University of Campinas
Institute of Computing

Qualification Exam

Computer Security by Hardware-Intrinsic Authentication

Ph.D. Student: **Caio Hoffman**
Principal advisor **Prof. Guido Costa Souza de Araújo**
Co-advisor: **Prof. Mario Lúcio Côrtes**

September of 2015

Contents

List of Figures

List of Tables

Abstract

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

1 Introduction

Standard design techniques to secure code execution in SoCs are based on known cryptographic mechanisms (primitives like block ciphers and hash functions), and on (micro) architecture techniques which can be used to encode bus transactions [9], or isolate secure code into trusted platforms [13], among others. Although such techniques usually provide good levels of security, most of them are either slow, considerably impact processor (micro) architecture design, require extensive changes in the programming tool-chain [30, 32], or are so complex that may create unexpected security loopholes.

Any solution up to this challenge should be able to allow for a seamless integration to the current processor/programming paradigms, achieve very high-level security under low-cost and low-power. More than incremental approaches which try to re-use current cryptographic mechanisms to fill in security holes, the new generation of IoT devices will require novel solutions which deeply integrate the hardware-intrinsic features to program execution, across the whole architecture and software stacks.

Physical Unclonable Functions (PUFs) are devices which exploit the statistical distribution of hardware intrinsic physical parameters, to design functions capable of (uniquely) mapping a set of inputs (*challenges*) to outputs (*responses*) [25]. Built upon PUFs' theoretical models, several constructions of essential cryptographic primitives have been proposed, mainly to support key exchange [5, 20, 34], device authentication [31], intellectual property protection [14], oblivious transfer [5, 27] and commitment schemes [5]. The myriad of cryptographic primitives which could benefit from PUFs has driven the search for efficient real-world implementation of these devices.

Although silicon PUFs have gained a lot of attention, they are still under strong scrutiny, as they can undergo a number of attacks like: (1) reverse engineering [24], (2) characterization of the physical parameter [33], (3) modeling [26], and (4) emulation [15]. Even though there are still many concerns about the overall security of PUFs, their simplicity, low-power consumption and speed are very attractive design features [22] for some application domains (e.g. IoT devices). One of the potential applications of PUFs in IoT devices would enable program code and data integrity. Yet very few works have addressed that using PUFs [32]. Thus additional research needs to be done in order not only to improve PUF security, but also to allow its integration into processor architecture and software stacks.

This proposal proposes *Computer Security by Hardware-Intrinsic Authentication* (CSHIA), a design of a new secure program execution model. The approach is a new PUF-based mechanism which aims at ensuring firmware authenticity and integrity for a given program/processor pair – while confidentiality is left for future work. Specifically, for authentication, the system generates an authentication tag (called PTAG) to every instruction and data cache line at the very first moment that it runs in the processor. For integrity, it provides a new architecture that ensures that program instructions and data are not violated, and that the program will execute correctly during the lifetime of the device.

The contributions of this work are:

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

This proposal is organized as follows. Section 3 discusses related work. Section 4 describes the proposed (micro) architecture and authentication mechanism. Section 5 discusses how CSHIA addresses typical attacks. Section 6 describes how CSHIA tackles replay attacks and provides an analysis of caching nodes strategies for authentication trees, from which this work leverages a performance-efficient strategy. Section 7 details CSHIA key extraction. Finally, Section 8 concludes the proposal and Section 9 discusses future work.

2 Goals

2.1 Main Goal

The Ph.D. project aims at providing the foundations of a new architecture model that is tamper-evident and tamper-resistant, while the following properties are ensured:

- The architecture is not processor-cycle consuming.
- The software tool-chain (compiler, operating systems, applications) will not be changed.
- Hardware modifications are low-power consuming.
- The architecture is resilient against software and physical attacks.

The proposed architecture addresses tamper-evident by code and data integrity verification, and tamper-resistant by code and data authentication.

2.2 Side Goals

For this project, we also intend to:

1. Build tools of simulation, modeling, and statistical analysis.
2. Make the tools free-content available in the internet.
3. Publish our results in international events and journals.
4. Patent the project.

3 Related Work

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

Although the dedicated nature of SoCs allows the adoption of more intrusive protection, it also imposes challenging energy and performance requirements. Unfortunately, traditional security solutions based on typical cryptographic mechanisms can have an expensive impact in device cost, energy efficiency and performance. One way to go around that is to consider approaches which enable a deep integration of device hardware-intrinsic mechanisms and program execution, as those offered by PUFs.

Qualitative analyses of PUFs have already been done in the literature [18] motivated by several reasons like cryptographic key generation [4, 31] and true random number generation [16, 19]. 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 to secure code execution aimed at keeping instructions and data secure from scrutiny, by using mechanisms like bus encryption. In [9] Elbaz *et al.*did a comprehensive survey of bus encryption, where they describe many possible ways of using cryptographic algorithms in SoC architectures, so as to ensure that no malicious instruction/ data would be executed by the CPU. From many alternatives, the authors discarded public key encryption because of its high overhead. The remaining solutions store encrypted data in external memory (or even at higher level cache memories). Such schemes require on-chip secret key storage, a major shortcoming since the usage of non-volatile memories to store keys is susceptible to side-channel attacks [28].

AEGIS, the proposed secure processor by Suh *et al.*in [32], uses PUFs as a cryptography primitive to uniquely authenticate code and data in order to prevent both software and physical attacks. They present a tool-chain for developing secure software for their

architecture which includes a secure operating system to manage different levels of memory protection. Although the presented tool-chain does not require modifications in the processor architecture, it demands extensive changes in the SoC architecture, in addition to changes in the compiler and operating system tool-chains. Even though the described set of tools enables different security levels, thus minimizing performance degradation, their architecture requires 30 % more processor cycles when running a case study: Sensor Networks. Besides that, performance degradation becomes prohibitive for programs with high cache miss rates. Code and data memory overheads are considerably low, and stayed below 5 % in the case study. Nevertheless, AEGIS does not ensure full-time security from power-on to power-off; i.e. the system runs unprotected until the security kernel loads the system.

One of the most difficult issues against active attacks on secure processor architectures is to preserve memory integrity. Memory placed outside of a secure area is exposed to any kind of manipulation an attacker could perform. Despite that, the secure area still needs to verify the integrity of the memory when communicating with it. Recently, many solutions came up in the literature proposing different approaches, costs, and overheads [10, 11, 17]. Section 5 discusses memory integrity issues in detail.

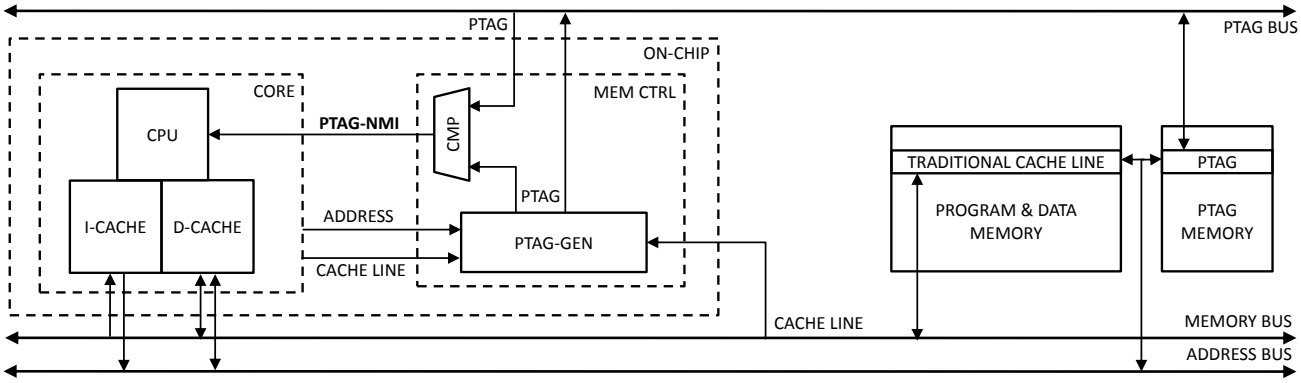


Figure 1: A system overview of the CSHIA system.

