Taking into account our estimative for area and power consumption, and the comparison against related work, the *CSHIA-TS* instance seems to be a great option for secure embedded systems. Finally, we provided an in depth security analyses, as well as a discussion of relevant works in literature. CSHIA's FPGA implementation has a preliminary version available for the research community here [26].

Acknowledgment

The authors would like to thank the São Paulo Research Foundation (FAPESP), grant 2015/06829-2, the National Council for Scientific and Technological Development (CNPq), grant 308736/2017-6, and Intel, grant "Energy-Efficient Security for SoC Devices – Physical Unclonable Functions", for supporting this work.

References

References

- [1] G. E. Suh, C. W. O'Donnell, I. Sachdev, S. Devadas, [Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions](#), in: Proceedings of the 32Nd Annual International Symposium on Computer Architecture, ISCA '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 25–36. doi:[10.1109/ISCA.2005.22](https://doi.org/10.1109/ISCA.2005.22). URL <http://dx.doi.org/10.1109/ISCA.2005.22>
- [2] R. Vaslin, G. Gogniat, J.-P. Diguët, E. Wanderley, R. Tessier, W. Burleson, [A security approach for off-chip memory in embedded microprocessor systems](#), Microprocessors and Microsystems 33 (1) (2009) 37 – 45, selected Papers from ReCoSoC 2007 (Reconfigurable Communication-centric Systems-on-Chip). doi:<https://doi.org/10.1016/j.micpro.2008.08.008>. URL <http://www.sciencedirect.com/science/article/pii/S0141933108000823>
- [3] M. Hong, H. Guo, FEDTIC: A Security Design for Embedded Systems with Insecure External Memory, in: T.-h. Kim, Y.-h. Lee, B.-H. Kang, D. Šlizek (Eds.), Future Generation Information Technology, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 365–375.
- [4] S. D. Bobade, V. R. Mankar, Developing Configurable Security Algorithms for Embedded System Storage, International Journal of Computer Science and Telecommunications 6 (7).