4 The CSHIA Architecture

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

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

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

This section is divided in two parts. First, Section 4.1 details the main operations executed by CSHIA, from the generation of the PTAG to its validation. Second, Section 4.2 describes the mechanism used to extract the key used to generate the PTAGs and discusses

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

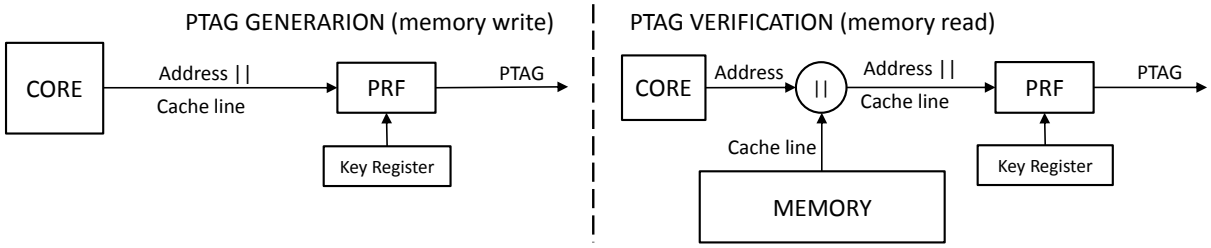


Figure 2: The PTAG-GEN during PTAG Generation (write) and PTAG Verification (read) operations.

firmware installation and update.

4.1 PTAG-GEN Operation

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

4.1.1 PTAG Generation (memory write)

During a write operation the MCTRL writes data/instruction cache lines to memory while the PTAG-GEN computes the PTAG and stores it into the PTAG Memory. To generate the PTAG a *Pseudorandom Function* (PRF) [12] module is used and takes as input the concatenation (||) of the cache line bits and the base address of the cache line provided by the core (see Figure 2). In order to ensure uniqueness, the PRF is configured using a *unique-per-device key*. As Section 4.2 describes, this key is produced by the intrinsic hardware features of a SRAM PUF. Such authentication tag is specific to the core running that specific cache line, as SRAM PUF outputs are dependent on the statistical variations of the manufacturing process, and these are unique to each processor. Hence identical cache lines running on different processors will produce different PTAG values for the same inputs. Only code in the cache, for which integrity has been ensured, will be able to write to the memory. In other words, a chain of trust exists from the execution of one instruction to another, starting on the first instruction which has been validated into the system. Moreover, as only ensured code can write into memory, not only code but also program data will have its own PTAG computed and stored into the PTAG Memory. Therefore, the chain of trust created by the flow of execution of ensured instructions will also be transferred to program data. Notice that this approach allows for the existence of multithreaded programs.

4.1.2 PTAG Verification (memory read)

During a read operation the MCTRL reads data/instruction cache lines from memory while the PTAG-GEN computes a PTAG for each cache line. As shown in Figure 2, during a read operation the cache line base address produced by the core is appended to the cache line bits read from memory and the resulting bits are fed to the PRF module. The PTAG produced in this way is compared to the PTAG read from memory for equality. If the previously stored PTAG and the recently computed value do not match, a *Non-Maskarable Interrupt* (NMI) is generated to the core (called PTAG-NMI), as code/data integrity might have been violated. As shown in Figure 1, in order to hide PUF latency, the data/instruction is sent to the respective cache (I$ or D$) at the same time that the PTAG-GEN computes the PTAG for that cache line and compares it to its PTAG, previously stored into the PTAG Memory.

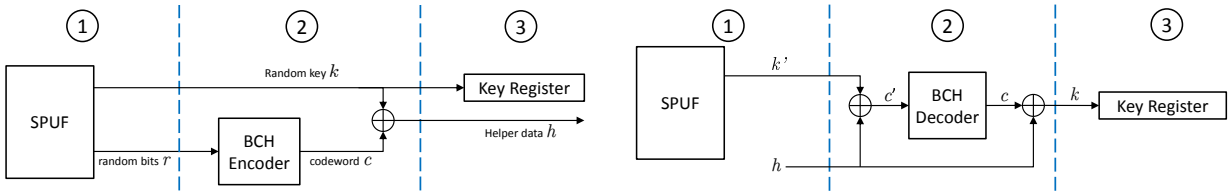
4.1.3 I/O

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

Overall, from an architecture perspective, the main advantages of the CSHIA execution model, when compared to other techniques, like [30] are: (a) simplicity and easy of integration to current architecture/programming models; (b) separation of the system and PTAG buses, isolation of the PTAG Memory from program scrutiny; (c) improved performance.

4.2 PTAG-GEN Configuration

As discussed above, the PTAG-GEN should produce a unique PTAG for each combination of processor, cache line data and address. To achieve that, the PRF is configured by means of a PUF generated key, which is unique to each processor. This key is generated only once and regenerated at each device turn-on time. Any quality PUF available in the literature could be used to produce the key. Nevertheless, given its simplicity and good statistical metrics [18] the SRAM PUF has been selected for the purpose of this work, and will be called from now



(a) Fuzzy extractor steps during the Enrollment. (b) Fuzzy extractor steps during the System Reboot.

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

on SPUF.

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

4.2.1 Enrollment Process

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

• **Assessment:** During this stage, the processor's cache is turned on and off tens of times to find a set A of SPUF addresses that have stable bits to compose the PRF key k; this assessments seeks to choose bits that will have a very low bit-flip probability until the end of the hardware lifetime – Section 7 shows experiments that support the existence of such addresses in SRAMs. Although the bits selected this way will not present bit-flips during chip assessment, one cannot assure that flips will never happen as physical parameters of the SPUF cells can degrade over time. To compensate for that, a fuzzy extractor architecture [1, 21, 23] is implemented in PTAG-GEN to recover the key k, in case of some key bits are incorrect when the processor turns on.

• **Extractor Setup:** In this stage, bits are selected from the SPUF, using addresses

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

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

4.2.2 The System Reboot Process

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

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

• **Key Recovery:** Given that there is a non-zero probability that the SPUF cells flip to different values as the device is turned on, the recovered key k' might not be the same as k. To correct k' (step 2), the MCTRL computes the exclusive-or of k' and h, thus generating

a codeword c' that differs from c in as many bits as k differs from k'. A BCH decoder is then used to correct c' to c. By computing $c \oplus h$ the system recovers the original value k.

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

5 Security Analysis

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

A security analysis across the whole (micro) architecture requires a proper selection of the parameters that compound CSHIA. For security and high performance of the PTAG generator, the chosen PRF is SipHash-2-4 [2]. The SipHash PRF uses 128-bit keys and produces 64-bit outputs. Thus the secret key k is 128 bits long and a PTAG p has 64 bits. A hardware implementation of SipHash-2-4 can produce an output for a 72-byte input (memory address concatenated with cache line) in 10 cycles, which is crucial for the CSHIA performance. As the helper data h has the same length of the key, it also has 128 bits. The random bits r, used in the generation of the codeword c, have length $|r| \geq |p| = 64$ bits. The proper value of r in CSHIA is explained in Section 7.

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

5.1 Brute-force/Forgery attacks

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

5.2 Enrollment/Reboot time attacks

The security of CSHIA enrollment and reboot process is also relevant. First, one should notice that two SPUF instances have different intrinsic physical parameters and thus are not equal. The probability that the key k and the random value r will repeat for any pair of SPUFs is extremely small. In addition to the literature [18, 19], Section 7 provides data that confirm the randomness of k and r over different instances of SPUF. Consequently, even if an attacker obtains the key of a CSHIA instance, valid PTAGs for other instances of the architecture could not be generated. Moreover, modifications of the values of addresses A or the helper data h stored in memory will recover wrong keys and the PTAG-GEN will not validate them. Such situations will generate a PTAG-NMI signal interrupting processor execution. An attacker could also try to manipulate the PTAGs of A and h in order to authenticate malicious set of addresses or helper data. In order to succeed, he/she needs to know the key, which is not available.

5.3 Modeling attacks

Although an attacker could collect as many cache lines and PTAGs as possible, creating a model of the PTAG mechanism should not be possible, given that the PTAG-GEN uses a PRF that was evaluated to be indistinguishable from a uniformly random function [2]. Thus, modeling does not apply. Besides that, numerical modeling attacks are not effective against SRAM-PUFs [26]. Without physical access to the device, the only way out for an attacker is guessing the PRF key, which is not an easy task when the key is randomly chosen and its size has at least 80 bits [3] – remembering that the key k is 128 bits long. An easier way to attack CSHIA would be to recover the random word r from the helper data h. Once the attacker recovers r, the same syndrome in the codeword c could be generated and then k recovered. Nonetheless, such attacks requires manipulating the helper data [23], which is not possible in CSHIA, since a PTAG protects its integrity. Thus, the only option left to the attacker is to guess r which is at least as hard as guessing a PTAG, given that $|r| \geq |p| = 64$ bits.

5.4 Side-channel/Physical attacks

Literature abounds on studies of different side-channel attacks in SRAM [15, 24] and fuzzy extractors [8, 23]. While invasive and semi-invasive attacks may completely break CSHIA se-

curity, attacks like Differential Power Analysis (DPA) [23] that needs to manipulate the system's parameter – like the addresses A and/or the helper data h – will not succeed, because all off-chip data will have their integrity verified through PTAGs. Protecting SRAM from leaking instructions or dumping data during the system reboot will ensure protection against key extraction. Besides that, some mechanisms like fully resettable SRAM [15] may difficult semi-invasive attacks. Of course, fully physical protection demands novel VLSI countermeasures to prevent invasive attacks like in [24].

5.5 Memory integrity attacks

There are basically three types of memory integrity attacks against CSHIA. Using the same terminology as in [10], those are (1) *Spoofing*, (2) *Splicing* or *Relocation*, and (3) *Replay* attacks.

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

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

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

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

presents how CSHIA tackles this attack.

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

6 Dealing with Replay Attack

Dealing with replay attack requires keeping on-chip status of all data memory blocks. Obviously, that needs a large memory on-chip, which is impractical. To reduce the on-chip memory requirements, the standard approach is to use an *Authentication Tree*. There are different forms of building authentication trees [11], [10], [17]. However, since CSHIA uses a PRF to authenticate memory blocks, a suitable approach is a Merkle Tree [11]. Section 6.1 discusses a naïve tree implementation and Section 6.2 shows how to improve it. Finally, Section 6.3 justifies why adopting caching-node policies for authentication trees would improve the overall system performance.

6.1 Merkle Tree

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

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

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

To illustrate the process, suppose that the processor requests MB-2. As a regular integrity check procedure, the MCTRL computes the PTAG of the MB and verifies it against

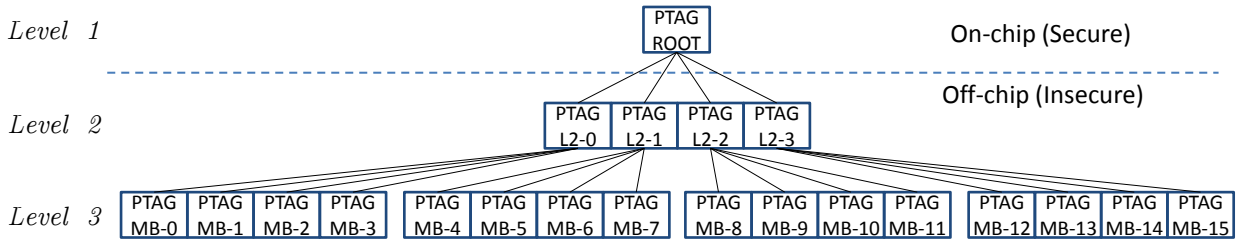


Figure 4: A pictorial example of the Merkle tree implemented in CSHIA.

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

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

To illustrate an eviction situation, using Figure 4 again, suppose the processor evicted the data cache line associated to the MB-9. The MCTRL computes the PTAG MB-9 and stores it in the PTAG Memory. Now the MCTRL has to bring all PTAG MB-9 siblings in order to recompute the PTAG L2-2. After that, the MCTRL updates the PTAG L2-2 and brings all its siblings to compute PTAG ROOT, which is updated only on chip. Thus, there were 2 writes and $2 \cdot 3$ reads in the PTAG Memory, which amounts to $2 \cdot 4$ accesses to the PTAG Memory, or $(L-1) \cdot d$ in general. The entire eviction process, including the verification of the requested MB by the processor, demands $2 \cdot d \cdot (L-1)$ accesses to the

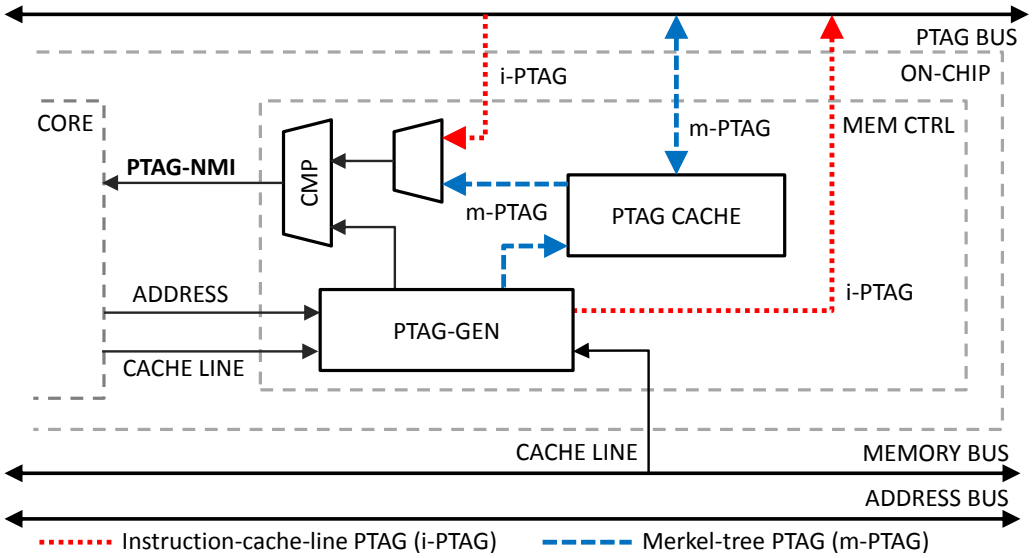


Figure 5: Modifications of the MCTRL and its behavior regarding PTAGs of instruction cache lines (i-PTAG, small dashes) and PTAGs of the Merkle tree (m-PTAG, larger dashes).

PTAG Memory. For typical memory sizes, this access rate will indeed cause PTAG-Bus contention. The adoption of a cache to store PTAGs is an attractive solution for this type of problem. This is discussed next.

6.2 PTAG Cache

In [11], Gassend *et al.*propose to use the L2 processor's cache to cache the tree while allowing speculative execution until a hit happens in either the cache or the root (in the secure area). Despite being an efficient solution, since CSHIA has a dedicated bus for PTAGs, a PTAG Cache on the MCTRL fits better the CSHIA needs. In addition to that, reducing the useful space of the processor's caches is prohibitive, given the limited resources in SoCs typical of IoT devices. See the MCTRL modifications in Figure 5.