- [5] t. C. Hoffman, M. Cortes, D. Aranha, G. Araujo, Computer security by hardware-intrinsic authentication, in: Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2015 International Conference on, 2015, pp. 143–152. doi:10.1109/CODESISSS.2015.7331377.
- [6] LEON3 Processor - Gaisler, <http://www.gaisler.com/index.php/products/processors/leon3>, website. Accessed: 08-Feb-2017.
- [7] B. Gassend, D. Clarke, M. van Dijk, S. Devadas, **Silicon Physical Random Functions**, in: Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS '02, ACM, New York, NY, USA, 2002, pp. 148–160. doi:10.1145/586110.586132. URL <http://doi.acm.org/10.1145/586110.586132>
- [8] A.-R. Sadeghi, D. Naccache, Towards Hardware-Intrinsic Security: Foundations and Practice, 1st Edition, Springer-Verlag New York, Inc., New York, NY, USA, 2010.
- [9] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, Y. Jin, **HAFIX: Hardware-assisted Flow Integrity Extension**, in: Proceedings of the 52Nd Annual Design Automation Conference, DAC '15, ACM, New York, NY, USA, 2015, pp. 74:1–74:6. doi:10.1145/2744769.2744847. URL <http://doi.acm.org/10.1145/2744769.2744847>
- [10] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, J. Zambreno, CODESSEAL: Compiler/FPGA Approach to Secure Applications, in: P. Kantor, G. Muresan, F. Roberts, D. D. Zeng, F.-Y. Wang, H. Chen, R. C. Merkle (Eds.), Intelligence and Security Informatics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 530–535.
- [11] A. K. Kanuparthi, M. Zahran, R. Karri, Architecture Support for Dynamic Integrity Checking, IEEE Transactions on Information Forensics and Security 7 (1) (2012) 321–332. doi:10.1109/TIFS.2011.2166960.
- [12] R. Elbaz, D. Champagne, R. Lee, L. Torres, G. Sassatelli, P. Guillemain, **TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks**, in: P. Paillier, I. Verbauwhede (Eds.), Cryptographic Hardware and Embedded Systems - CHES 2007, Vol. 4727 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 289–302. doi:10.1007/978-3-540-74735-2_20. URL http://dx.doi.org/10.1007/978-3-540-74735-2_20
- [13] M. Hong, H. Guo, S. X. Hu, **A Cost-effective Tag Design for Memory Data Authentication in Embedded Systems**, in: Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '12, ACM, New York, NY, USA, 2012, pp. 17–26. doi:10.1145/2380403.2380414. URL <http://doi.acm.org/10.1145/2380403.2380414>
- [14] B. Gassend, G. Suh, D. Clarke, M. van Dijk, S. Devadas, Caches and hash trees for efficient memory integrity verification, in: High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on, 2003, pp. 295–306. doi:10.1109/HPCA.2003.1183547.
- [15] L. Su, S. Courcambeck, P. Guillemain, C. Schwarz, R. Pacalet, SecBus: Operating System controlled hierarchical page-based memory bus protection, in: Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09., 2009, pp. 570–573. doi:10.1109/DATE.2009.5090729.
- [16] B. Rogers, S. Chhabra, M. Prvulovic, Y. Solihin, Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly, in: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), 2007, pp. 183–196. doi:10.1109/MICRO.2007.16.
- [17] K. Kursawe, A. Sadeghi, D. Schellekens, B. Skoric, P. Tuyls, Reconfigurable Physical Unclonable Functions - Enabling technology for tamper-resistant storage, in: 2009 IEEE International Workshop on Hardware-Oriented Security and Trust, 2009, pp. 22–29. doi:10.1109/HST.2009.5225058.
- [18] Z. Liu, Q. Zhu, D. Li, X. Zou, Off-Chip Memory Encryption and Integrity Protection Based on AES-GCM in Embedded Systems, IEEE Design Test 30 (5) (2013) 54–62. doi:10.1109/MDAT.2013.2255912.
- [19] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, Z. Wang, **Architecture for Protecting Critical Secrets in Microprocessors**, SIGARCH Comput. Archit. News 33 (2) (2005) 2–13. doi:10.1145/1080695.1069971. URL <http://doi.acm.org/10.1145/1080695.1069971>
- [20] J. Aumasson, D. J. Bernstein, **SipHash: A Fast Short-Input PRF**, in: Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9–12, 2012. Proceedings, 2012, pp. 489–508. doi:10.1007/978-3-642-34931-7_28. URL http://dx.doi.org/10.1007/978-3-642-34931-7_28
- [21] G. T. Becker, Robust Fuzzy Extractors and Helper Data Manipulation Attacks Revisited: Theory vs Practice, IEEE Transactions on Dependable and Secure Computing PP (99) (2017) 1–1. doi:10.1109/TDSC.2017.2762675.