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

To compare how the PTAG Cache reduces the number of access to the PTAG Memory in relation to a CSHIA implementation without the PTAG Cache (that is, the naïve

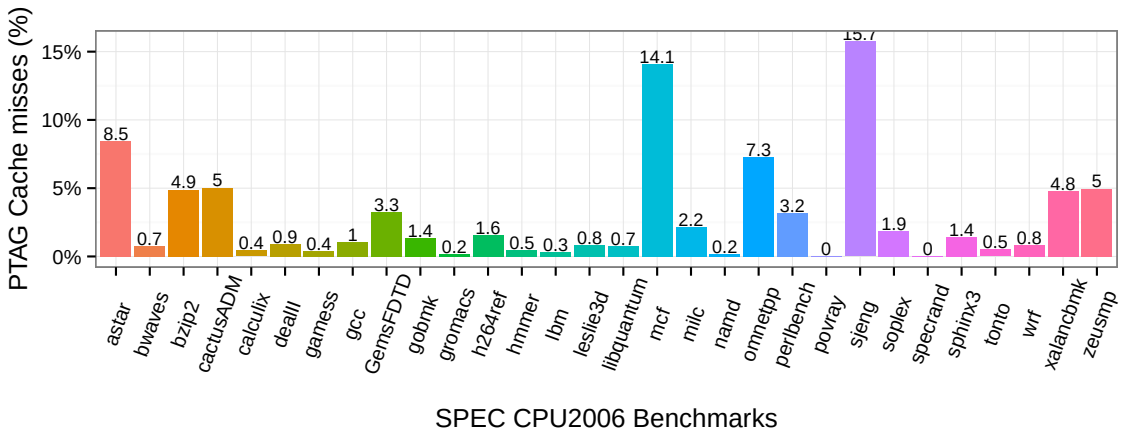


Figure 6: PTAG Cache miss rate for the SPEC 2006 benchmarks.

CSHIA implementation), one can measure, for each data cache miss, what is the hit rate of the PTAG Cache. To do that, assume the following experimental scenario: a 64-KB processor data cache memory with 8-way set associativity, 128 lines, and blocks of 64 bytes; and a 64-KB PTAG Cache with 16-way set associativity, 64 lines, and blocks of the same size. Notice that while a 64-byte memory block represents a cache line, 64-byte PTAG-Cache block represents a chunk of $d = 8$ PTAGs – remembering that a PTAG is 64 bits long.

The experiment was performed using a PIN TOOL to simulate the PTAG Cache and a L1 Data Cache. The PIN TOOL runs all benchmarks of the SPEC CPU2006. The experiment assumed 48 bits of virtual address space. In CSHIA, a memory with 2^{48} bytes requires 2^{42} PTAGs to authenticate all memory blocks. Now, consider a Merkle Tree with degree $d = 8$ and $L = 15$ levels. For a rough estimate of the size of a complete PTAG Memory, one should take into account $2^{42} \cdot$ 8 bytes of memory block PTAGs plus no more than $2^{42} \cdot$ 8 bytes to store all ancestors, which results in 2^{46} bytes. This is equivalent to 25 % of the size of the entire main memory.

Due to computing limitations, the experiment did not use a complete 8-ary Merkle tree for simulation. Main memory regions that would result in a subtree of PTAGs in the Merkle Tree were not allocated in the PTAG Memory and were represented by a phantom node in the tree (similar what was done in [6]). Nonetheless, there is still a high number of PTAGs to verify in the simulation and that would also call into question the efficiency of the PTAG Cache. Fortunately, such high-overhead configuration is very unlikely to be found in an IoT SoC, but serves for illustrative purposes as a pessimistic reference point.

Figure 6 shows the Miss Rate (MR) of the PTAG Cache. One can clearly identify

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

6.3 Caching Policies

During the aforementioned experiment, different strategies for copying nodes of the Merkle Tree to the PTAG Cache resulted in different miss rates. For instance, adopting the strategy of stopping verifying/updating nodes of a leaf-root path at the first node found in cache, as used by Gassend *et al.* in [11], resulted in a miss rate generally four times greater than the results showed in the Figure 6. The strategy that resulted in the Figure 6 always cached all nodes from the leaf-root path while the integrity of a MB is verified/updated.

One can argue that a strategy which does not stop copying nodes of a tree's path, after finding a descendant node of the path in the cache, would increase the cache's miss rate rather than decreasing it. To verify this argument, the miss rate of four caching nodes strategy was analyzed. The strategies are:

• **CHTree**. This policy is the same adopted by Gassend *et al.* in [11]. It consists in stopping verifying/updating nodes of a leaf-root path at the first node found in cache. This policy leverages from the location of the cache, which lies inside of the processor and therefore can not be under the control of an adversary.

• **Caching.Path** always caches nodes from a leaf-root path, while the integrity of a MB is verified/updated. Thus, regardless a descendant node is found in the PTAG Cache, this policy will not stop verifying/updating (and bringing those nodes that are not in the cache) until it reaches the PTAG Root.

• **Read.Hit**. This policy mixes the CHTree and Caching.Path policies. For verifying a leaf-root path, this policy adopts the CHTree's process. For updating, this policy adopts Caching.Path's process.

• **SecBus** [29] adds to the CHTree policy (which was named to *as soon as possible* (ASAP) policy) the *as late as possible* (ALAP) policy. The ALAP policy delays the eviction

of dirty nodes by vacating clean nodes in the cache instead. When all nodes in the cache are dirty the cache controller should use an eviction policy in order to avoid deadlock. This proposal uses the least recently used (LRU) as eviction policy to SecBus.

The same experiment before (from Section 6.2) was performed for each policy and the results are shown in the Figure 7. The Caching.Path results are the same which Figure 6 shows. One can notice in the Figure 7 that the results of Caching.Path are clearly superior compared to the other policies. A deeper analysis in the results showed that the number of hits in the Caching.Path policy was too large due to the repeating process of always verifying/updating all nodes in a leaf-root path. Namely, nodes of leaf-root paths frequently were in the cache. Thus the only inconvenience of Caching.Path is to need multiple accesses in the cache, which is very unlikely to be a problem since cache access demands only a few clock cycles.

Nevertheless, it seems that miss rates as a performance metric are not reliable to determine whether a caching-node strategy is performance-efficient, since one would like to know how a caching policy affects the number of accesses to external memories. A better metric would be the ratio between PTAG Cache misses and L1 Data Cache misses[3]. Namely, for each access to main memory how many accesses are need to the PTAG Memory in order to verify/update the integrity of a MB. The results of this metric for the four caching policies are presented in the Figure 8.

Notice in the Figure 8 that the performance of Caching.Path is not superior anymore. For almost all benchmarks the performance of all policies were sightly similar. Three benchmarks are distinctive: *cactusADM*, *omnetpp*, and *sjeng*. In the *omnetpp* and *sjeng* benchmarks, the policies Caching.Path and Read.Hit perform better than CHTree and SecBus due to high number of evictions that those benchmarks present. As the policies CHTree and SecBus delay updating ancestor nodes of the nodes present in the cache, an eviction demands that all ancestors nodes that are not in cache have to be updated in order to keep coherence in the authentication tree. Every node that has to be brought to the PTAG Cache may provoke another eviction and this scheme can go on and on. Specifically, for *omnetpp* and *sjeng* that happens very often. In the case of the *cactusADM*, the number of reads overwhelms the number of writes, thus there is a very few frequency of eviction and that benefits

[3]Notice that for the experiments in this section there is no L2 Caches.

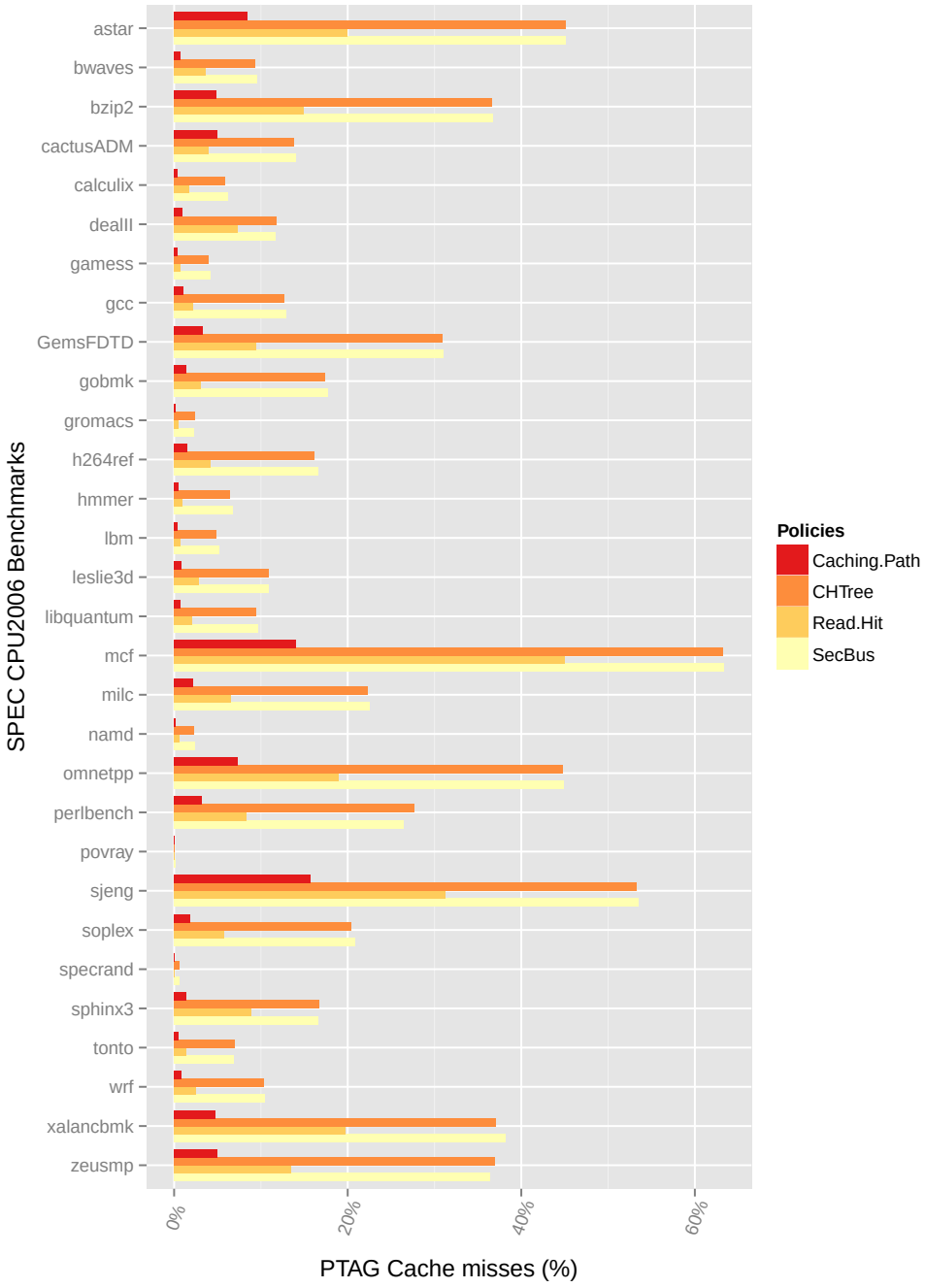


Figure 7: PTAG Cache miss rate for the SPEC 2006 benchmarks.

the CHTree and SecBus policies. Notice that the Read.Hit policy adopts the same caching strategy for reading of the CHTree and SecBus policies, which results in good performance on the simulation the benchmark *cactusADM*.

To shed light on the comparison among the policies, the occurrence of evictions in the PTAG Cache was compared to L1 Data Cache misses. This can give a verdict of which

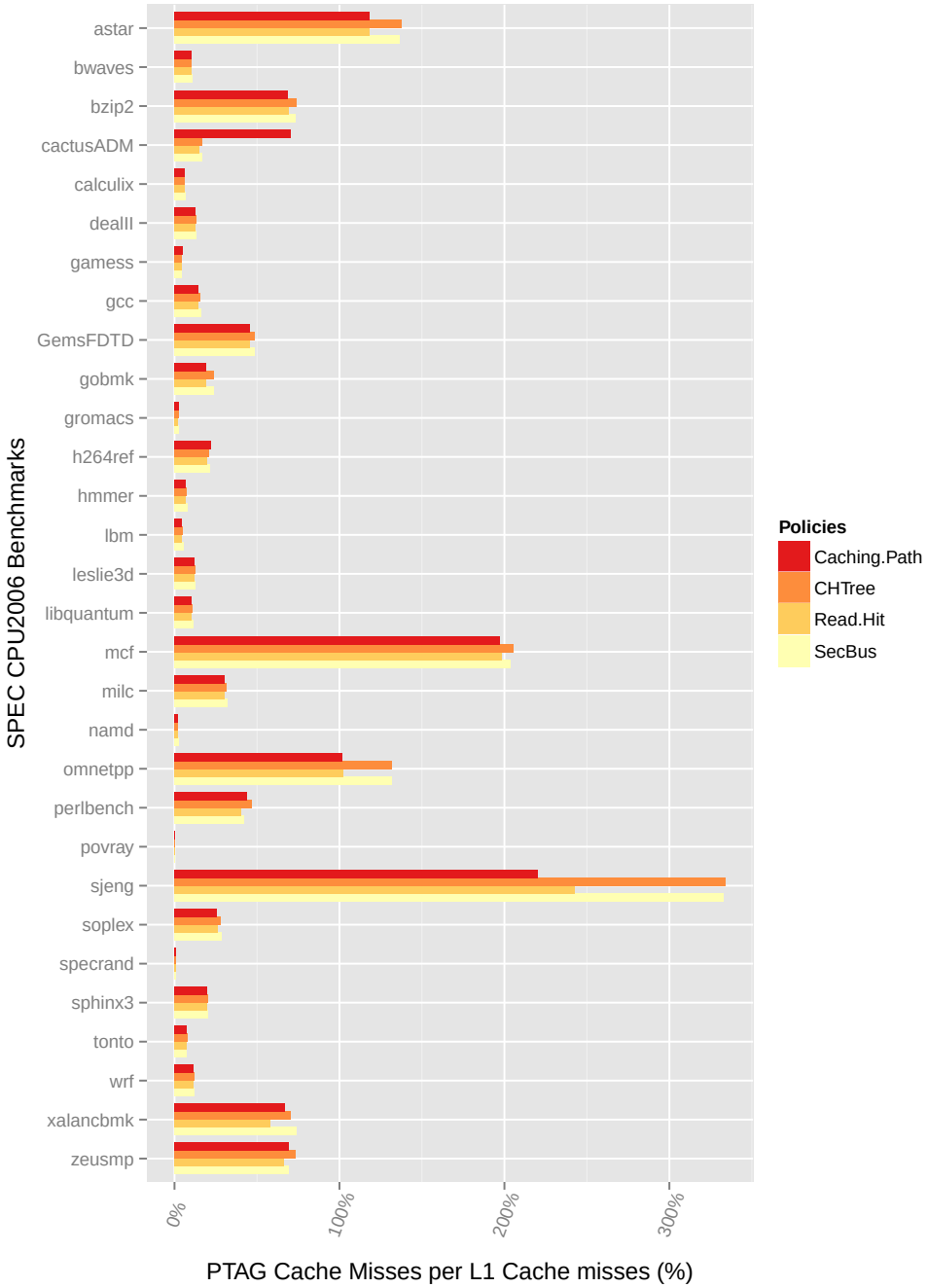


Figure 8: PTAG Cache miss rate for the SPEC 2006 benchmarks.

policies will produce less memory writes. Regarding IoT devices, the presence of non-volatile memories as main memory (and PTAG Memory) might not be uncommon, which means that a high number of memory writes increases the power consumption and decreases the lifetime of IoT devices. Thus any policy that reduces the number of evictions is performance-friendly.

Figure 9 shows that Caching.Path and Read.Hit policies have lower numbers of evic-

tions compared to the CHTree and SecBus policies in almost all benchmark. The only exception is to the *cactusADM* benchmark in which Caching.Path presents a poor performance. From the Figures 8 and 9, one can notice that the benchmarks *astar*, *omnetpp*, and *sjeng* present a significant proportion of evictions (Figure 9) from the misses that happen (Figure 8) in the PTAG Cache. From such results, one can conclude that Caching.Path and Read.Hit are the best caching policies for devices that will implement the CSHIA architecture. Besides, taking into account the results from the *cactusADM* benchmark, the best caching policy among those tested seems to be the Read.Hit policy.