- [22] F. Armknecht, R. Maes, A.-R. Sadeghi, F.-X. Standaert, C. Wachsmann, [A Formalization of the Security Features of Physical Functions](#), in: Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 397–412. doi:10.1109/SP.2011.10. URL <http://dx.doi.org/10.1109/SP.2011.10>
- [23] H. S. G. Pussewalage, J. Hu, J. Pieprzyk, A survey: Error control methods used in bio-cryptography, in: 2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 2014, pp. 956–962. doi:10.1109/FSKD.2014.6980968.
- [24] M.-D. M. Yu, S. Devadas, [Secure and Robust Error Correction for Physical Unclonable Functions](#), IEEE Des. Test 27 (1) (2010) 48–65. doi:10.1109/MDT.2010.25. URL <http://dx.doi.org/10.1109/MDT.2010.25>
- [25] J.-B. Fischer, J. Stern, An Efficient Pseudo-Random Generator Provably as Secure as Syndrome Decoding, in: U. Maurer (Ed.), Advances in Cryptology — EUROCRYPT '96, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 245–255.
- [26] Implementation of Computer Security by Hardware-Intrinsic Authentication, <http://cshia.lsc.ic.unicamp.br>, website. Accessed: 10-Jul-2019.
- [27] SipHash-2-4 VHDL Implementation, <https://github.com/pemb/siphash>, web Site. Accessed: 2-Oct-2018.
- [28] MiBench Version 1.0 – Welcome to the homepage for MiBench: a free, commercially representative embedded benchmark suite., <http://vhosts.eecs.umich.edu/mibench/>, website. Accessed: 08-May-2018.
- [29] CACTI – An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model, <http://www.hpl.hp.com/research/cacti/>, website. Accessed: 08-May-2018.
- [30] S. Z. Ahmed, J. Eydoux, L. Rouge, J. B. Cuelle, G. Sasatelli, L. Torres, Exploration of power reduction and performance enhancement in LEON3 processor with ESL reprogrammable eFPGA in processor pipeline and as a co-processor, in: 2009 Design, Automation Test in Europe Conference Exhibition, 2009, pp. 184–189. doi:10.1109/DATE.2009.5090655.
- [31] E. B. Barker, A. L. Roginsky, SP 800-131A. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths, Tech. rep., Gaithersburg, MD, United States (2011).
- [32] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, J. Schmidhuber, [Modeling Attacks on Physical Unclonable Functions](#), in: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10, ACM, New York, NY, USA, 2010, pp. 237–249. doi:10.1145/1866307.1866335. URL <http://doi.acm.org/10.1145/1866307.1866335>
- [33] G. T. Becker, A. Wild, T. Güneysu, Security analysis of index-based syndrome coding for PUF-based key generation, in: 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2015, pp. 20–25. doi:10.1109/HST.2015.7140230.
- [34] C. Paar, J. Pelzl, [Understanding Cryptography: A Textbook for Students and Practitioners](#), 2009. URL <http://books.google.com.br/books?id=f24wFELSzkoC>
- [35] J. Delvaux, D. Gu, I. Verbauwhede, M. Hiller, M.-D. M. Yu, Efficient Fuzzy Extraction of PUF-Induced Secrets: Theory and Applications, in: B. Gierlichs, A. Y. Poschmann (Eds.), Cryptographic Hardware and Embedded Systems – CHES 2016, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 412–431.
- [36] D. Merli, D. Schuster, F. Stumpf, G. Sigl, [Side-Channel Analysis of PUFs and Fuzzy Extractors](#), in: J. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, Y. Beres (Eds.), Trust and Trustworthy Computing, Vol. 6740 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 33–47. doi:10.1007/978-3-642-21599-5_3. URL http://dx.doi.org/10.1007/978-3-642-21599-5_3
- [37] D. Karakoyunlu, B. Sunar, Differential template attacks on PUF enabled cryptographic devices, in: 2010 IEEE International Workshop on Information Forensics and Security, 2010, pp. 1–6. doi:10.1109/WIFS.2010.5711445.
- [38] E. Jamro, The design of a VHDL based synthesis tool for BCH codecs, Ph.D. thesis, The university of Huddersfield, http://home.agh.edu.pl/~jamro/bch_thesis/bch_thesis.html. Website. Accessed: 06-May-2018 (1997).
- [39] A. J. Menezes, P. C. Van Oorschot, S. A. Vanstone, Handbook of applied cryptography, CRC press, 1996.
- [40] J. Yang, Y. Zhang, L. Gao, [Fast Secure Processor for Inhibiting Software Piracy and Tampering](#), in: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, IEEE Computer Society, Washington, DC, USA, 2003, pp. 351–. URL <http://dl.acm.org/citation.cfm?id=956417.956576>
- [41] A. Rogers, M. Milenkovic, A. Milenkovic, A low overhead hardware technique for software integrity and confidentiality, in: 2007 25th International Conference on Computer Design, 2007, pp. 113–120. doi:10.1109/ICCD.2007.4601889.

- [42] J. Sepulveda, F. Willgerodt, M. Pehl, [SEPUFSoc: Using PUFs for Memory Integrity and Authentication in Multi-Processors System-on-Chip](#), in: Proceedings of the 2018 on Great Lakes Symposium on VLSI, GLSVLSI '18, ACM, New York, NY, USA, 2018, pp. 39–44. doi:[10.1145/3194554.3194562](https://doi.org/10.1145/3194554.3194562).
URL <http://doi.acm.org/10.1145/3194554.3194562>