To the best of our knowledge, CSHIA is the first architecture with a dedicated bus to deal with memory authentication and integrity verification. The results from simulating a dedicated cache to the PTAG Bus seems promising. Differently from the work in [11], in which the L2 processor's cache stores nodes of the authentication tree, instructions, and data, a dedicated cache does not affect the cache miss-rate of programs, mainly when the L2 cache is small. Nonetheless, further studies on performance, power consumption, and area are needed for a thorough comparison with the work of Gassend *et al.* This is left for future work.

Notice also that the PTAG-Cache replay attack solution specifically employs authentication trees. However, there are other solutions to replay attack that uses caches [7], and a comparison involves figures of merit that are beyond the scope of this proposal.

Finally, a high number of accesses to the PTAG Cache might increase the frequency of execution stalls since, for each miss, the MCTRL may have to verify an entire leaf-root path. However, after finding the first node of the path in cache, the PTAG-Cache controller would keep its policy process and allow the processor to proceed with its execution. In addition to that, speculative execution would reduce the time that the processor is kept stalled when many nodes of a leaf-root path are absent from the cache. One would only stall the processor when data need to leave the secure area, which seems a promising approach. Measuring the real impact and efficiency of such policies requires simulation and implementation. Nevertheless, preliminary results allow concluding that the PTAG Cache and its caching policies may contribute to improve CSHIA performance.

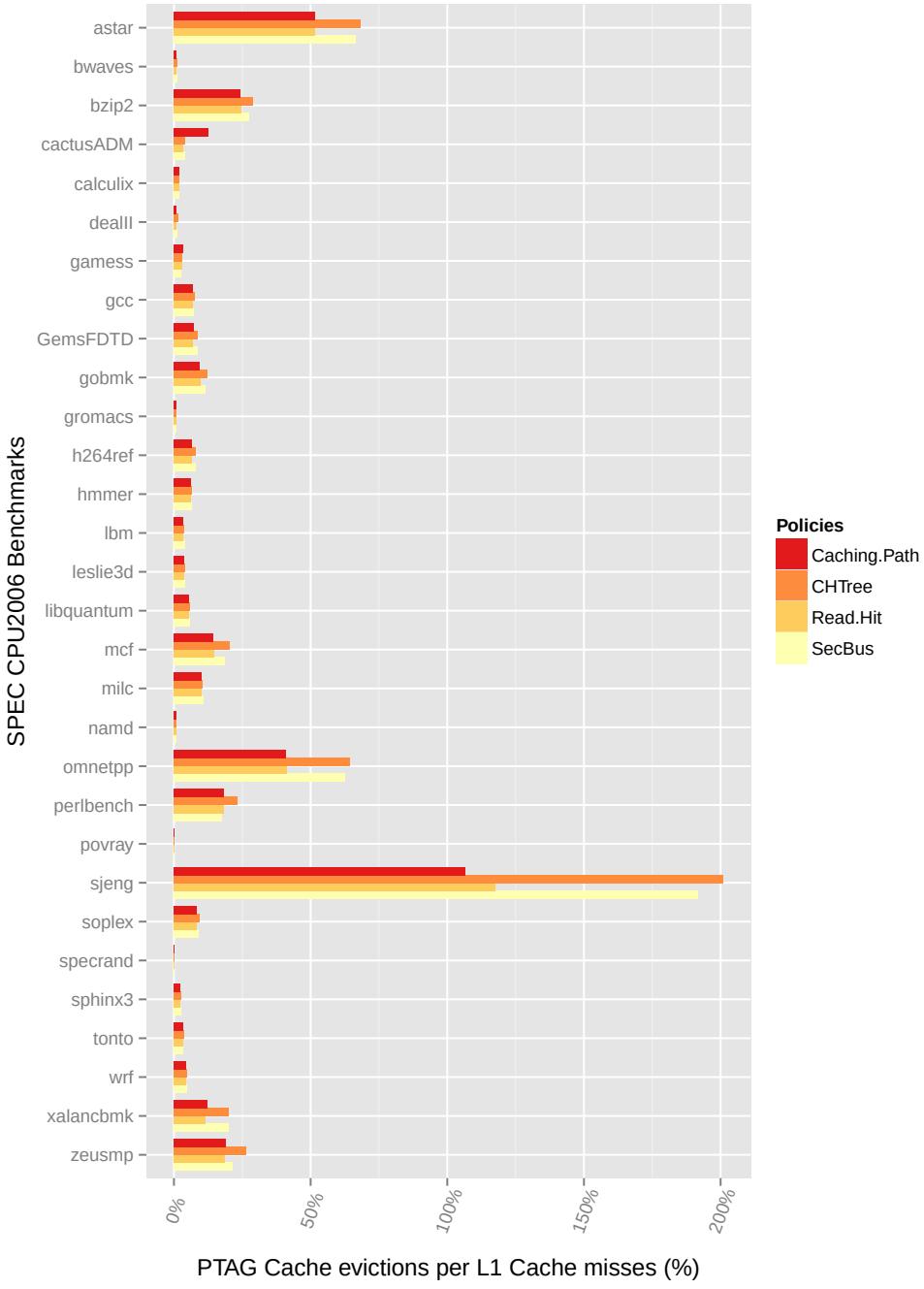


Figure 9: PTAG Cache miss rate for the SPEC 2006 benchmarks.

7 SRAM PUF key Extraction

The main requirements of the key extraction process presented in Section 4 have to be validated by experimental results. This section shows how to determine the number of assessments needed to extract a stable key for the PRF. It also shows how the statistical distribution of the possible keys behaves across different SPUFs.

7.1 SPUF Evaluation

As stated in Section 4, even choosing stable bits for the key, one cannot ensure that they will not flip and make the fuzzy extractor recover a wrong value for k. The BCH code to be used in CSHIA's fuzzy extractor (Figure 3) must satisfy two properties: (a) codeword length with the same bit-length as the key k (i.e. 128 bits); (b) syndrome length such that the random word r is at least as long as the PTAG length (i.e., $|r| \geq |p| = 64$ bits). A (127, 64, 10) instance of a BCH code has both properties and is able to detect and correct up to 10 errors. Since that code produces 127-bit codewords, a parity bit is computed over the 127-bit codeword and concatenated to it, resulting in c (Figure 3). It is worth noticing that the parity bit value is completely unknown to an attacker, since both r and the 127-bit codeword are unknown. Thus, neither a bit of c nor a bit of k will leak any information through this scheme.

At this point, one has to devise a strategy to extract stable bits and to measure the probability that these bits would not exhibit more bit-flips than the maximum allowed by the BCH-based fuzzy extractor. In a related article [19], Leest *et al.*measured the stability of SRAMs. They ran 500 power-up operations on a SRAM with the purpose of extracting random numbers and concluded that, after the first 100 power-up rounds, the amount of variability (entropy) of the measured SRAM contents becomes stable. Leest *et al.*then used this lower bound of 100 assessments to estimate the minimum entropy of other SRAMs, allowing them to discover the amount of bits they should extract in order to obtain random words with the same length. This result is useful for CSHIA implementation, because it establishes that stable random bits can be extracted from a SRAM if their locations are known.

However, even after 100 assessments, there is no guarantee that, after picking 128 bits to compose the key k, the number of bit-flip errors will not exceed the limit of the

chosen BCH code (10 errors). To estimate the probability of more than 10 errors in a 128-bit key, an experiment with SRAMs available in 10 Altera DE1 development kit was conducted. Although each SRAM had 512 KB, a much larger size than a conventional L1 cache in modern processors, only the first 64 KB were analyzed.

In the first experiment, each SPUF is subjected to 200 power-up cycles (assessments), in which the first one is considered the reference assignment. Figure 10 shows the percentage amount of bits that present a different value as compared to the reference, at least once. One can notice that after around 100 assessments the curves start to flatten off, which reproduces the result by Leest *et al.*

Now, one has to determine the probability of finding more than 10 errors with respect to the reference assessment, when reading 128 bits out of these potentially stable bits. Recall that finding such a high number of errors would be a problem because the BCH code in the fuzzy extractor deals with at most 10 errors. Figure 11 shows the probability of having more than 10 bit flips when choosing 128 bits at every assessment after the 100th round. The measurements were conducted in the range 100 to 200 rounds of assessments, given that the SRAM behavior is not expected to change significantly after 200 assessments. In order to decide what is the recommended number of assessments, one can take the reference probability of one part in a million (1 p.p.m) [21] – shown as a horizontal line in the Figure 11.

One can notice in Figure 11 that, after selecting 128 bits from those that have not flipped yet after 160 assessments, it is highly unlikely to obtain unstable keys from any

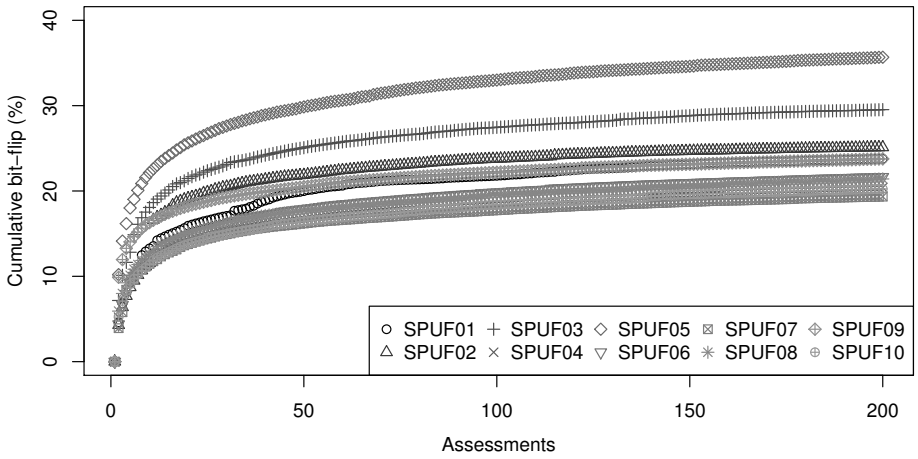


Figure 10: The fraction of SPUF bits that have flipped during 200 assessments.

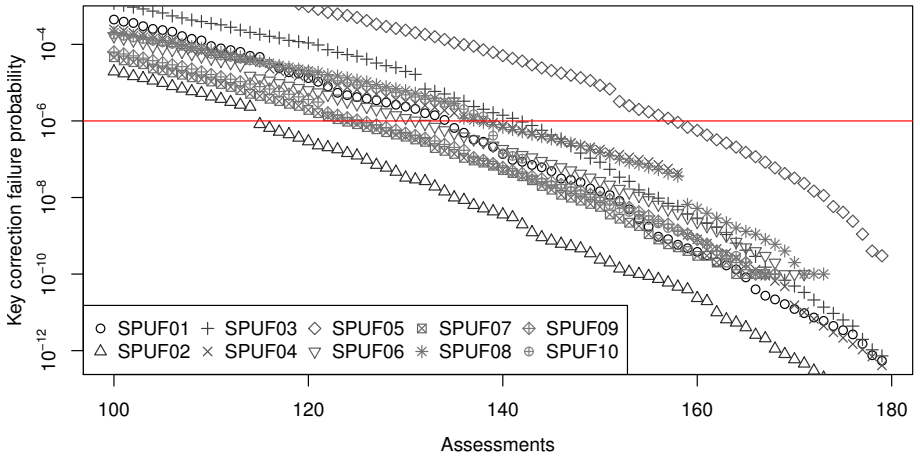


Figure 11: The probability of (127, 64, 10) BCH code failure in correcting a 128-bit key when composing it from single bits extracted from SPUFs.

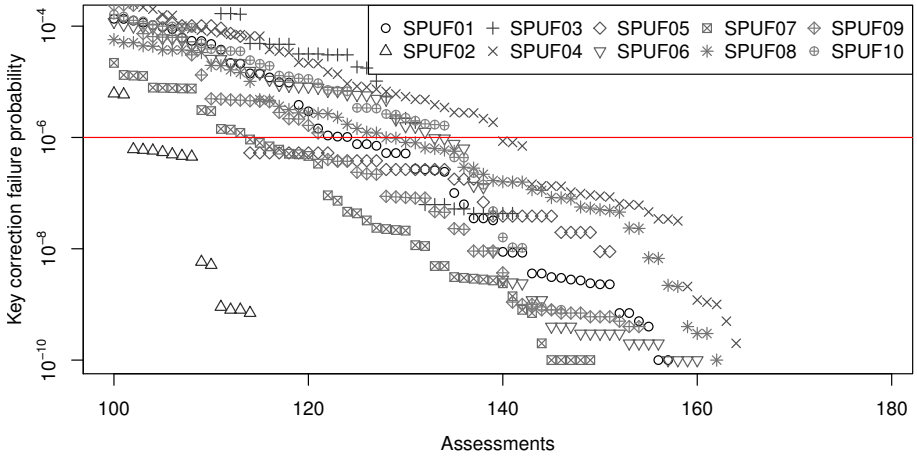


Figure 12: The probability of (127, 64, 10) BCH code failure in correcting a 128-bit key when compounding it with 16-bit words picked from SPUFs.

SPUF. However, this method requires keeping track of the addresses of 128 bit locations. Each address takes 19 bits in a 64 KB memory, resulting in an overhead of $128 \cdot 19$ bits $=$ 2432 bits – equivalent to 5 cache lines. To reduce this overhead, one can look for stable memory words, rather than looking for single bits. The experiments show that picking stable words reduces not only the address storage overhead, but also the number of assessments to reach the probability of 10^{-6}. Figure 12 shows the key correction failure probability with respect to the number of assessments when picking 16-bit stable words. Unfortunately, it was not possible to find stable words with more than 16 contiguous bits for any SPUF.

7.2 Key Extraction

This section presents the key extraction algorithm that can be automated and used over the previously described data, and an analysis of the statistical properties of the resulting keys. Figure 12 clearly shows that there is strong correlation between the key correction failure probability and the number of assessments. Thus, one can run a linear regression model to predict the number of assessments that produces the expected low failure probability. After conducting such experiment, with 99% confidence at the prediction interval, 149 was found to be a conservative number of assessments to reach the 10^{-6} probability.

This analysis may be conducted during the learning phase (technology maturation) over all evaluated SRAM instances, and results in a single number, the maximum number of assessments. Once this number is determined, the Algorithm 1 can be executed for each processor instance, producing as result a list of reliable and stable memory addresses.

Data: **m** ← number of assessments, L1Data, L1Instruction
Result: Stable word addresses.
1 L1Data ← *reset*; $i \leftarrow 0$;
2 Copy(L1Data, L1Instruction);
3 SetValid(Words(L1Instruction));
4 **for** $i := 1$ ***to*** m **do**
5 | L1Data ← *reset*; $j \leftarrow 0$;
6 | **for** $j := 1$ ***to*** *Length(Words(L1Data))* **do**
7 | | **if** *L1Data[j]* $\neq$ *L1Instruction[j]* **then** SetInvalid(L1Instruction[j]);
8 | **end**
9 **end**
10 **return** ValidAddresses(L1Instruction);

Algorithm 1: Algorithm to determine the addresses of stable words in the L1 Data Cache.

The execution of the Algorithm 1 in each processor instance would take at most $O(n \cdot m)$ operations, for a memory with n words and running m assessments. However, as the algorithm execution proceeds, the number of valid lines to be verified decreases. For instance, after 100 assessments, less than 5% of n words are still valid for a 64-KB SPUF with 16-bit words. Therefore, the algorithm should run faster than $O(n \cdot m)$. After running m assessments, one can select the appropriate set of addresses of SPUF that are valid for composing the set A of addresses (see Section 4.2). In the case of 128-bit key and 16-bit words, eight addresses would be needed to compose the set of addresses A. This information

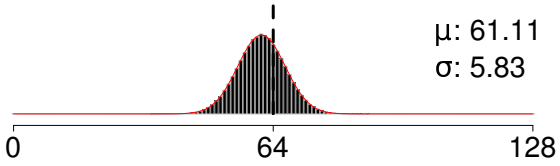


Figure 13: Hamming Distance distribution of 128-bit keys extracted from SPUFs.

is then recorded in non-volatile memory (probably the PTAG Memory) to be used every time the system is powered up.

It is also critical to validate whether the keys extracted as described have adequate statistical properties. In particular, it is important to analyze how extracted keys differ across different SRAM instances. The established method in the literature for measuring this property is the Hamming Distance. The ideal behavior would be a normal distribution centered at 64, for a 128-bit long key. The Hamming Distance of the extracted keys presented in Figure 13 shows a random distribution centered at 61.11. This is slightly skewed towards zero, but still close to the ideal distribution. Finally, one can notice that the same analysis and result applies to the extraction of r (see Figure 3).

Differently from previous architectures that use PUFs [32], CSHIA does not need additional PUF circuitry, since the processor's SRAMs cache are used as PUFs. Also, CSHIA key extraction is deeply integrated to the architecture and is the first of this kind, to the best of our knowledge. Nonetheless, comparison with others SPUF key extraction processes in the literature [21] still needs to be done. Experimental results presented in this section showed that good quality keys can be extracted from SRAMs with relatively straightforward procedures.

8 Conclusions and discussions

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

9 Future Work

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

10 Project Timeline

CSHIA is a long term project from its specification to a final silicon prototype. This proposal aims at specification in addition to some simulation and implementation. Specifically, for specification, we address all details that ensure high level security. For simulation, we generate data that under analyses may support or not decisions made in the system specification. Finally, for implementation, our goal is to develop a countermeasure for physical and side-channel attacks.

Table 1 describes the project timeline that contemplates the doctoral degree requirements and the research activities. We marked requirements and activities in regard to the semester either we finished them or we expect to finish. The items (a) through (e) are requirements, the latter items are activities.

Table 1: Timeline for the doctoral program.

Activities	**Semesters**								
	2013		**2014**		**2015**		**2016**		**2017**
	1º	2º	3º	4º	5º	6º	7º	8º	9º
(a) English Test Proficiency	✓								
(b) Ph.D. requirements (Courses credits)	✓	✓	✓						
(c) Qualifying Examination						✓			
(d) Publishing					✓	✓		✓	✓
(e) Defense of the Thesis									✓
(f) Patent				✓	✓				
(g) Internship or Ph.D. sandwich							✓	✓	

We would like to draw your attention to some items that we will explain in the following:

(d) We have an accepted paper at the Internation Conference on Hardware/Software Codesign and System Synthesis (October 4-9, 2015). We also intend to produce an extended version of the conference paper with additional results. This extended version will have some results that we presented in the Section 6 and our goal is to submit it to a journal by the end of this year (2015). Additionally, we expect to publish results obtained during the Ph.D. Sandwich.

(f) The agency of innovation at the UNICAMP (INOVA) has deposited a patent resulted from this work in the Brazilian National Industrial Property Institute (INPI

BR 10.2015.016831.4).

(g) We submitted a proposal of a one-year research internship to the FAPESP. This research internship aims at using ultimate equipment to study physical and side-channel attacks to CSHIA and PUFs. The internship will be under supervision of the Prof. Catherine H. Gebotys in the Department of Electrical and Computer Engineering at the University of Waterloo, where equipment to perform reverse engineering, fault injection via Laser/EM, and microprobing, has been recently acquired.

References

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

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

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

[4] M. Bhargava and K. Mai, "An efficient reliable puf-based cryptographic key generator in 65nm cmos," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–6.

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

[6] D. Champagne, R. Elbaz, and R. Lee, "The reduced address space (ras) for application memory authentication," in *Information Security*, ser. Lecture Notes in Computer Science, T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, Eds. Springer Berlin Heidelberg, 2008, vol. 5222, pp. 47–63, Available: http://dx.doi.org/10.1007/978-3-540-85886-7_4.

[7] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. Suh, "Incremental multiset hash functions and their application to memory integrity checking," in *Advances in Cryptology - ASIACRYPT 2003*, ser. Lecture Notes in Computer Science, C.-S. Laih, Ed. Springer Berlin Heidelberg, 2003, vol. 2894, pp. 188–207.

[8] J. Delvaux and I. Verbauwhede, "Attacking puf-based pattern matching key generators via helper data manipulation," in *Topics in Cryptology – CT-RSA 2014*, ser. Lecture Notes in Computer Science, J. Benaloh, Ed. Springer International Publishing, 2014, vol. 8366, pp. 106–131, Available: http://dx.doi.org/10.1007/978-3-319-04852-9_6.

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

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

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

[12] O. Goldreich, *Foundations of Cryptography: Basic Tools*. New York, NY, USA: Cambridge University Press, 2004.

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

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

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

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

[17] M. Hong, H. Guo, and 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*, ser. CASES '12. New York, NY, USA: ACM, 2012, pp. 17–26, Available: http://doi.acm.org/10.1145/2380403.2380414.

[18] S. Katzenbeisser, U. Kocabaş, V. Rožić, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann, "Pufs: Myth, fact or busted? a security evaluation of physically unclonable functions (pufs) cast in silicon," in *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 283–301, Available: http://dx.doi.org/10.1007/978-3-642-33027-8_17.

[19] V. Leest, E. Sluis, G.-J. Schrijen, P. Tuyls, and H. Handschuh, "Efficient implementation of true random number generator based on sram pufs," in *Cryptography and Security: From Theory to Applications*, ser. Lecture Notes in Computer Science, D. Naccache, Ed. Springer Berlin Heidelberg, 2012, vol. 6805, pp. 300–318, Available: http://dx.doi.org/10.1007/978-3-642-28368-0_20.

[20] D. Lim, J. W. Lee, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas, "Extracting secret keys from integrated circuits," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, no. 10, pp. 1200–1205, Oct. 2005.

[21] R. Maes, P. Tuyls, and I. Verbauwhede, "Low-overhead implementation of a soft decision helper data algorithm for SRAM pufs," in *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, 2009, pp. 332–347, Available: http://dx.doi.org/10.1007/978-3-642-04138-9_24.

[22] M. Majzoobi, F. Koushanfar, and M. Potkonjak, "Techniques for design and implementation of secure reconfigurable pufs," *ACM Trans. R. Tech. S.*, vol. 2, no. 1, pp. 1–33, 2009.

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

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

[25] R. S. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical one-way functions," *Science*, vol. 297, pp. 2026–2030, 2002, Available: http://web.media.mit.edu/~brecht/papers/02.PapEA.powf.pdf.

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

[27] U. Rührmair, "Oblivious transfer based on physical unclonable functions," in *Trust and Trustworthy Computing*, ser. LNCS, A. Acquisti, S. Smith, and A.-R. Sadeghi, Eds., 2010, vol. 6101, pp. 430–440, Available: http://dx.doi.org/10.1007/978-3-642-13869-0_31.

[28] A. Sadeghi and D. Naccache, Eds., *Towards Hardware-Intrinsic Security - Foundations and Practice*, ser. Information Security and Cryptography, 2010, Available: http://dx.doi.org/10.1007/978-3-642-14452-3.

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

[30] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th Annual International Conference on Supercomputing*, ser. ICS '03. New York, NY, USA: ACM, 2003, pp. 160–171, Available: http://doi.acm.org/10.1145/782814.782838.

[31] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *Proceedings of the 44th annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 9–14, Available: http://doi.acm.org/10.1145/1278480.1278484.

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

[33] S. Tajik, E. Dietz, S. Frohmann, J.-P. Seifert, D. Nedospasov, C. Helfmeier, C. Boit, and H. Dittrich, "Physical characterization of arbiter pufs," in *Cryptographic Hardware and Embedded Systems – CHES 2014*, ser. Lecture Notes in Computer Science, L. Batina and M. Robshaw, Eds. Springer Berlin Heidelberg, 2014, vol. 8731, pp. 493–509, Available: http://dx.doi.org/10.1007/978-3-662-44709-3_27.

[34] B. Škorić, P. Tuyls, and W. Ophey, "Robust key extraction from physical uncloneable functions," in *Applied Cryptography and Network Security*, ser. LNCS, 2005, vol. 3531, pp. 407–422.